Delta state replicated data types

Paulo Sérgio Almeida ¹, Ali Shoker *¹, Carlos Baquero ¹
Departamento de Informática, Universidade do Minho, Campus de Gualtar, 4710-057 Braga, Portugal

HIGHLIGHTS

• Definition of delta-state CRDTs and their relation to state CRDTs.
• Proofs of conditions to attain equivalence to state based CRDTs.
• Anti-entropy algorithm for basic and causally consistent convergence.
• Portfolio of delta state CRDTs including optimized sets, and recursive map.

ARTICLE INFO

Article history:
Received 19 April 2016
Received in revised form 13 June 2017
Accepted 13 August 2017
Available online 18 August 2017

Keywords:
Distributed systems
Eventual consistency
State-based
Delta
Conflict-free replicated data types
CRDT

ABSTRACT

Conflict-free Replicated Data Types (CRDTs) are distributed data types that make eventual consistency of a distributed object possible and non ad-hoc. Specifically, state-based CRDTs ensure convergence through disseminating the entire state, that may be large, and merging it to other replicas. We introduce Delta State Conflict-Free Replicated Data Types (δ-CRDT) that can achieve the best of both operation-based and state-based CRDTs: small messages with an incremental nature, as in operation-based CRDTs, disseminated over unreliable communication channels, as in traditional state-based CRDTs. This is achieved by defining δ-mutators to return a delta-state, typically with a much smaller size than the full state, that to be joined with both local and remote states. We introduce the δ-CRDT framework, and we explain it through establishing a correspondence to current state-based CRDTs. In addition, we present an anti-entropy algorithm for eventual convergence, and another one that ensures causal consistency. Finally, we introduce several δ-CRDT specifications of both well-known replicated datatypes and novel datatypes, including a generic map composition.

© 2017 Elsevier Inc. All rights reserved.

1. Introduction

Eventual consistency (EC) is a relaxed consistency model that is often adopted by large-scale distributed systems [15,18,34] where availability must be maintained, despite outages and partitioning, whereas delayed consistency is acceptable. A typical approach in EC systems is to allow replicas of a distributed object to temporarily diverge, provided that they can eventually be reconciled into a common state. To avoid application-specific reconciliation methods, costly and error-prone, Conflict-Free Replicated Data Types (CRDTs) [32,33] were introduced, allowing the design of self-contained distributed data types that are always available and eventually converge when all operations are reflected at all replicas. Though CRDTs are deployed in practice and support millions of users worldwide [9,21,30], more work is still required to improve their design and performance.

CRDTs support two complementary designs: operation-based (or op-based) and state-based. In op-based designs [26,33], the execution of an operation is done in two phases: prepare and effect. The former is performed only on the local replica and looks at the operation and current state to produce a message that aims to represent the operation, which is then shipped to all replicas. Once received, the representation of the operation is applied remotely using effect. On the other hand, in a state-based design [5,33] an operation is only executed on the local replica state. A replica periodically propagates its local changes to other replicas through shipping its entire state. A received state is incorporated with the local state via a merge function that deterministically reconciles both states. To maintain convergence, merge is defined as a join: a least upper bound over a join-semilattice [5,33].

Op-based CRDTs have some advantages as they can allow for simpler implementations, concise replica state, and smaller messages; however, they are subject to some limitations: First, they
assume a message dissemination layer that guarantees reliable exactly-once causal broadcast; these guarantees are hard to maintain since large logs must be retained to prevent duplication even if TCP is used \[20\]. Second, membership management is a hard task in op-based systems especially once the number of nodes gets larger or due to churn problems, since all nodes must be coordinated by the middleware. Third, the op-based approach requires operations to be executed individually (even when batched) on all nodes.

The alternative is to use state-based systems, which are free from these limitations. However, a major drawback in current state-based CRDTs is the communication overhead of shipping the entire state, which can get very large in size. For instance, the state size of a counter CRDT (a vector of integer counters, one per replica) increases with the number of replicas; whereas in a grow-only set, the state size depends on the set size, that grows as more operations are invoked. This communication overhead limits the use of state-based CRDTs to data-types with small state size (e.g., counters are reasonable while large sets are not). Recently, there has been a demand for CRDTs with large state sizes (e.g., in RiAK DT Maps \[10\] that can compose multiple CRDTs and that we formalize in Section 7.4.9). In this paper, we rethink the way state-based CRDTs should be designed, having in mind the problematic shipping of the entire state. Our aim is to ship a representation of the effect of recent update operations on the state, rather than the whole state, while preserving the idempotent nature of \textit{join}. This ensures convergence over unreliable communication (on the contrary to op-based CRDTs that demand exactly-once delivery and are prone to message duplication). To achieve this, we develop in detail the concept of Delta State-based CRDTs (\(\delta\)-CRDT) that we initially introduced in \[2\]. In this new (delta) framework, the state is still a join-semilattice that now results from the join of multiple fine-grained states, i.e., deltas, generated by what we call \(\delta\)-mutators. \(\delta\)-mutators are new versions of the datatype mutators that return the effect of these mutators on the state. In this way, deltas can be temporarily retained in a buffer to be shipped individually (or joined in groups) instead of shipping the entire object. The changes to the local state are then incorporated at other replicas by joining the shipped deltas with their own states.

The use of “deltas” (i.e., incremental states) may look intuitive in state dissemination; however, this is not the case for state-based CRDTs. The reason is that once a node receives an entire state, merging it locally is simple since there is no need to care about causality, as both states are self-contained (including meta-data). The challenge in \(\delta\)-CRDT is that individual deltas are now “state fragments” and usually must be causally merged to maintain the desired semantics. This raises the following questions: is merging deltas semantically equivalent to merging entire states in CRDTs? If not, what are the sufficient conditions to make this true in general? And under what constraints causal consistency is maintained? This paper answers these questions and presents corresponding proofs and examples.

We address the challenge of designing a new \(\delta\)-CRDT that conserves the correctness properties and semantics of an existing CRDT by establishing a relation between the novel \(\delta\)-mutators with the original CRDT mutators. We prove that eventual consistency is guaranteed in \(\delta\)-CRDT as long as all deltas produced by \(\delta\)-mutators are delivered and joined at other replicas, and we present a corresponding simple anti-entropy algorithm. We then show how to ensure causal consistency using deltas through introducing the concept of delta-interval and the causal delta-merging condition. Based on these, we then present an anti-entropy algorithm for \(\delta\)-CRDT, where sending and then joining delta-intervals into another replica state produces the same effect as if the entire state had been shipped and joined.

We illustrate our approach through a simple counter CRDT and a corresponding \(\delta\)-CRDT specification. Later, we present a portfolio of several \(\delta\)-CRDTs that adapt known CRDT designs and also introduce a generic kernel for the definition of CRDTs that keep a causal history of known events and a CRDT map that can compose them. All these \(\delta\)-CRDT datatypes, and a few more, are available online in a reference C++ library \[3\]. Our experience shows that a \(\delta\)-CRDT version can be devised for all CRDTs, but this requires some design effort that varies with the complexity of different CRDTs. This refactoring effort can be avoided for new datatypes by writing all mutations as delta-mutations, and only deriving the standard mutators if needed; these can be trivially obtained from the delta-mutators.

