UiT The Arctic University of Norway

The Faculty of Science and Technology
Department of Computer Science

Towards Improved Support for Conflict-Free Replicated Data Types

—

Sigbjørn Rostad

INF-3981 Master's Thesis in Computer Science - June 2020

Abstract

Conflict-free Replicated Data Types (CRDTs) are distributed data types which ensure Strong Eventual Consistency (SEL), and also has properties such as commutativity and idempotence. There are many variations of CRDTs, and the ones we will study are state-based delta CRDTs. State-based CRDTs is a variation where the CRDT instances synchronize by sending their state to each other. An improvement to this is delta CRDTs which is yet another variation where only the difference of mutations is disseminated. We will explore some of the existing CRDTs, but also present some new CRDT designs and their implementations. One of the new CRDTs presented is Causal Length Set (CLSet), which is a simple, yet effective state-based delta CRDT. It does not use dots as causal context as many other CRDTs; it merely uses a single integer. With this minimalist CRDT design, we can achieve both great performance and a small memory footprint. Another CRDT presented is Multiple Value Map (MVMap), a CRDT which focuses on convenience as opposed to performance.

Contents

List of Figures

List of Listings

List of Abbreviations

AWLWWMap	Add-Wins Last-Write-Wins Map
AWSet	Add-Wins Set
CRDTs	Conflict-free Replicated Data Types
GSet	Grow-Only Set
LWW	Last-Write-Wins
MVMap	Multi-Value Map
MVReg	Multi-Value Register
ORSet	Observed-Remove Set
PID	Process Identifier
Poset	Partially Ordered Set
SEC	Strong Eventual Consistency
TFAWSet	Tombstone-Free Add-Wins Set

1 Introduction

The CAP theorem [3] describes three different properties for distributed systems, consistency (C) - which refer to multiple sites having the same copy of a piece of data, availability (A) - which refer to the data always being available and partition-tolerance (P) - which refer to the system still working after being partitioned on the network. These properties are not binary but continuous. The CAP theorem states that one can only achieve at most two of these properties. Conflict-free Replicated Data Types (CRDTs) can achieve both availability and partition-tolerance because of replication of data over multiple sites. However, CRDTs lack consistency as data can take some time to synchronize across sites; this makes them eventually consistent.

There are many variations of CRDTs. State-based, operation-based, delta, add-wins, last-write-wins and many more, even combinations of multiple variations is possible. We will focus more on state-based delta CRDTs. This CRDT variation creates δs (deltas) based on the mutations it receives, which in turn is disseminated to the other CRDT sites. The other sites will incorporate updates requiring only the δ to reach the same state as its neighbours. The article *Delta State Replicated Data Types* [15] has been a source for multiple of the existing CRDT designs mentioned in this thesis. In addition, an already existing CRDT implementation [5] has been useful for implementing new CRDT implementations.

We will look at some of the theory behind CRDTs. We will explore some existing CRDT designs and their properties, then try to bring something new to the table by mixing existing designs with some new ideas.

Existing set CRDTs today are expensive, and multi-value register CRDTs push the responsibility of resolving conflicts to the applications. The contribution of this thesis is CLSet, a new set CRDT, and MVMap, a new multi-value CRDT.

The Causal Length Set (CLSet) design is based on causal lengths. Causal length is a property observed on a CRDT called Grow-only Set (GSet). The causal length is a by-product of its causal context. With this knowledge, we can create CLSet. The main contribution of CLSet is a performant general-purpose set CRDT. The results can be seen in Chapter 6. The CLSet CRDT has been published in the *PaPoC 2020 conference proceeding* [17].

Multi-value CRDTs is a particular type of CRDT, where concurrent writes to the same key across multiple sites will all be included in the CRDT, as such there are multiple values on a key, and it is up to the application to decide which value to use. The new CRDT design MVMap has a slight change such that whenever the CRDT is read, the values are automatically sent through a 'resolver' which decide what to do with the values. The main contribution of MVMap is a multi-value CRDT where the responsibility of resolving conflicts is shifted away from the application and more towards the CRDT itself.

1.1 Outline

Theoretical Background (Chapter 2) gives a basic introduction to theory behind some CRDTs. It also highlights different CRDT designs and their variations. Reading this chapter helps the reader get up to speed and get an overview of CRDTs and make it easier to understand the later chapters.

Exisiting CRDTs (Chapter 3) explains some existing CRDT designs we have today.

New CRDTs (Chapter 4) introduces some new CRDT designs which solve some of the existing CRDT problems.

Implementation (Chapter 5) go deeper into the implementation details of the new and some existing CRDTs. This chapter includes code samples from an implementation written in Elixir.

Experiments & Results (Chapter 6) is the chapter where we put the CRDTs discussed in Chapter 5 to the test. The CRDTs are benchmarked both in terms of response time and memory footprint, to get a more accurate representation of how these CRDTs might fare in actual applications.

Finally the thesis is summarized in the Conclusion (Chapter 7).

/2 Theoretical Background

2.1 Join-Semilattice

Conflict-free Replicated Data Types (CRDTS) is based on the theory of set, order and lattice, thus understanding how these work is going to be important for understanding CRDTs on a more fundamental level. On the surface, with only a little pre-existing knowledge, CRDTs are relatively simple to understand, but to understand the 'how' we might need to learn some set, order and lattice theory.

2.1.1 Partially Ordered Set

A Partially Ordered Set (POset) is a way to say that some elements in a set precede others. The word 'partial' in 'Partially Ordered Set' indicates that not every element in the set have to precede another element [14].

2.1.2 Hasse Diagram

Hasse Diagram is a diagram used in order theory. *Wolfram Mathworld* [8] defines it in the following way:

> *A Hasse diagram is a graphical rendering of a partially ordered set displayed via the cover relation of the partially ordered set with an*

implied upward orientation. A point is drawn for each element of the poset, and line segments are drawn between these points according to the following two rules:

- *If $x < y$ in the poset, then the point corresponding to x appears lower in the drawing than the point corresponding to y.*
- *The line segment between the points corresponding to any two elements x and y of the poset is included in the drawing if and only if x covers y or y covers x*

The definition explains the Hasse diagram as a display of a poset. As explained in section 2.1.1, elements in posets might precede others. This is also the reason for the Hasse diagram having an 'implied upward orientation'. Elements at the bottom come before elements at the top. The line segments in the diagram link two elements together only if the higher-order element contains the lower-order element.

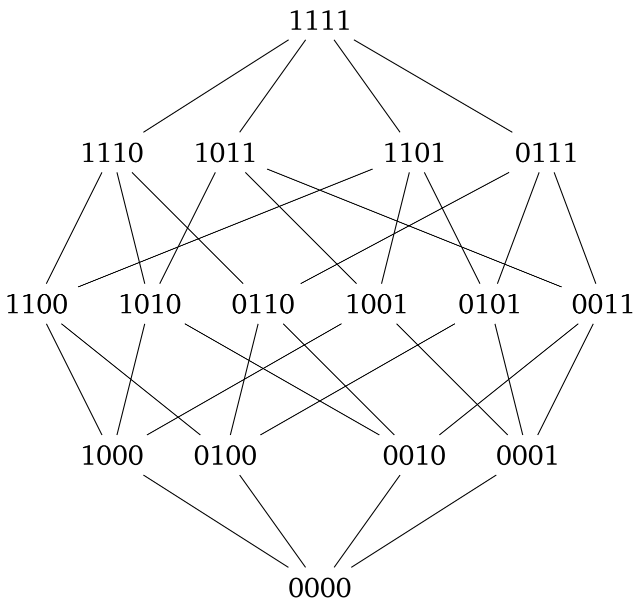


Figure 2.1: Hasse diagram

Figure 2.1 shows an example of a Hasse diagram. Notice that the element at the bottom says 0000 and the element at the top says 1111. The other elements in between are all possible permutations and their corresponding links. The Hasse diagram displays the minimum amount of information necessary to describe the order of the elements. Hasse diagrams also have a natural upwards direction. If we look at the bottom element, 0000, it has four links. Each link is leading to a possible state change. In this case, the logic of the diagram is that only one digit can change at a time, between 0 and 1. This means that the bottom element has four links because all four of its digits is subject to change. If we move one state upwards to 1000, this node has three links upwards as one of the digits has already changed, but it also has one link going back to the previous state. If we keep going upwards, the number of links above will decrease, and the number of links below will continue to increase until all digits have changed to 1.

2.1.3 Semilattice

Semilattice can be defined as such:

> *A join-semilattice is a poset which admits all finite joins, or equivalently which admits a bottom element $\perp$ and binary joins $a \vee b$. If we think of a poset as a category, a join-semilattice is the same as a poset with finite colimits, or equivalently, a poset with finite coproducts.*
>
> *Dually, a meet-semilattice is a poset which admits all finite meets, including a top element $\top$ and binary meets $\wedge$. Once again, $\wedge$ is commutative, associative, unital for $\top$, and idempotent. Once again we can recover the order from it, but this time defining $a \leq b$ to mean $a \wedge b = a$. If we think of a poset as a category, a meet-semilattice is the same as a poset with finite limits, or equivalently, a poset with finite products* [12].

To elaborate on the definition; a join-semilattice is a construct for displaying the state transitions (joins) of a system. The semilattice is a poset and implies convergence. Thus, it starts with a bottom $\perp$ element and ends with convergence. The meet-semilattice is similar to the join-semilattice; it starts with a top $\top$ element, and displays which two elements meet for every element.

$$\begin{tikzcd}
 & \{a,b,c\} \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
\{a,b\} \arrow[d, dash] \arrow[dr, dash] & \{a,c\} \arrow[dl, dash] \arrow[dr, dash] & \{b,c\} \arrow[dl, dash] \arrow[d, dash] \\
\{a\} \arrow[dr, dash] & \{b\} \arrow[d, dash] & \{c\} \arrow[dl, dash] \\
 & \perp &
\end{tikzcd}$$

Figure 2.2: Hasse diagram of a semilattice

Figure 2.2 show an example of a semilattice. Notice it begins with a bottom $\perp$ element. From the bottom element, we have in this example, three elements being added a, b and c, illustrated by the three upward links. Now look at state $\{a\}$, and notice that there are two possible links upwards. The first link adds the element b, and the second link adds the element c. This effectively means that each link illustrates a join $\vee$ of the other permutations of the state. After the fact, all states in the second-order $\{a, b\}$, $\{a, c\}$ and $\{b, c\}$ have two elements. From this point, there is only one possibility, and that is that each state adds the last remaining element; $\{a, b\}$ joined $\vee$ with any of the other states will receive c , $\{a, c\}$ joined $\vee$ with any of the other states will receive b, and $\{b, c\}$ joined $\vee$ with any of the other states will receive a. The result is convergence as the final state contains all elements, $\{a, b, c\}$.

2.2 Idempotence

Idempotence in math and computer science means that an operation applied to a variable does not affect the value beyond the initial application [9].

```
let data = {x: 10}
{x: 10}

data.x = 20
{x: 20}

data.x = 20
{x: 20}
```

Listing 2.1: Idempotent example in JavaScript

Listing 2.1 show an example of idempotence, where only the first application changed the value of 'x' from 10 to 20, the second operation changed nothing. This shows that the operations in Figure 2.1 are idempotent.

2.3 Commutativity

Commutative means that a series of operations can be applied in any order still give the same result [4].

```
let data = {x: 10}
{x: 10}

data.x += 10
{x: 20}

data.x += 5
{x: 25}
```

Listing 2.2: Commutativity example 1/2 in JavaScript

```
let data = {x: 10}
{x: 10}

data.x += 5
{x: 15}

data.x += 10
{x: 25}
```

Listing 2.3: Commutativity example 2/2 in JavaScript

Listing 2.2 show an example of commutativity. Two operations are applied to a set-element. First, the value is incremented by 10, and then it is incremented by 5. Listing 2.3 show the same operations, just in the opposite order. As can be seen, the result is the same. Since the result is the same in both Figure 2.2 and Figure 2.3, we can say that the operations are commutative.

2.4 Causality

Causal means that something relates to, or is caused by something else. If x and y have a causal relation, the value of x is going to be affected by the value of y. Causality is essential for ensuring correctness in data structures. For example, when joining two sets, causality can help decide which elements to keep and which elements to throw away. Causality in terms of maintaining data structures revolves around keeping metadata for items added to the data structure, then afterwards whenever there is a conflict of some sort, utilize the information from the metadata to resolve the conflicts. How exactly causality is implemented is very application-specific, so for more details, one should focus on understanding the implementation of the application itself.

Figure 2.3 from [15] defines their version of causal context used in research they performed on CRDTs. They call it causal context which compared to causality is the same thing when we are talking about CRDTs, state and context. Figure

$$\begin{aligned}
\mathsf{CausalContext} &= \mathcal{P}(\mathbb{I} \times \mathbb{N}) \\
\mathsf{max}_i(c) &= \mathsf{max}(\{n \mid (i,n) \in c\} \cup \{0\}) \\
\mathsf{next}_i(c) &= (i, \mathsf{max}_i(c) + 1)
\end{aligned}$$

Figure 2.3: Causal context from *Delta State Replicated Data Types* [15]

2.3 is described as following in their article:

> *As seen in the Figure, a causal context is a set of dots. We define two functions over causal contexts: $max_i(c)$ gives the maximum sequence number for pairs in c from replica i, or 0 if there is no such dot; $next_i(c)$ produces the next available sequence number for replica i given set of events in c* [15].

The takeaway here is that causal context defines sequence numbers for data-pairs in the CRDT. There is one function for getting the highest sequence number, and one function for generating the next sequence number based on the current state of the CRDT.

2.5 Conflict-free Replicated Data Types

CRDTs are data abstractions that guarantee convergence for replicated data. CRDTs associate every element in the CRDT with causal contexts as metadata. Having a context for every element is expensive and will scale poorly when the number of items in the CRDT increases. Thus reducing memory footprint and reducing operation (addition and removal) time is of high interest.

