Delta State Replicated Data Types

Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero

HASLab/INESC TEC and Universidade do Minho, Portugal

Abstract. 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; whereas operation-based CRDTs disseminate operations (i.e., small states) assuming an exactly-once reliable dissemination layer. We introduce *Delta State Conflict-Free Replicated Data Types* (δ-CRDT) that can achieve the best of both worlds: 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.

1 Introduction

Eventual consistency (EC) is a relaxed consistency model that is often adopted by large-scale distributed systems [1,2,3] 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) [4,5] 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 [6,7,8], 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 [9,5], 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 [10,5] 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 [10,5].

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 [11]. 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 [12] that can compose multiple CRDTs and that we formalize in Section 7.4).

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 *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* (δ-CRDT) that we initially introduced in [13]. 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 *δ-mutators*. *δ-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 δ-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 δ-CRDT that conserves the correctness properties and semantics of an existing CRDT by establishing a relation between the novel δ-mutators with the original CRDT mutators. We prove that eventual consistency is guaranteed in δ-CRDT as long as all deltas produced by δ-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 δ-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 δ-CRDT specification. Later, we present a portfolio of several δ-CRDT*s* 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 δ-CRDT datatypes, and a few more, are available online in a reference C++ library [14]. Our experience shows that a δ-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.

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 $\mathbb{I}$.

2.1 Notation

We use mostly standard notation for sets and maps, including set comprehension of the forms $\{f(x) \mid x \in S\}$ or $\{x \in S \mid Pred(x)\}$. 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 $m\{k \mapsto v\}$ denotes m updated by mapping k to v. The domain and range of a map m is denoted by $\mathsf{dom}\, m$ and $\mathsf{ran}\, m$, respectively, i.e., $\mathsf{dom}\, m = \{k \mid (k, v) \in m\}$ and $\mathsf{ran}\, m = \{v \mid (k, v) \in m\}$. We use $\mathsf{fst}\, p$ and $\mathsf{snd}\, p$ to denote the first and second component of a pair p, respectively. We use $\mathbb{B}$, $\mathbb{N}$, and $\mathbb{Z}$, for the booleans, natural numbers, and integers, respectively; also $\mathbb{I}$ for some unspecified set of node identifiers. Most sets we use are partially ordered and have a least element $\bot$ (the bottom element). We use $A \hookrightarrow B$ for a partial function from A to B; given such a map m, then $\mathsf{dom}\, m \subseteq A$ and $\mathsf{ran}\, m \subseteq B$, and for convenience we use $m(k)$ when $k \notin \mathsf{dom}\, m$ and B has a bottom, to denote $\bot_B$; e.g., for some $m : \mathbb{I} \hookrightarrow \mathbb{N}$, then $m(k)$ denotes 0 for any unmapped key k.

3 A Background of State-based CRDTs

Conflict-Free Replicated Data Types [4,5] (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 [15], 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

$$
\begin{aligned}
\mathsf{GCounter} &= \mathbb{I} \hookrightarrow \mathbb{N} \\
\bot &= \{\} \\
\mathsf{inc}_i(m) &= m\{i \mapsto m(i) + 1\} \\
\mathsf{value}(m) &= \sum_{j \in \mathbb{I}} m(j) \\
m \sqcup m' &= \{j \mapsto \mathsf{max}(m(j), m'(j)) \mid j \in \mathsf{dom}\, m \cup \mathsf{dom}\, m'\}
\end{aligned}
$$

Fig. 1: State-based Counter CRDT; replica i.

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_i, 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_i, 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 δ-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 1 (Delta-mutator). *A delta-mutator m^δ 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 2 (Delta-group). *A delta-group is inductively defined as either a delta-mutation or a join of several delta-groups.*

Definition 3 (δ-CRDT). *A δ-CRDT consists of a triple (S, M^δ, Q), where S is a join-semilattice, M^δ 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 δ-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 δ rather than the entire resulting state X'. All state transitions in a δ-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 δ-CRDT (S, M^δ, 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 $X \in S$:

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

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., $m(X) = X \sqcup m(X)$, as mutators are inflations), it will be useful to find a non-trivial decomposition such that delta-states returned by delta-mutators in M^δ are smaller than the resulting state:

$$\mathsf{size}(m^\delta(X)) \ll \mathsf{size}(m(X))$$

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^δ should be minimal in following sense: for any other alternative choice of delta-mutator $m^{\delta'}$, for any X, $m^{\delta'}(X) \not\sqsubset m^\delta(X)$. 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 $m^{\delta_\bot}$, i.e., with $m^{\delta_\bot} \sqsubseteq m^{\delta'}$ (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: δ-CRDT Counter

Fig. 2 depicts a δ-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 inc^δ is newly defined, which increments the map entry corresponding to the local replica and only returns that entry, instead of the full map as 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 $\mathbb{I}$, which are assumed to have zero values. The decomposition is easy to understand in this example since the equation $\mathsf{inc}_i(X) = X \sqcup \mathsf{inc}_i^\delta(X)$ holds as $m\{i \mapsto m(i) + 1\} = m \sqcup \{i \mapsto m(i) + 1\}$. 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 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

$$
\begin{aligned}
\mathsf{GCounter} &= \mathbb{I} \hookrightarrow \mathbb{N} \\
\bot &= \{\} \\
\mathsf{inc}_i^\delta(m) &= \{i \mapsto m(i) + 1\} \\
\mathsf{value}(m) &= \sum_{j \in \mathbb{I}} m(j) \\
m \sqcup m' &= \{j \mapsto \mathsf{max}(m(j), m'(j)) \mid j \in \mathsf{dom}\, m \cup \mathsf{dom}\, m'\}
\end{aligned}
$$

Fig. 2: A δ-CRDT counter; replica i.

reduces data communication overhead, since multiple increments from a given source can be collapsed into a single state counter.

