Advanced Conflict-free Replicated DataTypes

A preprint part of Book Chapter 4: Data management techniques in Ultrascale Computing Systems[*]

Ali Shoker,[†]Anna Queralt[‡]and Toni Cortes[§]

February 26, 2019

One approach to achieve ultra-scale data management is to use a full data replication paradigm with a relaxed consistency model. This is advocated given the tradeoffs of Availability, Consistency, and network Partition addressed by the CAP theorem [10]. While a relaxed consistency model allows for prompt local updates, this can lead to potential conflicts in the data once merged elsewhere. With the premise to eventually converge to a single state, manual and case-tailored solutions are unproven correct and cumbersome to use. In this section, we present a more generic and mathematically proven method though Conflict-free Replicated DataTypes [16] that guarantee eventual convergence. We present four variants of CRDT models: operation-based, pure operation-based, state-based, and delta-state based CRDTs [16, 2, 3, 7, 5]. We aim to keep the presentation simple by addressing a common "set" datatype example throughout all CRDT variants to show their differences. We finally present a case study, on the *dataClay* [15, 14] distributed platform, demonstrating how CRDTs can be used in practice[1].

1 Scalability and Availability Tradeoffs

With the immense volumes of data generated by social networks, Internet of Things, and data science, scalability becomes one of the major data management challenges. To sustain such data volumes, a data management service must guarantee both a large data storage capacity and high availability. Although the former can be increased through scaling up, i.e., augmenting the storage capacity of a service though adding larger hard drives or using RAID technology,

[*]Book: `https://digital-library.theiet.org/content/books/10.1049/pbpc024e_ch4`

[†]HASLab, INESC TEC & University of Minho, Portugal

[‡]Barcelona Supercomputing Center (BSC), Spain

[§]BSC and Universitat Politcnica de Catalunya, Spain

[1]The research leading to these results has received funding from the European Union's Horizon 2020 - The EU Framework Programme for Research and Innovation 2014-2020, under grant agreement No. 732505, LightKone project.

the availability challenge remains, due to bottlenecks and single points of failure. A typical alternative is to scale out through distributing, a.k.a., replicating, the data over distinct nodes either within a small proximity, like a cluster, or scattered over a large geographical space. This however raises a challenge on the quality of data, subject to the speed of Read/Write operations and data freshness, and often governed through a data consistency model [17].

Traditionally, the common approach was to fully replicate the data and use a strong consistency model, e.g., sequential consistency or total order protocols like quorum-based consensus [11, 13]. However, the overhead of synchronization becomes intolerable under scale, especially in a geographically distributed settings or loosely coupled systems. In fact, the CAP theorem [10, 1] forces to choose between (strict) data consistency and availability, given that network partitions are hard to avoid in practice. Consequently, the recent trend is to adopt a relaxed data consistency model that trades strict consistency for high availability. This is advocated by applications that cannot afford large response times on *reads* and *writes*, and thus allow for stale reads as long as propagating local writes eventually lead to convergence [19]—assuming system quiescence.

The weakest consistency models that guarantee convergence often adopt a variant of Eventual Consistency [19] (EC) in which updates are applied locally, without prior synchronization with other replicas, such that they are eventually delivered and applied by all replicas. In practice, many applications require stronger guarantees on time and order, and advocate the causal consistency model, which enforces a *happens-before* relation [12]: A is delivered before B if A occurred before B on the same machine; or otherwise, A occurred before a *send* event and B occurred after a *receive* event (possibly by transitivity). To understand the need for causal consistency in applications, consider an example on a replicated messaging service where Bob commented on Alice's message, i.e., Alice's message happened before Bob's comment. Since, in a distributed setting, it can happen that different users read from different servers (a.k.a., replicas), without enforcing causal consistency some users may read Bob's comment before Alice's message.

Even if it boosts availablity, a relaxed consistency model can lead to conflicts when concurrent operations are invoked on different replicas. Traditionally, conflicts are reconciled manually or left to the application to decide on the order (all concurrent versions are retained and exposed) [9]. This process is very costly and subject to errors which necessitates a systematic way instead. Conflict-free Replicated DataTypes [16, 2, 7] (CRDTs) are mathematical abstractions that are proven to be conflict-free and easy to use, and they are recently being adopted by leading industry like Facebook, Tom Tom, Riak DB, etc.[2]. In the rest of this section, we introduce CRDTs and present some of the recent advanced models.

[2]Some of the industry use of CRDTs can be seen here: `https://en.wikipedia.org/wiki/Conflict-free_replicated_data_type#Industry_use`.

