Efficient Synchronization of State-based CRDTs

Vitor Enes
HASLab / INESC TEC and
Universidade do Minho
Portugal

Paulo Sérgio Almeida
HASLab / INESC TEC and
Universidade do Minho
Portugal

Carlos Baquero
HASLab / INESC TEC and
Universidade do Minho
Portugal

João Leitão
NOVA LINCS, FCT and
Universidade NOVA de Lisbon
Portugal

***Abstract*—To ensure high availability in large scale distributed systems, *Conflict-free Replicated Data Types* (CRDTs) relax consistency by allowing immediate query and update operations at the local replica, with no need for remote synchronization. State-based CRDTs synchronize replicas by periodically sending their full state to other replicas, which can become extremely costly as the CRDT state grows. Delta-based CRDTs address this problem by producing small incremental states (deltas) to be used in synchronization instead of the full state. However, current synchronization algorithms for delta-based CRDTs induce redundant wasteful delta propagation, performing worse than expected, and surprisingly, no better than state-based. In this paper we: 1) identify two sources of inefficiency in current synchronization algorithms for delta-based CRDTs; 2) bring the concept of join decomposition to state-based CRDTs; 3) exploit join decompositions to obtain optimal deltas and 4) improve the efficiency of synchronization algorithms; and finally, 5) experimentally evaluate the improved algorithms.**

***Keywords*-CRDTs; Optimal Deltas; Join Decomposition;**

I. Introduction

Large-scale distributed systems often resort to replication techniques to achieve fault-tolerance and load distribution. These systems have to make a choice between availability and low latency or strong consistency [1]–[4], many times opting for the first [5], [6]. A common approach is to allow replicas of some data type to temporarily diverge, making sure these replicas will eventually converge to the same state in a deterministic way. *Conflict-free Replicated Data Types* (CRDTs) [7], [8] can be used to achieve this. They are key components in modern geo-replicated systems, such as Riak [9], Redis [10], and Microsoft Azure Cosmos DB [11].

CRDTs come mainly in two flavors: *operation-based* and *state-based*. In both, queries and updates can be executed immediately at each replica, which ensures availability (as it never needs to coordinate beforehand with remote replicas to execute operations). In operation-based CRDTs [7], [12], operations are disseminated assuming a reliable dissemination layer that ensures exactly-once causal delivery of operations.

State-based CRDTs need fewer guarantees from the communication channel: messages can be dropped, duplicated, and reordered. When an update operation occurs, the local state is updated through a mutator, and from time to time (since we can disseminate the state at a lower rate than the rate of the updates) the full (local) state is propagated to other replicas.

Although state-based CRDTs can be disseminated over unreliable communication channels, as the state grows, sending the full state becomes unacceptably costly. Delta-based CRDTs [13], [14] address this issue by defining delta-mutators that return a delta (δ), typically much smaller than the full state of the replica, to be merged with the local state. The same δ is also added to an outbound δ-buffer, to be periodically propagated to remote replicas. Delta-based CRDTs have been adopted in industry as part of Akka Distributed Data framework [15] and IPFS [16], [17].

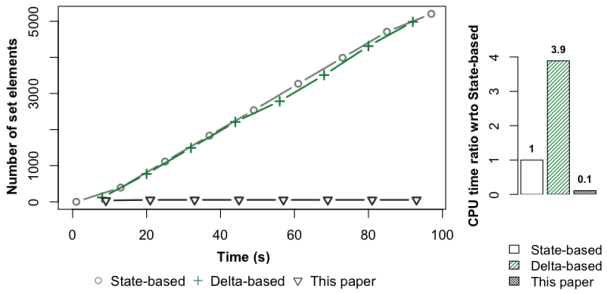


Fig. 1: Experiment setup: 15 nodes in a partial mesh topology replicating an always-growing set. The left plot depicts the number of elements being sent throughout the experiment, while the right plot shows the CPU processing time ratio with respect to state-based. Not only does delta-based synchronization not improve state-based in terms of state transmission, it even incurs a substantial processing overhead.

However, and somewhat unexpectedly, we have observed (Figure 1) that current delta-propagation algorithms can still disseminate much redundant state between replicas, performing worse than envisioned, and no better than the state-based approach. This anomaly becomes noticeable when concurrent update operations always occur between synchronization rounds, and it is partially justified due to inefficient redundancy detection in delta-propagation.

In this paper we identify two sources of redundancy in current algorithms, and introduce the concept of join decomposition of a state-based CRDT, showing how it can be used to derive optimal deltas ("differences") between states, as well as optimal delta-mutators. By exploiting these concepts, we also introduce an improved synchronization algorithm, and experimentally evaluate it, confirming that it outperforms current approaches by reducing the amount of state transmission, memory consumption, and processing time required for delta-based synchronization.

II. Background on State-based CRDTs

A state-based CRDT can be defined as a triple $(\mathcal{L}, \sqsubseteq, \sqcup)$ where $\mathcal{L}$ is a join-semilattice (lattice for short, from now on), $\sqsubseteq$ is a partial order, and $\sqcup$ is a binary join operator that derives the least upper bound for any two elements of $\mathcal{L}$. State-based CRDTs are updated through a set of *mutators* designed to be inflations, i.e. for mutator m and state $x \in \mathcal{L}$, we have $x \sqsubseteq \mathsf{m}(x)$.

Synchronization of replicas is achieved by having each replica periodically propagate its local state to other neighbour replicas. When a remote state is received, a replica updates its state to reflect the join of its local state and the received state. As the local state grows, more state needs to be sent, which might affect the usage of system resources (such as network) with a negative impact on the overall system performance. Ideally, each replica should only propagate the most recent modifications executed over its local state.

Delta-based CRDTs can be used to achieve this, by defining *delta-mutators* that return a smaller state which, when merged with the current state, generates the same result as applying the standard mutators, i.e. each mutator m has in delta-CRDTs a corresponding δ-mutator m^δ such that:

$$\mathsf{m}(x) = x \sqcup \mathsf{m}^\delta(x)$$

In this model, the deltas resulting from δ-mutators are added to a δ-buffer, in order to be propagated to neighbor replicas, as a δ-group, at the next synchronization step. When a δ-group is received from a neighbor, it is also added to the buffer for further propagation.

A. CRDT examples

In Figure 2 we present the specification of two simple state-based CRDTs, defining their lattice states, mutators, corresponding δ-mutators, and the binary join operator $\sqcup$. These lattices are typically bounded and thus a bottom value $\bot$ is also defined. (Note that the specifications do not define the partial order $\sqsubseteq$ since it can always be defined, for any lattice $\mathcal{L}$, in terms of $\sqcup$: $x \sqsubseteq y \Leftrightarrow x \sqcup y = y$.)

A CRDT counter that only allows increments is known as a *grow-only counter* (Figure 2a). In this data type, the set of replica identifiers $\mathbb{I}$ is mapped to the set of natural numbers $\mathbb{N}$. Increments are tracked per replica i, individually, and stored in a map entry $p(i)$. The value of the counter is the sum of each entry's value in the map. Mutator inc returns the updated map (the notation $p\{k \mapsto v\}$ indicates that only entry k in the map p is updated to a new value v, the remaining entries left unchanged), while the δ-mutator inc^δ only returns the updated entry. The join of two GCounters computes, for each key, the maximum of the associated values.

The lattice state evolution (either by mutation or join of two states) can also be understood by looking at the corresponding Hasse diagram (Figure 3). For example, state $\{\mathsf{A}_1, \mathsf{B}_1\}$ in Figure 3a (where A_1 represents entry $\{\mathsf{A} \mapsto 1\}$ in the map, i.e. one increment registered by replica A), can result from an increment on $\{\mathsf{A}_1\}$ by B, from an increment on $\{\mathsf{B}_1\}$ by A, or from the join of these two states.

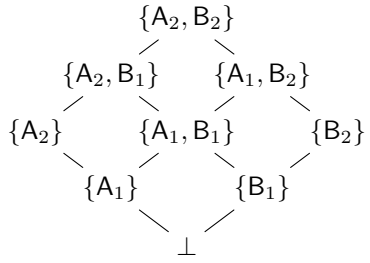
$$
\begin{aligned}
\mathsf{GCounter} &= \mathbb{I} \hookrightarrow \mathbb{N} \\
\bot &= \varnothing \\
\mathsf{inc}_i(p) &= p\{i \mapsto p(i) + 1\} \\
\mathsf{inc}_i^\delta(p) &= \{i \mapsto p(i) + 1\} \\
\mathsf{value}(p) &= \sum \{v \mid k \mapsto v \in p\} \\
p \sqcup p' &= \{k \mapsto \mathsf{max}(p(k), p'(k)) \mid k \in l\} \\
&\textbf{where } l = \mathsf{dom}(p) \cup \mathsf{dom}(p')
\end{aligned}
$$

(a) Grow-only Counter.

