Approaches to Conflict-free Replicated Data Types

PAULO SÉRGIO ALMEIDA, Universidade do Minho, Braga, Portugal and INESC TEC, Porto, Portugal

Conflict-free Replicated Data Types (CRDTs) allow optimistic replication in a principled way. Different replicas can proceed independently, being available even under network partitions and always converging deterministically: Replicas that have received the same updates will have equivalent state, even if received in different orders. After a historical tour of the evolution from sequential data types to CRDTs, we present in detail the two main approaches to CRDTs, operation-based and state-based, including two important variations, the pure operation-based and the delta-state based. Intended for prospective CRDT researchers and designers, this article provides solid coverage of the essential concepts, clarifying some misconceptions that frequently occur, but also presents some novel insights gained from considerable experience in designing both specific CRDTs and approaches to CRDTs.

CCS Concepts: • **Computing methodologies** → *Distributed computing methodologies*; • **Theory of computation** → *Distributed algorithms*; • **Computer systems organization** → *Availability*; • **Software and its engineering** → *Data types and structures*; • **Information systems** → *Data replication tools;*

Additional Key Words and Phrases: Eventual consistency, replicated data types, CRDT

ACM Reference Format:
Paulo Sérgio Almeida. 2024. Approaches to Conflict-free Replicated Data Types. *ACM Comput. Surv.* 57, 2, Article 51 (November 2024), 36 pages. https://doi.org/10.1145/3695249

1 Introduction

Classic distributed systems aim for strong consistency (e.g., linearizability [40]) through the state-machine replication approach proposed by Lamport [52]. But while they can achieve performance (throughput), achieving strong consistency in systems with large spatial spans comes at the cost of high response time and the loss of availability under network partitions, as expressed by the CAP theorem [17, 34].

The importance of always-on availability for real businesses motivated relaxing consistency. A seminal work, which popularized the NoSQL movement, was Amazon's Dynamo [27], which stresses the importance of availability: "even the slightest outage has significant financial consequences and impacts customer trust." But programming NoSQL data stores is difficult and error-prone, given a low-level read-write-based API and the need for ad hoc reconciliation of concurrent updates.

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project UIDB/50014/2020. DOI: 10.54499/UIDB/50014/2020.
Author's Contact Information: Paulo Sérgio Almeida, Universidade do Minho, Braga, Portugal and INESC TEC, Porto, Portugal; e-mail: psa@di.uminho.pt.

ACM 0360-0300/2024/11-ART51
https://doi.org/10.1145/3695249

Since their appearance [78], **Conflict-free Replicated Data Types (CRDTs)**, soon became very popular. The essential concept is (1) providing a higher-level API, as in classic data types, but for distributed, replicated objects, while (2) achieving availability through relaxing consistency and allowing immediate local replica updates and queries, with asynchronous communication to make replicas converge, (3) with data-type-specific concurrency semantics and synchronization specified as built-in, freeing programmers from writing ad hoc reconciliation code.

CRDTs are difficult to design, prone to subtle bugs, but allow the majority of distributed application programmers to use them, from some library, with little effort, while only a minority of expert CRDT designers need to go through the intricate process of creating new CRDTs over time. This article is mostly aimed at prospective CRDT researchers or designers, but every CRDT user gains from having some knowledge about how they work. It describes the two main approaches to CRDTs, based on propagating operations or on propagating state, while also presenting two variants. The most important, delta-state CRDTs [4], aim to achieve, in a way, "the best of both worlds." Pure operation-based CRDTs [7, 8] are a relevant point in the design space that makes clear the role of specifications over a partially ordered set of operations (a partially ordered log) in the definition of the CRDT and of *causal stability* in achieving a small state, not achievable otherwise.

This article clarifies misconceptions regarding CRDTs, which frequently occur in papers or presentations, and shows novel depictions, e.g., to provide intuition about joining causal state-based CRDTs. One common misconception is regarding commutativity: "CRDTs are types with commutative operations," which is not true. Indeed, the more significant improvement over classic optimistic replication, like Lazy Replication [50], is the support for data types with non-commutative operations. We clarify the role of commutativity by presenting a better (than the usual) diagram showing the execution model of operation-based CRDTs. Another example is regarding monotonicity in state-based CRDTs: "mutators must be monotonic functions," when in fact they must be *inflations*, to result in the monotonic evolution of state. The article discusses the role of commutativity, idempotence, and inflations in the ability to reuse sequential data types for both operation- and state-based CRDT designs. The classic requirement of *prepare* in operation-based CRDTs to be side-effect-free is also addressed. We point out that if we distinguish the abstract state used in queries from the full CRDT concrete state, then the requirement can be relaxed, leading to better designs, and we present a novel *observed-remove set* that is better than any prior design.

After a historical tour showing the evolution from sequential data types to CRDTs, the subsequent sections present: operation-based CRDTs, pure operation-based CRDTs, state-based CRDTs, and delta-state based CRDTs. This is followed by a comparison of the approaches, a discussion of identity management towards scalability, and a presentation of practical applications. Along the article, we use classic examples (counters, registers, sets), which are relatively simple to explain and understand, while having enough subtlety to allow comparing approaches, avoiding more complex data types, such as lists (e.g., Treedoc [70], RGA [74]), which need considerably more involved algorithms. Collaborative Editing and the List data type are discussed in Section 9, Practical Applications.

2 From Sequential Data Types to CRDTs

2.1 From Sequential to Concurrent Data Abstractions

Data Abstractions. Abstraction is essential to tame complexity and scale. Two main types are functional and data abstraction. Functional abstractions (functions and procedures) were introduced first, roughly at the same time (1958) in Fortran, Lisp, and Algol. Data abstractions involved a longer evolution over time, with the two main variants being abstract data types and objects.

The ingredients to obtain data types are normally thought of as: procedures; the concept of record, introduced in the AED-1 language [75] (then named *plexes*) and adopted for Algol by Wirth

and Hoare [87]; the combination of procedures and records. What actually happened [63] was the generalization of Algol blocks to have lifetime not restricted to stack allocation, in Simula [26] processes (and later classes). The final ingredient was hiding the representation from client code, in CLU [56], leading to **abstract data types (ADTs)**. (Simula had some information hiding capability, through inner blocks, which was not adequate.)

Sequential Data Types. In imperative languages, sequential data types became the most common abstraction in libraries. The availability of types such as Set and Map, serving as "Swiss Army knifes" in solving many problems, diminished the number of times programmers had to "reinvent the wheel" and end up with slow and buggy implementations. The research effort in efficient imperative implementations of such data types has been immensely useful for the "real world."

For sequential data types, proving correctness is relatively easy (when no aliasing is involved), made possible with the introduction of axiomatic reasoning by Hoare [42], involving the notions of preconditions, postconditions and invariants, and its adaptation to data types [43]. The pervading occurrence of aliasing poses a problem; languages like Rust aim to avoiding mutable state sharing.

Concurrent Data Types. It was only natural to extend sequential data types to shared memory concurrency to obtain objects usable by concurrent threads. Inspired by Simula, Hansen [38] introduced monitors with the construct of *shared classes*, and Hoare [44] introduced a slight variant. Monitors enforce mutual-exclusion during operation execution, allowing the concept of *atomic objects*, which are easy to reason about using the same concepts of pre-/post-conditions and invariants, as no concurrency occurs during operation execution. Monitors also allow blocking mid-operation via the await primitive (Hansen) or condition variables (Hoare). This is something useful and essential for inter-process synchronization, to achieve cooperation through shared abstractions, the most well known being the *bounded-buffer*. Unfortunately, it complicates reasoning. Blocking abstractions, common in shared-memory concurrency, are less common in distributed systems (an example being a distributed lock) and do not suit CRDTs, which aim to be always available locally.

Lock-free and Wait-free Data Structures. Towards achieving performance and fault tolerance in shared-memory multiprocessors, lock-free data structures were proposed. These do no use locks but resort directly to low-level atomic operations, such as *compare-and-swap*, provided by hardware. This allows several data type operations in progress concurrently, not in mutual exclusion (as in monitors). Moreover, if wait-free [39], then it guarantees that an operation completes in a finite number of steps, regardless of what other processes do. Their emphasis is on multiprocessors, not distributed systems, not being of further concern here, except for one question they raise: how to express correctness criteria when several actions are happening concurrently. This issue is relevant for distributed systems.