A CRDT in practice consists of multiple instances, sometimes also called sites or nodes. The sites are all independent without any sort of leader- or master-site. There is no need for a master-site since the CRDTs are ensured to converge. Because of convergence, the end-result in all the CRDT sites is the same even if additions/removals is received in a differing order. Even though we have these challenges, the CRDT sites can still reach the same state. When all sites have the same state, the CRDT has converged.

The sites are not connected through the CRDT, often most the CRDTs are entirely decoupled from any other pieces of code, and then there is an additional layer built on top of the CRDT to handle synchronization between the different sites. For reference, the CRDT implementations can be called the *CRDT layer*, and the synchronization implementation can be called the *synchronization layer*. It is good to decouple these two layers such that they can be modified or replaced without affecting each other. The synchronization layer can be seen as having two phases, the *operation phase* and the *synchronization phase*.

The operation phase is the phase where the user can do operations such as adding elements to the CRDT, removing elements from the CRDT and reading the content of the CRDT. The synchronization phase is the phase where a site sends data to other sites. What this data entails depends on the implementation. It usually is either a set of operations to be executed or a piece of state to be merged into the other sites' state.

They way CRDTs reach convergence is through a synchronization phase which will send data from one site of a CRDT to the other sites of the CRDT. The CRDT site will look at the new data, and decide if they should merge its data with the newly received data. The received data differs between *state-based CRDTs* and *operation-based CRDTs*; this is further explained in the next two sub-sections.

The uses of CRDTs are many. One example is to incorporate it into an application to handle and synchronize data. With a CRDT in the application structure, data would automatically synchronize between the different units in the application, and one would only need to read the CRDT occasionally to get the most recent updates.

CRDTs are also a promising technology for building *Local-First Software*.

> *In local-first applications, we treat the copy of the data on the users' local device — laptop, tablet, or phone — as the primary copy. Servers still exist, but they hold secondary copies of the data in order to assist with access from multiple devices* [10].

Using CRDTs in local first software, users can do changes locally without an internet connection. At a later time when the connection is recovered, multiple sites can concurrently and asynchronously update data stored locally and automatically merge updates performed at other sites. With these properties we can categorize the CRDT as *Strongly Eventually Consistent*; all sites have the same data state when the sites have incorporated the same set of updates, regardless of the order in which they incorporated the updates. A site queries and updates its local replica without coordination with other sites. The sites will synchronize with each other until they reach convergence.

There are two main categories of CRDTs; state-based and operation-based. Within these categories, there are subcategories with different implementations. Therefore, when considering CRDTs, one should study the differences and pick the CRDT with the properties that fit the application best.

2.5.1 State-Based CRDTs

State-based CRDTs is one of the two main categories of CRDTs. The main idea is that when something is updated within the CRDT, it will merely send its state to the other sites. State-based CRDTs are both commutative and idempotent.

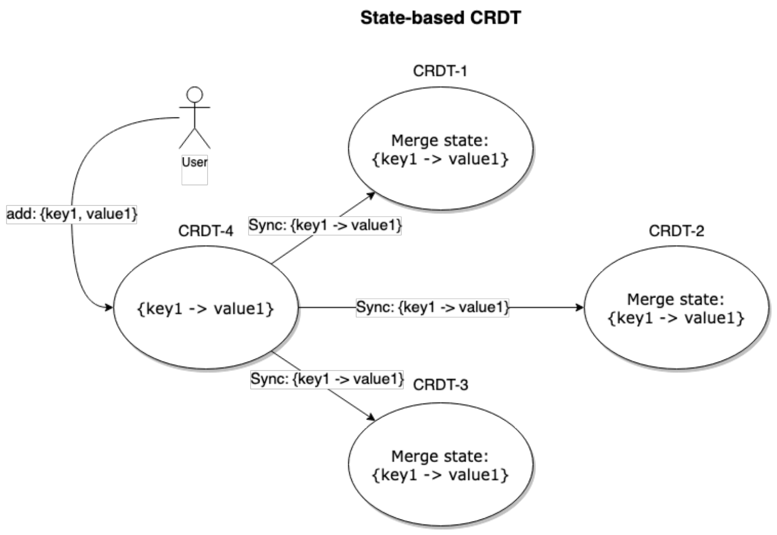


Figure 2.4: State-Based CRDT Example

Figure 2.4 shows an example of a state-based CRDT. The user performs an add-operation, inserting a key-value pair into the CRDT. The CRDT performs the add-operation, adding the pair into its state. During the next sync, the CRDT will send its updated state to the neighbouring CRDTs. The neighbours notice that the state they received is newer than their state, this is typically through a timestamp or some sort of vector clock. Furthermore, the received state is merged with its current state.

> *However, a major drawback in current state-based CRDTs is the communication overhead of shipping the entire state, which can get very large. For instance, the size of a counter CRDT (a vector of integer counters, one per replica) increases with the number of replicas. In contrast, in a grow-only Set, the state size depends on the set size, that grows as more operations are invoked* [15].

The solution to this is delta CRDTs. Delta CRDTs instead only send a δ (delta) of the state to the other sites. In delta CRDTs, the δ is the difference in the state before and after an update (add/remove). However, since δs is only fragments of the actual CRDT state, it requires specific merge algorithms to maintain the semantics of the CRDTs. The merge algorithm joins any pair of states so that the set of join operations create a join-semilattice. The merge algorithm is decided by the CRDT type.

CRDTs internal state must always monotonically increase through the update function. The reason for this is as soon as some piece of information is removed, the join algorithm might not be able to join a pair of states correctly. The result of an incorrect join might be that an item that was previously removed might return.

2.5.2 Operation-Based CRDTs

Operation-based CRDTs is the second category of CRDTs. The main idea behind operation-based CRDTs is that whenever something is updated, only the operation itself propagates to the other sites. The other sites execute the operation just as the original did. The sites should now be converged. Operation-based CRDTs are commutative, meaning that the order of the applied operations does not matter. However, operation-based CRDTs are not idempotent, meaning that operations applied multiple times would change the value beyond the first application.

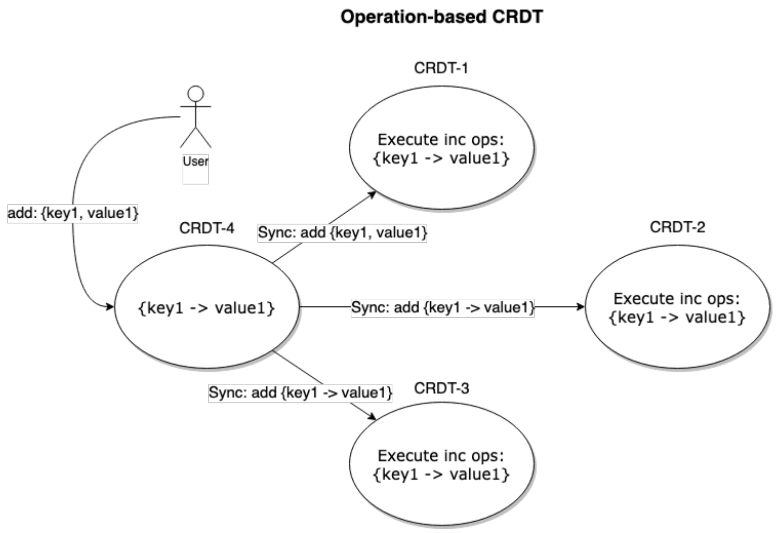

Operation-based CRDT
CRDT-1
User
Execute inc ops: {key1 -> value1}
add: {key1, value1}
CRDT-4
Sync: add {key1, value1}
CRDT-2
{key1 -> value1}
Sync: add {key1 -> value1}
Execute inc ops: {key1 -> value1}
Sync: add {key1 -> value1}
CRDT-3
Execute inc ops: {key1 -> value1}

Figure 2.5: Operation-Based CRDT Example

Figure 2.5 shows an example of an operation-based CRDT. The user performs an add-operation with a key-value pair. 'CRDT-4' executes the add-operation, inserting it into its state. Afterwards, it will send the operation to the neighbouring CRDTs. They will receive the add-operation and execute it the same way 'CRDT-4' did.

The drawback of operation-based CRDTs is that they are not idempotent. This means that operations can only be applied once. Therefore when syncing it requires additional logic to make sure all neighbours receive precisely one of each operation.

2.5.3 Other Variations of CRDTs

There are many variations of CRDTs, some of those not mentioned yet are:

- Delta CRDTs
- Named CRDTs
- Add-Wins (AW)
- Remove-Wins (RW)
- Last-Write-Wins (LWW)

Delta CRDTs refers to the data being synchronized between multiple sites should only be the difference since the last synchronization. Each time a mutation (add/remove) is applied to the CRDT, a δ is created. During the next synchronization, only this δ will be distributed to the other sites. The benefit to this is much less data transfer, especially for state-based CRDTs, but also a lower amount of memory overhead since the CRDT will not be working with any redundant data.

As mentioned, delta CRDTs are a variation of CRDTs, but there is no limiting factor for which CRDT that can become a delta CRDT. Any CRDT can be transformed into a delta CRDT with some modification to their mutator and their join function. The mutator will instead return only the change that has been done. The next synchronization will now instead simply distribute the δs gathered since the last synchronization. The CRDTs will also have a different join function. This join function will still look at the incoming δ to decide if it should incorporate it. As they are just incorporating δs, it might result in very minor changes, but the result is precisely the same as a normal CRDT, just with less overhead.

Named CRDTs refers to the mutators add_i and $remove_i$ being denoted by an i (replica identifier). This means that the mutators are *named* as opposed to being *anonymous*. Named CRDT and named mutators are synonymous. This means that mutations can only update the part of the state that is specific to the replica performing the mutation [15].

Add-Wins refers to a CRDT prioritizing add-operations when receiving multiple concurrent operations on the same key (maps) or value (sets). If a CRDT were to receive two concurrent operations, one remove and one add, for a key. The remove-operation would be disregarded while the add-operation would be performed.

Remove-Wins refers to a CRDT prioritizing remove-operations. It is the same as the Add-Wins, except for prioritizing remove instead.

Last-Write-Wins refers to CRDTs prioritizing later writes over earlier writes. The prioritizing is both in terms of concurrent operations and single operations. Every operation is paired with a timestamp, vector clock or any other way of measuring time. Then the CRDT will utilize this to decide which operations to apply.

These variations can be used by CRDTs. Also, multiple variations can be used at once, but there are exceptions such as: add-wins and remove-wins at the same time. The variations should be seen as defining a consistent behaviour for a CRDT. Having a defined behaviour is vital for keeping the state of the CRDT consistent with the other sites.

2.6 Further Reading

The following articles/papers would help for a better understanding of CRDTs.

Conflict-free Replicated Data Types [16], is an article by Marc Shapiro, Nuno Preguiça, Carlos Baquero and Marek Zawirski. They discuss large distributed systems and the importance of replication and Strong Eventual Consistency (SEC) for these systems to be reliable and available. CRDTs can provide these properties, and also ease problems of unreliable internet connections. They then propose an approach which is a solution to these problems.

Scalable and Accurate Causality Tracking for Eventually Consistent Stores [1], is an article by Paulo Sérgio Almeida, Carlos Baquero, Ricardo Gonçalves, Nuno Preguiça and Victor Fonte. This article discusses causality between multiple replicas and how performance takes a big hit whenever concurrent updates happen. They propose a new logical clock mechanism and a logical clock framework that together support a traditional key-value store API, while capturing causality in an accurate and scalable way, avoiding false conflicts.

Delta State Replicated Data Types [15] is an article by Paulo Sérgio Almeida, Ali Shoker, Carlos Baquero. This article presents different CRDT designs that are based on a new strategy where only δs (deltas) of updates are propagated through a system. This drastically reduces the traffic and overhead that the CRDT produces, which in turn increases the performance, while still having the same properties. These ideas have been implemented by a Add-Wins Last-Write-Wins Map (AWLWWMap) CRDT implementation [5], which have been a basis for the implementation of CLSet in this thesis.

3 Existing CRDTs

CRDTs is an abstraction for data replicated at multiple sites and for merging updates from these sites without any conflicts. CRDTs data is guaranteed to be strongly eventually consistent. This means that all sites have the same state when they have incorporated the same set of updates, regardless of the order. Differing applications call for different CRDT implementations.

Some of the significant differences are how the data is stored or represented, how data of multiple sites are merged and how data gets output when reading. There exists many different CRDT implementations and even many undiscovered. Some of these implementations use 'set' as their baseline data structure. In contrast, others might use 'map'. The difference between set and map is implementation-specific and does not matter as long as the properties of the CRDT fulfil the requirements of the application.

Set and map are the two fundamental data types used in the CRDTs which are going to be discussed. When these data types are used in CRDTs, elements are associated with some causal context.

> *Causal means that something relates to, or is caused by something else. If x and y have a causal relation, the value of x is going to be affected by the value of y.*

This is the general meaning of causal. In terms of CRDTs, causal context refers to values kept in state, additional to the actual elements themselves. During

$$\begin{aligned} \mathsf{CausalContext} &= \mathcal{P}(\mathbb{I} \times \mathbb{N}) \\ \mathsf{max}_i(c) &= \mathsf{max}(\{n \mid (i,n) \in c\} \cup \{0\}) \\ \mathsf{next}_i(c) &= (i, \mathsf{max}_i(c) + 1) \end{aligned}$$

Figure 3.1: Causal context from *Delta State Replicated Data Types* [15]

$$\begin{aligned} &\mathsf{DotStore} \\ \mathsf{dots}\langle S : \mathsf{DotStore}\rangle &: S \to \mathcal{P}(\mathbb{I} \times \mathbb{N}) \\[1em] \mathsf{DotSet} : \mathsf{DotStore} &= \mathcal{P}(\mathbb{I} \times \mathbb{N}) \\ \mathsf{dots}(s) &= s \\[1em] \mathsf{DotFun}\langle V : \mathsf{Lattice}\rangle : \mathsf{DotStore} &= \mathbb{I} \times \mathbb{N} \hookrightarrow V \\ \mathsf{dots}(s) &= \mathsf{dom}\ s \\[1em] \mathsf{DotMap}\langle K,\ V : \mathsf{DotStore}\rangle : \mathsf{DotStore} &= K \hookrightarrow V \\ \mathsf{dots}(m) &= \bigcup \{\mathsf{dots}(v) \mid (\_, v) \in m\} \end{aligned}$$

Figure 3.2: Dot stores from *Delta State Replicated Data Types* [15]

synchronization, elements and the causal context will propagate through the CRDT sites. The context will help other sites decide if they need to apply an update or not.