This paper is an extended version of \[2\], adding the following material: Proofs of conditions to attain equivalence to state based CRDTs; Anti-entropy algorithm for basic convergence; Portfolio of delta state CRDTs including simple compositions and anonymous replicated types (grow only sets, two phase sets, lexicographic pairs (Soundcloud \[9\]) last-writer-wins sets), named types (positive-negative counters, Cassandra \[16\]) lexicographic counters; Kernel for causal CRDTs, with a universal join function; Optimized causal CRDTs (remove-wins sets, (RiAK) flags \[6\]); Recursive map data type for causal CRDTs.

2. System model

Consider a distributed system with nodes containing local memory, with no shared memory between them. Any node can send messages to any other node. The network is asynchronous; there is no global clock, no bound on the time a message takes to arrive, and no bounds on relative processing speeds. The network is unreliable: messages can be lost, duplicated or reordered (but are not corrupted). Some messages will, however, eventually get through: if a node sends infinitely many messages to another node, infinitely many of these will be delivered. In particular, this means that there can be arbitrarily long partitions, but these will eventually heal. Nodes have access to durable storage; they can crash but will eventually recover with the content of the durable storage just before the crash occurred. Durable state is written atomically at each state transition. Each node has access to its globally unique identifier in a set \(I\).

2.1. Notation

We use mostly standard notation for sets and maps, including set comprehension of the forms \(\{x \in S \mid \varphi(x)\}\) or \(\{x \in S \mid \text{Pred}(x)\}\). A map is a set of \((k, v)\) pairs, where each \(k\) is associated with a single \(v\). Given a map \(m\), \(m(k)\) returns the value associated with key \(k\), while \(m[k \rightarrow v]\) denotes \(m\) updated by mapping \(k\) to \(v\). The domain and range of a map \(m\) is denoted by \(\text{dom} m\) and \(\text{ran} m\), respectively, \(\text{dom} m = \{k \mid (k, v) \in m\}\) and \(\text{ran} m = \{v \mid (k, v) \in m\}\). We use \(\text{fmap}\) and \(\text{snmap}\) to denote the first and second component of a pair \(p\), respectively. We use \(\text{B}\), \(\text{N}\), and \(\text{Z}\), for the boolean, natural numbers, and integers, respectively; also \(\text{f}\) for some unspecified set of nodes identifiers. Most sets we use are partially ordered and have a least element \(\bot\) (the bottom element). We use \(\text{A} \rightarrow \text{B}\) for a partial function from \(\text{A}\) to \(\text{B}\); given such a map \(m\), then \(\text{dom} m \subseteq \text{A}\) and \(\text{ran} m \subseteq \text{B}\), and for convenience we use \(m(k)\) when \(k \notin \text{dom} m\) and \(\text{B}\) has a bottom, to denote \(\bot:\); e.g., for some \(m : I \rightarrow \text{N}\), then \(m(k)\) denotes \(0\) for any unmapped key \(k\).

3. A background of state-based CRDTs

\textit{Conflict-Free Replicated Data Types} \[32,33\] (CRDTs) are distributed datatypes that allow different replicas of a distributed CRDT instance to diverge and ensures that, eventually, all replicas converge to the same state. State-based CRDTs achieve this through propagating updates of the local state by disseminating the entire state across replicas. The received states are then
merged to remote states, leading to convergence (i.e., consistent states on all replicas).

A state-based CRDT consists of a triple $(S, M, Q)$, where $S$ is a join-semilattice [17], $Q$ is a set of query functions (which return some result without modifying the state), and $M$ is a set of mutators that perform updates; a mutator $m \in M$ takes a state $X \in S$ as input and returns a new state $X' = m(X)$. A join-semilattice is a set with a partial order $\sqsubseteq$ and a binary join operation $\sqcup$ that returns the least upper bound (LUB) of two elements in $S$; a join is designed to be commutative, associative, and idempotent. Mutators are defined in such a way to be inflations, i.e., for any mutator $m$ and state $X$, the following holds:

$$X \sqsubseteq m(X).$$

In this way, for each replica there is a monotonic sequence of states, defined under the lattice partial order, where each subsequent state subsumes the previous state when joined elsewhere.

Both query and mutator operations are always available since they are performed using the local state without requiring inter-replica communication; however, as mutators are concurrently applied at distinct replicas, replica states will likely diverge. Eventual convergence is then obtained using an anti-entropy protocol that periodically ships the entire local state to other replicas. Each replica merges the received state with its local state using the join operation in $S$. Given the mathematical properties of join, if mutations stop being issued and anti-entropy proceeds, all replicas eventually converge to the same state, i.e., the least upper-bound of all states involved. State-based CRDTs are interesting as they demand little guarantees from the dissemination layer, working under message loss, duplication, reordering, and temporary network partitioning, without impacting availability and eventual convergence.

Fig. 1 represents a state-based increment-only counter. The GCounter CRDT state is a map from replica identifiers to positive integers. Initially, the bottom state $\bot$ is an empty map (unmapped keys implicitly mapping to zero). A single mutator, i.e., inc, is defined that increments the value corresponding to the local replica $i$ (returning the updated map). The query operation value returns the counter value by adding the integers in the map entries. The join of two states is the point-wise maximum of the maps. Mutators, like inc, are in general parameterized by the replica id $i$, so that their exact behavior can depend on it, while queries, like value, are typically replica agnostic and only depend on the CRDT state, regardless of in which replica they are invoked.

The main weakness of state-based CRDTs is the cost of dissemination of updates, as the full state is sent. In this simple example of counters, even though increments only update the value corresponding to the local replica $i$, the whole map will always be sent in messages, even when the other map entries remained unchanged (e.g., if no messages have been received and merged).

It would be interesting to only ship the recent modification incurred on the state, and possibly any received modifications that effectively changed it. This is, however, not possible with the current model of state-based CRDTs as mutators always return a full state. Approaches which simply ship operations (e.g., an "increment n" message), like in operation-based CRDTs, require reliable communication (e.g., because increment is not idempotent). In contrast, the modification that we introduce in the next section allows producing and encoding recent mutations in an incremental way, while keeping the advantages of the state-based approach, namely the idempotent, associative, and commutative properties of join.

4. Delta-state CRDTs

We introduce Delta-State Conflict-Free Replicated Data Types, or $\delta$-CRDT for short, as a new kind of state-based CRDTs, in which delta-mutators are defined to return a delta-state: a value in the same join-semilattice which represents the updates induced by the mutator on the current state.

Definition 4.1 (Delta-mutator). A delta-mutator $m^\delta$ is a function, corresponding to an update operation, which takes a state $X$ in a join-semilattice $S$ as parameter and returns a delta-mutation $m^\delta(X)$, also in $S$.

Definition 4.2 (Delta-group). A delta-group is inductively defined as either a delta-mutation or a join of several delta-groups.