2 Conflict-free Replicated Datatypes

Conflict-free Replicated Datatypes (CRDTs) [16, 2, 7] are data abstractions (e.g, counters,sets, maps, etc), that are usually replicated over (often loosely-connected) network of nodes, and are mathematically formalized to guarantee that replicas converge to the same state, provided they eventually apply the same updates. The assumption is that state updates should be commutative, or designed to be so. This ensures that applying concurrent updates on different nodes is independent from their application order. For instance, applying "Add A" and "Add B" to a *set* will eventually lead to both elements A and B in the two replicas. Whereas, concurrent non-commutative operations "Add A" and "Remove A" may lead to two different replicas (one is empty and the other has element A).

There are two main approaches for CRDTs: operation-based (a.k.a., op-based) and state-based. The former is based on disseminating operations whereas the latter propagates a state that results from locally applied operations. In the systems where the message dissemination layer guarantees Reliable Causal Broadcast (RCB), op-based CRDTs are desired as they allow for simpler implementations, concise replica state, and smaller messages. On the other hand, state-based CRDTs are more complex to design, but can handle duplicates and out-of-order delivery of messages without breaking causality, and thus are favored in hostile networks.

In the following, we elaborate on these models and their optimizations focusing on the Add-wins Set (AWSet) datatype (in which a concurrent add dominates a remove). We opt for AWSet being a very common datatype that has causality semantics, which helps explaining the different ordering guarantees in CRDT models. More formally, in an AWSet that retains add and rmv operations together with their timestamps t, the concurrent semantics can be defined by a read operation that returns the following elements:

$$\{v \mid (t, [\mathsf{add}, v]) \in s \wedge \forall (t', [(\mathsf{add} \mid \mathsf{rmv}), v]) \in s \cdot t \not< t'\} \quad (1)$$

Op-based CRDTs. In op-based CRDTs [16], each replica maintains a local state that is type-dependent. A replica is subject to clients' operations, i.e., *query* and *update*, that are executed locally as soon as they arrive. While query operations read the local (maybe stale) state without modifying it, updates often modify the state and generate some message to be propagated to other replicas through a reliable causal broadcast (RCB) middleware. The RCB handles the exactly-once dissemination across replicas, mainly through an API composed of causal broadcast function cbcast and causal deliver function cdeliver.

To explain the process further, we consider the general structure of an op-based CRDT in Figure 1, and convey Algorithm 1 in Figure 2 that depicts the interplay between the RCB middleware and the CRDT. In particular, when an update operation o is issued at some node i having state σ_i, the function $\mathsf{prepare}_i(o, \sigma_i)$ produces a message m that includes some ordering meta-data in addition to the operation. This message m is then broadcast by calling

Σ	: State type, σ_i is an instance
$\mathsf{prepare}_i(o, \sigma_i)$	: Prepares a message m given an operation o
$\mathsf{effect}_i(m, \sigma_i)$	: Applies a *prepared* message m on a state
$\mathsf{val}_i(q, \sigma_i)$	: Read-only evaluation of query q on a state

Figure 1: The general composition of an op-based CRDT.

```
state:
    $\sigma_i \in \Sigma$
on operation$_i(o)$:
    $m := \mathsf{prepare}_i(o, \sigma_i)$
    cbcast$_i(m)$
on cdeliver$_i(m)$:
    $\sigma_i := \mathsf{effect}_i(m, \sigma_i)$
on query$_i()$:
    val$_i(\sigma_i)$
```

Algorithm 1: CRDT & RCB

1

```
state:
    $\sigma_i = (s, \pi) \in \mathcal{P}(O) \times (O, \leq)$
on operation$_i(o)$:
    tcbcast$_i(o)$
on tcdeliver$_i(m, t)$:
    $\pi_i := \mathsf{effect}_i(m, t, \pi_i)$
on tcstable$_i(t)$:
    $\sigma_i := \mathsf{prune}_i(\pi_i, s, t)$
on query$_i()$:
    val$_i(\sigma_i)$
```

Algorithm 2: Pure CRDT & TRCB

Figure 2: Distributed algorithms for node i showing the interplay of classical and pure op-based CRDTs given a standard versus tagged reliable causal broadcast middlewares, respectively.

$\mathsf{cbcast}_i(m)$, and is delivered via $\mathsf{cdeliver}_j(m)$ at each destination node j (including i itself). cdeliver triggers $\mathsf{effect}_j(m, \sigma_j)$ that returns a new replica state σ'_j. Finally, a query operation q is issued, $\mathsf{val}_i(q, \sigma_i)$ is invoked, and no corresponding broadcast occurs.

Example. We further exemplify op-based CRDTs though an "Add-wins Set" (AWSet) CRDT design depicted in Figure 3. The state Σ is composed of a local sequence number $n \in \mathbb{N}$ and a set of values in V together with some meta-data in $I \times \mathbb{N}$ that is used to guarantee the causality information of the datatype. The prepare of an add operation produces a tuple, to be disseminate by RCB, composed of the element to be added together with the ID and the incremented sequence number of the local node i. The effect function is invoked on RCB delivery and leads to adding this tuple to the state and incrementing the sequence number only if the prepare was local at i. To the contrary, preparing a rmv returns all the tuples containing the removed item, which allows the corresponding effect to remove all these tuples. Finally, the val function returns all the elements in the set.