Figure 3.1 show an example of a causal context. Whenever elements are added to the CRDT, an identifier is added. The identifier is a replica-unique integer which is appended to a globally unique replica identifier. The integer represents the event of adding this element. The elements being added to the CRDT is paired with identifiers in the following way, *(i, 1), (i, 2),...*. The identifiers can be collected such that we can know which elements are known to which replica. This pair can be called a dot, and the collection of dots can be called causal context [15]. During synchronization, other sites can compare the identifiers to decide if the element should be added or not.

Figure 3.2 show an example of dot stores [15]. A dot store can be seen as an extension of a causal context seen in Figure 2.3. The dot store will contain all event identifiers. The dot store can be queried about the identifiers corresponding to relevant events. Figure 3.2 show three examples of dot stores:

- DotSet - Is a set of dots
- DotFun - Is a function that can map from dots to some lattice V
- DotMap - Is a map from some K to some dot store V

The rest of this chapter will discuss some different CRDT designs and their properties.

3.1 Grow-only Set

Grow-Only Set (GSet) is one of the more simplistic CRDTs. It is a set CRDT, so it does not do any mapping between keys and values. It only supports add-operations which means that it is not fully a general-purpose CRDT, but there are still situations where a GSet is sufficient.

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

Figure 3.3: GSet CRDT

Figure 3.3 show the definition of a GSet. We can see that the add-operation is simply a $\cup$ (union) of the set and the new element. Therefore the δ (delta) of the add is either a set containing only $\{e\}$ or an empty set. The merge/join of GSet is also simply a $\cup$.

GSet only relies on the elements themself as causal context. This is because as previously mentioned, GSet is a set CRDT which only support add-operations. This combination of properties enables GSet to only store the elements with no additional metadata. During addition and join operations, GSet will match the actual elements with each other to decide which elements to keep and which to discard.

$$\begin{tikzcd}
 & \{a,b,c\} \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
\{a,b\} \arrow[d, dash] \arrow[dr, dash] & \{a,c\} \arrow[dl, dash] \arrow[dr, dash] & \{b,c\} \arrow[dl, dash] \arrow[d, dash] \\
\{a\} \arrow[dr, dash] & \{b\} \arrow[d, dash] & \{c\} \arrow[dl, dash] \\
 & \{\} &
\end{tikzcd}$$

Figure 3.4: GSet Hasse Diagram of States

Figure 3.4 is a Hasse diagram of GSet states. Between joins, no information is ever lost. Every new state subsumes the old one through a ∪-operation.

3.2 Add-Wins Set

Add-Wins Set (AWSet) is not a specific CRDT, but instead a general idea that when the CRDT receives multiple concurrent operations, it should favour add-operations. Add-Wins was previously mentioned in 2.5.3 - *Theoretical Background, Other Variations of CRDTs*, as one of the defined behaviours for keeping CRDTs consistent. In this case, it is a set designed with the add-wins behaviour. As there are multiple ways to design an AWSet, we are going to look at a specific CRDT from the article *Delta State Replicated Data Types* [15] shown in Figure 3.5.

$$\begin{aligned}
\mathsf{AWSet}\langle E\rangle &= \mathsf{Causal}\langle\mathsf{DotMap}\langle E, \mathsf{DotSet}\rangle\rangle \\
\mathsf{add}_i^\delta(e,(m,c)) &= (\{e \mapsto d\}, d \cup m(e)) \quad \textbf{where } d = \{\mathsf{next}_i(c)\} \\
\mathsf{remove}_i^\delta(e,(m,c)) &= (\{\}, m(e)) \\
\mathsf{clear}_i^\delta((m,c)) &= (\{\}, \mathsf{dots}(m)) \\
\mathsf{elements}((m,c)) &= \mathsf{dom}\, m
\end{aligned}$$

Figure 3.5: Delta CRDT Add-Win Set, Replica i

Figure 3.5 uses a map from elements to sets of dots as dot store. When an element is added, all dots in the corresponding entry will be replaced by a singleton set containing a new dot. If a DotSet for some element were to become empty, such as when removing the element, the join for DotMap ⟨E, DotSet⟩ will remove the entry from the resulting map. Concurrently created dots are preserved when

> *joining. The clear delta mutator will put all dots from the dot store in the causal context, to be removed when joining. As only non-empty entries are kept in the map, the set of elements corresponds to the map domain* [15].

AWSet is a delta CRDT denoted by the δ (delta) symbol in the add_i^δ, $remove_i^\delta$ and $clear_i^\delta$, also notice that the mutators are denoted by an i (replica identifier) which means that the mutators are named as opposed to being anonymous, this can also be called a named CRDT. This means that mutations update only the part of the state that is specific to that replica. The clear-mutator will remove all elements in the CRDT, but it will not remove dots. The reason that it does not remove dots is that if it were to receive a late synchronization from a different site, it would check its dots, and see that this element has already been seen and cleared, thus ignoring the element.

3.3 Tombstone-Free Add-Wins Set

Elements added to CRDTs are often stored alongside some metadata associated with that element. Tombstones are metadata associated with elements after they have been removed from the CRDT. Thus tombstone-free means not to store metadata after removals. This is also an AWSet, which means that add-operations are prioritized above others when receiving multiple concurrent operations.

Tombstone-Free Add-Wins Set (TFAWSet) is an alteration of a CRDT originally from *Delta State Replicated Data Types* [15], but with the name *Add-Wins Set (AWSet)*. It has since been implemented by [5]. They modified the CRDT to be a map instead of a set and also improved it to be tombstone-free. After these modifications, it was given the name *Add-Wins Last-Write-Wins Map (AWLWWMap)*, but a more fitting name would be TFAWMap. For the purpose of this thesis, TFAWMap has been modified to TFAWSet for a better comparison with the other CRDTs.

This means that TFAWSet is originally from the article *Delta State Replicated Data Types* [15] and it is described in the following way:

> *In an add-wins set, removals do not affect elements that have been concurrently added. In this sense, under concurrent updates, add will win over a remove of the same element. The implementation uses a map from elements to sets of dots as dot store. This data-type can be seen as a map from elements to enable-wins flags, but with a single*

$$
\begin{aligned}
\mathsf{TFAWSet} &\overset{\text{def}}{=} (E \hookrightarrow \mathcal{P}(dots)) \times \mathcal{P}(dots) \\
\mathsf{add}_i^{\delta}((m,c),e) &\overset{\text{def}}{=} \langle \{e \mapsto d\}, \{d\} \rangle \text{ where } d = \mathsf{next}_i(c) \\
\mathsf{remove}_i^{\delta}((m,c),e) &\overset{\text{def}}{=} \langle \{\}, m(e) \rangle \\
(m,c) \sqcup (m',c') &\overset{\text{def}}{=} \langle \{e \mapsto d'' \mid e \in \mathsf{dom}(m) \cup \mathsf{dom}(m') \\
&\qquad\qquad\qquad \wedge\, d'' \neq \{\}\}, \\
&\qquad c \cup c' \rangle \\
&\quad \text{where } d = m(e), d' = m'(e) \text{ and} \\
&\qquad d'' = (d \cap d') \cup (d - c') \cup (d' - c) \\
\mathsf{in}((m,c),e) &\overset{\text{def}}{=} e \in \mathsf{dom}(m) \\
\mathsf{all}((m,c)) &\overset{\text{def}}{=} \mathsf{dom}(m)
\end{aligned}
$$

Figure 3.6: TFAWSet delta-state CRDT

common causal context, and keeping only elements mapped to an enabled flag. When an element is added, all dots in the corresponding entry will be replaced by a singleton set containing a new dot. If a DotSet for some element were to become empty, such as when removing the element, the join for DotMap<E, DotSet> will remove the entry from the resulting map. Concurrently created dots are preserved when joining. The clear delta mutator will put all dots from the dot store in the causal context, to be removed when joining. As only non-empty entries are kept in the map, the set of elements corresponds to the map domain [15].

Figure 3.6 shows a definition of a TFAWSet. It is a CRDT that utilizes dots for causal context. This definition in specific is a delta CRDT, which means it only operates on δs, and a delta CRDT's mutator will only return a δ for the change that was done. This δ is used by the join function to apply the update, and a delta CRDT can do that with the minimal amount of operations necessary. TFAWSet's add_i^{δ} mutator takes $m(mutator), c(context)$ and $e(entry)$ as input and returns a tuple with the element and the next dot for the context.

3.4 Observed-Remove Set

A CRDT for general-purpose sets with both addition and removal operations can be designed as causal CRDTs [15]. Observed-Remove Set (ORSet) is one of these causal CRDTs; by utilizing two causal contexts it can keep track of both added and removed elements. Figure 3.7 from the *CLSet* [17] paper is a

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

Figure 3.7: ORSet CRDT

state-based ORSet. Originally defined in *Lasp* [11].

In an ORSet, every element is associated with two causal contexts. This can be described the following way in terms of the partial function:

Given a (total) function $f\colon \mathsf{dom}(f) \longrightarrow Y$ where $\mathsf{dom}(f) \subseteq X$. A *partial function* $f\colon X \hookrightarrow Y$ maps x to $\bot_Y$ if $x \notin \mathsf{dom}(f)$, where $\bot_Y$ is the *bottom element* of Y. For natural numbers $\mathbb{N}$, $\bot_{\mathbb{N}} = 0$. For $\mathcal{P}(S)$ ordered with $\subseteq$, $\bot_{\mathcal{P}(S)} = \{\}$.

As mentioned, the ORSet's causal context consists of two sets. One of which keeps track of elements added to the CRDT, whereas the other keeps track of elements removed from the CRDT. When an element is added and later removed, it is not deleted. It is instead added to the remove-context, hence the name *Observed-Remove Set*.

The partial function conveniently simplifies the specification of some mutators and the join operation. A causal context is a set of event identifiers, also known as *dots* (typically a pair of a site identifier and a site-specific sequence number).

Additions and removals are achieved with inflationary updates of the associated causal contexts. Using causal contexts, we can 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 [17].

3.5 Multi-Value Register

Multi-Value Register (MVReg) supports read and write operations just like other CRDTs. However, during concurrent writes, the join operation will include all the values that were written concurrently. Note that the next write under the same key will overwrite all the current values.

$$
\begin{aligned}
\mathsf{MVRegister}\langle V\rangle &= \mathsf{Causal}\langle\mathsf{DotFun}\langle V\rangle\rangle \\
\mathsf{write}^{\delta}_{i}(v,(m,c)) &= (\{d\mapsto v\},\{d\}\cup\mathsf{dom}\,m) \quad \textbf{where } d=\mathsf{next}_i(c) \\
\mathsf{clear}^{\delta}_{i}((m,c)) &= (\{\},\mathsf{dom}\,m) \\
\mathsf{read}((m,c)) &= \mathsf{ran}\,m
\end{aligned}
$$

Figure 3.8: MVRegister CRDT [15]

Figure 3.8 show a definition of a MVReg. Initial implementations of these registers tagged each value with a full version vector; here we introduce an optimized implementation that tags each value with a single dot, by using a DotFun ⟨V⟩ as dot store. In Figure 3.8, we can see that the write delta mutator returns a causal context with all dots in the store. They are then removed upon join, together with a single mapping from a new dot to the value written; as usual, the new dot is also put in the context. A clear operation simply removes current dots, leaving the register in the initial empty state. Reading simply returns all values mapped in the store [15].

The MVReg CRDT works much like other CRDTs. The MVReg *write* mutator is similar to the AWSet's *add* mutator. The MVReg *clear* mutator is identical to the AWSet's *clear* mutator. The main difference is that concurrent writes will write all elements into a list. This specific implementation in Figure 3.8 will return a random element on *read*. However, a different and more versatile approach is to return the whole list and let the application itself select the value it wants.

4 New CRDTs

In this chapter, we will look at two new CRDT implementations, CLSet [17] and MVMap. These CRDT designs are new additions of CRDTs in the state-based general-purpose category. CLSet is a new, very simple CRDT based on GSet. MVMap is a new CRDT inspired by MVRegister with a new interesting feature.

4.1 CLSet

Before looking at CLSet, we should look at GSet and study it to find a way to create a new general-purpose CRDT. By understanding GSet, we can better understand the reasoning behind the CLSet design.

The fundamental issue that a general-purpose set CRDT must address is how to identify the causality between the different addition and removal operations. The logic behind CLSet originates from the GSet CRDT.