Definition 4.3 ($\delta$-CRDT). A $\delta$-CRDT consists of a triple $(S, M^\delta, Q)$, where $S$ is a join-semilattice, $M^\delta$ is a set of delta-mutators, and $Q$ a set of query functions, where the state transition at each replica is given by either joining the current state $X \in S$ with a delta-mutation:

$$X' = X \sqcup m^\delta(X),$$

or joining the current state with some received delta-group $D$:

$$X' = X \sqcup D.$$

In a $\delta$-CRDT, the effect of applying a mutation, represented by a delta-mutation $\delta = m^\delta(X)$, is decoupled from the resulting state $X' = X \sqcup \delta$, which allows shipping this $\delta$ rather than the entire resulting state $X'$. All state transitions in a $\delta$-CRDT, even upon applying mutations locally, are the result of some join with the current state. Unlike standard CRDT mutators, delta-mutators do not need to be inflations in order to inflate a state; this is however ensured by joining their output, i.e., deltas, into the current state:

$$X \sqsubseteq X \sqcup m^\delta(X).$$

In principle, a delta could be shipped immediately to remote replicas once applied locally. For efficiency reasons, multiple deltas returned by applying several delta-mutators can be joined locally into a delta-group and retained in a buffer. The delta-group can then be shipped to remote replicas to be joined with their local states. Received delta-groups can optionally be joined into their buffered delta-group, allowing transitive propagation of deltas. A full state can be seen as a special (extreme) case of a delta-group.

If the causal order of operations is not important and the intended aim is merely eventual convergence of states, then delta-groups can be shipped using an unreliable dissemination layer that may drop, reorder, or duplicate messages. Delta-groups can always be re-transmitted and re-joined, possibly out of order, or can simply be subsumed by a less frequent sending of the full state, e.g., for performance reasons or when doing state transfers to new members.

4.1. Delta-state decomposition of standard CRDTs

A $\delta$-CRDT$(S, M^\delta, Q)$ is a delta-state decomposition of a state-based CRDT $(S, M, Q)$, if for every mutator $m \in M$, we have a corresponding mutator $m^\delta \in M^\delta$ such that, for every state $X \in S$:

$$m(X) = X \sqcup m^\delta(X).$$

$$G\text{Counter} = I \to \mathbb{N}$$

$$\bot = \{\}$$

$$\text{inc}_i(m) = m(i \mapsto m(i) + 1)$$

$$\text{value}(m) = \sum_{j \in m} m(j)$$

$$m \sqcup m' = \{j \mapsto \max(m(j), m'(j)) \mid j \in \text{dom } m \cup \text{dom } m'\}$$

Fig. 1. State-based counter CRDT; replica $i$. 

$$X \sqsubseteq m(X).$$

$$G\text{Counter} = I \to \mathbb{N}$$

$$\bot = \{\}$$

$$\text{inc}_i(m) = m(i \mapsto m(i) + 1)$$

$$\text{value}(m) = \sum_{j \in m} m(j)$$

$$m \sqcup m' = \{j \mapsto \max(m(j), m'(j)) \mid j \in \text{dom } m \cup \text{dom } m'\}$$
GCounter = \{ m \mapsto i \mapsto m(i) + 1 \}

\text{value}(m) = \sum_{j \in \text{dom } m} m(j)

m \lor m' = \{ j \mapsto \max(m(j), m'(j)) \mid j \in \text{dom } m \lor \text{dom } m' \}

This equation states that applying a delta-mutator and joining into the current state should produce the same state transition as applying the corresponding mutator of the standard CRDT.

Given an existing state-based CRDT (which is always a trivial decomposition of itself, i.e., \( m(X) = X \lor m(X) \), as mutators are inflations), it will be useful to find a non-trivial decomposition such that delta-states returned by delta-mutators in \( M^\delta \) are smaller than the resulting state:

\text{size}(m^\delta(X)) \ll \text{size}(m(X)).

In general, there are several possible delta-state decompositions, with multiple possible delta-mutators that correspond to each standard mutator. In order to minimize the generated delta-states (which will typically minimize their size) each delta-mutator chosen \( m^\delta \) should be minimal in following sense: for any other alternative choice of delta-mutator \( m'^\delta \), for any \( X, m'^\delta(X) \not\subset m^\delta(X) \). Intuitively, minimal delta-mutators do not leak into the deltas they produce any redundant information that is already present in \( X \). Moreover (although in theory not always necessarily the case) for typical datatypes that we have come across in practice, for each mutator \( m \) there exists a corresponding minimum delta-mutator \( m^\delta_i \), i.e., with \( m^\delta_i \subseteq m^\delta \) (under the standard pointwise function comparison), for any alternative delta-mutator. As we will see in the concrete examples, typically minimum delta-mutators are found naturally, with no need for some special “search”.

4.2. Example: \( \delta \)-CRDT counter

Fig. 2 depicts a \( \delta \)-CRDT specification of a counter datatype that is a delta-state decomposition of the state-based counter in Fig. 1. The state, join and value query operation remain as before. Only the mutator \( \text{inc}^\delta \) is newly defined, which increments the full map entry corresponding to the local replica and only returns that entry, instead of the full map as in the inc in the state-based CRDT counter does. This maintains the original semantics of the counter while allowing the smaller deltas returned by the delta-mutator to be sent, instead of the full map. As before, the received payload (whether one or more deltas) might not include entries for all keys in \( \mathbb{L} \), which are assumed to have zero values. The decomposition is easy to understand in this example since the equation \( \text{inc}^\delta(X) = X \lor \text{inc}^\delta(X) \) holds as \( m(i) = m(i) + 1 \) in \( m \lor [i \mapsto m(i) + 1] \). In other words, the single value for key \( i \) in the delta, corresponding to the local replica identifier, will overwrite the corresponding one in \( m \) since the former maps to a higher value (i.e., using \( \text{max} \)). Here it can be noticed that: (1) a delta is just a state, that can be joined possibly several times without requiring exactly-once delivery, and without being a representation of the “increment” operation (as in operation-based CRDTs), which is itself non-idempotent; (2) joining deltas into a delta-group and disseminating delta-groups at a lower rate than the operation rate reduces data communication overhead, since multiple increments from a given source can be collapsed into a single state counter.

5. State convergence

In the \( \delta \)-CRDT execution model, and regardless of the anti-entropy algorithm used, a replica state always evolves by joining the current state with some delta: either the result of a delta-mutation, or some alternative delta-group (which itself can be expressed as a join of delta-mutations). Without loss of generality, we assume \( S \) has a bottom \( \bot \) which is also the initial state. (Otherwise, a bottom can always be added, together with a special init delta-mutator, which returns the initial state.) Therefore, all states can be expressed as joins of delta-mutations, which makes state convergence in \( \delta \)-CRDT easy to achieve: it is enough that all delta-mutations generated in the system reach every replica, as expressed by the following proposition.

Proposition 5.1 (\( \delta \)-CRDT Convergence). Consider a set of replicas of a \( \delta \)-CRDT object, replica \( i \) evolving along a sequence of states \( X^i_t = X^i_1, \ldots \), each replica performing delta-mutations of the form \( m^\delta_{ij}(X^i_k) \) at some subset of its sequence of states, and evolving by joining the current state either with self-generated deltas or with delta-groups received from others. If each delta-mutation \( m^\delta_{ij}(X^i_k) \) produced at each replica is joined (directly or as part of a delta-group) at least once with every other replica, all replica states become equal.

Proof. Trivial, given the associativity, commutativity, and idempotence of the join operation in any join-semilattice. □

This opens up the possibility of having anti-entropy algorithms that are only devoted to enforce convergence, without necessarily providing causal consistency (enforced in standard CRDTs); thus, making a trade-off between performance and consistency guarantees. For instance, in a counter (e.g., for the number of likes on a social network), it may not be critical to have causal consistency, but merely not to lose increments and achieve convergence. The same reasoning can be applied to other situations, specially when commutative data-types are involved, e.g., grow-only-sets.