5 State Convergence

In the δ-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 arbitrary 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 δ-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 1. *(δ-CRDT convergence) Consider a set of replicas of a δ-CRDT object, replica i evolving along a sequence of states $X_i^0 = \bot, X_i^1, \ldots$, each replica performing delta-mutations of the form $m_{i,k}^\delta(X_i^k)$ 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 $m_{i,k}^\delta(X_i^k)$ 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.

```
1  inputs:
2      n_i ∈ P(I), set of neighbors
3      t_i ∈ B, transitive mode
4      choose_i ∈ S × S → S, state/delta
5  durable state:
6      X_i ∈ S, CRDT state, X_i^0 = ⊥
7  volatile state:
8      D_i ∈ S, delta-group, D_i^0 = ⊥
9  on operation_i(m^δ)
10     d = m^δ(X_i)
11     X'_i = X_i ⊔ d
12     D'_i = D_i ⊔ d
13 on receive_{j,i}(d)
14     X'_i = X_i ⊔ d
15     if t_i then
16         D'_i = D_i ⊔ d
17     else
18         D'_i = D_i
19 periodically
20     m = choose_i(X_i, D_i)
21     for j ∈ n_i do
22         send_{i,j}(m)
23     D'_i = ⊥
```

Algorithm 1: Basic anti-entropy algorithm for δ-CRDT.

5.1 Basic Anti-Entropy Algorithm

A basic anti-entropy algorithm that ensures eventual convergence in δ-CRDT is presented in Algorithm 1. For the node corresponding to replica i, the durable state, which persists after a crash, is simply the δ-CRDT state X_i. The volatile state D 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^δ is applied to the current state of X_i, generating a delta d. This delta is joined both with X_i to produce a new state, and with D. 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 choose_i 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 δ-CRDT state. The basic algorithm operates in one of two modes: (1) a *transitive* mode (when t_i is true) in which d is also joined into D, 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 [16]: 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 δ-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 δ-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 4 (Delta-interval). *Given a replica i progressing along the states $X_i^0, X_i^1, \ldots$, by joining delta d_i^k (either local delta-mutation or received delta-group) into X_i^k to obtain X_i^{k+1}, a delta-interval $\Delta_i^{a,b}$ is a delta-group resulting from joining deltas $d_i^a, \ldots, d_i^{b-1}$:*

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

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 δ-CRDTs.

Definition 5 (Delta-interval-based anti-entropy algorithm). *A given anti-entropy algorithm for δ-CRDTs is delta-interval-based, if all deltas sent to other replicas are delta-intervals.*

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

Definition 6 (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 $\Delta_j^{a,b}$ 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 δ-CRDT that satisfies it can be used to obtain the same outcome of a standard CRDT; this is formally stated in Proposition 2.

Proposition 2. *(CRDT and δ-CRDT correspondence) Let (S, M, Q) be a standard state-based CRDT and (S, M^δ, Q) a corresponding delta-state decomposition. Any δ-CRDT state reachable by an execution E^δ over (S, M^δ, Q), by a delta-interval based anti-entropy algorithm A^δ 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^δ, and execution E of a standard CRDT of which (S, M^δ, Q) is a decomposition, as

follows: 1) the state $(X_i, D_i, \ldots)$ of each node in E^δ 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^δ in E^δ, E executes he corresponding mutation m, satisfying $m(X) = X \sqcup m^\delta(X)$; 3) whenever E^δ contains a send action of a delta-interval $\Delta_i^{a,b}$, execution E contains a send action containing the full state X_i^b; 4) whenever E^δ performs a join into some X_i of a delta-interval $\Delta_j^{a,b}$, execution E delivers and joins the corresponding message containing the full CRDT state X_j^b. 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^δ, 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 $m(X) = X \sqcup m^\delta(X)$. 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^δ satisfies the causal merging-condition, it only joins into state X_i^k a delta-interval $\Delta_j^{a,b}$ if $X_i^k \sqsupseteq X_j^a$. In this case, the outcome will be:

$$\begin{aligned}
X_i^{k+1} &= X_i^k \sqcup \Delta_j^{a,b} \\
&= X_i^k \sqcup \bigsqcup \{d_j^l \mid a \leq l < b\} \\
&= X_i^k \sqcup X_j^a \sqcup \bigsqcup \{d_j^l \mid a \leq l < b\} \\
&= X_i^k \sqcup X_j^a \sqcup d_j^a \sqcup d_j^{a+1} \sqcup \ldots \sqcup d_j^{b-1} \\
&= X_i^k \sqcup X_j^{a+1} \sqcup d_j^{a+1} \sqcup \ldots \sqcup d_j^{b-1} \\
&= \ldots \\
&= X_i^k \sqcup X_j^{b-1} \sqcup d_j^{b-1} \\
&= X_i^k \sqcup X_j^b
\end{aligned}$$

The resulting state X_i^{k+1} in E^δ 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^δ and E.