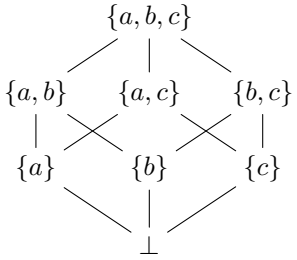
$$
\begin{aligned}
\mathsf{GSet}\langle E \rangle &= \mathcal{P}(E) \\
\bot &= \varnothing \\
\mathsf{add}(e, s) &= s \cup \{e\} \\
\mathsf{add}^\delta(e, s) &= \begin{cases} \{e\} & \textbf{if } e \notin s \\ \bot & \textbf{otherwise} \end{cases} \\
\mathsf{value}(s) &= s \\
s \sqcup s' &= s \cup s'
\end{aligned}
$$

(b) Grow-only Set.

Fig. 2: Specifications of two data types, replica $i \in \mathbb{I}$.



(a) GCounter, with two replicas $\mathbb{I} = \{\mathsf{A}, \mathsf{B}\}$.



(b) $\mathsf{GSet}\langle\{a, b, c\}\rangle$.

Fig. 3: Hasse diagram of two data types.

A *grow-only set*, Figures 2b and Figure 3b, is a set data type that only allows element additions. Mutator add returns the updated set, while add^δ returns a singleton set with the added element (in case it was not in the set already). The join of two GSets simply computes the set union.

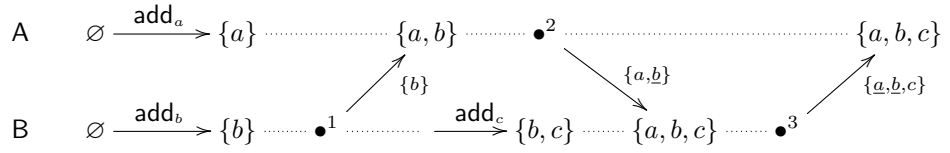


Fig. 4: Delta-based synchronization of a GSet with 2 replicas $\mathsf{A}, \mathsf{B} \in \mathbb{I}$. Underlined elements represent the BP optimization.

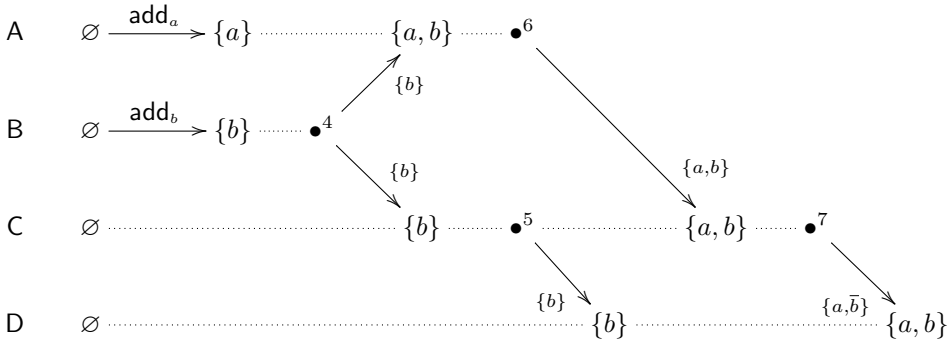


Fig. 5: Delta-based synchronization of a GSet with 4 replicas $\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D} \in \mathbb{I}$. The overlined element represents the RR optimization.

Although we have chosen as running examples very simple CRDTs, the results in this paper can be extended to more complex ones, as we show in Appendix B. For further coverage of delta-based CRDTs see [14].

B. Synchronization Cost Problem

Figures 4 and 5 illustrate possible distributed executions of the classic delta-based synchronization algorithm [14], with replicas of a *grow-only-set*, all starting with a bottom value $\bot = \varnothing$. (This classic algorithm is captured in Algorithm 1, covered in Section IV.) Synchronization with neighbors is represented by $\bullet$ and synchronization arrows are labeled with the state sent, where we overline or underline elements that are being redundantly sent and can be removed (thus improving network bandwidth consumption) by employing two simple and novel optimizations that we introduce next.

In Figure 4, we have two replicas $\mathsf{A}, \mathsf{B} \in \mathbb{I}$ and each adds an element to the replicated set. At $\bullet^1$, B propagates the content of the δ-buffer, i.e. $\{b\}$, to neighbour A. At $\bullet^2$, A sends to B $\{a, b\}$, i.e. the join of $\{a\}$ from a local mutation, and the received $\{b\}$ from B, even though $\{b\}$ came from B itself. By simply tracking the origin of each δ-group in the δ-buffer, replicas can **avoid back-propagation of δ-groups** (BP).

Before receiving $\{a, b\}$, B adds a new element c to the set, also adding $\{c\}$ to the δ-buffer. Upon receiving $\{a, b\}$, and since what was received produces changes in the local state, B adds it to the δ-buffer. At $\bullet^3$, B propagates all new changes since last synchronization with A: $\{c\}$ from a local mutation, and $\{a, b\}$ from B, even though $\{a, b\}$ came from replica B. When A receives $\{a, b, c\}$, it will also add it to the buffer to be further propagated. Note that as long as this pattern keeps repeating (i.e. there's always a state change between synchronizations), delta-based synchronization will propagate the same amount of state as state-based synchronization would, representing no improvement. This is illustrated in Figure 4, and demonstrated empirically in Section V.

In Figure 5, we have four replicas $\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D} \in \mathbb{I}$, and replicas A, B add an element to the set. At $\bullet^4$, B propagates the content of the δ-buffer to neighbours A and C. At $\bullet^5$, C propagates the received $\{b\}$ to D. At $\bullet^6$, A sends the join of $\{a\}$ from a local mutation and the received $\{b\}$ to C. Upon receiving the δ-group $\{a, b\}$, C adds it to the δ-buffer and sends it to D at $\bullet^7$. However, part of this δ-group has already been in the δ-buffer (namely b), and thus, has already been propagated. This observation hints for another optimization: **remove redundant state in received δ-groups** (RR), before adding them to the δ-buffer.

Both BP and RR optimizations are detailed in Section IV, where we incorporate them into the delta-based synchronization algorithm with few changes.

III. Join Decompositions and Optimal Deltas

In this section we introduce state decomposition in state-based CRDTs, by exploiting the mathematical concept of *irredundant join decompositions* in lattices. We then demonstrate how this concept can be used to derive deltas and delta-mutators that are optimal, in the sense that they produce the smallest delta-state possible. In Section IV we show how this same concept plays a key role in the RR optimization briefly described in the previous section.

A. Join Decomposition of a State-based CRDT

Definition 1 (Join-irreducible state)**.** *State $x \in \mathcal{L}$ is join-irreducible if it cannot result from the join of any finite set of states $F \subseteq \mathcal{L}$ not containing x:*

$$x = \bigsqcup F \Rightarrow x \in F$$

Example 1. *Let the following p_1, p_2 and p_3 be* GCounter *states, and s_1, s_2 and s_3 be* GSet *states.*

✓ $p_1 = \{\mathsf{A}_5\}$ ✗ $s_1 = \bot$
✓ $p_2 = \{\mathsf{B}_6\}$ ✓ $s_2 = \{a\}$
✗ $p_3 = \{\mathsf{A}_5, \mathsf{B}_7\}$ ✗ $s_3 = \{a, b\}$

States p_3 and s_3 are not join-irreducible states, since they can be decomposed into (i.e. result from the join of) two states different from themselves: $\{\mathsf{A}_5\}$ and $\{\mathsf{B}_7\}$ for p_3, $\{a\}$ and $\{b\}$ for s_3. Bottom (e.g., s_1) is never join-irreducible, as it is the join over an empty set $\bigsqcup \varnothing$.

In a Hasse diagram of a finite lattice (e.g., in Figure 3) the join-irreducibles are those elements with exactly one link below. Given lattice $\mathcal{L}$, we use $\mathcal{J}(\mathcal{L})$ for the set of all join-irreducible elements of $\mathcal{L}$.