5.1. Basic anti-entropy algorithm

A basic anti-entropy algorithm that ensures eventual convergence in \( \delta \)-CRDT is presented in Algorithm 1. For the node corresponding to replica \( i \), the durable state, which persists after a crash, is simply the \( \delta \)-CRDT state \( X_i \). The volatile state \( D_i \) stores a delta-group that is used to accumulate deltas before eventually sending it to other replicas. The initial value for both \( X_i \) and \( D_i \) is \( \bot \).

When an operation is performed, the corresponding delta-mutator \( m^\delta \) is applied to the current state of \( X_i \), generating a
This delta is joined both with $X_j$ to produce a new state, and with $D_i$. In the same spirit of standard state based CRDTs, a node sends its messages in a periodic fashion, where the message payload is either the delta-group $D_i$ or the full state $X_i$; this decision is made by the function $\text{choose}$, which returns one of them. To keep the algorithm simple, a node simply broadcasts its messages without distinguishing between neighbors. After each send, the delta-group is reset to $\bot$.

Once a message is received, the payload $d$ is joined into the current $\delta$-CRDT state. The basic algorithm operates in one of two modes: (1) a transitive mode (when $\delta_i = \bot$) in which $d$ is also joined into $D_i$, allowing transitive propagation of delta-mutations, where deltas received at node $i$ from some node $j$ can later be sent to some other node $k$; (2) a direct mode where a delta-group is exclusively the join of local delta-mutations ($j$ must send its deltas directly to $k$). The decisions of whether to send a delta-group versus the full state (typically less periodically), and whether to use the transitive or direct mode are out of the scope of this paper. In general, decisions can be made considering many criteria like delta-group size, state size, message loss distribution assumptions, and network topology.

6. Causal consistency

Traditional state-based CRDTs converge using joins of the full state, which implicitly ensures per-object causal consistency [12]: each state of some replica of an object reflects the causal past of operations on the object (either applied locally, or applied at other replicas and transitively joined).

Therefore, it is desirable to have $\delta$-CRDTs offer the same causal-consistency guarantees that standard state-based CRDTs offer. This raises the question about how can delta propagation and merging of $\delta$-CRDT be constrained (and expressed in an anti-entropy algorithm) in such a manner to give the same results as if a standard state-based CRDT was used. Towards this objective, it is useful to define a particular kind of delta-group, which we call a delta-interval:

**Definition 6.1 (Delta-interval).** Given a replica $i$ progressing along the states $X_i^0, X_i^1, \ldots$, by joining delta $d_i$ (either local delta-mutation or received delta-group) into $X_i^0$ to obtain $X_i^{0+1}$, a delta-interval $\Delta_i^{a:b}$ is a delta-group resulting from joining deltas $d_i^a, \ldots, d_i^{b-1}$:

$$\Delta_i^{a:b} = \{d_i^a | a \leq k < b\}.$$

The use of delta-intervals in anti-entropy algorithms will be a key ingredient towards achieving causal consistency. We now define a restricted kind of anti-entropy algorithm for $\delta$-CRDTs.

**Definition 6.2 (Delta-interval-based Anti-entropy Algorithm).** A given anti-entropy algorithm for $\delta$-CRDTs is delta-interval-based, if all deltas sent to other replicas are delta-intervals.

Moreover, to achieve causal consistency the next condition must be satisfied:

**Definition 6.3 (Causal delta-merging Condition).** A delta-interval based anti-entropy algorithm is said to satisfy the causal delta-merging condition if the algorithm only joins $\Delta_i^{a:b}$ from replica $j$ into state $X_i$ of replica $i$ that satisfy

$$X_i \sqsupseteq X_j^a.$$

This means that a delta-interval is only joined into states that at least reflect (i.e., subsume) the state into which the first delta in the interval was previously joined. The causal delta-merging condition is important, since any delta-interval based anti-entropy algorithm for a $\delta$-CRDT that satisfies this can be used to obtain the same outcome of a standard CRDT; this is formally stated in Proposition 6.4.

**Proposition 6.4 (CRDT and $\delta$-CRDT Correspondence).** Let $(S, M, Q)$ be a standard state-based CRDT and $(S, M^\delta, Q)$ a corresponding delta-state decomposition. Any $\delta$-CRDT state reachable by an execution $E^\delta$ over $(S, M^\delta, Q)$, by a delta-interval based anti-entropy algorithm $A^\delta$ satisfying the causal delta-merging condition, is equal to a state resulting from an execution $E$ over $(S, M, Q)$, having the corresponding data-type operations, by an anti-entropy algorithm $A$ for state-based CRDTs.

**Proof.** By simulation, establishing a correspondence between an execution $E^\delta$, and execution $E$ of a standard CRDT of which $(S, M^\delta, Q)$ is a decomposition, as follows: (1) the state $(X_i, D_i, \ldots)$ of each node in $E^\delta$ containing CRDT state $X_i$, information about delta-intervals $D_i$ and possibly other information, corresponds to only $X_i$ component (in the same join-semilattice); (2) for each action which is a delta-mutation $m^\delta$ in $E^\delta$, $E$ executes the corresponding mutation $m$, satisfying $m(X_i) = X_i \cup m^\delta(X_i)$; (3) whenever $E^\delta$ contains a send action of a delta-interval $\Delta_i^{a:b}$, execution $E$ contains a send action containing the full state $X_i^a$; (4) whenever $E^\delta$ performs a join into some $X_i$ of a delta-interval $\Delta_i^{a:b}$, execution $E$ delivers and joins the corresponding message containing the full CRDT state $X_i$. By induction on the length of the trace, assume that for each replica $i$, each node state $X_i$ in $E$ is equal to the corresponding component in the node state in $E^\delta$, up to the last action in the global trace. A send action does not change replica state, preserving the correspondence. Replica states only change either by performing data-type update operations or upon message delivery by merging deltas/states respectively. If the next action is an update operation, the correspondence is preserved due to the delta-state decomposition property $m(X_i) = X_i \cup m^\delta(X_i)$. If the next action is a message delivery at replica $i$, with a merging of delta-interval/state from other replica $j$, because algorithm $A^\delta$ satisfies the causal merging-condition, it only joins into state $X_i$ a delta-interval $\Delta_j^{a:b}$ if $X_j^a \sqsupseteq X_i^a$. In this case, the outcome will be

$$X_i^{i+1} = X_i^a \sqcup \Delta_i^{a:b} = X_i^a \sqcup \bigcup \{d_j^a | a \leq l < b\} = X_i^a \sqcup X_j^a \sqcup \bigcup \{d_j^a | a \leq l < b\} = X_i^a \sqcup X_j^a \sqcup d_j^a \sqcup \Delta_j^{a:b-1} \sqcup \ldots \sqcup d_j^{b-1} \sqcup \ldots \sqcup d_j^{b-1} = \ldots = X_i^a \sqcup X_j^{a:b-1} \sqcup d_j^{b-1} = X_i^a \sqcup X_j^b.$$

The resulting state $X_i^{i+1}$ in $E^\delta$ will be, therefore, the same as the corresponding one in $E$ where the full CRDT state from $j$ has been joined, preserving the correspondence between $E^\delta$ and $E$. ∎

**Corollary 6.5 ($\delta$-CRDT causal consistency).** Any $\delta$-CRDT in which states are propagated and joined using a delta-interval-based anti-entropy algorithm satisfying the causal delta-merging condition ensures causal consistency.