Corollary 1. *(δ-CRDT causal consistency) Any δ-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 2 and causal consistency of state-based CRDTs.

6.1 Anti-Entropy Algorithm for Causal Consistency

Algorithm 2 is a delta-interval based anti-entropy algorithm which enforces the causal delta-merging condition. It can be used whenever the causal consistency

```
1  inputs:
2      n_i ∈ P(I), set of neighbors
3  durable state:
4      X_i ∈ S, CRDT state, X_i^0 = ⊥
5      c_i ∈ N, sequence number, c_i^0 = 0
6  volatile state:
7      D_i ∈ N ↪ S, deltas, D_i^0 = {}
8      A_i ∈ I ↪ N, ack map, A_i^0 = {}
9  on receive_{j,i}(delta, d, n)
10     if d ⋢ X_i then
11         X_i' = X_i ⊔ d
12         D_i' = D_i{c_i ↦ d}
13         c_i' = c_i + 1
14     send_{i,j}(ack, n)
15 on receive_{j,i}(ack, n)
16     A_i' = A_i{j ↦ max(A_i(j), n)}
17 on operation_i(m^δ)
18     d = m^δ(X_i)
19     X_i' = X_i ⊔ d
20     D_i' = D_i{c_i ↦ d}
21     c_i' = c_i + 1
22 periodically // ship interval or state
23     j = random(n_i)
24     if D_i = {} ∨ min dom D_i > A_i(j)
       then
25         d = X_i
26     else
27         d = ⨆{D_i(l) | A_i(j) ≤ l < c_i}
28     if A_i(j) < c_i then
29         send_{i,j}(delta, d, c_i)
30 periodically // garbage collect deltas
31     l = min{n | (_, n) ∈ A_i}
32     D_i' = {(n, d) ∈ D_i | n ≥ l}
```

Algorithm 2: Anti-entropy algorithm ensuring causal consistency of δ-CRDT.

guarantees of standard state-based CRDTs are needed. For simplicity, it excludes some optimizations that are important in practice, but easy to derive. The algorithm distinguishes neighbor nodes, and only sends to each one appropriate delta-intervals that obey the delta-merging condition and can joined at the receiving node.

Each node i keeps a contiguous sequence of deltas $d_i^l, \ldots, d_i^u$ in a map D from integers to deltas, with $l = \min \mathsf{dom}\, D$ and $u = \mathsf{max}\, \mathsf{dom}\, D$. The sequence numbers of deltas are obtained from the counter c_i that is incremented when a delta (whether a delta-mutation or delta-interval received) is joined with the current state. Each node i keeps an acknowledgments map A that stores, for each neighbor j, the largest index b for all delta-intervals $\Delta_i^{a,b}$ acknowledged by j (after j receives $\Delta_i^{a,b}$ from i and joins it into X_j).

Node i sends a delta-interval $d = \Delta_i^{a,b}$ with a (delta, d, b) message; the receiving node j, after joining $\Delta_i^{a,b}$ into its replica state, replies with an acknowledgment message (ack, b); if an ack from j was successfully received by node i, it updates the entry of j in the acknowledgment map, using the max function. This handles possible old duplicates and messages arriving out of order.

Like the δ-CRDT state, the counter c_i is also kept in a durable storage. This is essential to cope with potential crash and recovery incidents. Otherwise, there would be the danger of receiving some delayed ack, for a delta-interval sent before crashing, causing the node to skip sending some deltas generated after recovery, thus violating the delta-merging condition.

The algorithm for node i periodically picks a random neighbor j. In principle, i sends the join of all deltas starting from the latest delta acked by j. Exceptionally, i sends the entire state in two cases: (1) if the sequence of deltas

D_i is empty, or (2) if j is expecting from i a delta that was already removed from D_i (e.g., after a crash and recovery, when both deltas and the ack map, being volatile state, are lost). A delta message is only sent if the counter c_i has advanced past the next delta expected by node j, i.e., if $A_i(j) < c_i$, to avoid sending the full state in local inactivity periods, when no local operations are being issued, all neighbor nodes have acked all deltas, and garbage collection has been applied, making the D_i map empty. To garbage collect old deltas, the algorithm periodically removes the deltas that have been acked by *all* neighbors.