2.2 Strongly Consistent Replication

Linearizability. Linearizability [40] is the more widely used correctness criterion when aiming for implementations that, even allowing concurrent execution of operations, mimic the behavior exposed by a sequential data type. An execution is linearizable, roughly, if (1) it is equivalent to some sequential execution; (2) it respects "real time" for non-overlapping operations.

Consider a register object (Figure 1) being accessed by two processes. The read from P_2 must return 0 if the register provides linearizability, even though w(2) was the last operation to return, before the read. The reason is that, because the read from P_1 returned 2, the write from P_2 must have taken effect before it, in the interval shaded in blue, being ordered before r():2 and w(0) from P_1.

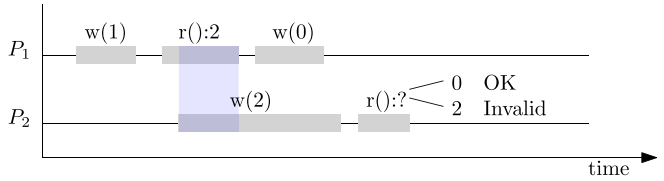

Fig. 1. A register object with read and write operations. To be linearizable, last read must return 0.

Replicated State Machines. Obtaining distributed implementations of data types that comply with linearizability can be done through the state-machine replication approach. This was proposed by Lamport [52] and consists essentially of:

(1) Replicate state on several nodes;
(2) Same deterministic state machine at each node, $S' = F(S, I)$, where the next state is a function of the current one and the input;
(3) Agree on a global total order of inputs from all nodes;
(4) Apply totally ordered inputs at each node.

The result is that the replicated system behaves as if it were a single machine. The most difficult part, specially to be fault-tolerant, is the agreement [68]. The extreme case of tolerating byzantine faults is applied in the now-popular blockchains, many of which use some variant of the PBFT algorithm [23]. Using a sequential data type for the state machine results in a replicated data type. One could think then that the problem of obtaining distributed implementations of data types is solved and there is nothing more that needs to be done.

2.3 The CAP Theorem and Consistency Models

The CAP Theorem. In a keynote at the PODC'00 conference, Brewer [17] stated a conjecture that in a distributed system we can only achieve, simultaneously, at most two of the three guarantees: strong Consistency, Availability, Partition tolerance. This conjecture was proved [34], more concretely, interpreting strong consistency as linearizability, and became known as the CAP theorem. This is commonly expressed as a trilemma, in which we can only pick two out of the three properties. But because network partitions may always occur and cannot be avoided, we can design either AP or CP systems: achieve either availability or strong consistency (linearizability).

CP vs. AP. The CAP theorem implies an important design choice for distributed systems. Whether to achieve linearizability (CP) or availability (AP). But even when there are no network partitions, the design choice has also implications on response times. As a rough summary, CP systems:

— aim for linearizability;
— may become unavailable under network partitions;
— have high response times in wide area.

In AP systems:

— operations can remain available, even when there are partitions;
— response times can be low even in wide area.

Forgoing linearizability, an important question is: what consistency model to aim for?

Consistency Models. The definition of consistency models has been an important and complex research topic going over many decades. It typically involves tradeoffs regarding consequences to relevant actors (e.g., hardware, compiler, programmers) in matters such as efficient use of "the

machine," ease of implementability, ease of reasoning, or useful guarantees. Viotti and Vukolic [82] present over 40 models. Of these, causal consistency [2] is of special importance, being (broadly) the strongest achievable while not losing availability [57].

Causal Consistency. **Causal Consistency (CC)** allows processes to see different orders, as long as they are consistent with a global *happens-before* partial order. Happens-before (hb) contains the session order so, which relates operations from each process, and the *visibility* order vis [19] (operation a is visible to b, written $a \,\mathsf{vis}\, b$, if the effect of a is visible to the process performing b), being the transitive closure of their union:

$$\mathsf{hb} \doteq (\mathsf{so} \cup \mathsf{vis})^+$$

and causal consistency essentially means that:

$$\mathsf{vis} = \mathsf{hb},$$

i.e., all operations from the causal past are visible, with no missing updates. Processes may possibly arbitrate operations in different total orders, each compatible with the global visibility partial order, not necessarily converging (causal consistency itself does not ensure convergence).

Convergence. The strongest guarantees that AP systems aim for are causal consistency together with what has become known as ***strong eventual consistency* (SEC)** [78], which guarantees that all updates will eventually become visible everywhere (eventual visibility/delivery) and that processes that see the same set of updates have equivalent state, regardless of the order in which they become visible (*strong convergence*).

We remark that the term SEC was an unfortunate choice of terminology, and the source of some confusion, due to the use of the word *strong*, usually associated with strong consistency models, such as sequential consistency and linearizability. Moreover, ***Eventual Consistency* (EC)**, as originally introduced by Terry et al. [80], already includes the SEC guarantees. As described in that paper, EC systems should include mechanisms to ensure two properties, on which EC relies:

- *total propagation*: each update is eventually propagated everywhere, by some anti-entropy mechanism (i.e., eventual delivery);
- *consistent ordering*: non-commutative updates are applied in the same order everywhere, which implies the *strong convergence* property of SEC.

These two properties mean, as described by Terry et al. [81], also addressing EC, that "all servers eventually receive all Writes via the pair-wise anti-entropy process and that two servers holding the same set of Writes will have the same data contents." So, originally EC meant SEC, and a similar definition of EC was also used by Roh et al. [74]. The informal meaning of EC "eventual convergence when updates stop being issued," since it became popular due to Vogels [83], is just a consequence of the above properties that EC systems ensure. As originally introduced, what is "eventual" in EC is operation visibility (delivery); replicas that have delivered the same set of updates have converged.

Spatial Scalability. More than just being available (AP), EC systems aiming for CC and convergence can be performant and usable at large spatial scales (very wide area), with low operation response time. This is an essential property for interactive systems, sometimes forgotten when thinking only of global throughput as a measure of success, often done when evaluating CP systems.

Causal consistency can be achieved while being as fast as possible in terms of physical limits (speed of light). We can say that these systems exhibit "mechanical sympathy" regarding our universe, i.e., the model suits the "machine," even at large spatial scales. (The term *mechanical sympathy* was introduced in software by Martin Thomsom to refer to software being built with

the understanding of how hardware works, so it can suit the hardware and run efficiently in the underlying machine.)

On the contrary, linearizability can be said to be *contra naturam*. There is a wide mismatch between model (what it aims to guarantee) and "machine" (our universe). It would suit an Aristotelian universe with infinite light speed. In our universe, we must slow things down (delay responses) to achieve it. The response time degrades with spatial span, becoming impractical for very wide distances; e.g., linearizability would be completely unsuitable for a system encompassing Earth and a future Mars colony. Large spatial spans require specialized techniques or protocols, which motivated research on **DTNs (delay-tolerant networks)** and **OppNets (opportunistic networks)**, for which CRDTs can be suitable. Guidec et al. [37] investigate the implementation of CRDTs for OppNets.

2.4 Optimistic Replication and CRDTs

Optimistic Replication. Well before the CAP theorem was introduced, systems relaxing coordination and providing availability were developed. This was the *optimistic replication* [76] approach. The main ingredients were:

— replicas are locally available for queries and updates;
— updates are propagated asynchronously, in the background, opportunistically.

Thus, a total order of operations was not attempted: Updates could become known in different orders by different replicas. This approach achieves both availability and low latency, but it poses a problem: Replicas may diverge, possibly forever. This problem was addressed using two approaches in two relevant early works:

— In Lazy replication [50], relax ordering, but only for commutative operations.
— In Bayou [81], have conflict detection, client-provided merge procedures, tentative writes for availability, but the same order at all replicas for committed writes (possibly undoing and reapplying if necessary) for eventual consistency.

Conflict-free Replicated Data Types. The introduction of CRDTs [78] was an important step over previous approaches to optimistic replication. A CRDT is exposed as a standard data type, providing operations. Each data type object is replicated and accessed locally by two kinds of operations:

— mutator operations update state;
— query operations look at the state and return a result.

Operations are always available, not depending on synchronization. A CRDT object is highly available, even under partitions, with essentially zero operation response time (only local computation).

As previous optimistic replication approaches, information is propagated asynchronously. The main novelty is that conflicts are dealt with semantically, making a replication-aware concurrent specification part of the data type definition. This specification expresses how conflicts are solved and, contrary to previous approaches, is not limited to commutative operations. Conflict resolution is encapsulated by the data type, freeing programmers from having to write application-specific ad hoc conflict resolution code. (The effort becomes choosing the appropriate CRDTs.)