Pure Op-based CRDTs. Op-based CRDTs can be optimized to reduce the dissemination and storage overhead. Indeed, since op-based CRDTs assume the presence of and RCB, one can take advantage of its time abstractions, i.e.,

$$\Sigma = \mathbb{N} \times \mathcal{P}(I \times \mathbb{N} \times V) \qquad \sigma_i^0 = (0, \{\})$$

$$\begin{aligned}
\mathsf{prepare}_i([\mathsf{add}, v], (n, s)) &= [\mathsf{add}, v, i, n+1] \\
\mathsf{effect}_i([\mathsf{add}, v, i', n'], (n, s)) &= (n' \wedge (i = i') \vee n, s \cup \{(v, i', n')\}) \\
\mathsf{prepare}_i[\mathsf{rmv}, v], (n, s)) &= [\mathsf{rmv}, \{(v', i', n') \in s \mid v' = v\}] \\
\mathsf{effect}_i([\mathsf{rmv}, r], (n, s)) &= (n, s \setminus r) \\
\mathsf{val}_i(n, s) &= \{v \mid (v, i', n') \in s\}
\end{aligned}$$

Figure 3: Operation-based Add-Wins Set CRDT, at node i.

often implemented via version vectors (VV), to guarantee causal delivery and thus avoid disseminating the meta-data produced by prepare. Consequently, this leads to disseminating the "pure" operations and possible arguments which makes the prepare useless, hence the name Pure op-based CRDTs [7, 5]. In addition, this "pure" mind-set leads to having a standard state for all datatypes: a partially order log: Polog. However, this will lead to storing the VV in the state which can be very expensive when the number of replicas in the systems is high. Fortunately, and contrary to the classical op-based approach, the time notion of VV is useful to transform the Polog into a sequential log which eventually helps to prune the—no longer needed—VVs.

Tagged Reliable Causal Broadcast (TRCB). To achieve the above optimizations, we consider an extended RCB, called Tagged RCB (TRCB), that provides two functionalities through the API functions: tcdeliver and tcstable [7, 5]. The former is a equivalent to the standard RCB cdeliver presented above with a simple extension to the API by exposing the VVs (used internally by the RCB) to the node upon delivery, which can be appended to the operation by the recipient. On the other hand, tcstable is a new function that returns a timestamp τ indicating that all operations with timestamp $t \leq \tau$ are *stable*: have been delivered on all nodes. The essence is that no concurrent operations to the stable operations in the Polog are expected to be delivered, and hence, the corresponding VVs in the Polog can be pruned without affecting the datatype semantics.

Given the TRCB, depicted in Algorthim 2 of Figure 2, the design of Pure CRDTs differs from the classical ones in different aspects. First, the state is common to all datatypes, and is represented by a set of stable operations and a Polog. Second, prepare has no role anymore as the operations and its arguments can be immediately disseminated though the TRCB. Third, the significant change is with the effect function that discards datatype-specific "redundant" operations before adding to the Polog (e.g., a duplicate operation). However, to the contrary of classical op-based CRDTs, only one effect function is required per datatype. Finally, prune function is required to move stable operations (triggered via the TRCB's tcstable) to the sequential log after pruning

$$\begin{array}{rcl}
\Sigma = \mathcal{P}(O) \times T \hookrightarrow O & & \sigma_i^0 = (s_i, \pi_i)^0 = (\{\}, \{\}) \\
\mathsf{effect}_i(o, t, s, \pi) & = & (s \setminus \{[\mathsf{add}, v] \mid o = [\mathsf{rmv}, v]\}, \\
 & & \pi \setminus \{(t', [\mathsf{add}, v]) \mid o = [\mathsf{rmv}, v] \wedge t' < t\} \cup \\
 & & \{(t, o) \mid o = [\mathsf{add}, v] \wedge (\_, o) \notin \pi \wedge o \notin s\}) \\
\mathsf{val}_i(s, \pi) & = & \{v \mid [\mathsf{add}, v] \in s \vee (t, [\mathsf{add}, v]) \in \pi\} \\
\mathsf{prune}_i(\pi, s, \tau) & = & (s \cup \{o\}, \pi \setminus \{(t, o)\}) \wedge t \leq \tau
\end{array}$$

Figure 4: Pure operation-based Add-Wins Set CRDT, at node i.

the VVs.