Figure 4.1, left side, show the definition of a GSet. We can see that the add-operation is simply a $\cup$ (union) of the set and the new element. Therefore the δ (delta) of the add is either a set containing only $\{e\}$ or an empty set. The merge/join of GSet is also simply a $\cup$. The right side of Figure 4.1 is a

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

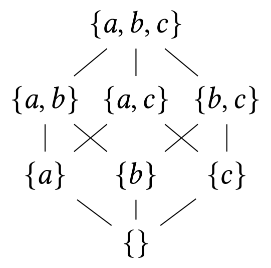
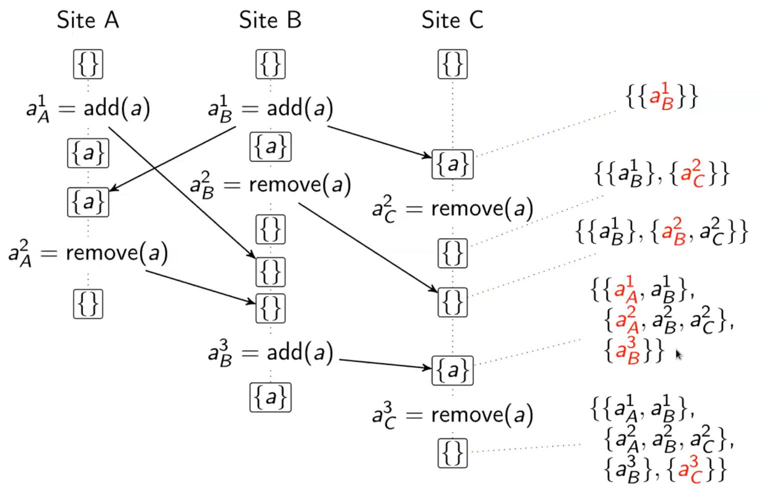


Figure 4.1: GSet CRDT and Hasse diagram of states



Site A
Site B
Site C
$a_A^1 = \mathsf{add}(a)$
$a_B^1 = \mathsf{add}(a)$
$a_B^2 = \mathsf{remove}(a)$
$a_C^2 = \mathsf{remove}(a)$
$a_A^2 = \mathsf{remove}(a)$
$a_B^3 = \mathsf{add}(a)$
$a_C^3 = \mathsf{remove}(a)$
$\{\{a_B^1\}\}$
$\{\{a_B^1\}, \{a_C^2\}\}$
$\{\{a_B^1\}, \{a_B^2, a_C^2\}\}$
$\{\{a_A^1, a_B^1\}, \{a_A^2, a_B^2, a_C^2\}, \{a_B^3\}\}$
$\{\{a_A^1, a_B^1\}, \{a_A^2, a_B^2, a_C^2\}, \{a_B^3\}, \{a_C^3\}\}$

Figure 4.2: GSet Equivalence Classes of Concurrent Updates

Hasse diagram of GSet states. The states are *partially ordered*[1], *inflationary*[2] and *join-irreducible*[3]. With this, we can also say that the GSet states form a join-semilattice.

A GSet CRDT is limited in its functionality because it can not remove elements. We would like a CRDT where we can both add and remove elements. This type of CRDT can be called a general-purpose CRDT. The challenge for a general-purpose CRDT lies in the consistency and causality of concurrent operations.

1. *A Partially Ordered Set* is a way to say that some elements in a set precede others.
2. *Inflationary* refer to the fact that the state is always increasing. No information is ever lost. Every new state subsumes the old one.
3. Join-irreducible refer to the join of two states where the resulting join is the minimal result. An element a in a lattice L is said to be join-irreducible if and only if a is not a bottom element, and, whenever $a = b \vee c$, then $a = b$ or $a = c$.

Figure 4.2 shows a scenario of concurrent updates. There are three sites, Site A, Site B and Site C. The top of the Figure is the beginning while the bottom of the Figure is the end. The right-most column shows the equivalence classes of the operations. All sites start with the same state. Site A and Site B start by adding a. We can see that two sites performing the same operation result in equivalent states. The arrows between the different sites indicate a synchronization between two sites. Synchronization of two equal states results in the same states as before the synchronization. Also, in terms of remove-operations, we can see that multiple remove-operations result in the same equivalent state.

Finally, look at the right-most column to see the equivalence classes of Site C at specific points in time, specified by the dotted line. Text in red indicates the most recent change. Whenever a site does an operation, it will update their local equivalence class data, then synchronize to the other sites.

The first look at the equivalence classes $\{\{a_B^1\}\}$ for Site C, we see that the first addition's equivalence class is added after the synchronization between Site B and Site C. The element also says that it originates from Site B and that the addition from Site A has not been received yet.

If we look at the second version of the equivalence classes $\{\{a_B^1\}, \{a_C^2\}\}$ we can see that Site C has removed element A, but not yet received the addition from Site A nor the removal from Site B. Also, the equivalence classes for these two operations are stored separately, because the first operation was an addition, the second was a removal, so the corresponding elements for these operations were not equal.

The third look at the equivalence classes $\{\{a_B^1\}, \{a_B^2, a_C^2\}\}$ show that it has received the remove-operation from Site B. We can see that the operation is placed together with the remove-operation from Site C.

The fourth look of the equivalence classes $\{\{a_A^1, a_B^1\}, \{a_A^2, a_B^2, a_C^2\}, \{a_B^3\}\}$ tells us that the first addition from Site A is received and that it is placed together with the first equivalence class. The first removal from Site A is also received and placed in the second equivalence class. The second add of element A is received from Site B, and is added in a new equivalence class.

The final look at the equivalence classes $\{\{a_A^1, a_B^1\}, \{a_A^2, a_B^2, a_C^2\}, \{a_B^3\}, \{a_C^3\}\}$ show us that Site C has removed a one last time, and this operation is placed in a new equivalence class.

The result is four different equivalence classes, after a set of operations on these three sites. The takeaway here is that at four places in time did this CRDT system have equivalent states, the same number of times as the number of equivalence classes. The CRDT system as a whole did the following operations (if we ignore concurrent operations as they have no effect), $add(a)$, $remove(a)$, $add(a)$, $remove(a)$, which sums up to four operations and therefore four equivalence classes.

By using the information just learned, we can create a new CRDT using the equivalence classes count, but in a much simpler form. The number of equivalence classes reflects the number of times a specific element is added and removed, and this number is precisely what the CLSet CRDT will track, we can call this count the causal length (CL).

$$
\begin{aligned}
\mathsf{CLSet}(E) &\stackrel{\text{def}}{=} E \hookrightarrow \mathbb{N} \\
\mathsf{add}(s,e) &\stackrel{\text{def}}{=} \begin{cases} s\{e \mapsto s(e)+1\} & \text{if } \mathsf{even}\big(s(e)\big) \\ s & \text{if } \mathsf{odd}\big(s(e)\big) \end{cases} \\
\mathsf{add}^{\delta}(s,e) &\stackrel{\text{def}}{=} \begin{cases} \{e \mapsto s(e)+1\} & \text{if } \mathsf{even}\big(s(e)\big) \\ \{\} & \text{if } \mathsf{odd}\big(s(e)\big) \end{cases} \\
\mathsf{remove}(s,e) &\stackrel{\text{def}}{=} \begin{cases} s & \text{if } \mathsf{even}\big(s(e)\big) \\ s\{e \mapsto s(e)+1\} & \text{if } \mathsf{odd}\big(s(e)\big) \end{cases} \\
\mathsf{remove}^{\delta}(s,e) &\stackrel{\text{def}}{=} \begin{cases} \{\} & \text{if } \mathsf{even}\big(s(e)\big) \\ \{e \mapsto s(e)+1\} & \text{if } \mathsf{odd}\big(s(e)\big) \end{cases} \\
(s \sqcup s')(e) &\stackrel{\text{def}}{=} \mathsf{max}\big(s(e), s'(e)\big) \\
\mathsf{in}(e,s) &\stackrel{\text{def}}{=} \mathsf{odd}\big(s(e)\big) \\
\mathsf{all}(s) &\stackrel{\text{def}}{=} \{e \mid \mathsf{odd}\big(s(e)\big)\}
\end{aligned}
$$

Figure 4.3: CLSet CRDT

CLSet is designed 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. Remove is an inverse of the last add 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 fulfil the same purpose and therefore are regarded as the same update.

Seeing one means seeing both. Two concurrent reverses of the same update are also regarded as the same one [17].

Figure 4.3 shows the CLSet CRDT. An element e is in the set when its causal length is an odd number. A local addition has an effect only when the element is not in the set. Similarly, a local removal has an 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'. The CLSet CRDT does not rely on dots for synchronization over multiple sites; it only uses causal length, which in turn is very similar to Figure 2.3's causal context [17].

4.2 Multi-Value Map

With many concurrent users using a CRDT system (multiple CRDT sites in use), we might often stumble into the situation where multiple sites write the same key at the same time. Typically, when multiple sites join their states, we would experience that only the value with the latest timestamp would remain (in the case of Last-Write-Wins (LWW)). A multi-value CRDT would instead join concurrently added values together to a list of values. From this list, the application itself can decide on what to do with the result. The aim with Multi-Value Map (MVMap) is to make it easier to work with multi-value CRDTs and to automate tasks such as processing information read from the CRDT.

Figure 4.4 show the definition of delta-state MVMap which is inspired by MVReg from section *3.5 - Multi-Value Register*. The functionality of MVMap is very much the same as MVReg. However, MVMap enables the user to prematurely define what happens to the result coming out of the read query. The way it works is that the user specifies which type of resolve function the CRDT should use. This function should accept a list of values, perform some operation, then return a result. This can be seen in the read function in Figure 4.4.

The add mutator creates new dots and maps keys to pairs of dot and value. The remove mutator removes values based on keys. The join of two states includes all keys and concurrently added values. If two different sites concurrently add something to the same key, the dots will be the same, resulting in the next join to include both values. The read function performs the resolve function from the application on the given key's value. This value sent to the resolver can both be a list of values or a single value.

$$\begin{aligned}
\mathsf{MVMap} &\overset{\text{def}}{=} (K \hookrightarrow (dots \to V)) \times \mathcal{P}(dots) \\
\mathsf{add}^{\delta}_{i}(\langle m, c\rangle, k, v) &\overset{\text{def}}{=} \langle \{k \mapsto \langle d, v\rangle\}, d\rangle \text{ where } d = \{\mathsf{next}_i(c)\} \\
\mathsf{remove}^{\delta}_{i}(\langle m, c\rangle, k) &\overset{\text{def}}{=} \langle \{\}, \mathsf{dom}(m(k))\rangle \\
\langle m, c\rangle \sqcup \langle m', c'\rangle &\overset{\text{def}}{=} \langle \{k \mapsto m_k \mid k \in \mathsf{dom}(m) \cup \mathsf{dom}(m') \wedge \mathsf{dom}(m_k) \neq \{\}\}, \\
&\qquad c \cup c'\rangle \\
&\quad \text{where } d = \mathsf{dom}(m(k)), d' = \mathsf{dom}(m'(k)) \text{ and} \\
&\qquad \mathsf{dom}(m_k) = (d \cap d') \cup (d - c') \cup (d' - c) \text{ and} \\
&\qquad m_k = \{\langle d_k, v\rangle \mid \quad d_k \in \mathsf{dom}(m(k)) \wedge (m(k))(d_k) = v \\
&\qquad\qquad\qquad \vee\, d_k \in \mathsf{dom}(m'(k)) \wedge (m'(k))(d_k) = v\} \\
\mathsf{read}(\langle m, c\rangle, k) &\overset{\text{def}}{=} \mathsf{resolve}(V_k) \\
&\quad \text{where } V_k = \{v \mid \exists d \in \mathsf{dom}(m(k)) : v = (m(k))(d)\}
\end{aligned}$$

Note: Because dots are globally unique, v in m_k is deterministically decided even when $d_k \in \mathsf{dom}(m(k)) \cup \mathsf{dom}(m'(k))$

Figure 4.4: MVMap CRDT

5 Implementation

The implementation for this thesis is written in the Elixir programming language [7]. The basis for the code is an implementation of AWLWWMap found on GitHub [5]. This implementation is then again based on AWSet from [15]. The code is written in a modular fashion, such that one can develop different CRDTs and attach them to the existing implementation. The code, even though it is modular, was not sufficient, so it has since been refactored to more easily pair with different CRDT implementations.

The project is structured as illustrated in Figure 5.1. The CRDTs are split into two categories, *dots CRDT* and *simple CRDT*. This is because TFAWSet, ORSet and MVMap all utilize dots for causal context, whereas CLSet does not. To avoid any confusion, we can give names to some of the parts and actions of the CRDTs. Parts such as TFAWSet, ORSet, MVMap and CLSet can be called the *CRDT layer*. The logic that takes δ's (deltas) from the CRDT layer and synchronizes them to neighbours can be called the *synchronization layer*. The synchronization layer, for the most part, does two things: synchronize and apply operations. As such, we can say that it has two phases, the *synchronization phase* and the *operation phase*. This was also mentioned in Section 2.5 - *Conflict-free Replicated Data Types*.

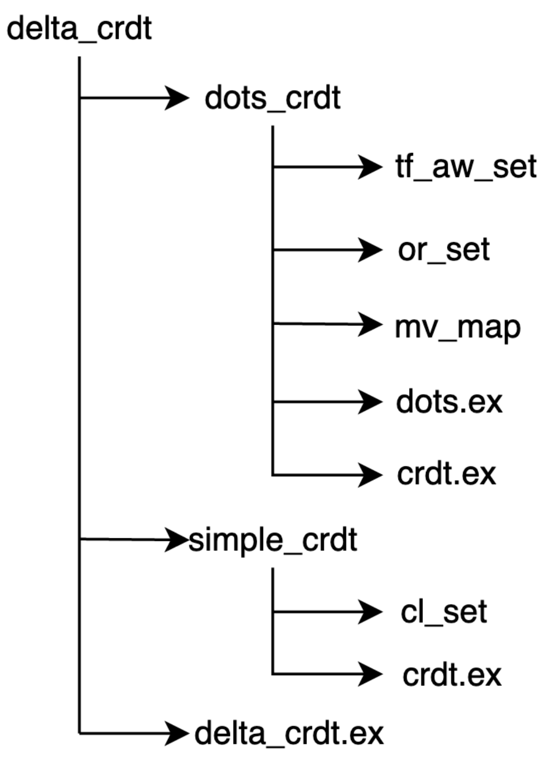


Figure 5.1: Project File-Structure

5.1 GenServer

The synchronization layers are implemented in an Elixir behaviour called *GenServer*. The GenServer is in simple terms a skeleton server, with a set of functions you can implement. Some of these functions include:

- `init` (automatically called whenever the GenServer process is created)
- `handle_info` (receives messages from other processes)
- `handle_call` (receives synchronous requests)
- `handle_cast` (receives asynchronous requests)

`Init` is a function implemented in GenServer. If implemented, it will be called during the creation of the GenServer process.