Proposition 3. *Algorithm 2 produces the same reachable states as a standard algorithm over a CRDT for which the δ-CRDT is a decomposition, ensuring causal consistency.*

Proof. From Proposition 2 and Corollary 1, it is enough to prove that the algorithm satisfies the causal delta-merging condition. The algorithm explicitly keeps deltas d_i^k tagged with increasing sequence numbers (even after a crash), according with the definition; node j only sends to i a delta-interval $\Delta_j^{a,b}$ if i has acked a; this ack is sent only if i has already joined some delta-interval (possibly a full state) $\Delta_j^{k,a}$. Either $k = 0$ or, by the same reasoning, this $\Delta_j^{k,a}$ could only have been joined at i if some other interval $\Delta_j^{l,k}$ had already been joined into i. This reasoning can be recursed until a delta-interval starting from zero is reached. Therefore, $X_i \sqsupseteq \bigsqcup\{d_j^k \mid 0 \leq k < a\} = \Delta_j^{0,a} = X_j^a$.

7 Portfolio of δ-CRDTs

Having established the equivalence to classic state based CRDTs we now derive a series of specifications based on delta-mutators. Although we cover a significant number of CRDTs, the goal is not to provide an exhaustive survey, but instead to illustrate more extensively the design of specifications with deltas. In our experience the intellectual effort of designing a delta-based CRDT is not much higher than designing it with standard mutators. Since standard mutators can be obtained from delta-mutators, by composing these with join, having delta-mutators as basic building blocks can only add flexibility to the designs.

First, we will cover simple CRDTs and CRDT compositions that do not require distinguished node identifiers for the mutation. Next, we cover CRDTs that require a unique identifier for each replica that is allowed to mutate the state, and make use of this identifier in one or more of the mutations. Then, we address the important class of what we denote by *Causal CRDTs*, presenting a generic design in which the state lattice is formed by a *dot store* and a *causal context*. We define three such dot stores and corresponding lattices, which are then used to defined several causal CRDTs. We conclude the portfolio with a Map design, a causal CRDT which can correctly embed any causal CRDT, including the map itself.

All of the selected CRDTs have delta implementations available in C++ [14], that complement the specifications. Most of the Causal CRDTs, including the

$$
\begin{aligned}
\mathsf{Pair}\langle A, B\rangle &= A \times B \\
\bot &= (\bot, \bot) \\
(a, b) \sqcup (a', b') &= (a \sqcup a', b \sqcup b')
\end{aligned}
$$

Fig. 3: Pair of join-semilattices.

$$
\begin{aligned}
\mathsf{LexPair}\langle A, B\rangle &= A \boxtimes B \\
\bot &= (\bot, \bot) \\
(a, b) \sqcup (a', b') &= \begin{cases} (a, b) & \text{if } a > a' \\ (a', b') & \text{if } a' > a \\ (a, b \sqcup b') & \text{if } a = a' \\ (a \sqcup a', \bot) & \text{otherwise} \end{cases}
\end{aligned}
$$

Fig. 4: Lexicographic pair of join-semilattices.

Map, are also available in Erlang and deployed in production as part of Riak DT [17].

7.1 Simple Lattice Compositions

To obtain composite CRDTs, a basic ingredient is being able to obtain states, which are join-semilattices, as composition of join-semilattices. Two common useful cases are the product and lexicographic product. Other examples of lattice composition are presented in [18,15].

Pair In Figure 3 we show the standard pair composition. The bottom is the pair of respective bottoms and the join is the coordinate-wise join of the components. This can be generalized to products of more than two components.

Lexicographic Pair A variation of the *pair* composition is to establish a *lexicographic pair*. In this construction, in Figure 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 in 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 δ-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

$$
\begin{aligned}
\mathsf{GSet}\langle E\rangle &= \mathcal{P}(E) \\
\bot &= \{\} \\
\mathsf{insert}_i^\delta(e, s) &= \{e\} \\
\mathsf{elements}(s) &= s \\
s \sqcup s' &= s \cup s'
\end{aligned}
$$

Fig. 5: δ-CRDT grow-only set, replica i.

$$
\begin{aligned}
\mathsf{2PSet}\langle E\rangle &= \mathcal{P}(E) \times \mathcal{P}(E) \\
\bot &= (\bot, \bot) \\
\mathsf{insert}_i^\delta(e, (s, t)) &= (\{e\}, \bot) \\
\mathsf{remove}_i^\delta(e, (s, t)) &= (\bot, \{e\}) \\
\mathsf{elements}((s, t)) &= s \setminus t \\
(s, t) \sqcup (s', t') &= (s \sqcup s', t \sqcup t')
\end{aligned}
$$

Fig. 6: δ-CRDT two-phase set, replica i.

for all replicas. (Although for uniformity of notation we will keep parameterizing mutators by replica identifier.)

GSet A simple example is illustrated by a grow-only set, in Figure 5. The single delta mutator $\mathsf{insert}_i^\delta(e, s)$ 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 $s \cup \{e\}$ that includes element e. The *join* of grow-only sets is trivially obtained by unioning the sets.

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 Figure 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.