Definition 2 (Join Decomposition)**.** *Given a lattice state $x \in \mathcal{L}$, a set of join-irreducibles D is a join decomposition [18] of x if its join produces x:*

$$D \subseteq \mathcal{J}(\mathcal{L}) \wedge \bigsqcup D = x$$

Definition 3 (Irredundant Join Decomposition)**.** *A join decomposition D is irredundant if no element in it is redundant:*

$$D' \subset D \Rightarrow \bigsqcup D' \sqsubset \bigsqcup D$$

Example 2. *Let $p = \{\mathsf{A}_5, \mathsf{B}_7\}$ be a* GCounter *state, $s = \{a, b, c\}$ a* GSet *state, and consider the following sets of states as tentative decompositions of p and s.*

✗ $P_1 = \{\{\mathsf{A}_5\}, \{\mathsf{B}_6\}\}$ ✗ $S_1 = \{\{b\}, \{c\}\}$
✗ $P_2 = \{\{\mathsf{A}_5\}, \{\mathsf{B}_6\}, \{\mathsf{B}_7\}\}$ ✗ $S_2 = \{\{a, b\}, \{b\}, \{c\}\}$
✗ $P_3 = \{\{\mathsf{A}_5, \mathsf{B}_6\}, \{\mathsf{B}_7\}\}$ ✗ $S_3 = \{\{a, b\}, \{c\}\}$
✓ $P_4 = \{\{\mathsf{A}_5\}, \{\mathsf{B}_7\}\}$ ✓ $S_4 = \{\{a\}, \{b\}, \{c\}\}$

Only P_4 and S_4 are irredundant join decompositions of p and s. P_1 and S_1 are not decompositions since their join does not result in p and s, respectively; P_2 and S_2 are decompositions but contain redundant elements, $\{\mathsf{B}_6\}$ and $\{b\}$, respectively; P_3 and S_3 do not have redundancy, but contain reducible elements (S_2 fails to be an irredundant join decomposition for the same reason, since its element $\{a, b\}$ is also reducible).

As we show in Appendix A and B, these irredundant decompositions exist, are unique, and can be obtained for CRDTs used in practice. Let $\Downarrow x$ denote the unique decomposition of element x. From the Birkhoff's Representation Theorem [19], decomposition $\Downarrow x$ is given by the maximals of the join-irreducibles below x:

$$\Downarrow x = \mathsf{max}\{r \in \mathcal{J}(\mathcal{L}) \mid r \sqsubseteq x\}$$

As two examples, given a GCounter state p and a GSet state s, their (quite trivial) irredundant decomposition is given by:

$$\Downarrow p = \{\{k \mapsto v\} \mid k \mapsto v \in p\} \qquad \Downarrow s = \{\{e\} \mid e \in s\}$$

We argue that these techniques can be applied to most (practical) implementations of CRDTs found in industry. The interested reader can find generic decomposition rules in Appendix C.

B. Optimal deltas and δ-mutators

Having a unique irredundant join decomposition, we can define a function which gives the minimum delta, or "difference" in analogy to set difference, between two states $a, b \in \mathcal{L}$:

$$\Delta(a, b) = \bigsqcup \{y \in \Downarrow a \mid y \not\sqsubseteq b\}$$

which when joined with b gives $a \sqcup b$, i.e. $\Delta(a, b) \sqcup b = a \sqcup b$. It is minimum (and thus, optimal) in the sense that it is smaller than any other c which produces the same result: $c \sqcup b = a \sqcup b \Rightarrow \Delta(a, b) \sqsubseteq c$.

If not carefully designed, δ-mutators can be a source of redundancy when the resulting δ-state contains information that has already been incorporated in the lattice state. As an example, the original δ-mutator add^δ of GSet presented in [13] always returns a singleton set with the element to be added, even if the element is already in the set (in Figure 2b we have presented a definition of add^δ that is optimal). By resorting to function Δ, minimum delta-mutators can be trivially derived from a given mutator:

$$\mathsf{m}^\delta(x) = \Delta(\mathsf{m}(x), x)$$

IV. Revisiting Delta-based Synchronization

Algorithm 1 formally describes delta-based synchronization at replica i. The algorithm contains lines that belong to classic delta-based synchronization [13], [14], and lines with BP and RR optimizations, while non-highlighted lines belong to both. In classic delta-based synchronization, each replica i maintains a lattice state $x_i \in \mathcal{L}$ (**line** 4), and a δ-buffer $B_i \in \mathcal{P}(\mathcal{L})$ as a set of lattice states (**line** 5). When an update operation occurs (**line** 6), the resulting δ is merged with the local state x_i (**line** 19) and added to the buffer (**line** 20), resorting to function store. Periodically, the whole content of the δ-buffer (**line** 11) is propagated to neighbors (**line** 12).

For simplicity of presentation, we assume that communication channels between replicas cannot drop messages (reordering and duplication is considered), and that is why the buffer is cleared after each synchronization step (**line** 13). This assumption can be removed by simply tagging each entry in the δ-buffer with a unique sequence number, and by exchanging acks between replicas: once an entry has been acknowledged by every neighbour, it is removed from the δ-buffer, as originally proposed in [13].

When a δ-group is received (**line** 14), then it is checked whether it will induce an inflation in the local state (**line** 16). If this is the case, the δ-group is merged with the local state and added to the buffer (for further propagation), resorting to the same function store. The precondition in **line** 16 appears to be harmless, but it is in fact, the source of most redundant state propagated in this synchronization algorithm. Detecting an inflation is not enough, since almost always there's something new to incorporate. Instead, synchronization algorithms must extract from the received δ-group the lattice state responsible for the inflation, as done by the RR optimization.

Few changes are required in order to incorporate this and the BP optimization in the classic algorithm, as we show

```
1  inputs:
2    n_i ∈ P(I), set of neighbors
3  state:
4    x_i ∈ L, x_i^0 = ⊥
5    B_i ∈ P(L), B_i^0 = ∅                 B_i ∈ P(L × I), B_i^0 = ∅
6  on operation_i(m^δ)
7    δ = m^δ(x_i)
8    store(δ, i)
9  periodically // synchronize
10   for j ∈ n_i
11     d = ⨆ B_i                           d = ⨆{s | ⟨s, o⟩ ∈ B_i ∧ o ≠ j}
12     send_{i,j}(delta, d)
13   B'_i = ∅
14 on receive_{j,i}(delta, d)
15                                         d = Δ(d, x_i)
16   if d ⋢ x_i                            if d ≠ ⊥
17     store(d, j)
18 fun store(s, o)
19   x'_i = x_i ⊔ s
20   B'_i = B_i ∪ {s}                      B'_i = B_i ∪ {⟨s, o⟩}
```

Algorithm 1: Delta-based synchronization algorithms at replica $i \in \mathbb{I}$: classic version and version with BP and RR optimizations.

next. This happens because our approach encapsulates most of its complexity in the computation of join decompositions and function Δ. The fact that few changes are required to the classic synchronization algorithm is a benefit, that will minimize the efforts in incorporating these techniques in existing implementations.

Avoiding back-propagation of δ-groups: For BP, each entry in the δ-buffer is tagged with its origin (**line** 5 and **line** 20), and at each synchronization step with neighbour j, entries tagged with j are filtered out (**line** 11).

Removing redundant state in received δ-groups: A received δ-group can contain redundant state, i.e. state that has already been propagated to neighbors, or state that is in the δ-buffer B_i still to be propagated. This occurs in topologies where the underlying graph has cycles, and thus, nodes can receive the same information through different paths in the graph. In order to detect if a δ-group has redundant state, nodes do not need to keep everything in the δ-buffer or even inspect the δ-buffer: it is enough to compare the received δ-group with the local lattice state x_i. In classic delta-based synchronization, received δ-groups were added to δ-buffer only if they would strictly inflate the local state (**line** 16). For RR, we extract from the δ-group what strictly inflates the local state x_i (**line** 15), and store it if it is different from bottom (**line** 16). This extraction is achieved by selecting which irreducible states from the decomposition of the received δ-group strictly inflate the local state, resorting to function Δ presented in Section III.

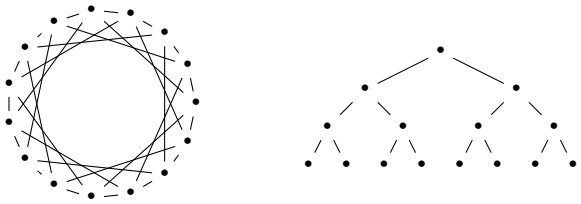


Fig. 6: Network topologies employed: a 15-node partial-mesh (to the left) and a 15-node tree (to the right).