`handle_info` is a function implemented in GenServer. It specifically receives messages which is sent in the form of `send`, `Process.send` or `Pro-`

`cess.send_after`. These send functions all sends messages from one process to another, and can then be received through a GenServer `handle_info`.

`handle_call` and `handle_cast` is similar to each other as they both respond to requests. The difference is that 'cast' is asynchronous, whereas 'call' is synchronous. They are called by calling `GenServer.call` or `GenServer.cast` accordingly. The GenServer-call and -cast functions take as input, the GenServer process which should receive the request, along with a message.

The return value of `handle_info`, `handle_call` and `handle_cast` should be a tuple in one of the following formats `{:reply, return_value, new_state}` or `{:noreply, new_state}`. So this is where the GenServer state is updated; by returning an updated version in the return tuple.

5.2 CRDT Layer API

The CRDT layer is connected to the CRDT synchronization layer. Whenever the synchronization layer needs to update the CRDTs state, it will perform actions through the CRDT layer API. The API consist of the following functions:

- `new`
- `add`
- `remove`
- `join`
- `read`

The `new` function is called inside the synchronization layer whenever the user creates a new CRDT instance. It simply makes an instance of the CRDT layer's module struct. `add` is used in the synchronization layer whenever the user does an add-mutation. `remove` is used when the user does a remove-mutation. `join`, joins two CRDT δs together. Usually one of the δs is the existing state, and the other is an incoming change. `read` returns the content of the CRDT. It does not return any form of context, or tombstone related data (removed values), which means that it is strictly values that are in the CRDT and have not been removed.

5.3 Delta CRDT API

The file `delta_crdt.ex` is the API which the user interfaces with to use the CRDTs. For increased modularity it is also placed outside of both dots and simple directories so there is one interface for both CRDT categories. It has the following API:

- `start_link` (starts a new CRDT instance)
- `set_neighbours` (set the neighbouring CRDTs for a CRDT)
- `mutate` (mutate a CRDT)
- `read` (read a CRDT)

In Elixir `start_link` is a normal name for a function which starts a process. The delta CRDT `start_link` calls a GenServer `start_link`, thus it has been given the same name since it indirectly starts a GenServer. The functions `set_neighbours`, `mutate` and `read` all sends calls and messages to the GenServer.

```
def start_link(crdt_module, backend, opts \\ []) do
  init_args =
    Keyword.put(opts, :crdt_module, crdt_module)
    |> Keyword.put_new(:sync_interval, @default_sync_int)

  case backend do
    :simple ->
      GenServer.start_link(SimpleCrdt.Crdt, init_args, opts)

    :dots ->
      GenServer.start_link(DotsCrdt.Crdt, init_args, opts)
  end
end

def set_neighbours(crdt, neighbours) when is_list(neighbours)
    do
  send(crdt, {:set_neighbours, neighbours})
  :ok
end

def mutate(crdt, f, a, timeout \\ 5000)
    when is_atom(f) and is_list(a) do
  GenServer.call(crdt, {:operation, {f, a}}, timeout)
end

def read(crdt, timeout \\ 5000) do
  GenServer.call(crdt, :read, timeout)
end
```

Listing 5.1: Delta CRDT API

Listing 5.1 show the implementation of the delta CRDT API. `start_link` takes one of the CRDT layers as input together with what type of backend it uses. The backend can either be `:simple` or `:dots`. It will set up a set of initial arguments followed by starting the appropriate GenServer process for the given backend. The GenServer will remember the CRDT module for any future mutations. The return value is a Process Identifier (PID) for the CRDT process.

`set_neighbours` takes two inputs. The first one is a CRDT PID, and the second one is a list of CRDT PIDs. It sends a message to the first PID asking it to update its state to the given list of PIDs.

`mutate` takes a CRDT PID, a function name in the form of an atom, and a list of two values. It does a `GenServer.call` to the PID, asking it to do an operation. Which operation it does is chosen by the value of `f`. Inside the GenServer, it will use `apply`, which requires both the CRDT module stored in state and the function name stored in `f`, to decide what function to call. The list `a` should consist of two values if the CRDT layer is a map CRDT, the first is going to be the key and the second is going to be the value stored under that key. However, if the CRDT layer is a set CRDT, only one value is needed.

`read` does a `GenServer.call` to a CRDT PID. It returns the result of the CRDT layer's read function.

5.4 Dots CRDT

The dots implementations consist of three different CRDTs; TFAWSet, ORSet and MVMap. In addition to these three implementations there is a file called `dots.ex`. This file is for managing dots, and is implemented separately such that multiple CRDT implementations can make use of it. The dots implementation contains an API for interacting with dots:

- `compress` (anti-entropy algorithm returning a version vector containing the highest sequence number for each replica)
- `decompress` (returns a list of all versions of all contexts)
- `next_dot` (computing the next dot based on given context)
- `union` (computes the union of two dot maps)
- `difference` (computes the difference of two dot maps)
- `member?` (check if a dot is a member of a dot map)

The last file in the dots_crdt directory is `crdt.ex`. This file implements the module which is responsible for synchronizing changes between the different CRDT sites, also called the synchronization layer. It stores the state of the CRDT, keeps a list of neighbours, performs operations, stores changes and synchronizes these changes to its neighbours in intervals. This file is the 'brain' of the CRDT and is implemented outside of the CRDT layer for modularity.

```
def handle_info(:sync, state) do
    state = sync_interval_or_state_to_all(state)

    Process.send_after(self(), :sync, state.sync_interval)

    {:noreply, state}
 end
```

Listing 5.2: Dots CRDT Synchronization Phase

The synchronization layer is implemented as an Elixir GenServer [6]. A GenServer behaves like a server; it can receive messages and perform operations based on which type of message it receives. For example, during initialization, the synchronization layer will send a message to itself containing the `:sync` event.

The synchronization layer will receive the message in a `handle_info` function, as can be seen in Listing 5.2. The function will synchronize state to its neighbours, followed by calling `Process.send_after` which then again sends a `:sync` message to itself after a delay, telling the synchronization layer to synchronize state again. The GenServer will repeat the `:sync` message repeatedly in an interval, as such it can be seen as a synchronization loop. The benefit of sending messages to itself is that the implementation is consistent in only using GenServer built-in functionality. Also, we forgo any race conditions as the GenServer keeps a message queue internally and processes the messages synchronously. `sync_interval_or_state_to_all` is the function which sends messages to neighbouring sites. Inside this function, it will call `send`, which sends messages. Send is similar to `Process.send_after`, but instead of being after a delay, it is instant.

As mentioned previously, the synchronization layer is split into two phases, the *operation phase* and the *synchronization phase*. In practice, the synchronization layer does not switch phases. However, it can instead be seen as being in the operation phase when processing an `:operation` message, likewise it can be seen as being in the synchronization phase while processing a `:sync` message.

```
def handle_info({:diff, diff, keys}, state) do
  new_crdt_state = state.crdt_module.join(state.crdt_state,
     diff, keys)

  new_state = Map.put(state, :crdt_state, new_crdt_state)

  {:noreply, new_state}
end
```

Listing 5.3: Dots CRDT Receive Delta

The synchronization phase sends out δs, which are then received in handle_info along with the :diff event. The dots CRDT implementation for :diff can be seen in Listing 5.3. There is not much to be done when receiving these messages, simply merge diff with the current state, through the CRDT module stored in state. Afterwards, insert new_crdt_state into the GenServer state and then return the updated state.

```
def handle_call({:operation, operation}, _from, state) do
    {:reply, :ok, handle_operation(operation, state)}
end

defp handle_operation({function, [key | rest_args]}, state)
    do
    delta =
      apply(state.crdt_module, function, [key | rest_args] ++
          [state.node_id, state.crdt_state])

    update_state_with_delta(state, delta, [key])
end
```

Listing 5.4: Dots CRDT Operation Phase

Listing 5.4 show how the CRDT handles :operation messages. It calls the handle_operation function, which in turn calls apply. apply is an Elixir function for programmatically calling functions where the name of the function can be stored in a variable. Notice the function uses the value of state.crdt_module to decide which module to search for the function. The CRDT module is set during initialization of the synchronization layer. The return value from each of the operations should be a δ. The δ can be used to update the CRDT state through the CRDT layer's join-function. The final step is done inside update_state_with_delta. In this function, the synchronization layer will merge the δ with the current state. This is also where the synchronization layer stores each new δ as an outstanding synchronization. The outstanding synchronizations will be sent to neighbours during the next synchronization.

The state of the CRDT is stored in the GenServer's state. During the operation phase a new δ is computed which is then merged with the old state to get the new state. The CRDT state is now updated to the new state. This means that the different CRDT layers (TFAWSet, ORSet, ...) is not stateful. Instead, the CRDT layer works like an interface taking the CRDT state as one of its parameters and returning a δ without doing any permanent changes. The changes happen after merging the state with a δ and storing the new state in the GenServer state.

5.4.1 TFAWSet

One of the CRDTs discussed in this thesis is TFAWSet. TFAWSet is a modification of AWLWWMap from [5]. This CRDT layer is made as a simple Elixir module, containing a data structure and a set of functions.

```elixir
defmodule DeltaCrdt.DotsCrdt.TFAWSet do
    defstruct dots: MapSet.new(),
              value: %{}

    ...
```

Listing 5.5: TFAWSet Module and Struct

Listing 5.5 show the TFAWSet module along with the definition of its struct. The struct contains two fields, `dots` which is a MapSet, and `value` which is a Map. The rest of the module implements the CRDT layer API mentioned in 5.2.

```elixir
def add(key, i, state) do
  case Map.fetch(state.value, key) do
    :error ->
      d = Dots.next_dot(i, state.dots)

      %TFAWSet{
        dots: MapSet.new([d]),
        value: %{key => MapSet.new([d])}
      }

    _ ->
      TFAWSet.new()
  end
end
```

Listing 5.6: TFAWSet Add

Listing 5.6 show the implementation of `add`. It takes a key, an identifier and a state as input. It does a `Map.fetch` in the state for the given key. If unable to find the key, it will return `:error`, which means that we wish to add the element. It generates a new dot. The new dot is based on the identifier and the current set of dots. Afterwards, it will create a new TFAWSet struct instance with the new dot and element; this is the δ. In the case of `Map.fetch` finding an existing element, the function will return an empty TFAWSet, which is the same as no changes (no δ).

```
def remove(key, state) do
  to_remove_dots =
    case Map.fetch(state.value, key) do
      {:ok, dots} -> dots
      :error -> []
    end

  %TFAWSet{
    dots: MapSet.new(to_remove_dots),
    value: %{}
  }
end
```

Listing 5.7: TFAWSet Remove

Listing 5.7 show the TFAWSet `remove`, which just `Map.fetch` for the key, if it were successfully found, it figures out which dots should be removed, followed by creating a new δ. If it returns `:error` it results in an empty δ.

```
def join(delta1, delta2) do
  new_dots = Dots.union(delta1.dots, delta2.dots)

  join_maps(delta1, delta2)
  |> Map.put(:dots, new_dots)
end
```

Listing 5.8: TFAWSet Join

The TFAWSet's `join` function can be seen in Listing 5.8. It takes two δ as input. The function generates a new set of dots which is the union of all dots. To join the values of the two δs, it calls `join_maps`. In there it will resolve any conflict by enumerating through all values, extracting that value from both δs and joining their respective dots with a union operation.

```
def read(%{value: values}) do
  Map.keys(values)
end

def read(%{value: values}, subset) when is_list(subset) do
  read(%{value: Map.take(values, subset)})
end

def read(crdt, key) do
  read(crdt, [key])
end
```

Listing 5.9: TFAWSet Read

With this implementation the `read` function is quite simple. Listing 5.9 shows the implementation. It consists of three functions, where it is possible to read a single value, a subset of values or the whole set. The appropriate function will be called depending on the input. Both the function which reads a single value and the function which reads a subset will call the last function, but with modified input, hence the last function can be seen as the 'main' read function.

5.4.2 ORSet

ORSet is a second CRDT discussed in this thesis. ORSet is implemented from a definition in [15]. This CRDT layer is made as an Elixir module, containing a data structure and a set of functions.

```
defmodule DeltaCrdt.DotsCrdt.ORSet do
    defstruct dots: MapSet.new(),
              value: %{}

    ...
```

Listing 5.10: ORSet Module and Struct

Listing 5.10 show the ORSet module and its struct. It has two fields, `dots` which is a MapSet, and `value` which is a Map.

```
def add(key, i, state) do
  delta_dots =
    case Map.fetch(state.value, key) do
      :error -> MapSet.new([Dots.next_dot(i, state.dots)])
      {:ok, {d, d}} -> MapSet.new([Dots.next_dot(i, state.
          dots)])
    end

  %ORSet{
    dots: delta_dots,
    value: Map.put(Map.new(), key, {delta_dots, MapSet.new()}
       )
  }
end
```

Listing 5.11: ORSet Add

Listing 5.11 show the implementation of `add`. It takes a key, an identifier and a state as input. With these parameters, it performs a `Map.fetch` after the given value in the state. If given an `:error`, it knows that the value does not already exist; thus, it generates the next dot and returns a δ containing the new dot and the new value.