If we look at the query function $\mathsf{elements}$ 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* [19] only does so if the element is already present in the *added* set.) The join is simply a pairwise join.

$$
\begin{aligned}
\mathsf{AWLWWSet}\langle E\rangle &= E \hookrightarrow \mathbb{N} \boxtimes \mathbb{B} \\
\bot &= \{\} \\
\mathsf{insert}^{\delta}_{i}(e,t,m) &= \{e \mapsto (t, \mathsf{True})\} \\
\mathsf{remove}^{\delta}_{i}(e,t,m) &= \{e \mapsto (t, \mathsf{False})\} \\
\mathsf{elements}(m) &= \{e \mid (e, (\_, \mathsf{True})) \in m\} \\
m \sqcup m' &= \{e \mapsto m(e) \sqcup m'(e) \mid e \in \mathsf{dom}\, m \cup \mathsf{dom}\, m'\}
\end{aligned}
$$

Fig. 7: δ-CRDT Add-Wins LWW Set, replica i.

Add-Wins Last-Writer-Wins Set This construction, depicted in Figure 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 elements 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 behaviour 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 $\mathsf{False} < \mathsf{True}$.

Notice that is 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 [20]. 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 [6].

7.3 Named δ-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.

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 Figure 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.

$$
\begin{aligned}
\mathsf{PNCounter} &= \mathsf{GCounter} \times \mathsf{GCounter} \\
\bot &= (\bot, \bot) \\
\mathsf{inc}_i^\delta((p,n)) &= (\mathsf{inc}_i^\delta(p), \bot) \\
\mathsf{dec}_i^\delta((p,n)) &= (\bot, \mathsf{inc}_i^\delta(n)) \\
\mathsf{value}((p,n)) &= \mathsf{value}(p) - \mathsf{value}(n) \\
(p,n) \sqcup (p',n') &= (p \sqcup p', n \sqcup n')
\end{aligned}
$$

Fig. 8: δ-CRDT positive-negative counter, replica i.

$$
\begin{aligned}
\mathsf{LexCounter} &= \mathbb{I} \hookrightarrow \mathbb{N} \boxtimes \mathbb{Z} \\
\bot &= \{\} \\
\mathsf{inc}_i^\delta(m) &= \{i \mapsto m(i) + (0,1)\} \\
\mathsf{dec}_i^\delta(m) &= \{i \mapsto m(i) + (1,-1)\} \\
\mathsf{value}(m) &= \sum_{j \in \mathbb{I}} \mathsf{snd}\, m(j) \\
m \sqcup m' &= \{j \mapsto m(j) \sqcup m'(j) \mid j \in \mathsf{dom}\, m \cup \mathsf{dom}\, m'\}
\end{aligned}
$$

Fig. 9: δ-CRDT Lexicographic Counter, replica i.

Lexicographic Counter While the PNCounter was one of the first CRDTs to be added to a production database, in Riak 1.4 [21], 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 behaviour of those early counters, and since Cassandra 2.1, a new counter was introduced [22]. We capture its main properties in the Figure 9 specification of a LexCounter.

This counter is updated by either incrementing or decrementing the second component of the lexicographic pair corresponding to the replica issuing the mutation. Decrements also increment the first component, to ensure that the pair will be inflated, making it (and therefore, the just updated second component) win upon a lexicographic join.

7.4 Causal δ-CRDTs

We now introduce a specific class of CRDTs, that we will refer to as *causal CRDTs*. Initial designs [5] 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 [3], they had sub-optimal scalability properties [16]. Later designs, such as in *observed-remove sets without*

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

Fig. 10: Causal Context.

tombstones [23], allow an efficient management of meta-data state and can be applied to a broad class of data-types.

We introduce the concept of *dot store* to be used together with a *causal context* to form the state (a join-semilattice) of a causal CRDT, presenting three such dot stores and lattices. These are then used to obtain several related data-types, including flags, registers, sets, and maps.

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 $i \in \mathbb{I}$ 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 Figure 10, a causal context is a set of dots. We define two functions over causal contexts: $\mathsf{max}_i(c)$ gives the maximum sequence number for pairs in c from replica i, or 0 if there is no such dot; $\mathsf{next}_i(c)$ produces the next available sequence number for replica i given set of events in c.

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 $j \in \mathbb{I}$, we have a contiguous sequence:

$$1 \leq n \leq \mathsf{max}_j(c_i) \Rightarrow (j,n) \in c_i.$$

Thus, under causal consistency the causal context can always be encoded as a compact *version vector* [24] $\mathbb{I} \hookrightarrow \mathbb{N}$ 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

$$\mathsf{DotStore}$$
$$\mathsf{dots}\langle S : \mathsf{DotStore}\rangle \ : \ S \to \mathcal{P}(\mathbb{I} \times \mathbb{N})$$

$$\mathsf{DotSet} : \mathsf{DotStore} = \mathcal{P}(\mathbb{I} \times \mathbb{N})$$
$$\mathsf{dots}(s) = s$$

$$\mathsf{DotFun}\langle V : \mathsf{Lattice}\rangle : \mathsf{DotStore} = \mathbb{I} \times \mathbb{N} \hookrightarrow V$$
$$\mathsf{dots}(s) = \mathsf{dom}\, s$$

$$\mathsf{DotMap}\langle K,\ V : \mathsf{DotStore}\rangle : \mathsf{DotStore} = K \hookrightarrow V$$
$$\mathsf{dots}(m) = \bigcup\{\mathsf{dots}(v) \mid (\_, v) \in m\}$$

Fig. 11: Dot Stores.

only transient and smaller than those in the actual replica states. Moreover, the same techniques that encodes contiguous sequences of dots can also be used for transient context compression [25].