V. Evaluation

In this Section we evaluate the proposed solutions and show the following:

- Classic delta-based synchronization can be as inefficient as state-based synchronization in terms of transmission bandwidth, while incurring an overhead in terms of memory usage required for synchronization (Section V-B).
- In acyclic topologies, BP is enough to attain the best results, while in topologies with cycles, only RR can greatly reduce the synchronization cost (Section V-B).
- Alternative synchronization techniques (such as Scuttlebutt [20] and operation-based synchronization [7], [8]) are metadata-heavy; this metadata represents a large fraction of all the data required for synchronization (over 75%) while for delta-based synchronization the metadata overhead can be as low as 7.7% (Section V-B).
- In moderate-to-high contention workloads, BP + RR can reduce transmission bandwidth and memory consumption by several GBs; when comparing with BP + RR, classic delta-based synchronization has an unnecessary CPU overhead of up-to 7.9× (Section V-C).

Instructions on how to reproduce all experiments can be found in our public repository[1].

A. Experimental Setup

The evaluation was conducted in a Kubernetes cluster deployed in Emulab [21]. Each machine has a Quad Core Intel Xeon 2.4 GHz and 12GB of RAM. The number of machines in the cluster is set such that two replicas are never scheduled to run in the same machine, i.e. there is at least one machine available for each replica in the experiment.

Network Topologies: Figure 6 depicts the two network topologies employed in the experiments: a partial-mesh, in which each node has 4 neighbors; and a tree, with 3 neighbors per node, with the exception of the root node (2 neighbors) and leaf nodes (1 neighbor). The first topology exhibits redundancy in the links and tests the effect of cycles in the synchronization, while the second represents an optimal propagation scenario over a spanning tree.

B. Micro-Benchmarks

We have designed a set of micro-benchmarks, in which each node periodically (every second) synchronizes with neighbors and executes an update operation over a CRDT. The

[1] https://github.com/vitorenesduarte/exp

TABLE I: Description of micro-benchmarks.

Type	Periodic event	Measurement
GCounter	single increment	number of entries in the map
GSet	addition of unique element	number of elements in the set
GMap K%	change the value of $\frac{K}{N}\%$ keys	number of entries in the map

update operation depends on the CRDT type. In GSet, the update event is the addition of a globally unique element to the set; in GCounter, an increment on the counter; and in GMap K% each node updates $\frac{K}{N}\%$ keys (N being the number of nodes/replicas), such that globally K% of all the keys in the *grow-only map* are modified within each synchronization interval. Note how the GCounter benchmark is a particular case of GMap K%, in which $K = 100$. For GMap K% we set the total number of keys to 1000, and for all benchmarks, the number of events per replica is set to 100.

These micro-benchmarks are summarized in Table I, along with the metric (to be used in transmission and memory measurements) we have defined: for GCounter and GMap K% we count the number of map entries, while for GSet, the number of set elements. We setup this part of the evaluation with 15-node topologies (as in Figure 6). As baselines, we have state-based synchronization, classic delta-based synchronization, Scuttlebutt, a variation of Scuttlebutt, and operation-based synchronization.

Scuttlebutt: Scuttlebutt [20] is an anti-entropy protocol used to reconcile changes in values of a key-value store. Each value is uniquely identified with a version $\langle i, s\rangle \in \mathbb{I} \times \mathbb{N}$, where the first component $i \in \mathbb{I}$ is the identifier of the replica responsible for the new value, and $s \in \mathbb{N}$ a sequence number, incremented on each local update, thus being unique. With this, the updates known locally can be summarized by a vector $\mathbb{I} \hookrightarrow \mathbb{N}$, mapping each replica to the highest sequence number it knows. When a node wants to reconcile with a neighbor replica, it sends the summary vector, and the neighbor replies with all the key-value pairs it has locally that have versions not summarized in the received vector. This strategy is performed in both directions, and in the end, both replicas have the same key-value pairs in their local key-value store (assuming no new updates occurred).

Scuttlebutt can be used to synchronize state-based CRDTs with few modifications. Using as values the CRDT state would be inefficient, since changes to the CRDT wouldn't be propagated incrementally, i.e. a small change in the CRDT would require sending the whole new state, as in state-based synchronization. Therefore, we use as values the optimal deltas resulting from δ-mutators. As keys, we can simply resort to the version pairs. When reconciling two replicas, a replica receiving new key-delta pairs, merges all the deltas with the local CRDT. If CRDT updates stop, eventually all replicas converge to the same CRDT state. We label this approach `Scuttlebutt`.

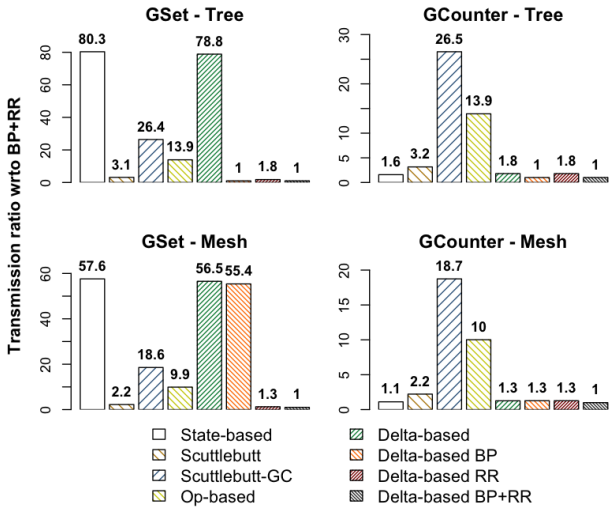


Fig. 7: Transmission of GSet and GCounter with respect to delta-based BP + RR – tree and mesh topologies.

This strategy is potentially inefficient in terms of memory: a replica has to keep in the Scuttlebutt key-value store *all* the deltas it has ever seen, since a neighbor replica can at any point in time send a summary vector asking for *any* delta. Since the original Scuttlebutt algorithm does not support deleting keys from the key-value store, we add support for *safe* deletes of deltas, in order to reduce its memory footprint. If each node keeps track of what each node in the system has seen (in a map $\mathbb{I} \hookrightarrow (\mathbb{I} \hookrightarrow \mathbb{N})$ from replica identifiers to the last seen summary vector), once a delta has been seen by all nodes, it can be safely deleted from the local Scuttlebutt store. We compare with this improved Scuttlebutt variant (labeled `Scuttlebutt-GC`) that allows nodes to only be connected to a subset of all nodes, not requiring all-to-all connectivity, while supporting safe deletes. For completeness, we also compare with the original Scuttlebutt design that is unable to garbage-collect unnecessary key-delta pairs.

Operation-based: Operation-based CRDTs [7], [8] resort to a causal broadcast middleware [22] that is used to disseminate CRDT operations. This middleware tags each operation with a vector clock that summarizes the causal past of the operation. Such vector is then used by the recipient to ensure causal delivery of operations, i.e. each operation is only delivered when every operation in its causal past has been delivered as well.

In topologies with all-to-all connectivity, each node is only responsible for disseminating its own operations. In order to relax this requirement, we have implemented a middleware that *stores-and-forwards* operations: when an operation is seen for the first time, it is added to a transmission buffer to be further propagated in the next synchronization step; if the same operation is received from different incoming neighbors, the middleware simply updates which nodes have seen this

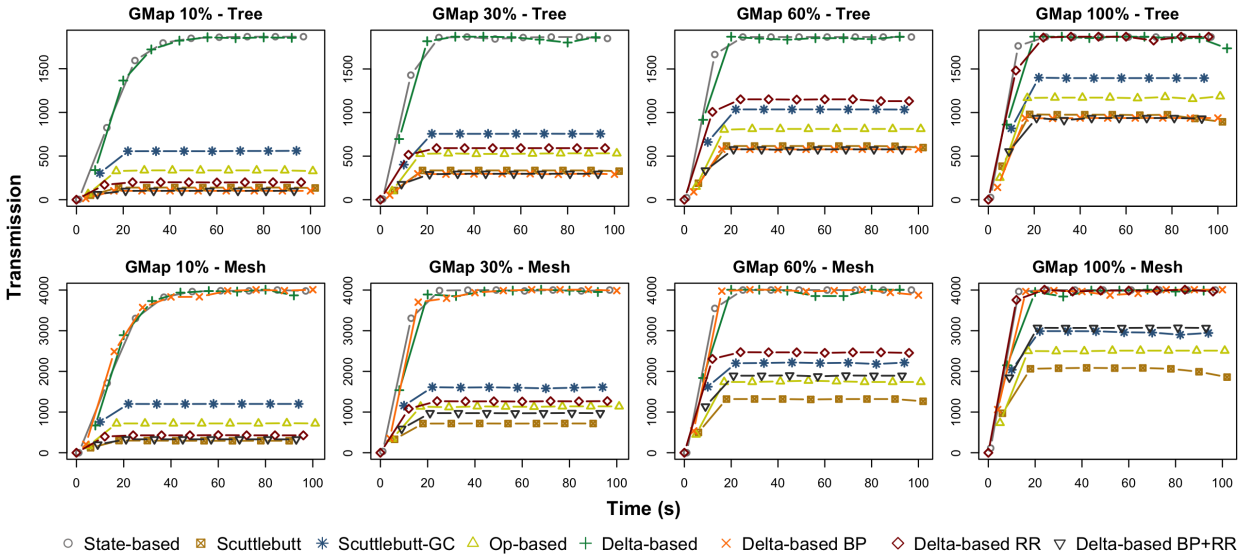


Fig. 8: Transmission of GMap 10%, 30%, 60% and 100% – tree and mesh topologies.

operation so that unnecessary transmissions are avoided. To the best of our knowledge, this is the best possible implementation of such a middleware. We label this approach `Op-based`.