ADTs vs. Objects. Evolution of object-oriented languages involved many concepts, such as abstract data type, parametric polymorphism, object, class, inheritance, and subtyping, as described in the classic by Cardelli and Wegner [22], with tools such as existential and bounded universal quantification. As well summarized by Cook [25], there are two main kinds of data abstractions: ADTs and objects.

ADTs can be modeled as existential types, can access multiples instances, have efficient binary operations, and different implementations cannot mix. Objects can be modeled with recursive

higher-order functions, access only the self state, binary operations are "slow" or even impossible, and multiple implementations can be used together.

Most of the above is irrelevant to this article, except for one aspect: ADT implementations can have access to multiple instances (e.g., binary operations, like multiply), while objects can only access the self state, with other objects being only accessible through their interfaces.

For distributed systems, this means that only with full replication would replicated ADTs be viable. In general, with only a subset of objects being replicated in each node, we cannot rely on being able to access several specific objects together. This is why CRDTs do not provide binary operations involving two (or more) instances, but only operations on the "self" object. Therefore, CRDTs normally are really "Conflict-free Replicated Objects," which would have been a more suitable name.

Operation-based vs. State-based Approaches. Concerning CRDT implementation (both the data type itself and the propagation mechanism), there are two main approaches: operation-based and state-based.

Operation-based approaches propagate information about operations to other replicas, using a reliable messaging algorithm for propagation. This normally needs some ordering guarantees, but weaker than a total order. Normally causal delivery is chosen to achieve the goal of having causal consistency. A special case is the *pure* operation-based approach.

State-based approaches propagate replica states, as opposed to propagating operations. A merge function is defined to be able to reconcile replica states. State propagation is opportunistic, by "background" communication, typically much less frequent than per-operation, to amortize the cost of propagating full states. An important variant, to make the propagation more incremental, is the delta-state-based approach, partially combining the advantages of both approaches.

3 Operation-based CRDTs

The core concept of op-based (for short) CRDTs is to send operations, not state, to other replicas, towards replica convergence. So, when an update operation is invoked, in addition to being applied to the replica where it was invoked, it is sent to all other replicas, asynchronously, upon which they are applied at those replicas when they arrive. Query operations (that do not cause state changes) can make use of the local state and be responded to immediately, causing no inter-replica messaging.

Because operations are not, in general, idempotent, it is essential that an exactly-once messaging mechanism is used. For some CRDTs, no ordering guarantees at all would be needed for convergence. But towards ensuring, in addition to convergence, causal consistency—the strongest possible consistency model while remaining available under partitions—a causal broadcast [14] mechanism is normally adopted, making causally dependent operations become visible in the correct order.

The essential improvement over previous attempts at optimistic replication is the treatment of non-commutative operations. Two concurrently invoked non-commutative operations could arrive and be applied in different orders in different replicas, which would lead to divergence. This is dealt with by sending more than just the operation when "broadcasting operations" to other replicas.

3.1 Execution Model and Concurrency Semantics

Standard Execution Model of Operation-based CRDTs. To achieve convergence even for data types containing non-commutative operations, the execution model for op-based CRDTs, presented in Figure 2, divides the execution of an update operation in two phases: *prepare* and *effect*.

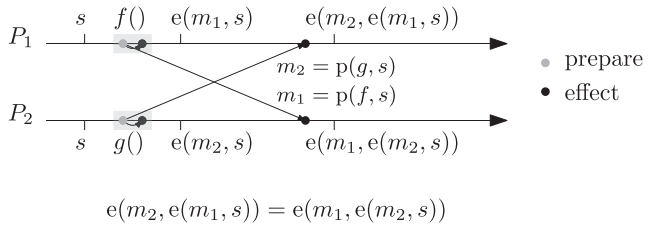

Fig. 2. Execution model for op-based CRDTs, stressing the commutativity of effect for concurrently invoked operations.

(1) When an update operation is invoked, prepare is performed locally:
   — it looks at the state and the operation;
   — it must have no side effects (on the abstract state);
   — the result from prepare is disseminated with reliable causal broadcast.
(2) Upon message delivery at each replica, effect is applied:
   — takes message (result from prepare) and state and produces new state;
   — it is designed to be commutative for concurrently invoked operations;
   — it assumes immediate self-delivery on sender replica.

Immediate self-delivery is important to ensure "Read Your Writes" [80], making the state-change immediately reflected locally and visible to subsequent query operations that follow the update.

Reusing Sequential Data Types with Commutative Operations. Data types that only have commutative operations can be implemented as op-based CRDTs trivially: the state is the same as for the sequential data type; prepare returns the operation identifier and arguments; effect invokes the corresponding sequential data type operation. These data types respect the *Principle of Permutation Equivalence* [10]: If all sequential permutations of updates lead to the same state, then concurrent execution of those operations should converge to that same state. For such data types, not even FIFO order is needed for convergence, just exactly-once delivery. Causal delivery is normally used to achieve causal consistency. Three examples are illustrated in Figure 3: GCounter, with only an increment update operation; a PNCounter, which may be negative, having both increment and decrement; GSet, a grow-only set having just an add update operation.

CRDTs with Non-commutative Operations. Where the CRDT approach becomes more interesting is for data types with non-commutative operations. An example is a set data type, having add and remove operations. These are not commutative, as:

$$\mathsf{add}(v, \mathsf{rmv}(v, s)) \neq \mathsf{rmv}(v, \mathsf{add}(v, s)).$$

If the state were the same as for the sequential data type, and effect defined as simply applying the corresponding operation, then the possibility of different delivery orders for concurrently invoked add and remove of the same element would lead to divergence.

Therefore, the state must be more involved, and effect must be defined such that it is commutative for concurrently invoked operations. But not only convergence is relevant. How can we define what is supposed to happen given two concurrently invoked add and remove? I.e., how can we define the data type semantics for such CRDTs?

Defining Concurrent Semantics. A first design criteria for CRDT semantics is preserving the sequential semantics of the original data type; i.e., under a sequential execution, the CRDT should produce the same outcome as the corresponding sequential data type. Then, we must define how

GCounter

CRDT state:
$n : \mathbb{N} = 0$
query value() : $\mathbb{N}$
return n
update inc()
prepare
return inc
effect inc
$n \leftarrow n + 1$

PNCounter

CRDT state:
$v : \mathbb{Z} = 0$
query value() : $\mathbb{Z}$
return v
update inc()
prepare
return inc
effect inc
$v \leftarrow v + 1$
update dec()
prepare
return dec
effect dec
$v \leftarrow v - 1$

Grow-only set (GSet$\langle E \rangle$)

CRDT state:
$s : \mathcal{P}(E) = \emptyset$
query elements() : E
return s
query contains($e : E$) : $\mathbb{B}$
return $e \in s$
update add($e : E$)
prepare
return (add, e)
effect (add, e)
$s \leftarrow s \cup \{e\}$

Fig. 3. Simple CRDTs with only commutative operations: GCounter, PNCounter and GSet. State and effect the same as for a sequential data type.

to handle conflicts for concurrently invoked operations. In the set example, given concurrent add and remove of the same element, we want to define which will "win." We have several possibilities:

— add wins;
— remove wins;
— **last-writer-wins (LWW)**, using a totally ordered arbitration.

For CRDTs, in general, the outcome may not be equivalent to some sequential execution. Data type semantics are usually defined resorting to the causal past, i.e., the result from a query depends on the set of update operations that are visible to it, and their partial order under *happens-before*.

3.2 Observed-cancel CRDTs

Observed-cancel Semantics. We may define different CRDTs, with different semantics, for each sequential data type (e.g., for a set). A particularly interesting concept, for choosing in which way a conflict between two operations is handled, is what can be called *observed-cancel* semantics. Essentially, "cancel observed (visible) operations, as if they were never issued."

Using causal delivery, as usual for op-based CRDTs, this means that an operation that cancels another will cancel the operations in its causal past, but not the ones concurrently issued. This is the most appealing choice, because it makes an operation act upon a closed, well-defined set of other operations that have already taken effect on the state. It will not "blindly discard" updates not yet seen and that could not have been accounted for yet. (Such updates will eventually become visible and can always be canceled subsequently.) This prevents undesired "lost updates" that CRDTs aim to avoid. Two examples are the observed-remove set [11]:

— has add and remove operations;
— remove only cancels the adds visible to it;
— concurrent adds will not be canceled and will "win";

and the observed-reset counter [84]:

— has increment and reset;
— a reset cancels the observed increments.