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 Figure 11 we define three kinds of dot stores: a DotSet is simply a set of dots; the generic $\mathsf{DotFun}\langle V\rangle$ is a map from dots to some lattice V; the generic $\mathsf{DotMap}\langle K, V\rangle$ is a map from some set K into some dot store V.

Causal δ-CRDTs In figure 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 [23,26]. 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 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 $\mathsf{DotFun}\langle V\rangle$ case is analogous,

$$\mathsf{Causal}\langle T : \mathsf{DotStore}\rangle = T \times \mathsf{CausalContext}$$

$$\sqcup \;:\; \mathsf{Causal}\langle T\rangle \times \mathsf{Causal}\langle T\rangle \to \mathsf{Causal}\langle T\rangle$$

$$\begin{aligned}
&\textbf{when} && T : \mathsf{DotSet}\\
&(s,c) \sqcup (s',c') = && ((s \cap s') \cup (s \setminus c') \cup (s' \setminus c), c \cup c')
\end{aligned}$$

$$\begin{aligned}
&\textbf{when} && T : \mathsf{DotFun}\langle _ \rangle\\
&(m,c) \sqcup (m',c') = && (\{k \mapsto m(k) \sqcup m'(k) \mid k \in \mathsf{dom}\, m \cap \mathsf{dom}\, m'\} \cup\\
& && \quad \{(d,v) \in m \mid d \notin c'\} \cup \{(d,v) \in m' \mid d \notin c\}, c \cup c')
\end{aligned}$$

$$\begin{aligned}
&\textbf{when} && T : \mathsf{DotMap}\langle _, _ \rangle\\
&(m,c) \sqcup (m',c') = && (\{k \mapsto \mathsf{v}(k) \mid k \in \mathsf{dom}\, m \cup \mathsf{dom}\, m' \wedge \mathsf{v}(k) \neq \bot\}, c \cup c')\\
& && \textbf{where } \mathsf{v}(k) = \mathsf{fst}\,((m(k),c) \sqcup (m'(k),c'))
\end{aligned}$$

Fig. 12: Lattice for Causal δ-CRDTs.

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 $\mathsf{DotMap}\langle K, V\rangle$, 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 $\mathsf{Causal}\langle V\rangle$, 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 $\bot_V$. 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 $\mathsf{DotMap}\langle K, V\rangle$, any unmapped key corresponds effectively to the bottom $\bot_V$.

Enable-Wins Flag The flags are simple, yet useful, data-types that were first introduced in Riak 2.0 [17]. Figure 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 $\mathsf{next}_i(c)$, 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

$$
\begin{aligned}
\mathsf{EWFlag} &= \mathsf{Causal}\langle\mathsf{DotSet}\rangle \\
\mathsf{enable}_i^\delta((s,c)) &= (d, d \cup s) \quad \textbf{where } d = \{\mathsf{next}_i(c)\} \\
\mathsf{disable}_i^\delta((s,c)) &= (\{\}, s) \\
\mathsf{read}((s,c)) &= s \neq \{\}
\end{aligned}
$$

Fig. 13: δ-CRDT Enable-wins Flag, replica i.

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

Fig. 14: δ-CRDT Multi-value register, replica i.

is added. It is possible to construct a dual data-type with *disable-wins* semantics and its code is also available [14].

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 [3], and the usual operation of Riak (when not using CRDT data-types). Initial implementations of these registers tagged each value with a full version vector [5]; here we introduce an optimized implementation that tags each value with a single dot, by using a $\mathsf{DotFun}\langle V\rangle$ as dot store. In Figure 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.

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 Figure 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 $\mathsf{DotMap}\langle E, \mathsf{DotSet}\rangle$

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

Fig. 15: δ-CRDT Add-wins set, replica i.

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

Fig. 16: δ-CRDT Remove-wins set, replica i.

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.

Remove-Wins Set Under concurrent adds and removes of the same element, a *remove-wins set* will make removes win. To obtain this behaviour, the implementation in Figure 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.

A Map Embedding Causal δ-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 [10], the creation of a map that allows removal of entries and supports recursive composition is not trivial. Riak 2.0 introduced a map design that provides a clear

$$
\begin{aligned}
\mathsf{ORMap}\langle K, \mathsf{Causal}\langle V\rangle\rangle &= \mathsf{Causal}\langle\mathsf{DotMap}\langle K, V\rangle\rangle \\
\mathsf{apply}_i^\delta(o_i^\delta, k, (m, c)) &= (\{k \mapsto v\}, c') \quad \textbf{where } (v, c') = o_i^\delta((m(k), c)) \\
\mathsf{remove}_i^\delta(k, (m, c)) &= (\{\}, \mathsf{dots}(m(k))) \\
\mathsf{clear}_i^\delta((m, c)) &= (\{\}, \mathsf{dots}(m))
\end{aligned}
$$

Fig. 17: δ-CRDT Map embedding Causal δ-CRDTs, with observed removes, replica i.