Example. Using the same example of the AWSet, we provide the Pure CRDT version in Figure 4. The state is composed of a set of sequential operations s and Polog π: a map from timestamps to operations. The prepare does not exist due the reasons mentioned above, and hence the operation and its arguments are sent to the destination. Once a rmv operation is delivered, the effect deletes the corresponding operation from s and those in the causal past of rmv $\in \pi$. (Remember that all operations in s are in the causal past of delivered operations.) Finally, effect only adds an add to π if the element is not in s or π. Once the tcstable triggers prune, given a stable timestamp τ, all operations with timestamp $t \leq \tau$ become stable; and are thus removed by prune from π and added to s without the (now useless) timestamp.

Additional pedantic details. A catalog of many op-based CRDT specifications like counters, sets, registers, maps, etc., can be found in [5]. There are also several optimizations that are beyond the scope of this book. For instance, the pure op-based specifications can be generalized further to have a common framework for all CRDTs in such a way the user only needs to define simple datatype-specific rules to truncate the Polog. In addition, one can go deeper and optimize each datatype aside. An example is to replace the stable state with a classical datatype instead of retaining the set of operations. On the other hand, datatypes that are natively commutative can be easier to implement as classical op-based CRDTs. Finally, we avoid presenting the details of the TRCB for presentation purposes. The reader can refer to [7, 5] for these details.

State-based CRDTs. While op-based CRDTs are based on the dissemination of operations that are executed by every replica, a “state” is disseminated in the state-based CRDTs [16]. A received state is incorporated with the local state via a *merge* function that, deterministically, reconciles any existing conflicts.

As depicted in Figure 5, a state-based CRDT consists of a state, mutators, join, and query functions. The state Σ is designed as a join-semilattice [8]: a set

Σ	: State defined as *join semi-lattice*, σ_i is an instance
$Mutators$	: mutating operations that inflate the state.
$s \sqcup s'$	: LUB to merge states s and s'
$\mathsf{val}_i(q, \sigma_i)$	: Read-only evaluation of query q on a state

Figure 5: The general composition of a state-based CRDT.

with a *partial order*, and a binary *join* operation $\sqcup$ that returns the *least upper bound* (LUB) of two elements in S, and is always designed to be commutative, associative, and idempotent. On the other hand, mutators are defined as *inflation*: for any mutator m and state X, $X \sqsubseteq m(X)$. This guarantees that the state never diminishes, and thus, each subsequent state subsumes the previous state when joined elsewhere. Finally, the query operation leaves no changes on the state. Note that the specification of all these functions, and the state, are datatype-specific.

Anti-entropy protocol. To the contrary of op-based CRDTs that assume the presence of RCB, state-based CRDTs can ensure eventual convergence using a simple *anti-entropy* protocol, as in Figure 6, that periodically ships the entire local state to other replicas. Each replica merges the received state with its local state using the *join* operation. (The algorithm in Figure 6 can be more sophisticated to include retransmissions, routing, or gossiping, but we keep it simple for presentation purposes.).

```
inputs:
    $n_i \in \mathcal{P}(\mathbb{I})$, set of neighbors
durable state:
    $X_i := \bot \in S$, CRDT state
on receive$_{j,i}(Y)$
    $X_i' = X_i \sqcup Y$
on operation$_i(m)$
    $X_i' = m(X_i)$
periodically // ship state
    $j$ = random$(n_i)$
    send$_{i,j}(X_i)$
```

1

Figure 6: A basic anti-entropy algorithm for state-based CRDTs.

Example. Again, we exemplify on state-based CRDTs via Figure 7 that depicts the design of AWSet. The state Σ is composed of two sets. One set is for the addition of elements with unique tags defined by unique ID and sequence number, and another tombstones set that serves for collecting the removed tags. This design is crucial to achieve the semi-lattice inflation properties subject to mutators: add and rmv. The former adds the new element to the addition set together with a new tag leaving the tombstones set intact. An element is removed by rmv through adding its unique tag to the tombstones set. Notice that the element must not be removed from the addition set which violates inflation. Given these specifications, the query function val will simply return all the added elements that do not have corresponding tags in the tombstones set. Finally, the merge function $\sqcup$ joins any two (disseminated or not) states by sim-

$$
\begin{array}{rcl}
\Sigma & = & \mathcal{P}(\mathbb{I} \times \mathbb{N} \times E) \times \mathcal{P}(\mathbb{I} \times \mathbb{N}) \\
\sigma_i^0 & = & (\{\}, \{\}) \\
\mathsf{add}_i(e, (s, t)) & = & (s \cup \{(i, n+1, e)\}, t) \\
 & & \text{with } n = \max(\{k \mid (i, k, \_) \in s\}) \\
\mathsf{rmv}_i(e, (s, t)) & = & (s, t \cup \{(j, n) \mid (j, n, e) \in s\}) \\
\mathsf{val}_i((s, t)) & = & \{e \mid (j, n, e) \in s \wedge (j, n) \notin t\} \\
(s, t) \sqcup (s', t') & = & (s \cup s', t \cup t')
\end{array}
$$