1) Transmission bandwidth: Figure 7 shows, for GSet and GCounter, the transmission ratio (of all synchronization mechanisms previously mentioned) with respect to delta-based synchronization with BP and RR optimizations enabled. The first observation is that classic delta-based synchronization presents almost no improvement, when compared to state-based synchronization. In the tree topology, BP is enough to attain the best result, because the underlying topology does not have cycles, and thus, BP is sufficient to prevent redundant state to be propagated. With a partial-mesh, BP has little effect, and RR contributes most to the overall improvement. Given that the underlying topology leads to redundant communication (desired for fault-tolerance), and classic delta-based can never extract that redundancy, its transmission bandwidth is effectively similar to that of state-based synchronization.

Scuttlebutt and Scuttlebutt-GC are more efficient than classic delta-based for GSet since both can precisely identify state changes between synchronization rounds. However, the results for GCounter reveal a limitation of this approach. Since Scuttlebutt treats propagated values as opaque, and does not understand that the changes in a GCounter compress naturally under lattice joins (only the highest sequence for each replica needs to be kept), it effectively behaves worse than state-based and classic delta-based in this case. Operation-based synchronization follows the *same trend* for the *same reason*: it improves state-based and classic delta-based for GSet but not for GCounter since the middleware is unable to compress multiple operations into a single, equivalent, operation. Supporting generic operation-compression at the middleware level in operation-based CRDTs is an open research problem. The difference between these three approaches is related with the metadata cost associated to each, as we show in Section V-B2.

Even with the optimizations BP + RR proposed, the best result for GCounter is not much better than state-based. This is expected since most entries of the underlying map are being updated between each synchronization step: each node has almost always something new from every other node in the system to propagate (thus being similar to state-based in some cases). This pattern represents a special case of a map in which 100% of its keys are updated between state synchronizations.

In Figure 8 we study other update patterns, by measuring the transmission of GMap 10%, 30%, 60%, and 100%. These results are further evidence of what we have observed in the case of GSet: BP suffices if the network graph is acyclic, but RR is crucial in the more general case.

As seen previously, Scuttlebutt and Scuttlebutt-GC behave much better than state-based synchronization, yielding a reduction in the transmission cost between 46% and 91%, and 20% and 65%, respectively. This is due to the underlying precise reconciliation mechanism of Scuttlebutt. Operation-based synchronization leads to a transmission reduction between 35% and 80% since it is able to represent incremental changes to the CRDT as small operations. Finally, delta-based BP + RR is able reduce the transmission costs by up-to 94%.

In the extreme case of GMap 100% (every key in the map is updated between synchronization rounds, which is a less likely workload in practical systems) and considering a partial-mesh, delta-based BP + RR provides a modest improvement in relation to state-based of about 18% less transmission, and its performance is below Scuttlebutt variants and operation-based synchronization.

Vector-based protocols (Scuttlebutt and operation-based) however, have an inherent scalability problem. When increasing the number of nodes in the system, the transmission costs may become dominated by the size of metadata required for

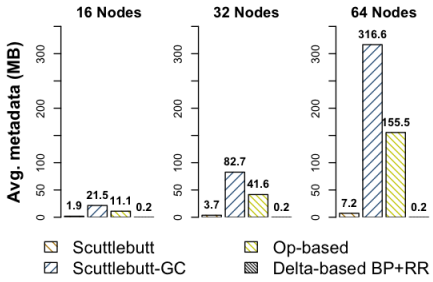


Fig. 9: Metadata required per node when synchronizing a GSet in a mesh topology. Each node has 4 neighbours (as in Figure 6) and each node identifier has size 20B.

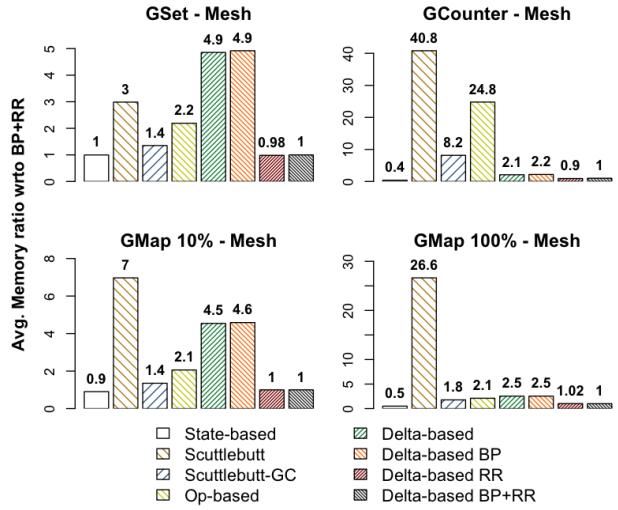


Fig. 10: Average memory ratio with respect to BP + RR for GCounter, GSet, GMap 10% and 100% – mesh topology

synchronization, as we show next.

2) Metadata Cost: Figure 9 shows the size of metadata required for synchronization per node while varying the total number of replicas (i.e. nodes). The results show a linear and quadratic cost (in terms of number of nodes) for Scuttlebutt and Scuttlebutt-GC (respectively), and a linear cost for operation-based synchronization (in terms of both number of nodes and pending updates still to be propagated). Given N nodes, P neighbors, and U pending updates, the metadata cost per node is:

- Scuttlebutt: NP (a vector per neighbor)
- Scuttlebutt-GC: N^2P (a map of vectors per neighbor)
- Operation-based: NPU (a vector per neighbor per pending update)
- Delta-based: P (a sequence number per neighbor)

This cost may represent a large fraction of all data propagated during synchronization. For example, in our measurements with 32 nodes, this metadata represents 75%, 99%, and 97% of the transmission costs for Scuttlebutt, Scuttlebutt-GC and operation-based, respectively, while the overhead of delta-based synchronization is only 7.7%.

TABLE II: Retwis workload characterization: for each operation, the number of CRDT updates performed and its workload percentage.

Operation	#Updates	Workload %
Follow	1	15%
Post Tweet	1 + #Followers	35%
Timeline	0	50%

3) Memory footprint: In delta-based synchronization, the size of δ-groups being propagated not only affects the network bandwidth consumption, but also the memory required to store them in the δ-buffer for further propagation. During the experiments, we periodically measure the amount of state (both CRDT state and metadata required for synchronization) stored in memory for each node.

Figure 10 reports the average memory ratio with respect to BP + RR. State-based does not require synchronization metadata, and thus it is optimal in terms of memory usage. Classic delta-based and delta-based BP have an overhead of 1.1x-3.9x since the size of δ-groups in the δ-buffer is larger for these techniques. For GSet and GMap 10%, Scuttlebutt-GC is close to BP + RR since deltas are removed from the key-value store as soon as they are seen by all replicas. Key-delta pairs are never pruned in the original Scuttlebutt, leading to an increasing memory usage. As long as new updates exist, the memory consumption for Scuttlebutt can only deteriorate, ultimately to a point where it will disrupt the system operation. Operation-based has a higher memory cost than Scuttlebutt-GC, since each operation in the transmission buffer is tagged with a vector, while in Scuttlebutt and Scuttlebutt-GC each delta is simply tagged with a version pair.

Considering the results for GCounter, the three vector-based algorithms exhibit the highest memory consumption. This is justified by the same reason they perform poorly in terms of transmission bandwidth in this case (Figure 7): these protocols are unable to compress incremental changes. Overall, and ignoring state-based which doesn't present any metadata memory costs, BP + RR attains the best results.

C. Retwis Application

We now compare classic delta-based with delta-based BP + RR using Retwis [23], a popular [24]–[26] open-source Twitter clone. In Table II we describe the application workload, similar to the one used in [24]: user a can follow user b by updating the set of followers of user b; users can post a new tweet, by writing it in their wall and in the timeline of all their followers; and finally, users can read their timeline, fetching the 10 most recent tweets.