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

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_i^δ 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 [27,28], 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 [9,10,4,5]. These are also closely related to Bloom^L [29] and Cloud Types [30]. State join-semilattices were used for deterministic parallel programming in LVars [31], where variables progress in the lattice order by joining other values, and are only accessible by special threshold reads.

8.2 Message size.

A key feature of δ-CRDT is message size reduction and coalescing, using small-sized deltas. The general old idea of using differences between things, called "deltas" in many contexts, can lead to many designs, depending on how exactly a delta is defined. The state-based deltas introduced for Computational CRDTs [32] require an extra delta-specific merge (in addition to the standard join) which does not ensure idempotence. In [33], an improved synchronization method for non-optimized OR-set CRDT [4] is presented, where delta information is propagated; in that paper deltas are a collection of items (related to update events between synchronizations), manipulated and merged through a protocol, as opposed to normal states in the semilattice. No generic framework is defined (that could encompass other data types) and the protocol requires several communication steps to compute the information to exchange. Operation-based CRDTs [4,5,34] also support small message sizes, and in particular, *pure* flavors [34] that restrict messages to the operation name, and possible arguments. Though pure operation-based CRDTs allow for compact states and are very fast at the source (since operations are broadcast without consulting the local state), the model requires more systems guarantees than δ-CRDT do, e.g., exactly-once reliable delivery and membership information, and impose more complex integration of new replicas. The work in [35] shows a different trade-off among state deltas and pure operations, by tagging operations and creating a globally stable log of operations while allowing local transient logs to preserve availability. While having other advantages, the creation of this global log requires more coordination than our gossip approach for causally consistent delta dissemination, and can stall dissemination.

8.3 Encoding causal histories.

State-based CRDT are always designed to be causally consistent [10,5]. Optimized implementations of sets, maps, and multi-value registers can build on this assumption to keep the meta-data small [16]. In δ-CRDT, however, deltas and

delta-groups are normally not causally consistent, and thus the design of *join*, the meta-data state, as well as the anti-entropy algorithm used must ensure this. Without causal consistency, the causal context in δ-CRDT can not always be summarized with version vectors, and consequently, techniques that allow for gaps are often used. A well known mechanism that allows for encoding of gaps is found in Concise Version Vectors [36]. Interval Version Vectors [25], later on, introduced an encoding that optimizes sequences and allows gaps, while preserving efficiency when gaps are absent.

9 Conclusion

We introduced the new concept of δ-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 δ-CRDT is that it preserves the crucial properties (idempotence, associativity and commutativity) of standard state-based CRDT. In addition, we have shown how δ-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 δ-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 δ-CRDT allows shipping delta-states as well as the entire state. Another interesting observation is that δ-CRDT can mimic the behavior of operation-based CRDTs, by shipping individual deltas on the fly but with weaker guarantees from the dissemination layer.

References

1. Cribbs, S., Brown, R.: Data structures in Riak. In: Riak Conference (RICON), San Francisco, CA, USA (oct 2012)
2. Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J., Hauser, C.H.: Managing update conflicts in Bayou, a weakly connected replicated storage system. In: Symp. on Op. Sys. Principles (SOSP), Copper Mountain, CO, USA, ACM SIGOPS, ACM Press (December 1995) 172–182
3. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: Symp. on Op. Sys. Principles (SOSP). Volume 41 of Operating Systems Review., Stevenson, Washington, USA, Assoc. for Computing Machinery (October 2007) 205–220
4. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: A comprehensive study of Convergent and Commutative Replicated Data Types. Rapp. Rech. 7506, INRIA, Rocquencourt, France (January 2011)
5. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In Défago, X., Petit, F., Villain, V., eds.: Int. Symp. on Stabilization, Safety,