Figure 7: State-based AWSet CRDT, at node i.

$$
\begin{array}{rcl}
\Sigma & = & \mathcal{P}(\mathbb{I} \times \mathbb{N} \times E) \times \mathcal{P}(\mathbb{I} \times \mathbb{N}) \\
\sigma_i^0 & = & (\{\}, \{\}) \\
\mathsf{add}_i^\delta(e, (s, t)) & = & (\{(i, n+1, e)\}, \{\}) \\
 & & \text{with } n = \max(\{k \mid (i, k, \_) \in s\}) \\
\mathsf{rmv}_i^\delta(e, (s, t)) & = & (\{\}, \{(j, n) \mid (j, n, e) \in s\}) \\
\mathsf{val}_i((s, t)) & = & \{e \mid (j, n, e) \in s \wedge (j, n) \notin t\} \\
(s, t) \sqcup (s', t') & = & (s \cup s', t \cup t')
\end{array}
$$

Figure 8: Delta-state AWSet CRDT, at node i.

ply computing the union of the sets, thus respecting the semi-lattice properties.

Delta-state CRDTs. Despite the simplicity and robustness of state-based CRDTs, the dissemination overhead is high as the entire state is always propagated even with small local state updates. Delta-state CRDTs [2, 3] are state-based CRDT variants that allow to "isolate" the recent updates on a state and ship the corresponding *delta*, i.e., a state in the semi-lattice corresponding only to the updates, and shipped to be merged remotely. The trick is to find new delta-mutators (a.k.a., δ-mutators) m^δ that return deltas instead of entire states (and again, these are datatype specific). Given a state X, the relation between mutators m in state-based CRDTs and m^δ is as follows:

$$X' = m(X) = X \sqcup m^\delta(X) \tag{2}$$

This represents the main change in the design of state-based CRDTs in Figure 5.

Example. Considering the AWSet example, the only difference between the delta CRDT version in Figure 8 and its state-based counterpart in Figure 7 is with mutators. One can simply notice that add^δ returns the recent update that represents the added element whereas add returns the entire state. A similar logic holds for the difference between rmv^δ and rmv. As for the $\sqcup$ and val, their design is the same in both versions; however, notice that the propagated and merged message is a delta-state in Figure 8 rather than a whole state. Indeed, although a delta-mutator returns a single delta, it more practical to join deltas locally and ship them in groups (which must not affect the $\sqcup$ in any case) as we explain next.

Causal anti-entropy protocol. This delta CRDT optimization comes at a cost: it is not longer safe to blindly merge received (delta) states when the datatype requires causal semantics. Indeed, state-based CRDTs implicitly ensure per-object causal consistency since the state itself retains all the causality information, whereas a delta state includes the tags of individual changes without any memory about the causal past. This requires a little more sophisticated anti-entropy protocol that enforces causal delivery on received deltas and supports coarse-grained shipping of delta batches, called "delta-intervals". A delta-interval $\Delta_i^{a,b}$ is a group of consecutive deltas correspOnding to a sequence of all the local delta mutations from a through $b-1$, and merged together via $\sqcup$ before shipping:

$$\Delta_i^{a,b} = \bigsqcup\{d_i^k \mid a \leq k < b\} \tag{3}$$

Given this, an anti-entropy algorithm can guarantee causal order if it respects the "causal delta-merging condition": $X_i \sqsupseteq X_j^a$. This means that a receiving replica can only join a remote delta-interval if it has already seen (and merged) all the causally preceding deltas for the same sender. The algorithm in Figure 9 is a basic anti-entropy protocol that satisfies the causal delta-merging property. (We discarded many optimization to focus on the core concept.) In addition to the state, a node retains a sequence number that, together with the acknowledgments map, helps the node to identify the missing deltas to be sent to a destination. In addition, the delta-interval D serves to batch deltas locally before sending them periodically. Once an operation is received from a client, a corresponding delta is returned by m^δ, which is then merged to the local state and joined to D for later dissemination to a random node. Once a delta interval is received, it gets merged to the local state as well as the local delta-interval buffer—to be sent to other nodes. Finally, deltas that have been sent to all nodes are garbage-collected from D.