The appeal of observed-cancel semantics can be seen in the observed-reset counter: It allows a sample-and-reset pattern, where a process periodically samples the counter value and resets

```
CRDT state:
    s : 𝒫(E × ...) = ∅
query elements() : E
    return {e | (e, _) ∈ s}
query contains(e : E) : 𝔹
    return ∃x · (e, x) ∈ s

update add(e : E)
    prepare
        let u = [some unique id]
        return (add, e, u)
    effect (add, e, u)
        s ⟵ s ∪ {(e, u)}
update remove(e : E)
    prepare
        let r = {(x, u) ∈ s | x = e}
        return (remove, r)
    effect (remove, r)
        s ⟵ s \ r
```

Fig. 4. Op-based observed-remove set, ORSet⟨E⟩, naive implementation.

it. This allows grouping increments in a sequence, without losing any increment, even the ones concurrently issued, which will be accounted for in the next sample-and-reset. An alternative semantics where reset cancels concurrent increments would not allow accurate accounting to be achieved.

Observed-remove Set. A well-known example is precisely the observed-remove set, also called add-wins set, because adds win over concurrent removes. Several different implementations were presented in the literature, from very naive to "optimized" [11]. An even more optimized version, not previously published, is presented in Figure 5.

Many CRDTs indeed start from a naive version, being further optimized along time. The observed-remove set is a good example of possible improvements. A vanilla, naive, version is shown in Figure 4. In this CRDT, the state is a set of pairs and an element e is considered to belong to the set if there is a pair (e, u), for some unique identifier u, in the set of pairs. This version has some open issues and several possible improvements.

The first issue is that it assumes the generation of unique IDs, but does not specify how to do so. This is a frequent need and can be achieved by using unique replica IDs and a counter per replica, incremented at each operation. Unique IDs are obtained as pairs (replica ID, counter); this is what we call a "dot" (from Dotted Version Vectors [69]). The second issue is that, being the state a set of pairs, most operations need set traversal, which is inefficient. The solution is to use a map from elements to sets of IDs. A third issue is that adds keep accumulating state, adding a new pair to the ones from previous adds of the same element. An improvement is to replace the current pairs, for the given element, with a single pair for the new unique ID. A fourth issue is that prepare for remove sends the element being removed repeated in each pair. The improvement is to collect the set of IDs separately.

The optimized implementation, in Figure 5, contains all the above improvements. It assumes a unique replica identifier i in the set $\mathbb{I}$ of possible replica identifiers and assumes the standard op-based execution model using causal delivery. Each replica has a counter, incremented per add, which allows, together with the replica ID, to generate unique IDs. It must be noticed that this counter is auxiliary state, not part of the CRDT state used in queries or effect. This auxiliary state does not converge, and it can be updated in prepare. This allows not respecting the classic rule that prepare must be free from side effects and obtaining a better CRDT implementation. Currently published versions that do not make this distinction are less elegant.

The state is a map from elements in the set to sets of operation identifiers. Here, we assume that the map stores only non-empty sets, implicitly returning $\emptyset$ for unmapped keys. Prepare returns a tuple with operation, argument, unique ID in the case of an add, and the set of IDs that the map

types:
$\mathbb{I}$, set of replica identifiers
parameters:
$i \in \mathbb{I}$, replica identifier
CRDT state:
$m : E \rightharpoonup \mathcal{P}(\mathbb{I} \times \mathbb{N}) = \emptyset$
$c : \mathbb{N} = 0$, auxiliary state
query elements() : E
return dom m
query contains($e : E$) : $\mathbb{B}$
return $m[e] \neq \emptyset$
update add($e : E$)
prepare
$c \leftarrow c + 1$
return (add, e, (i, c), $m[e]$)
effect (add, e, d, r)
$m[e] \leftarrow m[e] \setminus r \cup \{d\}$
update remove($e : E$)
prepare
return (remove, e, $m[e]$)
effect (remove, e, r)
$m[e] \leftarrow m[e] \setminus r$

Fig. 5. Op-based observed-remove set ORSet$\langle E \rangle$, optimized implementation. Algorithm for replica i.

holds, for the given element. When effect is applied for remove, it subtracts the set of IDs sent from the ones in the map for the corresponding element. This has the desired outcome of removing the adds that have been observed at the replica where the remove was invoked, at the time it was invoked. Concurrently issued adds will "survive." For add, effect adds the newly generated ID and subtracts the set of IDs present when the add was issued (similar to remove), as the new ID makes the others redundant.

This CRDT is more optimized than the one previously published [11]: It avoids computation cost by organizing entries in a map; avoids sending the element repeatedly in a remove; and removes all observed identifiers for the element in an add, while the previously published only discards entries of previous adds from the same source. This exemplifies how, even for a relatively simple CRDT, many different versions with subtle variations may exist. This example also shows the role of commutativity: Not only the operations themselves (add/remove) are not commutative, but the effect of add/remove is also not commutative; only the effect of concurrently issued add/remove is so.

3.3 Beyond Sequential Semantics and API

Lack of Equivalence to Sequential Executions. CRDTs aim to preserve the sequential semantics for sequential executions, but for concurrent invocations not always is it possible to achieve equivalence to some sequential execution. In some cases, the CRDT:

— has behavior not possible by any sequential execution;
— the interface itself is different from the sequential data type.

The observed-remove set allows executions that are not equivalent to any sequential execution, considering the corresponding sequential data type semantics (of a set). This is easily seen by a run with two replicas, involving two elements a and b, where concurrently each replica adds an element and removes the other, i.e., add(a); remove(b) $\parallel$ add(b); remove(a). By the observed-remove set semantics, upon convergence, both elements will be in the set, which is impossible in a sequential execution, as some remove will be the last operation.

Moreover, there are CRDTs for which even the interface itself was changed from the corresponding sequential data type. The most well-known example with a modified interface is the multi-value register, made popular by the Dynamo [27] key-value store from Amazon.

Multi-value Register. The multi-value register keeps the set of the most recent concurrent writes. A read returns that set of values, and a write overwrites that set, in the current replica into a singleton. A multi-value register implementation using the same technique of generating unique IDs as for the observed-remove set is presented in Figure 6.

The state is a set of pairs, each with the written value and corresponding unique ID. Prepare returns a tuple with: the operation, a pair with value and unique ID, and the set of IDs in the state.

types:
$\mathbb{I}$, set of replica identifiers
parameters:
$i \in \mathbb{I}$, replica identifier
CRDT state:
$s : \mathcal{P}(E \times (\mathbb{I} \times \mathbb{N})) = \emptyset$
$c : \mathbb{N} = 0$, auxiliary state
query read() : $\mathcal{P}(E)$
return $\{e \mid (e, \_) \in s\}$

update write($e : E$)
prepare
$c \leftarrow c + 1$
let $r = \{d \mid (\_, d) \in s\}$
return $(\text{write}, (e, (i, c)), r)$
effect (write, v, r)
$s \leftarrow \{(e, d) \in s \mid d \notin r\} \cup \{v\}$

Fig. 6. Op-based multi-value register, MVReg$\langle E \rangle$. Algorithm for replica i.

When effect is applied, it keeps the pairs in the state with ID not present in the set of IDs in the message and adds the new pair. This has the desired outcome of removing writes made obsolete, only preserving the most recent concurrent writes. At the replica where the write is invoked only a singleton will remain.

4 Pure Operation-based CRDTs

Pure Op-based CRDTs. Pure op-based CRDTs [7] are a subset of general op-based CRDTs, restricted to the essence of "send only operations." Pure op-based CRDTs use the same prepare-effect execution model. What defines them is the restriction that:

— Prepare simply returns the operation (including arguments), ignoring current state.

Given operation o and state s:

$$\text{prepare}(o, s) = o.$$

This is unlike the general op-based approach, which can depart too much from the op-based spirit, allowing implementations that are, for all practical purposes, state-based: We can pick any state-based CRDT, define prepare as applying the operation and returning the resulting state, and define effect as merging states. However, pure op-based CRDTs may be less efficient than the general op-based approach. The pure-op based approach is relevant as an important point in the design space.