and Security of Distributed Systems (SSS). Volume 6976 of Lecture Notes in Comp. Sc., Grenoble, France, Springer-Verlag (October 2011) 386–400
6. Peter Bourgon: Consistency without Consensus: CRDTs in Production at SoundCloud. URL http://www.infoq.com/presentations/crdt-soundcloud (Retrieved 22-dec-2015)
7. Todd Hoff: How League of Legends Scaled Chat to 70 Million Players - It akes a lot of Minions. URL http://highscalability.com/blog/2014/10/13 (Retrieved 22-dec-2015)
8. Michael Owen: Using Erlan, Riak and the ORSWOT CRDT at bet365. URL http://www.erlang-factory.com/euc2015/michael-owen
9. Letia, M., Preguiça, N., Shapiro, M.: CRDTs: Consistency without concurrency control. Rapp. Rech. RR-6956, INRIA, Rocquencourt, France (June 2009)
10. Baquero, C., Moura, F.: Using structural characteristics for autonomous operation. Operating Systems Review **33**(4) (1999) 90–96
11. Helland, P.: Idempotence is not a medical condition. Queue **10**(4) (April 2012) 30:30–30:46
12. Brown, R., Cribbs, S., Meiklejohn, C., Elliott, S.: Riak dt map: A composable, convergent replicated dictionary. In: Proceedings of the First Workshop on Principles and Practice of Eventual Consistency. PaPEC '14, New York, NY, USA, ACM (2014) 1:1–1:1
13. Almeida, P.S., Shoker, A., Baquero, C.: Efficient state-based crdts by delta-mutation. In: Networked Systems - Third International Conference, NETYS 2015, Agadir, Morocco, May 13-15, 2015. (2015)
14. Baquero, C.: Delta-enabled-crdts. URL http://github.com/CBaquero/delta-enabled-crdts (Retrieved 22-dec-2015)
15. Davey, B.A., Priestley, H.A.: Introduction to Lattices and Order (2. ed.). Cambridge University Press (2002)
16. Burckhardt, S., Gotsman, A., Yang, H., Zawirski, M.: Replicated data types: specification, verification, optimality. In Jagannathan, S., Sewell, P., eds.: POPL, ACM (2014) 271–284
17. Basho: Riak datatypes. URL http://github.com/basho (Retrieved 22-dec-2015)
18. Kemme, B., Schiper, A., Ramalingam, G., Shapiro, M.: Dagstuhl seminar review: Consistency in distributed systems. SIGACT News **45**(1) (March 2014) 67–89
19. Zeller, P., Bieniusa, A., Poetzsch-Heffter, A.: Formal specification and verification of crdts. In Ábrahám, E., Palamidessi, C., eds.: Formal Techniques for Distributed Objects, Components, and Systems - 34th IFIP WG 6.1 International Conference, FORTE 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014. Proceedings. Volume 8461 of Lecture Notes in Computer Science., Springer (2014) 33–48
20. Kyle Kingsbury: The trouble with timestamps. URL https://aphyr.com/posts/299-the-trouble-with-timestamps
21. Basho: Riak 1.4. URL https://github.com/basho/riak/blob/1.4/RELEASE-NOTES.md (Retrieved 4-jan-2016)
22. Datatax: Whats New in Cassandra 2.1: Better Implementation of Counters. URL http://www.datastax.com/dev/blog/whats-new-in-cassandra-2-1-a-better-implementation-of-counter (Retrieved 4-jan-2016)
23. Bieniusa, A., Zawirski, M., Preguiça, N., Shapiro, M., Baquero, C., Balegas, V., Duarte, S.: An optimized conflict-free replicated set. Rapp. Rech. RR-8083, INRIA, Rocquencourt, France (October 2012)

24. Parker, D.S., Popek, G.J., Rudisin, G., Stoughton, A., Walker, B.J., Walton, E., Chow, J.M., Edwards, D., Kiser, S., Kline, C.: Detection of mutual inconsistency in distributed systems. IEEE Trans. Softw. Eng. **9**(3) (May 1983) 240–247
25. Mukund, M., R., G.S., Suresh, S.P.: Optimized or-sets without ordering constraints. In: Proceedings ot the International Conference on Distributed Computing and Networking, New York, NY, USA, ACM (2014) 227241
26. Almeida, P.S., Baquero, C., Gonçalves, R., Preguiça, N.M., Fonte, V.: Scalable and accurate causality tracking for eventually consistent stores. In Magoutis, K., Pietzuch, P., eds.: 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. Volume 8460 of Lecture Notes in Computer Science., Springer (2014) 67–81
27. Wuu, G.T.J., Bernstein, A.J.: Efficient solutions to the replicated log and dictionary problems. In: Symp. on Principles of Dist. Comp. (PODC), Vancouver, BC, Canada (August 1984) 233–242
28. Johnson, P.R., Thomas, R.H.: The maintenance of duplicate databases. Internet Request for Comments RFC 677, Information Sciences Institute (January 1976)
29. Conway, N., Marczak, W.R., Alvaro, P., Hellerstein, J.M., Maier, D.: Logic and lattices for distributed programming. In: Proceedings of the Third ACM Symposium on Cloud Computing, ACM (2012) 1
30. Burckhardt, S., Fähndrich, M., Leijen, D., Wood, B.P.: Cloud types for eventual consistency. In: ECOOP 2012–Object-Oriented Programming. Springer (2012) 283–307
31. Kuper, L., Newton, R.R.: Lvars: lattice-based data structures for deterministic parallelism. In: Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing, ACM (2013) 71–84
32. Navalho, D., Duarte, S., Preguiça, N., Shapiro, M.: Incremental stream processing using computational conflict-free replicated data types. In: Proceedings of the 3rd International Workshop on Cloud Data and Platforms, ACM (2013) 31–36
33. Deftu, A., Griebsch, J.: A scalable conflict-free replicated set data type. In: Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems. ICDCS '13, Washington, DC, USA, IEEE Computer Society (2013) 186–195
34. Baquero, C., Almeida, P.S., Shoker, A.: Making operation-based CRDTs operation-based. In: Proceedings of Distributed Applications and Interoperable Systems: 14th IFIP WG 6.1 International Conference, Springer (2014)
35. Burckhardt, S., Leijen, D., Fahndrich, M.: Cloud types: Robust abstractions for replicated shared state. Technical Report MSR-TR-2014-43 (March 2014)
36. Malkhi, D., Terry, D.: Concise version vectors in winfs. Distributed Computing **20**(3) (2007) 209–219