**Proof.** From Proposition 6.4 and causal consistency of state-based CRDTs. ∎

This means that using any delta-interval-based anti-entropy algorithm respecting the causal delta-merging condition ensures the same properties concerning causal consistency as a standard
1 inputs: 
2 \( n_i \in \mathcal{P}(\mathbb{N}) \), set of neighbors 
3 
4 \( X_i \in \mathcal{S} \), CRDT state, \( X_0 = \bot \) 
5 \( c_i \in \mathbb{N} \), sequence number, 
6 \( c_0 = 0 \) 
7 
8 \( \text{volatile state:} \) 
9 \( D_i \in \mathbb{N} \mapsto \mathcal{S} \), deltas, 
10 \( D_0 = \{ \} \) 
11 \( A_i \in \mathbb{E} \mapsto \mathbb{E} \), ack map, 
12 \( A_0 = \{ \} \) 
13 
14 \( \text{on receive}_{j,d}(\delta, d, n) \) 
15 
16 \( \text{on receive}_{i,\text{ack}}(\delta, n) \) 
17 \( \text{on operation}(m^2) \) 
18 \( d = m^2(X_i) \) 
19 \( X_i' = X_i \cup d \) 
20 \( D_i' = D_i \cup \{ c_i \mapsto d \} \) 
21 \( c_i' = c_i + 1 \) 
22 
23 \( \text{periodically } \) // shift interval 
24 or state 
25 \( j = \text{random}(n_i) \) 
26 
27 \( \text{if } D_i > A_j(\text{then}) \) \( \) \( \) 
28 \( d = X_i \) 
29 
30 \( \text{else} \) 
31 
32 \( \text{periodically } \) // garbage 
33 collect deltas 
34 
35 \( l = \min(n[\ldots, n] \in A_i) \) 
36 \( D_i' = \{(n, d) \in D_i \mid n \geq l \} \) 
37 
38 \( \text{Algorithm 2: } \) Anti-entropy algorithm ensuring causal 
39 consistency of \( \delta \)-CRDT. 
40 
41 state-based CRDT. Next we present such an anti-entropy algo-
42 rithm. Regardless of the choice of such an algorithm, as we stated 
43 in Section 5, if all delta-mutations are joined at every-replica, 
44 we have convergence. 
45 
46 6.1. Anti-Entropy algorithm for causal consistency 
47 
48 Algorithm 2 is a delta-interval based anti-entropy algorithm 
49 which enforces the causal delta-merging condition. It can be used 
50 whenever the causal consistency guarantees of standard state-
51 based CRDTs are needed. For simplicity, it excludes some opti-
52 mizations that are important in practice, but easy to derive. The 
53 algorithm distinguishes neighbor nodes, and only sends to each 
54 one appropriate delta-intervals that obey the delta-merging con-
55 dition and can be joined at the receiving node. 
56 
57 Each node \( i \) keeps a contiguous sequence of deltas \( d_i^1, \ldots, d_i^n \) 
58 in a map \( D \) from integers to deltas, with \( l = \min \text{dom } D \) and 
59 \( u = \max \text{dom } D \). The sequence numbers of deltas are obtained from 
60 the counter \( c_i \) that is incremented when a delta (whether a delta-
61 mutation or delta-interval received) is joined with the current 
62 state. Each node \( i \) keeps an acknowledgments map \( A \) that stores, 
63 for each neighbor \( j \), the largest index \( b \) for all delta-intervals \( \Delta_j^{a,b} \) 
64 acknowledged by \( j \) (after receiving \( \Delta_j^a \) from \( i \) and joins it into \( X_j \) ). 
65 
66 Node \( i \) sends a delta-interval \( d = \Delta_j^{a,b} \) with a \( \delta \), \( d, b \) 
67 message; the receiving node \( j \), after joining \( \Delta_j^{a,b} \) into its replica 
68 state, replies with an acknowledgment message (\( \text{ack}, b \)); if an ack 
69 from \( j \) was successfully received by node \( i \), it updates the entry of \( j \) 
70 in the acknowledgment map, using the \text{max} function. This handles 
71 possible old duplicates and messages arriving out of order. 
72 
73 Like the \( \delta \)-CRDT state, the counter \( c_i \) is also kept in a durable 
74 storage. This is essential to cope with potential crash and recovery 
75 incidents. Otherwise, there would be the danger of receiving some 
76 delayed ack, for a delta-interval sent before crashing, causing the 
77 node to skip sending some deltas generated after recovery, thus 
78 violating the delta-merging condition. 
79 
80 The algorithm for node \( i \) periodically picks a random neighbor 
81 \( j \). In principle, \( i \) sends the join of all deltas starting from the latest 
82 delta acked by \( j \). Exceptionally, \( i \) sends the entire state in two cases: 
83 (1) if the sequence of deltas \( D_j \) is empty, or (2) if \( j \) is expecting 
84 from \( i \) a delta that was already removed from \( D_j \) (e.g., after a crash 
85 and recovery, when both deltas and the ack map, being volatile 
86 state, are lost). A delta message is only sent if the counter \( c_i \) has 
87 advanced past the next delta expected by node \( j \), i.e., if \( A_j(\delta) < c_i \), 
88 to avoid sending the full state in local inactivity periods, when no 
89 local operations are being issued, all neighbor nodes have acked all 
90 deltas, and garbage collection has been applied, making the \( D \) map 
91 empty. To garbage collect old deltas, the algorithm periodically 
92 removes the deltas that have been acked by all neighbors. 
93 
94 Proposition 6.6. Algorithm 2 produces the same reachable states as 
95 a standard algorithm over a CRDT for which the \( \delta \)-CRDT is a decompo-
96 sition, ensuring causal consistency. 
97 
98 Proof. From Proposition 6.4 and Corollary 6.5, it is enough to prove 
99 that the algorithm satisfies the causal delta-merging condition. The 
100 algorithm explicitly keeps deltas \( d_i \) tagged with increasing se-
101 quence numbers (even after a crash), according with the definition; 
102 node \( j \) only sends to \( i \) a delta-interval \( \Delta_j^{a,b} \) if \( i \) has acked \( a \); this ack is 
103 sent only if \( i \) has already joined some delta-interval (possibly a full 
104 state) \( \Delta_j^{a,\ast} \). Either \( k = 0 \) or, by the same reasoning, this \( \Delta_j^{a,\ast} \) could 
105 only have been joined at \( i \) if some other interval \( \Delta_j^{a,k} \) had already 
106 been joined into \( i \). This reasoning can be recursed until a delta-
107 interval starting from zero is reached. Therefore, \( X_j \equiv \bigcup \{d_i^j \mid 0 \leq j < k \} \) 
108 \( X_j^{a,\ast} \). 
109 
110 7. Portfolio of \( \delta \)-CRDTs 
111 
112 Having established the equivalence to classic state based CRDTs 
113 we now derive a series of specifications based on delta-mutators. 
114 Although we cover a significant number of CRDTs, the goal is not 
115 to provide an exhaustive survey, but instead to illustrate more 
116 extensively the design of specifications with deltas. In our exper-
117 ience the intellectual effort of designing a delta-based CRDT is not 
118 much higher than designing it with standard mutators. Since stan-
119 dard mutators can be obtained from delta-mutators, by composing 
120 these with join, having delta-mutators as basic building blocks can 
121 only add flexibility to the designs. 
122 
123 First, we will cover simple CRDTs and CRDT compositions that 
124 do not require distinguished node identifiers for the mutation. 
125 Next, we cover CRDTs that require a unique identifier for each 
126 replica that is allowed to mutate the state, and make use of this 
127 identifier in one or more of the mutations. Then, we address the 
128 important class of what we denote by \textit{Causal CRDTs}, presenting a 
129 generic design in which the state lattice is formed by a \textit{dot store} 
130 and a \textit{causal context}. We define three such dot stores and corre-
131 sponding lattices, which are then used to define several causal 
132 CRDTs. We conclude the portfolio with a design, a causal CRDT which 
133 can correctly embed any causal CRDT, including the map itself. 
134 
135 All of the selected CRDTs have delta implementations available 
136 in C++ [3], that complement the specifications. Most of the Causal 
137 CRDTs, including the Map, are also available in Erlang and deployed 
138 in production as part of Riak DT [6]. 
139 
140 7.1. Simple lattice compositions 
141 
142 To obtain composite CRDTs, a basic ingredient is being able to 
143 obtain states, which are join-semilattices, as composition of 
144 join-semilattices. Two common useful cases are the product and 
145 lexicographic product. Other examples of lattice composition 
146 are presented in [17,23]. 
147 
148 7.1.1. Pair 
149 
150 In Fig. 3 we show the standard pair composition. The bottom is 
151 the pair of respective bottoms and the join is the coordinate–wise 
152 join of the components. This can be generalized to products of more 
153 than two components.
\[
\text{Pair}(A, B) = A \times B \\
\bot = (\bot, \bot) \\
(a, b) \sqcup (a', b') = (a \sqcup a', b \sqcup b')
\]