Pure Implementations of Commutative Data Types. For data types with only commutative operations, where for any operations f and g and state s:

$$f(g(s)) = g(f(s)),$$

operations can be applied in any order, producing the same result. As discussed in the general op-based model, the CRDT specification can be based on the sequential specification, with a trivial implementation, defining effect as simply applying the operation:

$$\text{effect}(o, s) = o(s).$$

This applies to CRDTs such as GCounter, PNCounter, and GSet, described before, which fit the pure-op model. Again, the challenge lies in non-commutative operations: How can we obtain pure op-based implementations for non-commutative data types? The solution is to use an augmented form of causal broadcast, called *tagged causal broadcast*, which provides knowledge about happens-before and about *causal stability*, which we describe below.

4.1 Tagged Causal Broadcast and Causal Stability

Tagged Causal Broadcast (TCB). Causal broadcast middleware already manages causality information, but normal APIs do not expose it to clients. *Tagged Causal Broadcast* [7] provides:

Table 1. Some Usages of "Stability" and Their Meanings

Term	Provenance	Meaning
Self stabilization	Dijkstra [29]	returns to valid state
Stable storage	Lampson and Sturgis [53]	durable storage, survives crashes
Multicast stability	Birman et al. [15]	message was received by all nodes
Write stability	Terry et al. [81]	a tentative write commits
Causal stability	Baquero, Almeida, and Shoker [7]	concurrent messages were delivered

— a partial order on messages in terms of an end-to-end happens-before;
— information about *causal stability* of messages, as we define below.

The TCB API defines delivery as providing, together with the message itself, a timestamp corresponding to a partially ordered logical clock value that reflects *happens-before*. This timestamp can be used in the implementation of the CRDT to distinguish between causally related and concurrently invoked operations and implement the desired semantics.

Causal Stability. We define causal stability as: *message with timestamp t is* causally stable *at node i when all messages subsequently delivered at i will have timestamp $t' \geq t$*. So, a message is causally stable when no more concurrent messages will be delivered. This is different from classic multicast/message stability [15].

A multicast is stable when it has been received by all nodes. Multicast stability is used internally by the messaging middleware, being useful for garbage collection when ensuring fault tolerance: Once a message has been delivered at some node and becomes stable, that node can discard it.

While classic multicast stability regards messages being received, being an implementation aspect, causal stability is a property involving delivery, visible to TCB clients. Also, while multicast stability is a global property, causal stability is a per-node property: Some messages may be causally stable in some node but not in others. Causal stability is stronger: A message only becomes causally stable in some node when it has become stable.

The Many Faces of Stability. "Stability" has been a much-overloaded expression in distributed systems. Table 1 shows some terms where the word stability is used and their rough meaning. Causal stability has not been properly recognized, being sometimes confused with message stability. Although the concept itself was not new when pure op-based CRDTs were introduced, coining the term "causal stability" will help avoid some confusion, especially in contexts where both may coexist, such as when describing a system involving pure op-based CRDTs, which rely on causal stability, making use of a TCB middleware, whose implementation may resort to multicast/message stability.

Distributed Algorithm for Pure Op-based CRDTs. The distributed algorithm for pure op-based CRDTs becomes simply reacting to the TCB middleware callbacks and invoking either prepare, effect, or a stable function to make use of causal stability information, as shown in Figure 7.

4.2 Resorting to a Partially Ordered Log

Naive PO-Log-based Implementations. A starting point for implementing pure op-based CRDTs is making the state a *PO-Log*: partially ordered log of operations. The PO-Log can be implemented as a map from timestamps to operations. Effect simply adds an entry (t, o) delivered by TCB to the PO-Log. This makes both prepare and effect have a universal definition. Only queries are data-type-dependent, being defined over the PO-Log. This approach is shown in Figure 8. It is very naive, but a starting point for subsequent optimization.

state:
$s \in S$
on operation(o):
tcbcast(prepare(o, s))
on tcdeliver(m, t):
$s \leftarrow$ effect(m, t, s)
on tcstable(t):
$s \leftarrow$ stable(t, s)

Fig. 7. Distributed algorithm for pure op-based CRDTs, making use of a TCB middleware.

$$
\begin{aligned}
S = T \rightharpoonup O \qquad & s^0 = \{\} \\
\text{prepare}(o, s) &= o \\
\text{effect}(o, t, s) &= s \cup \{(t, o)\} \\
\text{eval}(q, s) &= [\text{data type specific query function over PO-Log }]
\end{aligned}
$$

Fig. 8. PO-Log based implementation for pure op-based CRDTs.

$$
\begin{aligned}
S = T \rightharpoonup O \qquad & s^0 = \{\} \\
\text{prepare}(o, s) &= o \qquad\qquad (o \text{ either } [\text{add}, v] \text{ or } [\text{rmv}, v]) \\
\text{effect}(o, t, s) &= s \cup \{(t, o)\} \\
\text{eval}(\text{elements}, s) &= \{v \mid (t, [\text{add}, v]) \in s \ \wedge \nexists (t', [\text{rmv}, v]) \in s \cdot t < t'\}
\end{aligned}
$$

Fig. 9. PO-Log-based observed-remove (add-wins) set CRDT.

PO-Log-based Observed-remove (add-wins) Set. An example of a PO-Log-based CRDT, an observed-remove set, is shown in Figure 9. Prepare and effect have the universal definition shown before. Only query is data-type-specific. The query mimics the specification over the partial order of operations: An element is considered to be in the set if there exists an add for that element not canceled by a remove in its causal future. Essentially, this implementation is a naive runnable specification. But it shows the role of partial-order-based concurrent specifications in the design of CRDTs [20, 21] and how it fits directly the pure op-based model, as opposed to the totally ordered history of operations used for sequential specifications.

Semantically Based PO-Log Compaction. PO-Log-based CRDTs as described would be very inefficient and need optimizations to be actually usable. The idea is to avoid PO-Log growth, making it compact by keeping the smallest number of items that produce equivalent results when queries are performed. The compaction is data-type-specific, according to the specification, and makes use of the causality-related data provided by the TCB middleware:

— Causality: to prune the PO-Log after effect is performed, i.e., after operation delivery;
— Causal stability: to discard timestamps for operations that become causally stable and in some cases to remove operations that become redundant after causal stability.

Causality information can be exploited to achieve PO-Log compaction by redefining effect to use a data-type-specific obsolete relation between timestamps:

$$\text{effect}(o, t, s) \doteq \{x \in s \mid \neg\, \text{obsolete}(x, (t, o))\} \cup \{(t, o) \mid x \in s \Rightarrow \neg\, \text{obsolete}((t, o), x)\}.$$

When a new pair (t, o) is delivered, effect discards from the PO-Log all elements x such that obsolete$(x, (t, o))$ holds. Moreover, the delivered pair (t, o) is only inserted into the PO-Log if it

is not itself redundant, i.e., if obsolete$((t, o), x)$ is false for any x in the PO-Log. The obsolete relation is not restricted to being a partial-order to allow, e.g., a newly arrived operation to discard others in the PO-Log without necessarily being itself added.

Regarding exploiting causal stability, the basic improvement is to replace timestamps by $\perp$ when they become causally stable, as they will always compare as in the past of any operation that subsequently arrives, saving space. For some data types, we can also remove some causally stable operations themselves if they become redundant considering the data type semantics. We thus define stable as:

$$\mathsf{stable}(t, s) \doteq \mathsf{stabilize}(t, s)[(\perp, o)/(t, o)],$$

where stabilize is a data-type-specific function that discards operations from the PO-Log that become redundant upon causal stability.

Commonly, stabilize is the identity function, but it can be quite involved. A detailed description of how causal stability can be exploited is beyond the scope of this article. Details can be found in Baquero, Almeida, and Shoker [7], Baquero et al. [8]. Just to give an example, in a remove-wins set, a remove wins over a concurrent add of the same element and must be kept in the PO-Log for some time in case a concurrent add shows up. But once the remove becomes causally stable, it can be discarded in some cases.

PO-Log-based Observed-remove Set with PO-Log Compaction. To exemplify how causality information can be used to perform PO-Log compaction, for an ORSet, it is true that:

- a subsequent add obsoletes a previous add of the same value;
- a subsequent remove obsoletes a previous add of the same value;
- a remove is made obsolete by any other timestamped operation: after having made other operations obsolete, the remove itself becomes redundant.