If the `Map.fetch` returns `{:ok, {d, d}}`, it means that it found the value and that it has been previously removed. This is because the ORSet stores values in the following format `%{value => {add-dots, remove-dots}}`. Furthermore, when Elixir tries to match something in the following format `{d, d}` it looks for a tuple of two values where both are the same. As such, when that condition matches, it means that both the add-dots and the remove-dots are equal. This, in turn, means that we know that the value has previously been removed. In this case, it will re-add the element by generating the next dot and return a δ containing the new dot and value.

```
def remove(key, state) do
  {a, r} = Map.get(state.value, key, {MapSet.new(), MapSet.
      new()})

  %ORSet{
    dots: MapSet.new(),
    value: Map.put(Map.new(), key, {MapSet.new(), MapSet.
       difference(a, r)})
  }
end
```

Listing 5.12: ORSet Remove

Listing 5.12 show the implementation of `remove`. It takes a key and state as input. It performs a `Map.get` on the state, with the key in question. The last argument to `Map.get` is a default value which is returned if the key is not found. Notice that the default value is the same format as the entries stored in state, thus if the key is not found `Map.get` will essentially return an empty entry. Afterwards, it generates a δ with the key and appropriate dots dependent on if the key existed or not.

```
def join(%{dots: d1, value: v1}, %{dots: d2, value: v2}) do
  dots = Dots.union(d1, d2)

  v =
    Map.merge(v1, v2, fn _key, {a1, r1}, {a2, r2} ->
      {MapSet.union(a1, a2), MapSet.union(r1, r2)}
    end)

  %ORSet{
    dots: dots,
    value: v
  }
end
```

Listing 5.13: ORSet Join

ORSet's `join` function can be seen in Listing 5.13. It takes two δs as input. It will join both their dot-maps through a union. Afterwards, it does a `Map.merge`, which will automatically merge entries which are not present in both maps. Entries which are present in both maps will call the given merge function. The merge function will receive two tuples, the first one is the add-dots and the remove-dots for the entry in the first δ, whereas the second tuple is the add-dots and the remove-dots for the entry in the second δ. With this information, it will create a new entry-tuple. The new tuple consists of the union of the add-dots and the union of the remove-dots. Finally, the join function returns the new state for the ORSet.

```
def read(%{value: v}) do
  Enum.reduce(v, MapSet.new(), fn {k, {a, r}}, acc ->
    if MapSet.subset?(a, r) do
      acc
    else
      MapSet.put(acc, k)
    end
  end)
end

def read(crdt, keys) when is_list(keys) do
  new_values = Map.take(crdt.value, keys)
  read(Map.put(crdt, :value, new_values))
end

def read(crdt, key) do
  read(crdt, [key])
end
```

Listing 5.14: ORSet Read

The `read` function for the ORSet can be seen in Listing 5.14. It consists of three different functions; a function to read a single value, a subset of values or the whole set. Elixir will call the appropriate function depending on the input it receives. Both the function which reads a single value and the function which reads a subset will call the last function, but with modified input, hence the last function can be seen as the 'main' read function. The function will perform a reduction on the state, where it enumerates through the value map, and accumulates entries only where the add-dots are not a subset of the remove-dots.

5.4.3 MVMap

MVMap is implemented in a way that it is just the basis for other CRDTs. What this means in practice is that MVMap is almost fully implemented, and it requires the user to create a separate module which then implements a `resolve` function for the MVMap.

```elixir
defmodule DeltaCrdt.DotsCrdt.GenericMVMap do
  defmacro __using__(_opts) do
    quote do
      import unquote(GenericMVMap)

      defstruct dots: MapSet.new(),
                value: %{},
                f: nil
      ...

    end
  end
end

defmodule DeltaCrdt.DotsCrdt.MaxMap do
  use DeltaCrdt.DotsCrdt.GenericMvMap

  defp resolve(%{vs: vs}) do
    Enum.max(vs, fn -> nil end)
  end
end
```

Listing 5.15: MVMap Module, Struct and MaxMap Resolve

Listing 5.15 shows how the MVMap module is created, and how a module called MaxMap is using the MVMap to implement a `resolve` function which picks out the maximum entry. The MVMap struct has three fields; `dots` which is a MapSet, `value` which is a Map, and `f` which is a function. `f` will be further explained later.

```elixir
def add(key, value, i, %{dots: dots, value: kdv} = state) do
  dot = Dots.next_dot(i, dots)
  dv = Map.get(kdv, key, %{})

 %{state | value: %{key => %{dot => value}}, dots: MapSet.
     new([dot | Map.keys(dv)])}
end
```

Listing 5.16: MVMap Add

Furthermore, MVMap module still has the standard CRDT layer API. Listing 5.16 show the implementation of `add`. Notice that MVMap is a map CRDT, which means that it takes a value as input, in addition to the key, identifier and state. `add` generates the next dot and does a `Map.get` to see if the given key has any existing dots. Either way, the key-value pair is added, in the format `%{key => %{dot => value}}`. This is also the reason for the variable name `kdv` which means key-dot-value. If there were an existing dot for the given key, it is stored in the `dots` MapSet. This way, the CRDT can know that this dot has previously been seen. The returned value is a delta, containing a new value and any already existing dots for the entry.

```elixir
def remove(key, %{value: kdv} = state) do
  dots = Map.get(kdv, key, %{}) |> Map.keys() |> MapSet.new()
  %{state | value: %{}, dots: dots}
end
```

Listing 5.17: MVMap Remove

The `remove` function can be seen in Listing 5.17. It takes just a key and state as input. It performs a `Map.get` for the given key, to get the corresponding dots. It returns a δ containing the dots it found.

```elixir
def join(s1, s2) do
  %{dots: c1, value: kdv1} = s1
  %{dots: c2, value: kdv2} = s2

  {common1, only1} = Map.split(kdv1, Map.keys(kdv2))
  {common2, only2} = Map.split(kdv2, Map.keys(common1))

  new_common =
    Map.merge(common1, common2, fn _key, dv1, dv2 ->
      join_dv(dv1, c1, dv2, c2)
    end)

  new_only1 = join_kdv(only1, c2)
  new_only2 = join_kdv(only2, c1)

  new_kdv = new_common |> Map.merge(new_only1) |> Map.merge(
      new_only2)

  %{s1 | dots: Dots.union(c1, c2), value: new_kdv}
end
```

Listing 5.18: MVMap Join

Listing 5.18 show the implementation of `join`. It starts by finding all entries which are common in both δs and all entries which are unique for both. Afterwards, the common entries' maps are merged. This merge function merges the dot-values for each key using a `join_dv` function. This function does intersection- and difference-operations on dots to make sure every relevant dot is retrieved and saved. Also, since this CRDT is a multi-value CRDT, `join_dv` will include multiple values if the same keys are present in both δs. The result is that dots from both δs are merged into one big set of common keys and all their corresponding dots. The unique entries for both δs are also merged together with a `join_kdv` function. This function enumerates through each entry. For each key, it accumulates all dots which are not known to the other context. It does this with the help of `Dots.difference`. All these different maps can now easily be merged with `Map.merge` since there should be no conflicts, hence no merge function has been given to `Map.merge`. Finally, the join returns the new state.

```
def read(crdt) do
  Enum.reduce(crdt.value, %{}, fn {k, dvs}, acc ->
    Map.put(
      acc,
      k,
      resolve(%{vs: Map.values(dvs), f: crdt.f})
    )
  end)
  |> Enum.filter(fn {_k, v} -> v != [] end)
  |> Map.new()
end

def read(crdt, keys) when is_list(keys) do
  new_values = Map.take(crdt.value, keys)
  read(Map.put(crdt, :value, new_values))
end

def read(crdt, key) do
  read(crdt, [key])
end
```

Listing 5.19: MVMap Read (resolve)

The `read` function of MVMap is where `resolve` takes effect. Listing 5.19 shows three different read functions. They will get called depending on which input is given, but as can be seen, they end up calling the 'main' read function either way. The 'main' read function performs a reduction, where it accumulates the result of `resolve`. We saw an example implementation of a resolve function in Listing 5.15.

The resolve function receives each key's value and resolves it based on which type of CRDT the user chose to create. The value given to `resolve` can either be a single value, or a list of values. In cases when using CRDTs such as MapMap or a ReduceMap, the resolve function enumerates and performs operations on each value. For this reason, MVMap keeps a function `f` stored in state. `f` is passed on to the resolve function, which calls the function on each element. Results from `resolve` which yields empty lists are removed.

```
defmodule DeltaCrdt.DotsCrdt.AvgMap do
  use DeltaCrdt.DotsCrdt.GenericMvMap

  defp resolve(%{vs: vs}) when is_list(vs) do
    Enum.sum(vs) / length(vs)
  end
end

defmodule DeltaCrdt.DotsCrdt.SumMap do
  use DeltaCrdt.DotsCrdt.GenericMvMap

  defp resolve(%{vs: vs}) do
    Enum.sum(vs)
  end
end

defmodule DeltaCrdt.DotsCrdt.ReduceMap do
  use DeltaCrdt.DotsCrdt.GenericMvMap

  defp resolve(%{vs: vs, f: f}) when is_function(f) do
    Enum.reduce(vs, [], f)
  end
end

defmodule DeltaCrdt.DotsCrdt.MvMap do
  use DeltaCrdt.DotsCrdt.GenericMvMap

  defp resolve(%{vs: vs}) do
    vs
  end
end
```

Listing 5.20: MVMap - Resolve Examples

Listing 5.20 show some more examples of MVMap resolve modules. It is essentially a straightforward design, which only requires the user to specify which type of CRDT they want, and possibly provide a function `f` dependent on which type of CRDT they made. Afterwards, just reading the CRDT will execute `resolve` and yield the result.

5.5 Simple CRDT

The CLSet CRDT has a different synchronization layer (`crdt.ex`) than the dots CRDTs, since it does not utilize dots. For this reason, it can be simplified a fair bit, hence the name, 'simple'. The file, `crdt.ex`, looks similar to the dots counterpart in regards to also being a GenServer, with several functions doing the same type of work. The GenServer has a state containing a list of neighbours and a queue of outstanding synchronizations. It will perform operations, store changes as outstanding synchronizations and then synchronize the changes to its neighbours in intervals.

```
def handle_info(:sync, state) do
  new_state =
    case length(state.queue) > 0 do
      true ->
        Enum.each(state.queue, fn delta = %Delta{to: t} ->
          Enum.each(t, &send(&1, {:delta, delta}))
        end)

        Map.put(state, :queue, [])

      false ->
        state
    end

  Process.send_after(self(), :sync, state.sync_interval)
  {:noreply, new_state}
end
```

Listing 5.21: Simple CRDT Synchronization Phase

Listing 5.21 show the synchronization phase for the simple CRDT. Similar to the dots CRDT, it is first invoked in the GenServer initialization by sending a message to itself containing the `:sync` event. The GenServer will receive the `:sync` message in `handle_info`. It behaves like a loop since it will keep sending a new message to itself continuously with `Process.send_after`. The benefit of sending messages to itself is that the implementation is consistent in only using GenServer built-in functionality. Also, we forgo any race conditions as the GenServer keeps a message queue internally, and messages are processed synchronously. The synchronization phase is where we enumerate through the queue, which is kept in state. The queue consists of δ's. The deltas are created during the operation phases, where one of the CRDTs mutators are called. Each item in the queue contains a δ and a list of recipients. It then enumerates through each recipient and sends a message to them. Afterwards, the queue is cleared, and the process sends a new `:sync` message to itself.

```
def handle_call({:operation, {func, data}}, _from, state) do
  {:reply, :ok, do_operation(func, data, state)}
end

defp do_operation(func, data, state) do
  {delta, crdt_state} = apply(state.crdt_module, func, data
      ++ [state.crdt_state])

  state
  |> append_queue(state.neighbours, delta)
  |> Map.put(:crdt_state, crdt_state)
end
```

Listing 5.22: Simple CRDT Operation Phase

Whenever the GenServer receives calls, it looks for a `handle_call` with matching parameters. Listing 5.22 show the implementation of the `:operation` event, also called the operation phase. As mentioned in Section 5.3, the user does a mutation through the delta CRDT API, where one of the input parameters is the function name in the form of an atom. This function name is then received here and stored in `func`. During initialization, the CRDT has stored which CRDT module it should use in `state.crdt_module`. With this information, it can call the Elixir built-in function `apply` which takes a module, a function, a set of arguments as input. The result of calling the function will yield a δ and the new CRDT state. The δ is appended to the queue, and the new CRDT state is put into the GenServer state.

```
def handle_info({:delta, %Delta{from: f, data: d}}, state) do
  crdt_state =
    Enum.reduce(d, state.crdt_state, fn {k, v}, acc ->
      state.crdt_module.join(%{k => v}, acc)
    end)

  new_state = Map.put(state, :crdt_state, crdt_state)
  {:noreply, new_state}
end
```

Listing 5.23: Simple CRDT Receive Delta

After a synchronization phase and the δs have been sent out, they will be received in `handle_info` with the `:delta` event. In addition to the event tag, there is a δ. The GenServer will enumerate in-case it contains multiple changes, for each change, it will call `join` on the CRDT layer's module and accumulate the state after each join. Afterwards, the GenServer state is updated with the new CRDT state.

5.5.1 CLSet

CLSet is a new CRDT first mentioned in [17] and it was also presented on the *PaPoC Workshop 2020* [13]. It is not a dots CRDT, thus using its own simple CRDT synchronization layer. It is created in the same way as the other CRDT discussed earlier.

```elixir
defmodule DeltaCrdt.SimpleCrdt.CLSet do

  def new(), do: %{}

  ...
```

Listing 5.24: CLSet Module

Listing 5.24 show the implementation of the CLSet module and how new instances are created. Since CLSet does not utilize dots, the only thing CLSet needs to keep track of is its entries, which it does by using a normal map.

```elixir
def add(key, state) do
  case Map.get(state, key, 0) do
    0 ->
      {%{key => 1}, Map.put(state, key, 1)}

    l ->
      case rem(l, 2) do
        0 ->
          {%{key => l + 1}, Map.put(state, key, l + 1)}

        1 ->
          {%{}, state}
      end
  end
end
```

Listing 5.25: CLSet Add

The CLSet `add` function can be seen in Listing 5.25. It takes a key and the CRDT state as input. Firstly, it tries to get the key from the state with `Map.get`, specifying that it wants 0 as the default value. If the `Map.get` returns 0 it knows that the key does not exist in the CRDT, and proceeds by returning a tuple containing the δ and the new state. The data stored in the CRDT is in the format `%{key => length}`, where length is the *causal length* of the key; thus the first time the key is added it will have a causal length of 1.