Each user has 3 objects associated with it: 1) a set of followers stored in a GSet; 2) a wall stored in a GMap mapping

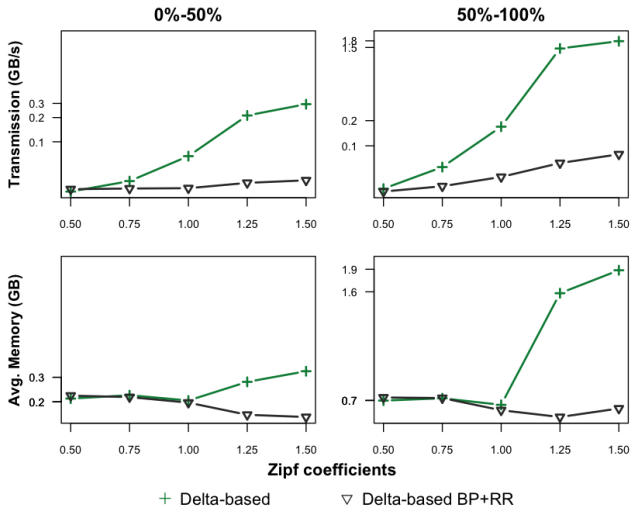


Fig. 11: Transmission bandwidth per node (top) and average memory per node (bottom) of classic delta-based and BP + RR for different Zipf coefficient values (log scale). The left and right side show these values for the first and second half of the experiment (respectively).

tweet identifiers to their content; and 3) a timeline stored in a GMap mapping tweet timestamps to tweet identifiers. We run this benchmark with 10K users, and thus, 30K CRDT objects overall. The size of tweet identifiers and content is 31B and 270B, respectively. These sizes are representative of real workloads, as shown in an analysis of Facebook's general-purpose key-value store [27]. The topology is a partial-mesh, with 50 nodes, each with 4 neighbors, as in Figure 6, and updates on objects follow a Zipf distribution, with coefficients ranging from 0.5 (low contention) to 1.5 (high contention) [24].

Figure 11 shows the transmission bandwidth and memory footprint of both algorithms, for different Zipf coefficient values. We can observe that in low contention workloads, classic delta-based behaves almost optimally when compared to BP + RR. Since updates are distributed almost evenly across all objects, there are few concurrent updates to the same object between synchronization rounds, and thus, the simple and naive inflation check in **line** 16 suffices. This phenomena was not observed in the previous set of benchmarks, since we had a single object, and thus, maximum contention.

As we increase contention, a more sophisticated approach like BP + RR is required, in order to avoid redundant state propagation. For example, with a 1.25 coefficient, bandwidth is reduced from 1.46GB/s to 0.06GB/s per node, and memory footprint per node drops from 1.58GB to 0.62GB (right side of the plots). Also, as we increase the Zipf coefficient, we note that the bandwidth consumption continues to rise, leading to an unsustainable situation in the case of classic delta-based, as it can never reduce the size of δ-groups being transmitted.

During the experiment we also measured the CPU time

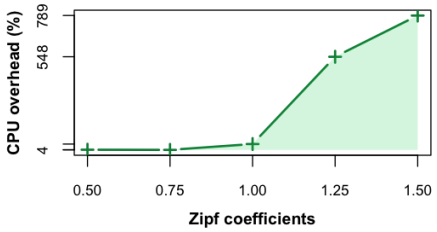


Fig. 12: CPU overhead of classic delta-based when compared to delta-based BP + RR.

spent in processing CRDT updates, both producing and processing synchronization messages. Figure 12 reports the CPU overhead of classic delta-based, when considering BP + RR as baseline. Since classic delta-based produces/processes larger messages than BP + RR, this results in a higher CPU cost: for the 1, 1.25 and 1.5 Zipf coefficients, classic delta-based incurs an overhead of 0.4x, 5.5x, and 7.9x respectively.

VI. Related Work

In the context of remote file synchronization, *rsync* [28] synchronizes two files placed on different machines, by generating file block signatures, and using these signatures to identify the missing blocks on the backup file. In this strategy, there's a trade-off between the size of the blocks to be signed, the number of signatures to be sent, and the size of the blocks to be received: bigger blocks to be signed implies fewer signatures to be sent, but the blocks received (deltas) can be bigger than necessary. Inspired by *rsync*, *Xdelta* [29] computes a difference between two files, taking advantage of the fact that both files are present. Consequently the cost of sending signatures can be ignored and the produced deltas are optimized.

In [30], we propose two techniques that can be used to synchronize two state-based CRDTs after a network partition, avoiding bidirectional full state transmission. Let A and B be two replicas. In *state-driven* synchronization, A starts by sending its local lattice state to B, and given this state, B is able to compute a delta that reflects the updates missed by A. In *digest-driven* synchronization, A starts by sending a digest (signature) of its local state (smaller than the local state), that still allows B to compute the delta. B then sends the computed delta along with a digest of its local state, allowing A to compute a delta for B. Convergence is achieved after 2 and 3 messages in *state-driven* and *digest-driven*, respectively. These two techniques also exploit the concept of join decomposition presented in this paper.

Similarly to *digest-driven* synchronization, Δ-CRDTs [31] exchange metadata used to compute a delta that reflects missing updates. In this approach, CRDTs need to be extended to maintain additional metadata for delta derivation, and if this metadata needs to be garbage collected, the mechanism falls-back to standard bidirectional full state transmission.

In the context of anti-entropy gossip protocols, *Scuttlebutt* [20] proposes a *push-pull* algorithm to be used to synchronize

a set of values between participants, but considers each value as opaque, and does not try to represent recent changes to these values as deltas. Other solutions try to minimize the communication overhead of anti-entropy gossip-based protocols by exploiting either hash functions [32] or a combination of Bloom filters, Merkle trees, and Patricia tries [33]. Still, these solutions require a significant number of message exchanges to identify the source of divergence between the state of two processes. Additionally, these solutions might incur significant processing overhead due to the need of computing hash functions and manipulating complex data structures, such as Merkle trees.

With the exception of *Xdelta*, all these techniques do not assume knowledge prior to synchronization, and thus delay reconciliation, by always exchanging state digests in order to detect state divergence.

VII. Conclusion

Under geo-replication there is a significant availability and latency impact [1] when aiming for strong consistency criteria such as linearizability [34]. Strong consistency guarantees greatly simplify the programmers view of the system and are still required for operations that do demand global synchronization. However, several other system's components do not need that same level of coordination and can reap the benefits of fast local operation and strong eventual consistency. This requires capturing more information on each data type semantics, since a read/write abstraction becomes limiting for the purpose of data reconciliation. CRDTs can provide a sound approach to these highly available solutions and support the existing industry solutions for geo-replication, which are still mostly grounded on state-based CRDTs.

State-based CRDT solutions quickly become prohibitive in practice, if there is no support for treatment of small incremental state deltas. In this paper we advance the foundations of state-based CRDTs by introducing minimal deltas that precisely track state changes. We also present and micro-benchmark two optimizations, *avoid back-propagation of δ-groups* and *remove redundant state in received δ-groups*, that solve inefficiencies in classic delta-based synchronization algorithms. Further evaluation shows the improvement our solution can bring to a small scale Twitter clone deployed in a 50-node cluster, a relevant application scenario.

Acknowledgments

We would like to thank Ricardo Macedo, Georges Younes, Marc Shapiro and the anonymous reviewers for their valuable feedback on earlier drafts of this work. Vitor Enes was supported by EU H2020 LightKone project (732505) and by a FCT - Fundação para a Ciência e a Tecnologia - PhD Fellowship (PD/BD/142927/2018). Carlos Baquero was partially supported by SMILES within TEC4Growth project (NORTE-01-0145-FEDER-000020). João Leitão was partially supported by project NG-STORAGE through FCT grant PTDC/CCI-INF/32038/2017, and by NOVA LINCS through the FCT grant UID/CEC/04516/2013.