So, the obsolete relation for the ORSet can be defined as:

$$\begin{aligned}
\mathsf{obsolete}((t, [\mathsf{add}, v]), (t', [\mathsf{add}, v'])) &= t < t' \wedge v = v' \\
\mathsf{obsolete}((t, [\mathsf{add}, v]), (t', [\mathsf{remove}, v'])) &= t < t' \wedge v = v' \\
\mathsf{obsolete}((t, [\mathsf{remove}, v]), x) &= \mathsf{True}.
\end{aligned}$$

The compacted PO-Log not also saves space but also allows a substantially simpler eval:

$$\mathsf{eval}(\mathsf{elements}, s) = \{v \mid (t, [\mathsf{add}, v]) \in s\}.$$

On Partition Tolerance. A possible criticism that can be made regarding the use of causal stability is that operations stop becoming stable under partitions. We observe that, while for strongly consistent (CP) systems a partition causes unavailability, in pure op-based CRDTs, a partition only impacts state size, which may grow while partitioned, but does not impact availability itself. If the partition does not take long, then it may not cause much harm and the system will recover. We also note that long partitions will be a problem in general for op-based (not only pure op-based) CRDTs, due to the need for buffering messages by the reliable messaging middleware. If long partitions or disconnected operation is the norm, then state-based CRDTs are preferable.

5 State-based CRDTs

State-based Propagation and Conflict Resolution. The state-based approach propagates replica states to other replicas. If the self state and a received state conflict, then they must be merged (reconciled) into a new state reflecting the conflict resolution. In classic optimistic replication, this was performed in some ad hoc way through a user-defined merge procedure.

Contrary to the operation-based approach, where each operation is immediately propagated, propagating states, which can be large, is performed much less frequently. Also, as opposed to

the usual causal broadcast to a well-known group of replicas in the operation-based approach, state-based propagation can have many forms, e.g., through an epidemic propagation [28] to an unknown set of participants. Given that states can be propagated in many different ways and arrive at a replica in different orders, conflict resolution (merge) should be:

— deterministic: result as a function of inputs (current and received state);
— obviously a commutative function;
— associative: to give the same result when merging in different orders;
— idempotent: merging equal states produces that state;
— merging with an older state produces the current state;
— monotonic: merging with a "newer" state produces a "newer" state.

The solution is to adopt for the replica state the mathematical concept of *join-semilattices* [13]. We now present a very brief introduction to the concepts of order and lattices relevant for CRDTs.

5.1 Lattices and Order

Partially ordered sets (posets). A poset has a binary relation $\sqsubseteq$, which is:

— (reflexive) $p \sqsubseteq p$;
— (transitive) $o \sqsubseteq p \land p \sqsubseteq q \Rightarrow o \sqsubseteq q$;
— (anti-symmetric) $p \sqsubseteq q \land q \sqsubseteq p \Rightarrow p = q$.

Two unordered elements are called concurrent:

— (concurrent) $p \parallel q \Leftrightarrow \neg(p \sqsubseteq q \lor q \sqsubseteq p)$.

Some posets have a *bottom* ($\bot$), an element smaller than any other:

$$\forall p \in P \cdot \bot \sqsubseteq p.$$

Join-semilattices. An upper bound of some subset S of some poset P is some u in P greater or equal than any element in S:

$$\forall s \in S \cdot s \sqsubseteq u.$$

If the *least upper bound* of S exists (an upper bound smaller than any other), then it is called the *join* of S, denoted $\bigsqcup S$. For two elements, $\bigsqcup\{p, q\}$ is denoted by $p \sqcup q$. A poset P is a join-semilattice if $p \sqcup q$ is defined for any two elements p and q in P. The join operator has the following properties:

— (idempotent) $p \sqcup p = p$;
— (commutative) $p \sqcup q = q \sqcup p$;
— (associative) $o \sqcup (p \sqcup q) = (o \sqcup p) \sqcup q$.

Examples and Non-examples of Join-semilattices. Some examples of join-semilattices are:

— natural numbers $\mathbb{N}$: $\sqsubseteq = \leq$; $\sqcup = \mathsf{max}$; $\bot = 0$;
— Booleans $\mathbb{B}$: $\mathsf{False} \sqsubset \mathsf{True}$; $\sqcup = \lor$; $\bot = \mathsf{False}$;
— any totally ordered set (called a *chain*).

Some posets that are not join-semilattices are:

— unordered posets (called *antichains*), where all elements are incomparable;
— strings under prefix ordering (e.g., *small* $\sqsubseteq$ *smallest* $\parallel$ *smaller*), where all concurrent elements are not joinable.

Lattice Compositions. An advantage of adopting (join-semi) lattices for the replica state is that we can use standard lattice composition constructs for obtaining complex states from simpler states. Some examples are given in Figure 10: the (Cartesian) product of two lattices, where join is performed component-wise; the lexicographic product of A and B, when elements are lexicographic pairs, compared first by the left-side component and only by the right-side one to break ties,

Cartesian product $A \times B$

$$(a_1, b_1) \sqsubseteq (a_2, b_2) = a_1 \sqsubseteq a_2 \wedge b_1 \sqsubseteq b_2$$
$$(a_1, b_1) \sqcup (a_2, b_2) = (a_1 \sqcup a_2, b_1 \sqcup b_2)$$

(join-semilattices A and B)

Lexicographic product $A \boxtimes B$

$$(a_1, b_1) \sqsubseteq (a_2, b_2) = a_1 \sqsubset a_2 \vee (a_1 = a_2 \wedge b_1 \sqsubseteq b_2)$$
$$(a_1, b_1) \sqcup (a_2, b_2) = \begin{cases} (a_1, b_1) & \text{if } a_2 \sqsubset a_1 \\ (a_2, b_2) & \text{if } a_1 \sqsubset a_2 \\ (a_1, b_1 \sqcup b_2) & \text{if } a_1 = a_2 \\ (a_1 \sqcup a_2, \bot) & \text{if } a_1 \parallel a_2 \end{cases}$$

(when B has a bottom or A is a chain)

Powerset $\mathcal{P}(S)$

$$\sqsubseteq = \subseteq$$
$$\sqcup = \cup$$

(set S)

Function space $A \to B$

$$f \sqsubseteq g = \forall x \in A \cdot f(x) \sqsubseteq g(x)$$
$$(f \sqcup g)(x) = f(x) \sqcup g(x)$$

(set A to join-semilattice B)

(map $K \rightharpoonup V$, V with bottom, where missing keys yield bottom is specially useful)

Fig. 10. Some classic lattice compositions.

being defined when B has a bottom or A is a chain; the powerset of any set, being join the set union; and the function space from any set A to lattice B, where both comparison and join are performed pointwise. Maps, where missing keys implicitly yield bottom can be considered a special case of functions, being especially useful. These and other examples are presented by Baquero et al. [6].

5.2 Basics of State-based CRDTs

State-based CRDTs. Like for op-based CRDTs, in state-based CRDTs, update operations are applied locally, and queries can be answered from the local state. The essential difference is how knowledge about operations is propagated. The propagation is indirect, through states: An operation updates the local state; from time to time, replicas send their full state to other replicas.

For this strategy to work, the essential aspect of state-based CRDTs is making the state be a join-semilattice. Replicas merge the received state using the join operator $\sqcup$. The properties of join give state-based CRDTs very good fault tolerance in terms of messaging faults: They work in unreliable networks subject to message loss, duplication, and reordering. Also, sending the full state ensures causal consistency, as it ensures a transitive propagation of the causal past.

Update Operations and State Mutators. In the state-based model, update operations invoke a *state mutator*, which returns the new state. A state mutator must be an *inflation*:

— (inflation) $x \sqsubseteq f(x)$;
— (strict inflation) $x \sqsubset f(x)$.

This, together with join being used for merging states, gives the essential property that replica states are always "going up" in the partial order, i.e., state evolves monotonically:

— when updates are locally invoked;
— when a remote state is received and merged through join.

Monotonic evolution of state is essential and what gives nice fault tolerance properties: A newer state always subsumes an older state; if a message is lost, then a newer one will subsume it; an old message can arrive as a duplicate causing no harm.