**Fig. 3.** Pair of join-semilattices.

\[
\text{LexPair}(A, B) = A \boxtimes B \\
\bot = (\bot, \bot) \\
(a, b) \sqcup (a', b') = \begin{cases} 
(a, b) & \text{if } a > a' \\
(a', b') & \text{if } a' > a \\
(a \sqcup a', b \sqcup b') & \text{if } a = a' \\
(a \sqcup a', \bot) & \text{otherwise}
\end{cases}
\]

**Fig. 4.** Lexicographic pair of join-semilattices.

\[
\text{GSet}(E) = \mathcal{P}(E) \\
\bot = \{\} \\
\text{insert}^i_e(x, s) = \{x\} \\
\text{elements}(s) = s \\
\text{union}(s, s') = s \sqcup s'
\]

**Fig. 5.** \(\delta\)-CRDT grow-only set, replica \(i\).

7.1.2. Lexicographic pair

A variation of the pair composition is to establish a lexicographic pair. In this construction, in Fig. 4, the first element takes priority in establishing the outcome of the join, and a join of the second component is only performed on a tie. One important special case is when the first component is a total order; it can be used, e.g., to define an outcome based on the comparison of a time-stamp, as will be shown later.

7.2. Anonymous \(\delta\)-CRDTs

The simplest CRDTs are anonymous. This occurs when the mutators do not make use of a globally unique replica identifier, having a uniform specification for all replicas. (Although for uniformity of notation we will keep parameterizing mutators by replica identifier.)

7.2.1. GSet

A simple example is illustrated by a grow-only set, in Fig. 5. The single delta mutator \(\text{insert}^i_e(x, s)\) does not even need to consider the current state of the replica, available in \(s\), and simply produces a delta with a singleton set containing the element \(e\) to be added. This delta \(\{e\}\) when joined to \(s\) produces the desired result: an inflated set \(s \sqcup \{e\}\) that includes element \(e\). The join of grow-only sets is trivially obtained by unioning the sets.

7.2.2. 2PSet

In case one needs to remove elements, there are multiple ways of addressing it. The simplest way is to include another (grow-only) set that gathers the removed elements. This is done in Fig. 6, which shows a two-phase set, with state being a pair of sets. The name comes from the fact that elements may go through two phases: the added phase and the removed phase. The shortcoming of this simple design is that once removed, elements cannot be re-added.

\[
\text{2PSet}(E) = \mathcal{P}(E) \times \mathcal{P}(E) \\
\bot = (\bot, \bot) \\
\text{insert}^i_e(x, (s, t)) = (\{x\}, t) \\
\text{remove}^i_e(x, (s, t)) = (\bot, \{x\}) \\
\text{elements}(s, t) = s \setminus t \\
(s, t) \sqcup (s', t') = (s \sqcup s', t \sqcup t')
\]

**Fig. 6.** \(\delta\)-CRDT two-phase set, replica \(i\).

\[
\text{AWLWSSet}(E) = E \mapsto \mathbb{N} \boxtimes B \\
\bot = \{\} \\
\text{insert}^i_e(x, t, m) = \{x \mapsto (t, \text{True})\} \\
\text{remove}^i_e(x, t, m) = \{x \mapsto (t, \text{False})\} \\
\text{elements}(m) = \{e \mid (e, (\_., \text{True})) \in m\} \\
m \sqcup m' = \{e \mapsto m(e) \sqcup m'(e) \mid e \in \text{dom } m \sqcup \text{dom } m'\}
\]

**Fig. 7.** \(\delta\)-CRDT Add-Wins LWW Set, replica \(i\).

If we look at the query function elements it is clear that the data-type is presenting to the user the set difference between the added elements and the removed elements (those stored in the tombstone set \(t\)). Removing an element simply adds it to the removed set. (A variant of 2PSet with guarded removes [36] only does so if the element is already present in the added set.) The join is simply a pairwise join.

7.2.3. Add-Wins last-writer-wins set

This construction, depicted in Fig. 7, manages a set of elements of type \(E\) by tagging them with timestamps from some total order—here we use natural numbers. Each time an element is added, it is tagged with a client supplied timestamp and the boolean True. Removed elements are similarly tagged, but with the boolean False. Elements marked with True are considered to be in the set. When joining two such sets, those elements in common will have to compete to define if they are in the set. By using lexicographic pairs, we obtain the behavior that elements with higher (more recent) time-stamps will win, defining the presence according to the boolean tag; if there is a tie in the time-stamp, adds will win, since we order False < True.

Notice that it is up to the client to ensure that supplied timestamps always grow monotonically. Failure to do so is a common source of errors in timestamp based systems [24]. A dual construction to the Add-Wins LWW Set is a Remove-Wins LWW Set, where remove operations take priority on the event of a time-stamp tie. This construction has been widely deployed in production as part the SoundCloud system [9].

7.3. Named \(\delta\)-CRDTs

Another design strategy for conflict-free data-types is to ensure that each replica only changes a specific part of the state. In Section 4, we defined a GCounter that, using a map from globally unique replica identifiers to natural numbers, keeps track of how many increment operations each replica did. This was the first example of a named CRDT, the construction covered in this section. The distinction from anonymous CRDTs is that mutations make use of the replica identifier \(i\).
7.3.1. PnCounter

By composing, in a pair, two grow-only counters we obtain a positive–negative counter that can track both increments and decrements. Shown in Fig. 8, the increment and decrement operations will update the first and second components of the pair, respectively. As expected, the value is obtained by subtracting the decrements from the increments.