If `Map.get` returns something that is not 0, the CRDT layer knows that the key exists. At this point, it checks whether the causal length is odd or even. If the causal length is even, it means the key has been removed. If it is odd instead, it means that the key has not been removed. The reason for this is since the causal length starts at 0 and the key is not in the CRDT. Then after the first add, the causal length is incremented to 1 (odd). Whenever it is removed, the causal length gets incremented to 2 (even), next time it is added the causal length gets incremented to 3 (odd), et cetera.

```elixir
def remove(key, state) do
  case Map.get(state, key, 0) do
    0 ->
      {%{}, state}

    l ->
      case rem(l, 2) do
        0 ->
          {%{}, state}

        1 ->
          {%{key => l + 1}, Map.put(state, key, l + 1)}
      end
  end
end
```

Listing 5.26: CLSet Remove

CLSet `remove`, shown in Listing 5.26 is very much like `add`. It does a `Map.get` with a default value of 0, to get the causal length. If it results in 0, the key does not exist. The function returns an empty δ along with the unchanged state. If it were to result in something other than 0, the key does exist, so it checks whether the causal length is odd or even. If it turns out to be even, it means that the key is removed, so the function returns an empty δ along with the unchanged state. If it instead turned out to be odd, it means that the key is not removed, so the causal length is incremented, and the δ and the new state is returned.

```elixir
def join(s1, s2) do
  Map.merge(s1, s2, fn _key, l1, l2 ->
    max(l1, l2)
  end)
end
```

Listing 5.27: CLSet Join

Listing 5.27 show the CLSet `join` function. The function takes two δs as input and performs a `Map.merge`. Unconflicting keys get merged automatically, and conflicting keys get merged by simply doing a `max` where the highest value is returned. This means that the highest causal length gets to join the CRDT, whenever there are conflicting keys.

```
def read(crdt) do
  Enum.reduce(crdt, MapSet.new(), fn {key, l}, acc ->
    case rem(l, 2) do
      0 -> acc
      1 -> MapSet.put(acc, key)
    end
  end)
end

def read(crdt, keys) when is_list(keys) do
  read(Map.take(crdt, keys))
end

def read(crdt, key) do
  read(crdt, [key])
end
```

Listing 5.28: CLSet Read

Listing 5.28 show the implementation of CLSet `read`. This read function, like the other CRDT implementations, consists of three different functions, each accepting different input. One function reads only one key, the second read function reads a subset of keys, and the final one reads all keys. The arguments decide which read function is called. Two of the read functions alter the input in such a way that it can use the 'main' read function to do the actual reading. The 'main' read function performs a reduction on the CRDT state. The reduction accumulates and returns all keys whose causal length is odd, i.e. not removed.

/6 Experiments & Results

There are four different CRDTs implemented in this thesis, TFAWSet, ORSet, MVMap and CLSet. In this chapter, we are going to look at their performance through some experiments. The experiments are written in Elixir as *Benchee* benchmarks. From Benchee's GitHub:

> *Library for easy and nice (micro) benchmarking in Elixir. Benchee allows you to compare the performance of different pieces of code at a glance. It is also versatile and extensible, relying only on functions. There are also a bunch of plugins to draw pretty graphs and more! Benchee runs each of your functions for a given amount of time after an initial warmup, it then measures their run time and optionally memory consumption. It then shows different statistical values like average, standard deviation etc* [2].

Benchee can wrap around functions making benchmarking specific scenarios relatively simple. The benchmarks are written in Elixir 1.9 (OTP 22), and run on a Macbook Pro 2019 with MacOS Catalina 10.15.5. The machine has the following hardware: Intel Core i5 1.4Ghz Quad-Core CPU, 8GB 2133Mhz DDR3 RAM. All CRDTs are run in a single Elixir process, so the number of CPU cores does not matter between the different CRDTs.

MVMap and all its different variations (MinMap, AvgMap, ReduceMap, et cetera) is not part of the performance testing. The MVMap CRDT is about convenience since it can automate operations such as *min, max, avg, reduce, map* through the read query. So this is why MVMap has not been compared to the other CRDTs in this section; it is a convenience CRDT and not a performance CRDT.

The first experiment is a benchmark where we study how well the CRDTs perform updates and joins. For each CRDT, we set up ten 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, only mutations which create a δ are considered.

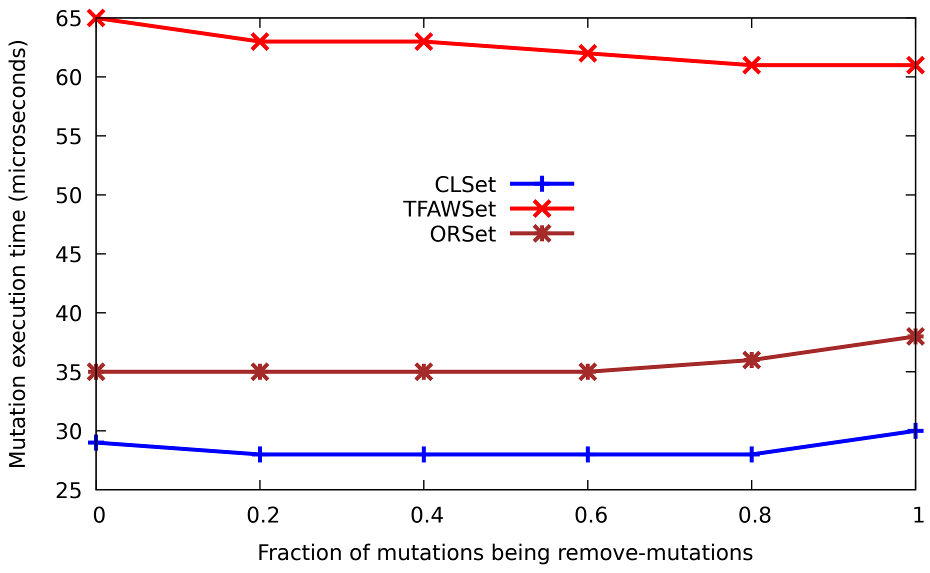


Figure 6.1: Concurrent Mutations and Merge Time

Figure 6.1 show the average time spent to finish each execution. TFAWSet was surprisingly slow, considering it has been altered to a set CRDT, and since it is tombstone-free. However, there is a downwards trend whenever the ratio of removal mutations increase. For ORSet its the other way around, it has an upwards trend whenever the ratio for removal mutations increase, which is understandable since ORSet is not tombstone-free and is storing data from removed elements. CLSet is faster than both TFAWSet and ORSet in all cases, which can be explained to CLSet not using dots for causal context, but instead only a single integer.

Take note that the number of mutations applied on a single element is typically few. The sizes of the dot sets in ORSet are therefore typically small in size, which can change ORSet performance quite much. In the case of TFAWSet, the causal context size is dependant on the number of CRDT instances (sites), which is typically higher than the number of mutations on a single element; thus the number of instances can change the TFAWSet performance quite much. CLSet does not have any apparent drawbacks. It has a one-to-one ratio between the number of elements stored and the amount of metadata stored, which is very good when dealing with a large number of elements. The graph shows the result of this specific implementation, which means that the results might be more even, with further performance improvements to the implementation.

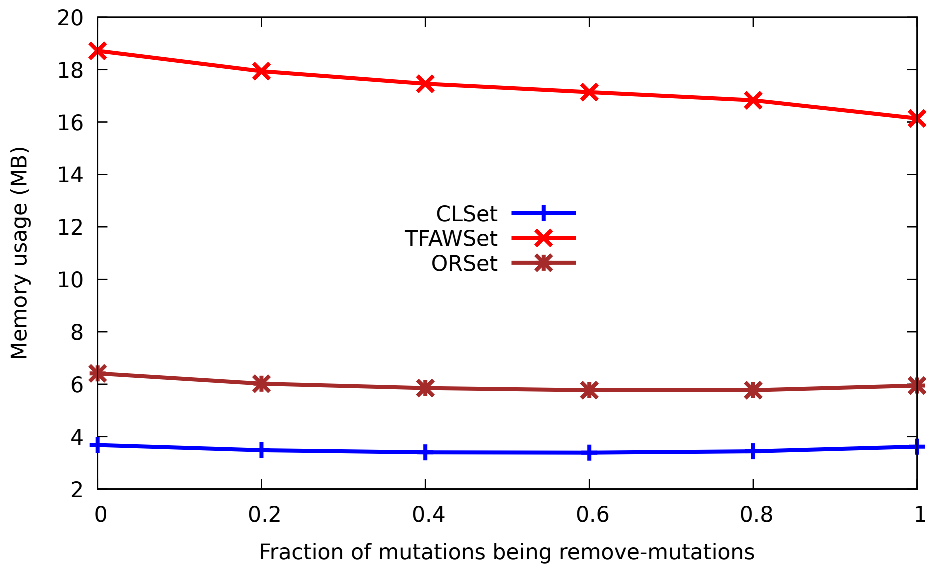


Figure 6.2: CRDT Memory Usage

The second experiment is studying memory consumption. The numbers are from memory usage during the first benchmark. The result can be seen in Figure 6.2. The memory consumption is very similar to the performance. TFAWSet ends up consuming a large amount of memory because of its join function, generating a significant amount of intermediate data. Since the benchmarks are run for a relatively short duration, the allocated memory might not have had a chance to be garbage collected yet, but even if it did, it would be at the cost of performance as garbage collection uses CPU resources.

Both ORSet and CLSet consume almost a static amount of memory over all the cases, but as ORSet is a dot CRDT, it has more context, and it will naturally use more than memory then CLSet. On the other hand, TFAWSet gradually uses less memory as more of the mutations are remove-mutations, this is since TFAWSet is tombstone-free and removed elements are entirely removed from context. There is probably a correlation between the results in the first and second experiment. A CRDT is fast/slow since it uses a low/high amount of memory.

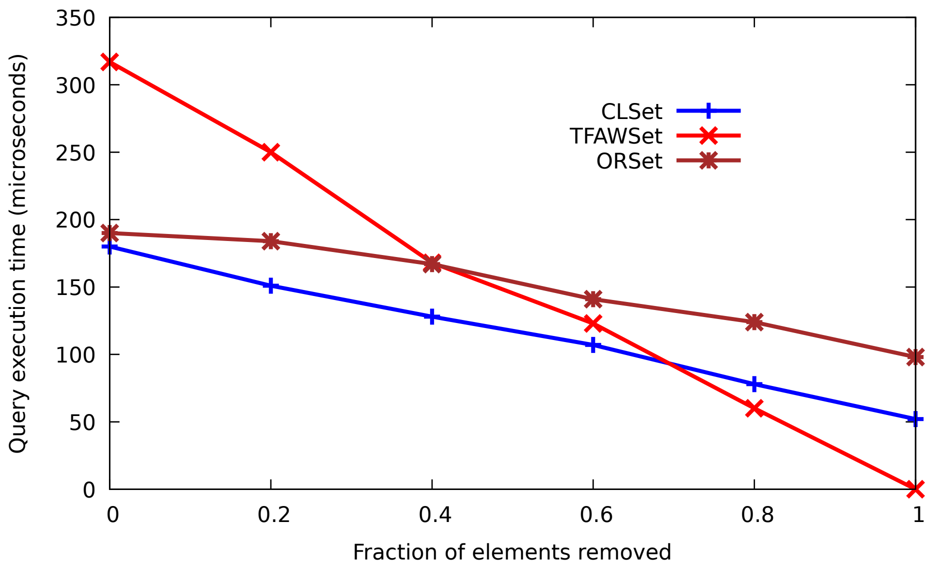


Figure 6.3: Query Execution Time

The last experiment is a benchmark to compare the execution time of the all-query (read). We set up one of each CRDT, then add 1000 elements to each, then gradually remove a bigger fraction of the elements before reading. The result can be seen in Figure 6.3. To begin with, CLSet and ORSet perform similarly, and TFAWSet is a fair bit slower. However, with 40% of the elements removed, TFAWSet reaches the same performance as ORSet, but CLSet has pulled ahead and is the fastest. At around 70% elements removed, TFAWSet is on par with CLSet and ORSet is lagging a bit behind. At 100% elements removed, TFAWSet's CRDT is empty, thus reaching a query speed of 0, whereas ORSet and CLSet both still have the context and metadata of removed elements in the state making them slower in this regard.

/7 Conclusion

The contribution of this thesis is the CLSet CRDT and its performance experiments, and also the MVMap CRDT. During the writing of this thesis, the CLSet CRDT has been published in the *PaPoC 2020 conference proceeding* [17].

To summarize, we have taken a look at some of the theory behind CRDTs and got an overview of different CRDT variations. We took a closer look at state-based delta CRDT where we looked at some existing CRDT designs like GSet, AWSet, TFAWSet, ORSet and MVReg. And then some new designs such as CLSet and MVMap, where we took ideas from existing CRDTs and then try and bring something new to the table. We looked at their respective implementations in Elixir, and then put the CRDTs to the test through experiments using Benchee benchmarks.

The results were interesting. Some observations were that CLSet performed the best in both mutation execution time and memory usage, whereas TFAWSet performed the worst in these two areas. However when it comes to reading, TFAWSet performed best when more than 70% of the elements are removed, but at the same time, it performed the worst when less than 40% of the elements are removed. The other CRDT implementations were in-between in the reading experiment. Overall the results were good for CLSet, and MVMap as mentioned, introduces a new convenience feature. This means that both CLSet and MVMap serves their purpose and might have a place in real applications.