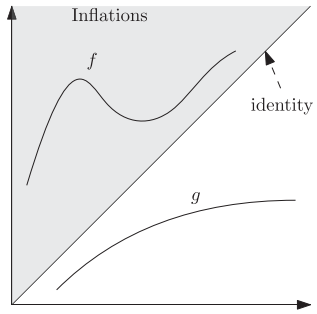

Fig. 11. Inflations vs. monotonic functions.

$$
\begin{aligned}
\mathsf{Advancer}\langle E\rangle &= E \rightharpoonup \mathbb{N} \\
s^0 &= \{\} \\
\mathsf{advance}(e, s) &= s\{e \mapsto \max\{s[e], 1 + \max\{v \mid (k, v) \in s \wedge k \neq e\}\}\} \\
\mathsf{ahead}(s) &= \{k \mid (k, v) \in s \wedge v = \max\{x \mid (\_, x) \in s\}\}
\end{aligned}
$$

Fig. 12. Sequential "advancer" data type, reusable as state-based CRDT.

Inflations vs. Monotonic Functions. Sometimes there is some confusion about monotonicity. Erroneously, many places say mutators need to be monotonic, when in fact mutators need to be inflations. Being monotonic is not necessary nor sufficient, as illustrated in Figure 11: f is an inflation but not monotonic, and g is monotonic but not an inflation. Decrement ($f(x) = x - 1$) is monotonic but not an inflation. What is monotonic is state evolution over time, as a result of applying mutators or join.

Reusing Sequential Data Types. As for op-based CRDTs, for some data types, we can have trivial state-based CRDTs, with state and operations being the same as the corresponding sequential data type. But for state-based CRDTs, this is considerably more difficult. It is necessary that:

— the state is already a join-semilattice (can be ordered, if not already, to be one);
— update operations are inflations;
— update operations are idempotent.

One may think that another condition for reuse would be that that the sequential data type should only have commutative operations, as for op-based CRDTs. This is not true. A counter-example is a sequential data type, presented in Figure 12, whose state is a map from some unspecified set of keys to integer values; with a single operation $\mathsf{advance}(k)$, which makes the corresponding key have a value at least one more than any other key, by increasing the value by the minimum needed (possibly 0); and with a single query $\mathsf{ahead}(k)$, which returns the set of keys with the maximum value (an empty set for the initial state or a singleton afterwards for sequential executions).

The update operation is not commutative as, starting from the initial state (empty map):

$$\mathsf{advance}(a); \mathsf{advance}(b) \quad \text{leads to} \quad \{a \mapsto 1, b \mapsto 2\},$$

and

$$\mathsf{advance}(b); \mathsf{advance}(a) \quad \text{leads to} \quad \{a \mapsto 2, b \mapsto 1\}.$$

But the original state (a map from some set to integers) is already a lattice, with join as the pointwise maximum, and the update operation is an inflation and idempotent. We can simply reuse

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

Fig. 13. State-based grow only set $\mathsf{GSet}\langle E\rangle$

the sequential data type state and operations and the original lattice join for merge. Concurrent execution of the operations above would be:

$$\mathsf{advance}(a) \;||\; \mathsf{advance}(b) \quad \text{leads to} \quad \{a \mapsto 1, b \mapsto 1\}.$$

This example is interesting, as it shows that for this data type concurrent executions can lead to states unreachable by any sequential execution. But this CRDT reuses the sequential data type and preserves its sequential semantics, while converging and leading to reasonable outcomes for concurrent executions: Concurrent invocations of advance may lead to ties, unlike sequential executions, which is a reasonable generalization of the sequential behavior.

In this example, reuse was possible because the update operation, even though not commutative, was an inflation and idempotent. For reuse, it is obviously necessary that all updates are inflations. Having to be idempotent is also easy to show, given the desire for permutation equivalence. Given a non-idempotent inflation o, the desired outcome for $o(s) \;||\; o(s)$ is $o(o(s))$, but

$$o(s) \;||\; o(s) \quad \text{converges to} \quad o(s) \sqcup o(s) = o(s) \sqsubset o(o(s)).$$

So, the reuse of a non-idempotent operation from the sequential data type would violate permutation equivalence. The simplest example of the impossibility of reusing non-idempotent operations is the counter data type: While a sequential counter could be trivially reused as an op-based CRDT, given commutativity, it cannot be reused for a state-based CRDT, given that increment is not idempotent.

Grow-only Set. The classic example of a trivial state-based CRDT is the **grow-only set (GSet)**, shown in Figure 13: a set having only the add update, where all operations, whether mutator (add) or queries (elements and contains) correspond to the original sequential data type operations, and the state is simply a set, as for the sequential data type. Merging replica states is simply the set union. This is possible because a set is a lattice and the only update operation (add) is an idempotent inflation. This is an example of an *anonymous CRDT*, where there is no need for node identifiers in the state.

Single-writer Principle and Named CRDTs. State-based CRDTs are less prone to have trivial implementations, namely, under the presence of non-idempotent operations. This precludes even seemingly trivial data types, such as counters, from having correspondingly trivial state-based CRDTs.

A powerful strategy in concurrent programming is the single writer principle. It amounts to each variable being updated only by a single process, has long been used to obtain elegant designs, such as the Bakery algorithm [51], and is especially relevant to the design of state-based CRDTs [31]:

— the state is partitioned in several parts;
— each replica updates a part dedicated exclusively to itself;
— the state is joined by joining respective parts.

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

Fig. 14. State-based GCounter.

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

Fig. 15. State-based PNCounter.

Unique replica identifiers can be used to partition the state, which becomes a map from IDs to parts. CRDTs that use replica IDs in the state can be called *named CRDTs*.

State-based GCounter. Contrary to op-based counters, state-based counters cannot use the trivial sequential implementation due to the non-idempotency of the increment operation. The state-based GCounter, shown in Figure 14, makes use of the single writer principle. The state maps replica identifiers to integers, the inc mutator for replica i increments the self entry (i), and join is the pointwise max. The counter is thus similar in structure to a version vector [67].

State-based PNCounter. A **positive-negative counter (PNCounter)**, having both increment and decrement operations, cannot have the same state as the GCounter, because decrement is not an inflation. This is solved through a pair of GCounters (product composition): Increments and decrements are tracked separately. The counter value is obtained as the difference. This CRDT is presented in Figure 15. (In practice, implementations use a single map to pairs.)

5.3 Causal CRDTs

The Problem of Forgetting Information. Many times, we want to remove things. An example is a set with add and remove operations. But, as we must use inflations for state mutators, in this set example, we cannot use a single set for the state and remove elements from it in the remove mutator. So, while a grow-only set is trivial, this more general set is not.

A common approach to solve this limitation is to use tombstones to mark removal, but it has the significant drawback of making the state grow forever. An example of such approach is the **2P-Set (two-phase set)** [77] CRDT: It allows add and remove, but once removed, an element cannot be re-added. The implementation uses a pair of sets to track adds and removes, each set only growing, making the state always growing and eventually large after some time, even when the number of elements considered to be in the set is small.

Causal CRDTs. A general approach to allow removing information while avoiding tombstones is used in *Causal CRDTs* [4], drawing inspiration from *Dotted Version Vectors* [69]. The state has

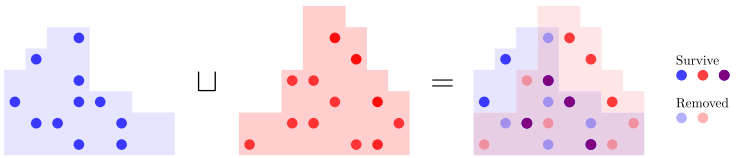

Fig. 16. Joining causal CRDTs.

two components: a *dot store* and a *causal context*. The **dot store (DS)** is a container for data-type-specific information; each information item is tagged with a unique event identifier, a *dot*, which is a pair (replica-identifier × counter). The **causal context (CC)** represents the causal history: the IDs of all visible updates, normally encoded by a version vector.

Remark. A dot is not merely a unique identifier. Being unique would be enough for the op-based ORSet CRDT and could have different forms, such as a pair (Lamport-timestamp × replica-identifier) or a UUID. A dot is a specific form of unique identifier, appropriate to be tested as belonging to a compressed causal history in the form of a version vector.

Causal CRDTs have a pair of components for state but they are not independent, as in a Cartesian product. The two components are related, conveying the knowledge that an information item with identifier covered by the causal context and not present in the dot store has already been removed.

Joining Causal CRDTs. To merge the information represented by two replica states, the join cannot be the standard one for product composition, i.e., we do not have the lattice resulting from the Cartesian product of the DS and CC components, but a more interesting one. The join is illustrated in Figure 16. The dot store that results from a join of x and y:

— has dots from x's DS not covered by y's CC;
— has dots from y's DS not covered by x's CC;
— has the dots present in both DSs.