References

[1] D. Abadi, "Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story," in *Computer*, 2012.
[2] E. Brewer, "A Certain Freedom: Thoughts on the CAP Theorem," in *PODC*, 2010.
[3] S. Gilbert and N. Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services," in *SIGACT News*, 2002.
[4] W. Golab, "Proving PACELC," in *SIGACT News*, 2018.
[5] P. Ajoux, N. Bronson, S. Kumar, W. Lloyd, and K. Veeraraghavan, "Challenges to Adopting Stronger Consistency at Scale," in *HOTOS*, 2015.
[6] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd, "Existential Consistency: Measuring and Understanding Consistency at Facebook," in *SOSP*, 2015.
[7] M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski, "Conflict-Free Replicated Data Types," in *SSS*, 2011.
[8] M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski, "Convergent and Commutative Replicated Data Types," in *Bulletin of the EATCS*, 2011.
[9] Basho, "Riak KV Concepts: Data Types." [Online]. Available: http://docs.basho.com/riak/kv/2.2.3/learn/concepts/crdts/
[10] R. Labs, "Under the Hood: Redis CRDTs." [Online]. Available: https://redislabs.com/docs/active-active-whitepaper/
[11] M. Azure, "Multi-master at global scale with Azure Cosmos DB." [Online]. Available: https://docs.microsoft.com/en-us/azure/cosmos-db/multi-region-writers
[12] C. Baquero, P. S. Almeida, and A. Shoker, "Pure Operation-Based Replicated Data Types," *CoRR*, 2017. [Online]. Available: http://arxiv.org/abs/1710.04469
[13] P. S. Almeida, A. Shoker, and C. Baquero, "Efficient State-Based CRDTs by Delta-Mutation," in *NETYS*, 2015.
[14] P. S. Almeida, A. Shoker, and C. Baquero, "Delta State Replicated Data Types," in *J. Parallel Distrib. Comput.*, 2018.
[15] Akka, "Distributed Data." [Online]. Available: https://doc.akka.io/docs/akka/2.5/scala/distributed-data.html
[16] IPFS, "Decentralized Real-Time Collaborative Documents." [Online]. Available: https://ipfs.io/blog/30-js-ipfs-crdts.md
[17] IPFS, "CRDT Research Repository." [Online]. Available: https://github.com/ipfs/research-CRDT/issues/31
[18] G. Birkhoff, "Rings of sets," in *Duke Mathematical Journal*, 1937.
[19] B. A. Davey and H. A. Priestley, "Introduction to Lattices and Order." Cambridge University Press, 1990.
[20] R. van Renesse, D. Dumitriu, V. Gough, and C. Thomas, "Efficient Reconciliation and Flow Control for Anti-entropy Protocols," in *LADIS*, 2008.
[21] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An Integrated Experimental Environment for Distributed Systems and Networks," in *SIGOPS Oper. Syst. Rev.*, 2002.
[22] R. Juan-Marín, H. Decker, J. E. Armendáriz-Íñigo, J. M. Bernabéu-Aubán, and F. D. Muñoz Escoí, "Scalability Approaches for Causal Multicast: A Survey," in *Distributed Computing*, 2016.
[23] Retwis. [Online]. Available: http://retwis.antirez.com
[24] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports, "Building Consistent Transactions with Inconsistent Replication," in *SOSP*, 2015.
[25] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *SOSP*, 2011.
[26] N. Crooks, Y. Pu, N. Estrada, T. Gupta, L. Alvisi, and A. Clement, "TARDiS: A Branch-and-Merge Approach To Weak Consistency," in *SIGMOD*, 2016.
[27] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload Analysis of a Large-Scale Key-Value Store," in *SIGMETRICS*, 2012.
[28] A. Tridgell and P. Mackerras, "The rsync algorithm," Australian National University, Tech. Rep., 1998.
[29] J. Macdonald, "Xdelta." [Online]. Available: http://xdelta.org
[30] V. Enes, C. Baquero, P. S. Almeida, and A. Shoker, "Join Decompositions for Efficient Synchronization of CRDTs after a Network Partition: Work in progress report," in *PMLDC@ECOOP*, 2016.
[31] A. van der Linde, J. Leitão, and N. Preguiça, "Δ-CRDTs: Making Δ-CRDTs Delta-based," in *PaPoC@EuroSys*, 2016.

[32] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic Algorithms for Replicated Database Maintenance," in *PODC*, 1987.
[33] J. Byers, J. Considine, and M. Mitzenmacher, "Fast Approximate Reconciliation of Set Differences," CS Dept., Boston University, Tech. Rep., 2002.
[34] M. P. Herlihy and J. M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," in *Trans. Program. Lang. Syst.*, 1990.
[35] C. Baquero, P. S. Almeida, A. Cunha, and C. Ferreira, "Composition in State-based Replicated Data Types," in *Bulletin of the EATCS*, 2017.
[36] V. Enes, P. S. Almeida, and C. Baquero, "The Single-Writer Principle in CRDT Composition," in *PMLDC@ECOOP*, 2017.
[37] DataStax, "What's New in Cassandra 2.1: Better Implementation of Counters." [Online]. Available: https://www.datastax.com/dev/blog/whats-new-in-cassandra-2-1-a-better-implementation-of-counters

APPENDIX

A. Existence of Unique Irredundant Decompositions

In this section we present sufficient conditions for the existence of unique irredundant join decompositions, and show how they can be obtained.

Definition 4 (Descending chain condition)**.** *A lattice $\mathcal{L}$ satisfies the* descending chain condition *(DCC) if any sequence $x_1 \sqsupset x_2 \sqsupset \cdots \sqsupset x_n \sqsupset \cdots$ of elements in $\mathcal{L}$ has finite length [19].*

Proposition 1. *In a distributive lattice $\mathcal{L}$ satisfying DCC every element $x \in \mathcal{L}$ has a unique irredundant join decomposition.*

Proof: Trivial, as corollary of the dual of Theorem 6 from [18]: a distributive lattice is modular; if it also satisfies DCC, then each element has a unique irredundant join decomposition. ∎

For almost all CRDTs used in practice, the state is not merely a join-semilattice, but a distributive lattice satisfying DCC (Appendix B). Therefore, from Proposition 1, we have a unique irredundant join decomposition for each CRDT state. Let $\Downarrow x$ denote this unique decomposition of an element x.

Proposition 2. *If $\mathcal{L}$ is a finite distributive lattice, then $\Downarrow x$ is given by the maximals of the join-irreducibles below x:*

$$\Downarrow x = \mathsf{max}\{r \in \mathcal{J}(\mathcal{L}) \mid r \sqsubseteq x\}$$

Proof: From the Birkhoff's Representation Theorem (see, e.g., [19]), each element x is isomorphic to $\{r \in \mathcal{J}(\mathcal{L}) \mid r \sqsubseteq x\}$, the set of join-irreducibles below it, which is isomorphic to the set of its maximals, containing no redundant element. ∎

Although Proposition 2 is stated for finite lattices, it can be applied to typical CRDTs defined over infinite lattices, as we show next.

B. Lattice Compositions in CRDTs

We now show that unique irredundant join decompositions (and therefore, optimal deltas and delta-mutators) can be obtained for almost all state-based CRDTs used in practice. Most CRDT designs define the lattice state starting from lattice chains (booleans and natural numbers), unordered sets, partial orders, and obtain more complex states by lattice composition through: cartesian product $\times$, lexicographic product $\boxtimes$, linear sum $\oplus$, finite functions $\hookrightarrow$ from a set to a lattice, powersets $\mathcal{P}$, and sets of maximal elements $\mathcal{M}$ (in a partial order). Note that two of the constructs, $\hookrightarrow$ and $\mathcal{P}$, were used in Section II-A to define GCounter and GSet, respectively. The use of these composition techniques and a catalog of CRDTs is presented in [35] but that presentation (as well as CRDT designs in general) simply considers building join-semilattices (typically with bottom) from join-semilattices, never examining whether the result is more than a join-semilattice.

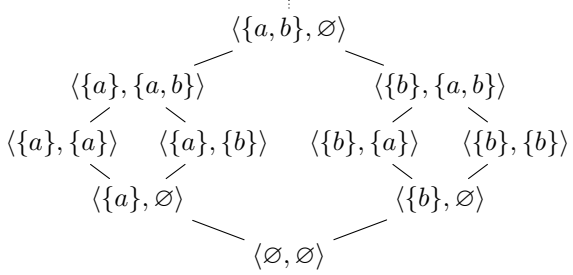


Fig. 13: Hasse diagram of $\mathcal{P}(\{a, b\}) \boxtimes \mathcal{P}(\{a, b\})$, a non-distributive lattice.