7.3.2. Lexicographic counter

While the PNCounter was one of the first CRDTs to be added to a production database, in Riak 1.4 [7], the competing Cassandra database had its own counter implementations based on the LWW strategy. Interestingly it proved to be difficult to avoid semantic anomalies in the behavior of those early counters, and since Cassandra 2.1, a new counter was introduced [16]. We capture its main anomalies in the behavior of those early counters, and since Cas-}

7.4. Causal δ-CRDTs

We now introduce a specific class of CRDTs, that we will refer to as causal CRDTs. Initial designs [32] introduced data types such as observed-remove sets and multi-value registers. While these made possible sets which allow adding and removing elements multiple times, and to model the design of the eventually consistent shopping cart, in Amazon Dynamo [18], they had sub-optimal scalability properties [12]. Later designs, such as in observed-remove sets without tombstones [8], allow an efficient management of metad-}

7.4.1. Causal context

A common property to causal CRDTs is that events can be assigned unique identifiers. A simple mechanism is to create these identifiers by appending to a globally unique replica identifier a replica-unique integer. For instance, in replica $i$ in we can create the sequence $(i, 1), (i, 2), \ldots$. Each of these pairs can be used to tag a specific event, or client action, and if we collect these pairs in a grow-only set, we can remember which events are known to each replica. The pair is called a dot and the grow-only set of pairs can be called a causal history, or alternatively a causal context, as we do here.

As seen in Fig. 10, 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 $c$.

7.4.2. Causal context compression

In practice, a causal context can be efficiently compressed without any loss of information. When using an anti-entropy algorithm that provides causal consistency, e.g., Algorithm 2, then for each replica state $X_i$ that includes a causal context $c_i$, and for any replica identifier $j \in I$, we have a contiguous sequence:

$$1 \leq n \leq \max_j(c_i) \Rightarrow (j, n) \in c_i$$

Thus, under causal consistency the causal context can always be encoded as a compact version vector $[31]$ $\langle I \rightarrow \mathbb{N} \rangle$ that keeps the maximum sequence number for each replica.

Even under non-causal anti-entropy, such as in Algorithm 1, compression is still possible by keeping a version vector that encodes the initial contiguous sequence of dots from each replica, together with a set for the non-contiguous dots. As anti-entropy proceeds, each dot is eventually encoded in the vector, and thus the set remains typically small. Compression is less likely for the causal context of delta-groups in transit or buffered to be sent, but those contexts are only transient and smaller than those in the actual replica states. Moreover, the same techniques that encode contiguous sequences of dots can also be used for transient context compression [28].

7.4.3. Dot stores

Together with a causal context, the state of a causal CRDT will use some kind of dot store, which acts as a container for data-type specific information. A dot store can be queried about the set of event identifiers (dots) corresponding to the relevant operations in the container, by function dots, which takes a dot store and returns a set of dots. In Fig. 11 we define three kinds of dot stores: a DotSet is simply a set of dots; the generic DotFun($V$) is a map from dots to some lattice $V$; the generic DotMap($K$, $V$) is a map from some set $K$ into some dot store $V$.

7.4.4. Causal δ-CRDTs

In Fig. 12 we define the join-semilattice which serves as state for Causal δ-CRDTs, where an element is a pair of dot store and causal context. We define the join operation for each of the three kinds of dot stores. These lattices are a generalization of techniques introduced in [1,8]. To understand the meaning of a state (and the
way join must behave), a dot present in a causal context but not in the corresponding dot store, means that the dot was present in the dot store, some time in the past, but has been removed meanwhile. Therefore, the causal context can track operations with remove semantics, while avoiding the need for individual tombstones.

When joining two replicas, a dot present in only one dot store, but included in the causal context of the other, will be discarded. This is clear for the simpler case of a DotSet, where the join preserves all dots in common, together with those not present in the other causal context. The DotFun case is analogous, but the container is now a map from dots to some value, allowing the value for a given dot to evolve with time, independently at each replica. It assumes the value set is a join-semilattice, and applies the corresponding join of values for each dot in common.

In the more complex case of DotMap, a map from some K to some dot store V, the join, for each key present in either replica, performs a join in the lattice Causal(V), by pairing the per-key value with the replica-wide causal context, and storing the resulting value (first component of the result) for that key, but only when it is not $\perp_V$. This allows the disassociation of a composite embedded value from a key, with no need for a per-key tombstone, by remembering in the causal context all dots from the composite value. Matching our notation, in a DotMap(K, V), any unmapped key corresponds effectively to the bottom $\perp_V$.

### 7.4.5. Enable-Wins flag

The flags are simple, yet useful, data-types that were first introduced in Riak 2.0 [6]. Fig. 13 presents an enable-wins flag. Enabling the flag simply replaces all dots in the store by a new dot; this is achieved by obtaining the dot through next(c), and making the delta mutator return a store containing just the new dot, together with a causal context containing both the new dot and all current dots in the store; this will make all current dots to be removed from the store upon a join (as previously defined), while the new dot is added. Concurrent enabling can lead to the store containing several dots. Reads will consider the flag enabled if the store is not an empty set. Disabling is similar to enabling, in that all current dots are removed from the store, but no new dot is added. It is possible to construct a dual data-type with disable-wins semantics and its code is also available [3].

### 7.4.6. Multi-Value register

A multi-value register supports read and write operations, with traditional sequential semantics. Under concurrent writes, a join makes a subsequent read return all concurrently written values, and a subsequent write will overwrite all those values. This data-type captures the semantics of the Amazon shopping cart [18], and the usual operation of Riak (when not using CRDT data-types).
MVRegister($V$) = Causal(DotFun($V$))
write$^i_c$(v, (m, c)) = $\langle \{d \to v\}, \{d\} \cup \text{dom } m \rangle$ where $d = \text{next}_c$($c$)
clear$^i_c$((m, c)) = $\langle \{\}, \text{dom } m \rangle$
read((m, c)) = ran m

Fig. 14. $\delta$-CRDT multi-value register, replica $i$.

AWSet($E$) = Causal(DotMap($E$, DotSet))
add$^i_c$((e, (m, c))) = $\langle \{e \to d\}, d \cup \text{dom } e_c \rangle$ where $d = \text{next}_c$($c$)
remove$^i_c$((e, (m, c))) = $\langle \{\}, m(c) \rangle$
clear$^i_c$((m, c)) = $\langle \{\}, \text{dots}(m) \rangle$
elements((m, c)) = $\text{dom } m$

Fig. 15. $\delta$-CRDT add-wins set, replica $i$.

Initial implementations of these registers tagged each value with a full version vector [33]; here we introduce an optimized implementation that tags each value with a single dot, by using a DotFun($V$) as dot store. In Fig. 14 we can see that the write delta mutator returns a causal context with all dots in the store, so that they are 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.

7.4.7. Add-Wins set

In an add-wins set removals do not affect elements that have been concurrently added. In this sense, under concurrent updates, an add will win over a remove of the same element. The implementation in Fig. 15, 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 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.