Additional pedantic details. A catalog of many delta-state CRDT specifications like counters, sets, registers, maps, etc., can be found in [2, 3]. There are also several optimizations that are beyond the scope of this book. For instance, the tags in the tombstone set can be compressed further in a single version vector and few tags. This helps generalizing the specifications to use a common causality abstraction per all datatypes [3]. Furthermore, the causal anti-entropy protocol can consider other conditions to improve performance, e.g., through

```
inputs:
    $n_i \in \mathcal{P}(\mathbb{I})$, set of neighbors
durable state:
    $X_i := \bot \in S$, CRDT state
    $c_i := 0 \in \mathbb{N}$, sequence number
volatile state:
    $D_i := \{\} \in \mathbb{N} \hookrightarrow S$, sequence of $\delta$s
    $A_i := \{\} \in \mathbb{I} \hookrightarrow \mathbb{N}$, ack map
on $\mathsf{operation}_i(m^\delta)$
    $d = m^\delta(X_i)$
    $X'_i = X_i \sqcup d$
    $D'_i = D_i\{c_i \mapsto d\}$
    $c'_i = c_i + 1$
on $\mathsf{receive}_{j,i}(\mathsf{delta}, d, n)$
    if $d \not\sqsubseteq X_i$ then
        $X'_i = X_i \sqcup d$
        $D'_i = D_i\{c_i \mapsto d\}$
        $c'_i = c_i + 1$
    $\mathsf{send}_{i,j}(\mathsf{ack}, n)$
on $\mathsf{receive}_{j,i}(\mathsf{ack}, n)$
    $A'_i = A_i\{j \mapsto \max(A_i(j), n)\}$
periodically // ship delta-interval
    $j = \mathsf{random}(n_i)$
    $d = \bigsqcup\{D_i(l) \mid A_i(j) \leq l < c_i\}$
    $\mathsf{send}_{i,j}(\mathsf{delta}, d, c_i)$
periodically // garbage collect $\delta$s
    $l = \min\{n \mid (\_, n) \in A_i\}$
    $D'_i = \{(n, d) \in D_i \mid n \geq l\}$
```

Figure 9: Basic causal anti-entropy protocol satisfying the delta-merging condition.

considering transitive propagation of deltas or sending a complete state once a delta does not help, e.g., a node was unavailable for a long time. In this particular case, other useful alternatives to define deltas by *join decomposition* can be found in [18].

3 A case study: dataClay distributed platform

We now present a case study to demonstrate the practical use of CRDTs in a real distributed system: dataClay distributed platform [15, 14]. The aim is to give the reader an applied example of CRDTs showing how they can make the developers life easier. For that purpose, we try to be direct and simple to help the reader getting started.

dataClay. A distributed platform aimed at storing, either persistently or in memory, Java and Python objects [15, 14]. This platform enables, on the one hand, to store objects as in an object-oriented database and, on the other hand, to build applications where objects are distributed among different nodes, while still being accessible from any of the nodes where the application runs. Furthermore, dataClay enables several applications to share the same objects as part of their data set.

dataClay has three interesting properties. The first is that it stores the class methods in addition to the data. This functionality has several implications that help application developers use the data in this platform: (1) data can only be accessed using the class methods (no direct field modifications) and thus class developers can take care, for instance, of integrity constraints that will be fulfilled by all applications using the objects; and (2) methods can be executed over the objects inside the platform, without having to move the data to the application. The second property is that objects in dataClay are not flat;

$$
\begin{array}{rcl}
\Sigma & = & I \hookrightarrow \mathbb{N} \\
\sigma_i^0 & = & \{\} \\
\mathsf{inc}_i(m) & = & m\{i \mapsto m(i)+1\} \\
\mathsf{val}_i(m) & = & \displaystyle\sum_{r \in \mathbf{dom}(m)} m(r) \\
m \sqcup m' & = & \mathbf{max}(m, m')
\end{array}
$$

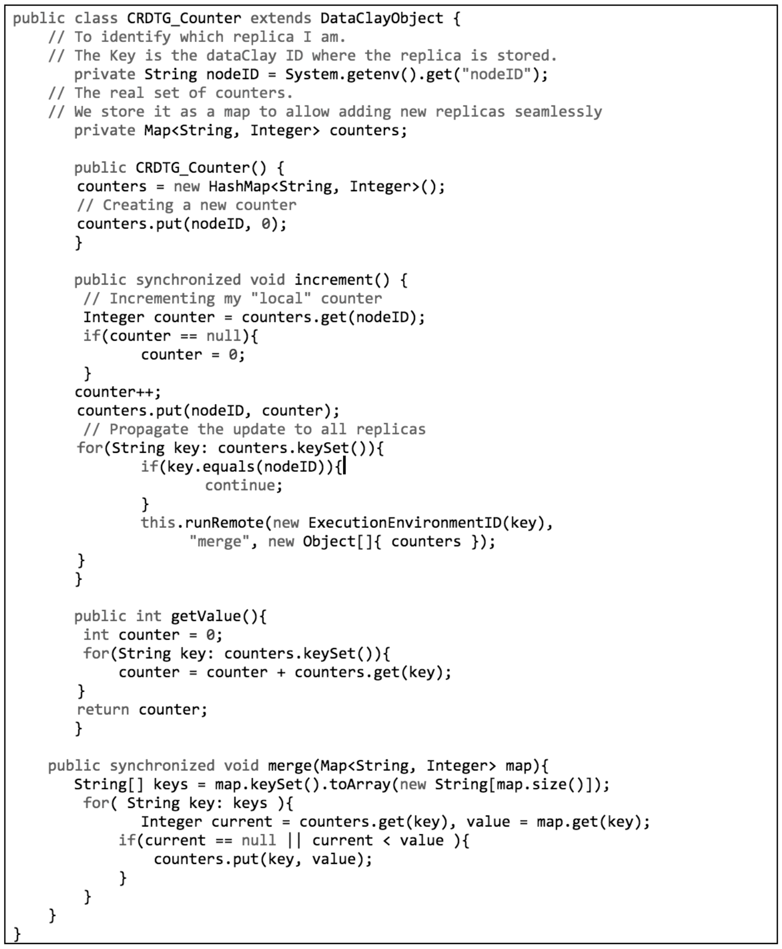
Figure 10: State based GCounter CRDT, on replica i.