In fact, all those constructs yield lattices with bottom when starting from lattices with bottom. Moreover, all these constructs yield lattices satisfying DCC, when starting from lattices satisfying DCC (such as booleans and naturals). Also, it is easily seen that most yield distributive lattices when applied to distributive lattices, with the exception of the lexicographic product with an arbitrary first component. As an example, in Figure 13 we depict the Hasse diagram of a non-distributive lexicographic pair. This lattice is non-distributive since, e.g., for $x = \langle\{a\}, \{a\}\rangle$, $y = \langle\{a\}, \varnothing\rangle$ and $z = \langle\{b\}, \varnothing\rangle$, we have $x = x \sqcap (y \sqcup z) \neq (x \sqcap y) \sqcup (x \sqcap z) = y$. For the join-reducible $\langle\{a, b\}, \varnothing\rangle$, the set of the maximals of the join-irreducibles below it (i.e. $\{\langle\{a\}, \{a\}\rangle, \langle\{a\}, \{b\}\rangle, \langle\{b\}, \{a\}\rangle, \langle\{b\}, \{b\}\rangle\}$) is a redundant decomposition (as well as some of its subsets), and there are several alternative irredundant decompositions:

- $\{\langle\{a\}, \varnothing\rangle, \langle\{b\}, \varnothing\rangle\}$
- $\{\langle\{a\}, \varnothing\rangle, \langle\{b\}, \{a\}\rangle\}$
- $\{\langle\{a\}, \varnothing\rangle, \langle\{b\}, \{b\}\rangle\}$
- $\{\langle\{a\}, \{a\}\rangle, \langle\{b\}, \varnothing\rangle\}$
- ...
- $\{\langle\{a\}, \{b\}\rangle, \langle\{b\}, \{b\}\rangle\}$

Fortunately, the typical use of lexicographic products to design CRDTs is with a chain (total order) as the first component, to allow an actor which is "owner" of part of the state (the single-writer principle [36]) to either inflate the second component, or to change it to some arbitrary value, while increasing a "version number" (first component). This principle is followed by Cassandra counters [37]. In such typical usages of the lexicographic product, with a chain as first component, the distributivity of the second component is propagated to the resulting construct. Table III summarizes these remarks about how almost always these CRDT composition techniques yield lattices satisfying DCC and distributive lattices, and thus, have unique irredundant decompositions, by Proposition 1.

TABLE III: Composition techniques that yield lattices satisfying DCC and distributive lattices, given lattices A and B, chain C, partial order P and (unordered) set U.

	$\mathcal{L}$						
	$A \times B$	$A \boxtimes B$	$C \boxtimes A$	$A \oplus B$	$U \hookrightarrow A$	$\mathcal{P}(U)$	$\mathcal{M}(P)$
A, B, P has DCC $\Rightarrow \mathcal{L}$ has DCC	✓	✓	✓	✓	✓	✓	✓
A, B distributive $\Rightarrow \mathcal{L}$ distributive	✓	✗	✓	✓	✓	✓	✓

TABLE IV: Composition techniques that yield finite ideals or quotients, given lattices A and B, chain C, partial order P, all satisfying DCC, and (unordered) set U.

	$\mathcal{L}$						
	$A \times B$	$A \boxtimes B$	$C \boxtimes A$	$A \oplus B$	$U \hookrightarrow A$	$\mathcal{P}(U)$	$\mathcal{M}(P)$
$\forall x \in \mathcal{L} \cdot x/\bot$ finite	✓	✗	✗	✗	✓	✓	✓
$\forall \langle x, y\rangle \in \mathcal{L} \cdot \langle x, y\rangle / \langle x, \bot\rangle$ finite	–	✓	✓	✓	–	–	–

Having DCC and distributivity, even if it always occurs in practice, is not enough to directly apply Proposition 2, as it holds for finite lattices. However if the sublattice given by the ideal $\downarrow x = \{y \mid y \sqsubseteq x\}$ is finite, then we can apply that proposition to this finite lattice (for which x is now the top element) to compute $\Downarrow x$. Again, finiteness yields from all constructs, with the exception of the lexicographic product and linear sum. For these two constructs, a similar reasoning can be applied, but focusing on a quotient sublattice in order to achieve finiteness.

Definition 5 (Quotient sublattice)**.** *Given elements $a \sqsubseteq b \in \mathcal{L}$, the* quotient sublattice b/a *is given by:*

$$b/a = \{x \in \mathcal{L} \mid a \sqsubseteq x \sqsubseteq b\}$$

Quotients generalize ideals, as we have $\downarrow x = x/\bot$. As an example, given some infinite set U and the lattice $\mathbb{N} \boxtimes \mathcal{P}(U)$, for each $x = \langle n, s\rangle$, the ideal $\downarrow x$ is still infinite when $n > 0$, as depicted in Figure 14. However, for each $\langle n, s\rangle$, the quotient $\langle n, s\rangle / \langle n, \bot\rangle$ is a finite lattice, and moreover, the elements given by $\Downarrow\langle n, s\rangle$ are the same either when considering the original lattice or the quotient sublattice. Therefore, we can use the formula for $\Downarrow x$ in Proposition 2. A similar reasoning can be used for linear sums. Table IV summarizes these remarks; the second row applies only to lexicographic products and linear sums[2].

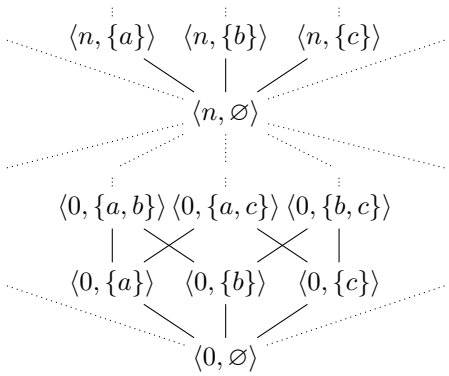


Fig. 14: Hasse diagram of $\mathbb{N} \boxtimes \mathcal{P}(U)$, for an infinite set U, where most ideals are infinite.

$$
\begin{aligned}
c \in C\text{:} \quad & \Downarrow c = \{c\} \\
\langle a, b\rangle \in A \times B\text{:} \quad & \Downarrow\langle a, b\rangle = \Downarrow a \times \{\bot\} \cup \{\bot\} \times \Downarrow b \\
\langle c, a\rangle \in C \boxtimes A\text{:} \quad & \Downarrow\langle c, a\rangle = \Downarrow c \times \Downarrow a \\
\mathsf{Left}\, a \in A \oplus B\text{:} \quad & \Downarrow \mathsf{Left}\, a = \{\mathsf{Left}\, v \mid v \in \Downarrow a\} \\
\mathsf{Right}\, b \in A \oplus B\text{:} \quad & \Downarrow \mathsf{Right}\, b = \{\mathsf{Right}\, v \mid v \in \Downarrow b\} \\
f \in U \hookrightarrow A\text{:} \quad & \Downarrow f = \{k \mapsto v \mid k \in \mathsf{dom}(f) \wedge v \in \Downarrow f(v)\} \\
s \in \mathcal{P}(U)\text{:} \quad & \Downarrow s = \{\{e\} \mid e \in s\} \\
s \in \mathcal{M}(P)\text{:} \quad & \Downarrow s = \{\{e\} \mid e \in s\}
\end{aligned}
$$

Note how the decompositions of GCounter and GSet presented in Section III-A are an application of these rules. As a further example, consider a *positive-negative counter* – PNCounter – a CRDT counter that allows both increments and decrements. In this CRDT, each replica identifier is mapped to a pair where the first component tracks the number of increments, and the second the number of decrements, i.e. $\mathsf{PNCounter} = \mathbb{I} \hookrightarrow (\mathbb{N} \times \mathbb{N})$. Given a PNCounter state $p = \{\mathsf{A} \mapsto \langle 2, 3\rangle, \mathsf{B} \mapsto \langle 5, 5\rangle\}$ (2 increments by A, 3 decrements by A, and an equal number of increments and decrements by B), the irredundant join decomposition of p is $\Downarrow p = \{\{\mathsf{A} \mapsto \langle 2, 0\rangle\}, \{\mathsf{A} \mapsto \langle 0, 3\rangle\}, \{\mathsf{B} \mapsto \langle 5, 0\rangle\}, \{\mathsf{B} \mapsto \langle 0, 5\rangle\}\}$.

C. Decomposing Compositions

In this section we show that for each composition technique there is a corresponding decomposition rule. As the lattice join $\sqcup$ of a composite CRDT is defined in terms of the lattice join of its components [35], decomposition rules of a composite CRDT follow the same idea and resort to the decomposition of its smaller parts. We now present such rules for all lattice compositions covered in Tables III and IV.

[2]In order to have a common notation for instances of $\boxtimes$ and $\oplus$, $\oplus$ instances are presented as pairs. For example, $\mathsf{Left}\, a \in A \oplus B$ becomes $\langle \mathsf{Left}, a\rangle$.