7.4.8. Remove-Wins set

Under concurrent adds and removes of the same element, a remove-wins set will make removes win. To obtain this behavior, the implementation in Fig. 16 uses a map from elements to a nested map from booleans to sets of dots. For both adding and removing of a given entry, the corresponding nested map is cleared (by the delta mutator inserting all corresponding dots into the causal context), and a new mapping from either True or False, respectively, to a singleton new dot is added.

When joining replicas, the nested map will collect the union of the respective sets in the corresponding entry (for dots not seen by the other causal context). As before, only non-bottom entries are kept, for both outer map (non-empty maps) and nested map (non-empty DotSets). Therefore, an element is considered to be in the set if it belongs to the outer map domain, and the corresponding nested map does not contain a False entry; thus, concurrent removes will win over adds.

7.4.9. A map embedding causal $\delta$-CRDTs

Maps are important composition tools for the construction of complex CRDTs. Although grow-only maps are simple to conceive and have been used in early state based designs [5], the creation of a map that allows removal of entries and supports recursive composition is not trivial. Rik 2.0 introduced a map design that provides a clear observed-remove semantics: a remove can be seen as an “undo” of all operations leading to the embedded value, putting it in the bottom state, but remembering those operations, to undo them in other replicas which observe it by a join. Key to the design is to enable removal of keys to affect (and remember) the dots in the associated nested CRDT, to allow joining with replicas that have concurrently evolved from the before-removal point, or to ensure that re-creating entries previously removed does not introduce anomalies.

In order to obtain the desired semantics it is not possible to simply map keys to causal CRDTs having their own causal contexts. Doing so would introduce anomalies when recreating keys, since old versions of the mappings in other replicas could be considered more recent than newer mappings, since the causal contexts of the re-created entries would start again at their bottom state. The solution is to have a common causal context to the whole map, to be used for all nested components, and never reset that single context.

For an arbitrary set of keys $K$ and a causal $\delta$-CRDT Causal($V$) that we want to embed (including, recursively, the map we are defining), the desired map can be achieved through Causal(DotMap($K$, V)), where a single causal context is shared by all keys and corresponding nested CRDTs, as presented in Fig. 17. This map can embed any causal CRDT as values. For instance we can define a map of type ORMap($S$, AWSets($E$)), mapping strings $S$ to add-wins sets of elements $E$; or define a more complex recursive structure that uses a map within a map ORMap($N$, ORMap($S$, MVReg($E$)))

The map does not support a specific operation to add new entries: it starts as an empty map, which corresponds to any key implicitly mapped to bottom; then, any operation from the embedded type can be applied, through a higher-order apply, which takes a delta mutator $o^{d}_k$ to be applied, the key $k$, and the map $(m, c)$. This mutator fetches the value at key $k$ from $m$, pairs it with the shared causal context $c$, obtaining a value from the embedded type, and invokes the operation over the pair; from the resulting pair, it extracts the value to create a new mapping for that key, which it pairs with the resulting causal context. Removing a key will recursively remove the dots in the corresponding embedded value, while the clear operation will remove all dots from the store. This simplicity was achieved by encapsulating most complexity in the join (and also the dots function) of the embedded type.

8. Related work

8.1. Eventually convergent data types

The design of replicated systems that are always available and eventually converge can be traced back to historical designs in [22,35], among others. More recently, replicated data types that always eventually converge, both by reliably broadcasting operations (called operation-based) or gossiping and merging states (called state-based), have been formalized as CRDTs [5,26,32,33]. These are also closely related to Bloom [14] and Cloud Types [11]. State join-semilattices were used for deterministic parallel programming in LVars [25], where variables progress in the lattice order by joining other values, and are only accessible by special threshold reads.
arenormally not causally consistent, and thus the design of join gaps, while preserving efficiency when gaps are absent.

Concise Version Vectors \[27\]. Interval Version Vectors \[28\], later known mechanism that allows for encoding of gaps is found in consequently, techniques that allow for gaps are often used. A well in must ensure this. Without causal consistency, the causal context of \( \delta \)-CRDT is that it preserves the crucial properties (idempotence, associativity and commutativity) of standard state-based CRDT. In addition, we have shown how \( \delta \)-CRDT can achieve causal consistency; and we presented an anti-entropy algorithm that allows replacing classical state-based CRDTs by more efficient ones, while preserving their properties. As an application of our approach we designed several novel \( \delta \)-CRDT specifications, including a general framework for causal CRDTs and composition in maps.

Our approach is more relaxed than classical state-based CRDTs, and thus, can replace them without losing their power since \( \delta \)-CRDT allows shipping delta-states as well as the entire state. Another interesting observation is that \( \delta \)-CRDT can mimic the behavior of operation-based CRDTs, by shipping individual deltas on the fly but with weaker guarantees from the dissemination layer.

\textbf{9. Conclusion}

We introduced the new concept of \( \delta \)-CRDTs and devised delta-mutators over state-based datatypes which can detach the changes that an operation induces on the state. This brings a significant performance gain as it allows only shipping small states, i.e., deltas, instead of the entire state. The significant property in \( \delta \)-CRDT is that it preserves the causal properties (idempotence, associativity and commutativity) of standard state-based CRDT. In addition, we have shown how \( \delta \)-CRDT can achieve causal consistency; and we presented an anti-entropy algorithm that allows replacing classical state-based CRDTs by more efficient ones, while preserving their properties. As an application of our approach we designed several novel \( \delta \)-CRDT specifications, including a general framework for causal CRDTs and composition in maps.

Our approach is more relaxed than classical state-based CRDTs, and thus, can replace them without losing their power since \( \delta \)-CRDT allows shipping delta-states as well as the entire state. Another interesting observation is that \( \delta \)-CRDT can mimic the behavior of operation-based CRDTs, by shipping individual deltas on the fly but with weaker guarantees from the dissemination layer.

\textbf{References}


---

Paulo Sérgio Almeida is an assistant professor at the Department of Informatics at University of Minho, and a researcher at HasLab/INESC TEC. He obtained a M.Sc. degree from University of Porto in 1994 and a Ph.D. degree in Computer Science from Imperial College London in 1998. His research interests have been in the area of distributed systems, namely in causality tracking mechanisms, eventually consistent non-relational databases, fault-tolerant distributed aggregation algorithms, bloom filters, and distributed algorithms in graphs. In recent years the main focus of research has been on Conflict-free Replicated Data Types.

Ali Shoker is a researcher at HasLab/INESC TEC and an invited assistant professor at the Department of Informatics at the University of Minho, Portugal. He obtained his Ph.D. degree in Informatics and Telecommunication from the University of Toulouse in Nov. 2012 and spent some time at EPFL LRD lab, Switzerland, and INSA de Lyon, France. His research is focused on Distributed Systems; in particular, large scale distributed data management and fault tolerance. More recently, his research is focused on Conflict-free Replicated Data Types, Edge/Fog Computing, and Internet of Things.

Carlos Baquero is currently an assistant professor at the Computer Science Department in Universidade do Minho (Portugal). He obtained his M.Sc. and Ph.D. degrees from Universidade do Minho in 1994 and 2000. His research interests are focused on distributed systems, in particular in causality tracking, peer-to-peer systems, distributed data aggregation, highly dynamic distributed systems, both in internet F2F settings and in mobile and sensor networks. Recent research is focused on Conflict-free Replicated Data Types, Edge/Fog Computing, and Internet of Things.