and they can rather be composed of other objects, or language basic types, like in any object-oriented language. Finally, dataClay enables objects to be replicated to several nodes managed by the platform in order to increase tolerance to faults and/or execution performance by exploiting parallelism.

The case for CRDTs. Despite fully replicating the data (and class definitions) to improve tolerance to faults, dataClay does not natively implement any synchronization scheme between replicas since some applications (or modules of an application) cannot afford paying the synchronization price [4, 17]. Consequently, to provide this flexibility, dataClay tries to offer mechanisms for class developers to implement the consistency model their objects may need. Nevertheless, building such mechanisms is always tedious and, most importantly, synchronization implies a performance penalty and lack of scalability [1]. Here is where CRDTs come into play to provide a relaxed consistency and seamless plug-and-play conflict resolution for the replicated objects across the platform. Furthermore, given that code is part of the replicated data, the class developers can implement CRDTs and dataClay itself will guarantee that the update rules will be followed regardless of the application using them.

Using CRDTs. For replicating objects, dataClay can provide the developer with a library for CRDTs to be used in the classes and maybe through composing objects. Given that dataClay's system model is a graph-like Peer-to-Peer system, it is more desirable to use the state-based CRDT model since no RCB middleware is required. By using CRDTs, any application can modify the data objects without prior synchronization with other replicas or applications. Once the changes are propagated, CRDTs can eventually converge to the same value. In order to show how CRDTs are mapped to dataClay, we provide a simple example on a Grow-only Counter (GCounter) CRDT [16, 2]. We choose the counter being a simple example and at the same time shows how a semi-lattice can be different from the AWSet discussed before.

The GCounter specification is conveyed in Figure 10. In this design, the state Σ is defined as a map from node IDs to a natural number corresponding



```java
public class CRDTG_Counter extends DataClayObject {
    // To identify which replica I am.
    // The Key is the dataClay ID where the replica is stored.
       private String nodeID = System.getenv().get("nodeID");
    // The real set of counters.
    // We store it as a map to allow adding new replicas seamlessly
       private Map<String, Integer> counters;

       public CRDTG_Counter() {
        counters = new HashMap<String, Integer>();
        // Creating a new counter
        counters.put(nodeID, 0);
        }

       public synchronized void increment() {
         // Incrementing my "local" counter
         Integer counter = counters.get(nodeID);
         if(counter == null){
                counter = 0;
         }
        counter++;
        counters.put(nodeID, counter);
         // Propagate the update to all replicas
        for(String key: counters.keySet()){
                if(key.equals(nodeID)){
                        continue;
                }
                this.runRemote(new ExecutionEnvironmentID(key),
                      "merge", new Object[]{ counters });
        }
        }

       public int getValue(){
         int counter = 0;
         for(String key: counters.keySet()){
             counter = counter + counters.get(key);
        }
        return counter;
        }

    public synchronized void merge(Map<String, Integer> map){
       String[] keys = map.keySet().toArray(new String[map.size()]);
        for( String key: keys ){
                Integer current = counters.get(key), value = map.get(key);
            if(current == null || current < value ){
                 counters.put(key, value);
            }
        }
    }
}
```

Figure 11: An implementation for GCounter CRDT in dataClay.

to the increments done locally. As inc mutator shows, a node can only increment its own key, whereas the val query function returns the sum of all keys from all nodes. Once a (whole) state is propagated, the merge is done through taking the maximum counter corresponding to each key. For instance, in a system of three nodes, the following two GCounters are merged as follows: $(1, 4, 5) \sqcup (3, 4, 2) = (3, 4, 5)$. In the *Java* implementation of the GCounter presented in Figure 11, the state is coded as a *hash map* to enable the addition of new replicas on the fly, without any kind of synchronization and/or notification as part of the CRDT. In this code, the increment method pushes the new version of the hash map to all existing replicas to have the last up-to-date version and recover any potentially missed update from another node. We leave the details of the code as an exercise to the reader.