References

[1] Gonçalves R. Preguiça N. Fonte V. Almeida P.S. Baquero C. "Scalable and Accurate Causality Tracking for Eventually Consistent Stores." In: *Distributed Applications and Interoperable Systems. DAIS 2014.* (2014). URL: https://link.springer.com/chapter/10.1007/978-3-662-43352-2_6#citeas.

[2] *Benchee, Benchmarking in Elixir*. URL: https://github.com/bencheeorg/benchee.

[3] *CAP theorem*. URL: https://www.ibm.com/cloud/learn/cap-theorem.

[4] *Commutativity*. URL: https://dictionary.cambridge.org/dictionary/english/commutative.

[5] *Delta CRDT Implementation*. URL: https://github.com/derekkraan/delta_crdt_ex.

[6] *Elixir GenServer Documentation*. URL: https://hexdocs.pm/elixir/GenServer.html.

[7] *Elixir Programming Language*. URL: https://elixir-lang.org.

[8] *Hasse Diagram*. URL: https://mathworld.wolfram.com/HasseDiagram.html.

[9] *Idempotence*. URL: https://ldapwiki.com/wiki/Idempotent.

[10] Martin Kleppmann. "Local-First Software:You Own Your Data, in spite of the Cloud." In: (2019). URL: https://martin.kleppmann.com/papers/local-first.pdf.

[11] Christopher Meiklejohn and Peter Van Roy. "Lasp: A Language for Distributed, Coordination-Free Programming." In: *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*. PPDP '15. Siena, Italy: Association for Computing Machinery, 2015, 184–195. ISBN: 9781450335164. DOI: 10.1145/2790449.2790525. URL: https://doi.org/10.1145/2790449.2790525.

[12] nLab authors. *semilattice*. http://ncatlab.org/nlab/show/semilattice. Revision 17. Mar. 2020.

[13] *PaPoC Workshop 2020*. URL: https://papoc-workshop.github.io/2020/.

[14] *Partially Ordered Set*. URL: https://ncatlab.org/nlab/show/partial+order.

[15] Carlos Baquero Paulo Sérgio Almeida Ali Shoker. “Delta State Replicated Data Types.” In: *Journal of Parallel and Distributed Computing, Volume 111, January 2018, Pages 162-173* (2016). URL: https://arxiv.org/abs/1603.01529.

[16] Preguiça N. M. Baquero C. Shapiro M. and M. Zawirski. “Conflict-Free Replicated Data Types.” In: *Stabilization, Safety, and Security of Distributed Systems. SSS 2011.* (2011). URL: https://pages.lip6.fr/Marc.Shapiro/papers/RR-7687.pdf.

[17] Weihai Yu and Sigbjørn Rostad. “A Low-Cost Set CRDT Based on Causal Lengths.” In: *Proceedings of the 7thWorkshop on Principles and Practice of Consistency for Distributed Data*. PaPoC ’20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 9781450375245. DOI: 10.1145/3380787.3393678. URL: https://doi.org/10.1145/3380787.3393678.

Appendix

.1 CLSet

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
© 2020 Copyright held by the owner/author(s).
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.

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

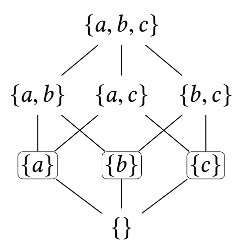


Figure 1. GSet CRDT and Hasse diagram of states

Figure 1 (left) shows GSet, a state-based CRDT for grow-only sets, where E is a set of possible elements, $\sqsubseteq \overset{\text{def}}{=} \subseteq$, $\sqcup \overset{\text{def}}{=} \cup$, 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 $\mathsf{add}(\{\}, a)$, 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.

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

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 $f\colon \mathsf{dom}(f) \to Y$ where $\mathsf{dom}(f) \subseteq X$. A *partial function* $f\colon X \hookrightarrow Y$ maps x to $\bot_Y$ if $x \notin \mathsf{dom}(f)$, 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

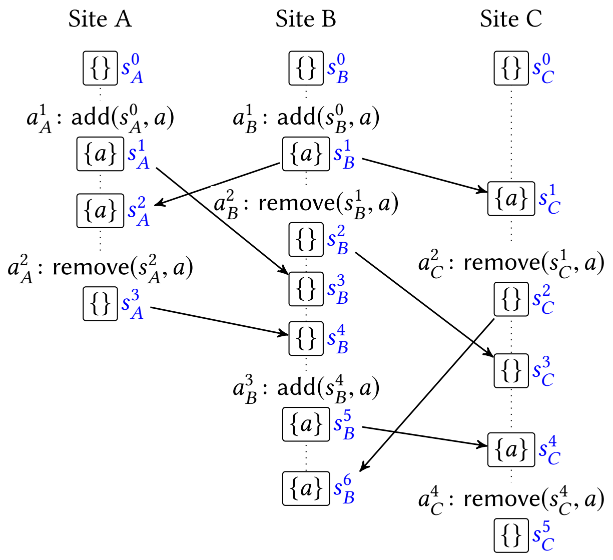


Figure 3. A scenario of concurrent set updates

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.

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

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

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

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

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

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

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

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

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

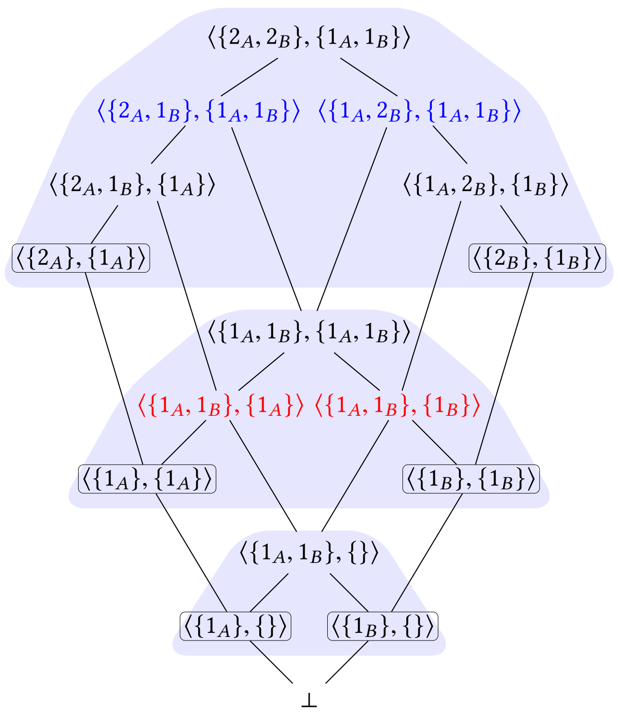

$\langle\{2_A, 2_B\}, \{1_A, 1_B\}\rangle$
$\langle\{2_A, 1_B\}, \{1_A, 1_B\}\rangle$
$\langle\{1_A, 2_B\}, \{1_A, 1_B\}\rangle$
$\langle\{2_A, 1_B\}, \{1_A\}\rangle$
$\langle\{1_A, 2_B\}, \{1_B\}\rangle$
$\langle\{2_A\}, \{1_A\}\rangle$
$\langle\{2_B\}, \{1_B\}\rangle$
$\langle\{1_A, 1_B\}, \{1_A, 1_B\}\rangle$
$\langle\{1_A, 1_B\}, \{1_A\}\rangle$
$\langle\{1_A, 1_B\}, \{1_B\}\rangle$
$\langle\{1_A\}, \{1_A\}\rangle$
$\langle\{1_B\}, \{1_B\}\rangle$
$\langle\{1_A, 1_B\}, \{\}\rangle$
$\langle\{1_A\}, \{\}\rangle$
$\langle\{1_B\}, \{\}\rangle$
$\perp$

Figure 5. States of a single element in ORSet

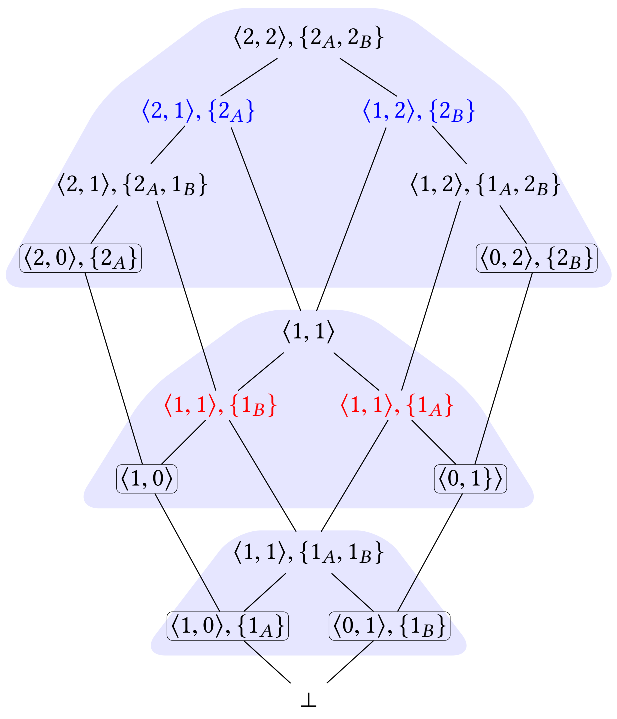

$\langle 2, 2\rangle, \{2_A, 2_B\}$
$\langle 2, 1\rangle, \{2_A\}$
$\langle 1, 2\rangle, \{2_B\}$
$\langle 2, 1\rangle, \{2_A, 1_B\}$
$\langle 1, 2\rangle, \{1_A, 2_B\}$
$\langle 2, 0\rangle, \{2_A\}$
$\langle 0, 2\rangle, \{2_B\}$
$\langle 1, 1\rangle$
$\langle 1, 1\rangle, \{1_B\}$
$\langle 1, 1\rangle, \{1_A\}$
$\langle 1, 0\rangle$
$\langle 0, 1\rangle$
$\langle 1, 1\rangle, \{1_A, 1_B\}$
$\langle 1, 0\rangle, \{1_A\}$
$\langle 0, 1\rangle, \{1_B\}$
$\perp$

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.

$$
\begin{aligned}
\mathsf{TFAWSet} &\overset{\text{def}}{=} (E \hookrightarrow \mathcal{P}(dots)) \times \mathcal{P}(dots)\\
\mathsf{add}_i^{\delta}(\langle m, c\rangle, e) &\overset{\text{def}}{=} \langle \{e \to d\}, d\rangle \text{ where } d = \{\mathsf{next}_i(c)\}\\
\mathsf{remove}_i^{\delta}(\langle m, c\rangle, e) &\overset{\text{def}}{=} \langle \{\}, m(e)\rangle\\
\langle m, c\rangle \sqcup \langle m', c'\rangle &\overset{\text{def}}{=} \langle \{e \to d'' \mid e \in \mathsf{dom}(m) \cup \mathsf{dom}(m')\\
&\qquad\qquad \wedge d'' \neq \{\}\},\\
&\qquad c \cup c'\rangle\\
&\qquad \text{where } d = m(e), d' = m'(e) \text{ and}\\
&\qquad d'' = (d \cap d') \cup (d - c') \cup (d' - c)\\
\mathsf{in}(\langle m, c\rangle, e) &\overset{\text{def}}{=} e \in \mathsf{dom}(m)\\
\mathsf{all}(\langle m, c\rangle) &\overset{\text{def}}{=} \mathsf{dom}(m)
\end{aligned}
$$

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 $(m(e), c_e) \sqsubset (m'(e), c'_e) \overset{\text{def}}{=} (c_e \subset c'_e) \vee (c_e = c'_e \wedge m(e) \supset m'(e))$. 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]

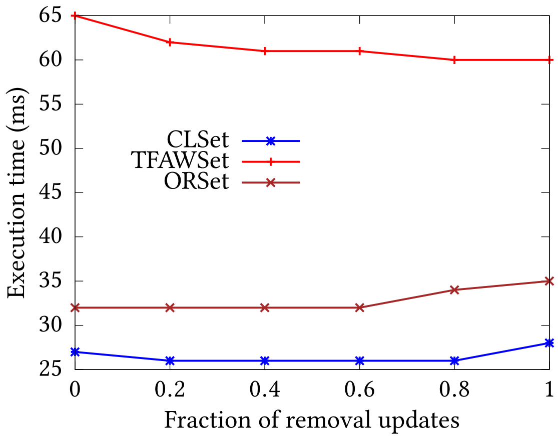


Figure 8. Time for concurrent updates and merges

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

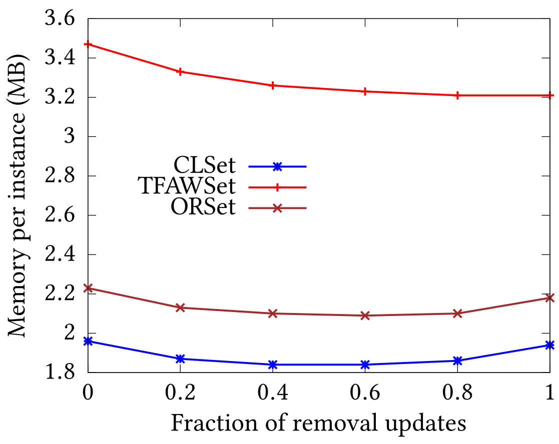


Figure 9. Memory consumption per CRDT instance

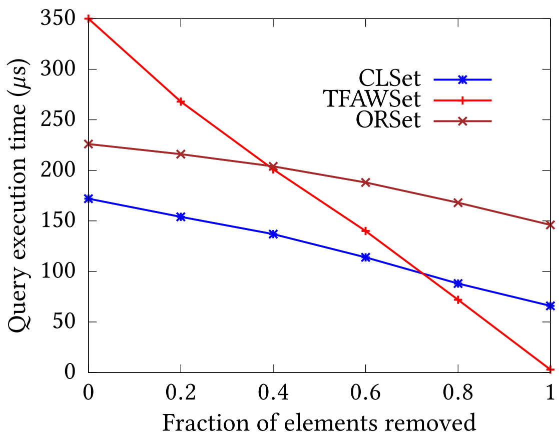


Figure 10. Time for performing the all query

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.