Class deployment. As mentioned above, dataClay also replicates the class definitions to be used by the applications across the systems and to allow method invocations close to the data. For this purpose, once a class is updated, the different versions must be coordinated to avoid conflicts in the semantics of the corresponding class instances. In order to allow for these updates in a loosely coordinated fashion, the classes can be designed as a Set CRDT associated with the class version. When a class update is made somewhere by any developer, the changes are deployed everywhere in the system, but they cannot be used until all replicas in the system see the new change. This concept is similar to the *causal stability* feature provided by the Tagged RCB presented before. In particular, although not all nodes detect causal stability of a version at the same time, once any node detects this, it is safe to start using that version. The reason is that causal stability ensures that the version has been delivered by all nodes in the system, and thus, the new class updates can be fetched to be used in the future. Notice that we are talking about class deployment here, but the designer must consider the compatibility between the old and the new versions, e.g., if some class instances already exist.

4 Conclusions and Future Directions

CRDTs make using replicated data less cumbersome to developers and correct being mathematically designed abstractions. However, they can only be useful when the application semantics allow for stale reads and favor immediate writes. CRDTs exist for many datatype variants of counters, sets, maps, registers, graphs, etc. They can however be extended to other types as long as operations are commutative or can be made so.

This section presented two main variants for CRDTs and their important optimizations. Some of the tradeoffs are understood, while others require future empirical investigation. Op-based CRDT designs are more intuitive to design and can be used once a reliable causal middleware is available. If it is possible to extend the middleware API, once can use pure op-based CRDTs to reduce the overhead of dissemination and storage. On the other hand, state-based CRDTs are more tolerant in hostile networks and gossip-like systems being natively idempotent: data can arrive though different nodes and get merged safely. Despite being easy to use in practice, state-based CRDTs can be expensive on dissemination when the state is not small. Consequently, it is recommended to use the delta-state CRDT alternative that significantly reduces the dissemination cost if a convenient causal anti-entropy protocol can be deployed. Furthermore, hybrid models of these variants can have tradeoff properties, and are interesting to study in the future work. Finally, it would be promising to investigate the feasibility of CRDTs in other system models and research areas like Edge Computing or Blockchain.

References

[1] Daniel Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *IEEE Computer*, 45(2):37–42, 2012.

[2] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Efficient state-based crdts by delta-mutation. In *International Conference on Networked Systems*, pages 62–76. Springer, 2015.

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

[4] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination avoidance in database systems. *PVLDB*, 8(3):185–196, 2014.

[5] Carlos Baquero, Paulo Sergio Almeida, and Ali Shoker. Pure operation-based replicated data types. *arXiv preprint arXiv:1710.04469*, 2017.

[6] Angelos Bilas. Data management techniques, 2019.

[7] Paulo S'ergio Almeida Carlos Baquero and Ali Shoker. Making operation-based crdts operation-based. In *Distributed Applications and Interoperable Systems - 14th IFIP WG 6.1 International Conference, DAIS 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proceedings*, pages 126–140, 2014.

[8] Brian A Davey and Hilary A Priestley. *Introduction to lattices and order.* Cambridge university press, 2002.

[9] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM, 2007.

[10] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News*, 33(2):51–59, 2002.

[11] James R Goodman. *Cache consistency and sequential consistency.* University of Wisconsin-Madison, Computer Sciences Department, 1991.

[12] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[13] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[14] Jonathan Martí, Anna Queralt, Daniel Gasull, Alex Barceló, Juan José Costa, and Toni Cortes. Dataclay: A distributed data store for effective inter-player data sharing. *Journal of Systems and Software*, 131:129–145, 2017.

[15] Jonathan Martí Fraiz. *dataClay: next generation object storage*. PhD thesis, Universitat Politècnica de Catalunya, 2017. PhD dissertation.

[16] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. *A comprehensive study of convergent and commutative replicated data types*. PhD thesis, Inria–Centre Paris-Rocquencourt; INRIA, 2011.

[17] Doug Terry. Replicated data consistency explained through baseball. *Commun. ACM*, 56(12):82–89, 2013.

[18] Vitor Enes, Carlos Baquero, Paulo Sergio Almeida, and Ali Shoker. Join Decompositions for Efficient Synchronization of CRDTs after a Network Partition. In *In the Proceedings of the ECOOP Programming Models and Languages for Distributed Computing Workshop*, PMLDC'16. ACM, July 2016.

[19] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.