The causal context upon a join is the join of the causal contexts.

Some Dot Stores. A dot store is a container; we can have different types of such containers. Three important dot stores are: DotSet – a set of dots; $\mathsf{DotFun}\langle V\rangle$ – a map from dots to values in some lattice V; and $\mathsf{DotMap}\langle K, V\rangle$ – a map from keys in some arbitrary set K to a dot store V:

$$\begin{aligned} \mathsf{DotSet} : \mathsf{DS} &= \mathcal{P}(\mathbb{I}\times\mathbb{N}) \\ \mathsf{DotFun}\langle V : \mathsf{Lattice}\rangle : \mathsf{DS} &= \mathbb{I}\times\mathbb{N} \rightharpoonup V \\ \mathsf{DotMap}\langle K,\ V : \mathsf{DS}\rangle : \mathsf{DS} &= K \rightharpoonup V. \end{aligned}$$

DotMaps are particularly powerful, as they allow map CRDTs that embed CRDTs as values:

$$\mathsf{DotMap}\langle K_1,\ \mathsf{DotMap}\langle K_2,\ \mathsf{DotFun}\langle\mathcal{P}(E)\rangle\rangle\rangle.$$

Joining Causal CRDTs—for Different Dot Stores. We can now define the join operator for causal CRDTs, considering the three kinds of dot stores above. The join, shown in Figure 17, simply returns the dot store containing the appropriate surviving items, as illustrated in Figure 16, paired with the join of the causal contexts.

Observed-remove Set ORSet$\langle E\rangle$. To illustrate the power of causal CRDTs, we give the example of a state-based observed-remove set, shown in Figure 18, where the state uses a DotMap from elements to DotSets. The add mutator replaces, for the key corresponding to the element being added, any set of dots by a new singleton. Remove simply removes the key from the map (domain subtraction), with no need for the introduction of a new event in the causal context. The causal

$$
\begin{aligned}
\mathsf{Causal}\langle T : \mathsf{DS}\rangle &= T \times \mathsf{CausalContext} \\
\sqcup &: \mathsf{Causal}\langle T\rangle \times \mathsf{Causal}\langle T\rangle \to \mathsf{Causal}\langle T\rangle \\
\\
\textbf{when} \quad & T : \mathsf{DotSet} \\
(s,c) \sqcup (s',c') &= ((s \cap s') \cup (s \setminus c') \cup (s' \setminus c), c \cup c') \\
\\
\textbf{when} \quad & T : \mathsf{DotFun}\langle \_ \rangle \\
(m,c) \sqcup (m',c') &= (\{d \mapsto m[d] \sqcup m'[d] \mid d \in \mathrm{dom}\, m \cap \mathrm{dom}\, m'\} \cup \\
&\quad \{(d,v) \in m \mid d \notin c'\} \cup \{(d,v) \in m' \mid d \notin c\}, c \cup c') \\
\\
\textbf{when} \quad & T : \mathsf{DotMap}\langle \_, \_ \rangle \\
(m,c) \sqcup (m',c') &= (\{k \mapsto \mathsf{v}(k) \mid k \in \mathrm{dom}\, m \cup \mathrm{dom}\, m' \wedge \mathsf{v}(k) \neq \bot\}, c \cup c') \\
&\quad \textbf{where } \mathsf{v}(k) = \mathrm{fst}\,((m[k],c) \sqcup (m'[k],c'))
\end{aligned}
$$

Fig. 17. Join operator for causal CRDTs, considering each kind of dot store.

$$
\begin{aligned}
\mathsf{ORSet}\langle E\rangle &= \mathsf{Causal}\langle \mathsf{DotMap}\langle E, \mathsf{DotSet}\rangle\rangle \\
\mathrm{add}_i(e,(m,c)) &= (m\{e \mapsto \{(i, c[i]+1)\}\}, c\{i \mapsto c[i]+1\}) \\
\mathrm{remove}_i(e,(m,c)) &= (\{e\} \lhd m, c) \\
\mathrm{elements}((m,c)) &= \mathrm{dom}\, m
\end{aligned}
$$

Fig. 18. State-based observed-remove set $\mathsf{ORSet}\langle E\rangle$.

context is in the form of a version vector. The join, not shown, is previously defined for the DotMap (and DotSet) case:

— it keeps concurrently added elements;
— it removes elements removed elsewhere if no dot survives.

Remark. Although not needed (and not done here), it would be advantageous to generate a new event, and advance the causal context, in a remove operation, even if there is no new dot being stored. The cost would be minuscule for a version vector representation of the causal context, and it would allow trivially comparing replica versions by only comparing the causal context, without the need to traverse the dot store, as in this implementation.

6 Delta State CRDTs

Operation- vs. State-based Approaches. Operation-based approaches have small messages but have strong requirements from the underlying messaging layer: It does not work if messages are lost or duplicated. However, state-based approaches work under weak messaging assumptions but send full states, which may have considerable sizes, causing a large messaging cost. Can we have the best of both worlds?

Delta-state CRDTs. In delta state CRDTs [4] the state is, like in normal state-based CRDTs, a join-semilattice, but messages are built from *delta-states*, which are states, typically small, which will make messages themselves hopefully small.

The essential difference is that a delta CRDT has *delta-mutators*, which return delta-states, to be: (1) joined with the current state; (2) joined with other delta-states, forming *delta-groups d*, to

State-based

$$X_i \leftarrow m(X_i)$$
$$X_i \leftarrow X_i \sqcup X_j$$

Delta-state-based

$$X_i \leftarrow X_i \sqcup m^\delta(X_i)$$
$$X_i \leftarrow X_i \sqcup d$$

Fig. 19. Normal state-based vs. delta-state-based approaches.

State-based GCounter

$$\mathsf{inc}_i(m) = m\{i \mapsto m[i] + 1\}$$

Delta-state-based GCounter

$$\mathsf{inc}_i^\delta(m) = \{i \mapsto m[i] + 1\}$$

State-based ORSet$\langle E \rangle$

$$\mathsf{add}_i(e, (m, c)) = (m\{e \mapsto d\}, c \cup d) \quad \textbf{where } d = \{(i, \max_i(c) + 1)\}$$
$$\mathsf{remove}_i(e, (m, c)) = (\{e\} \lhd m, c)$$

Delta-state-based ORSet$\langle E \rangle$

$$\mathsf{add}_i^\delta(e, (m, c)) = (\{e \mapsto d\}, d \cup m[e]) \quad \textbf{where } d = \{(i, \max_i(c) + 1)\}$$
$$\mathsf{remove}_i^\delta(e, (m, c)) = (\{\}, m[e])$$

Fig. 20. State- vs. delta-state examples: GCounter and ORSet.

send in messages. For each mutator m of a state-based CRDT, we can define a delta-mutator m^δ such that:

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

Figure 19 summarizes the approaches. While for normal state-based CRDTs mutators return the next state (and need to be defined as inflations), for delta-state CRDTs, delta-mutators return a value (delta) to be joined to the current state (causing an inflation). Also, state-based CRDTs join full states received as messages, while delta-state CRDTs join delta-groups (join of several deltas).

State vs. Delta-state—Examples. For each given state-based CRDT, we can define one (or more) corresponding delta-state-based CRDT. The state remains the same join-semilattice, and we only need to define corresponding delta-mutators. Figure 20 shows two examples: a GCounter and an ORSet. The corresponding delta-mutators returns, for the GCounter, a map with just the self-entry for the replica, instead of the full map. For the ORSet, it returns a pair where the dot store is a singleton or empty map, for add and remove, respectively, and the causal context has dots representing just the element being added/removed, plus a new dot for add.

This means that delta-states will be typically very small, with either singletons or very small sets. It also means that, unless an anti-entropy mechanism ensuring causal-consistency is used, causal contexts in replica states will not be downward closed and, therefore, not representable by a version vector. (In this example, causal contexts are a set of dots.) They can, nevertheless, be represented in a compact way by a version vector plus some extra dots.

Naive Delta Propagation Algorithm. As the standard state-based approach, delta-state CRDTs aim for allowing arbitrary topologies. Therefore, a basic propagation mechanism aims for transitive propagation of information. Such an algorithm is presented in Figure 21. It keeps a single "delta-buffer" to broadcast to neighbors; upon an update, it joins the delta produced by the

durable state:
$X_i \in S$, CRDT state; initially $X_i = \bot$
volatile state:
$D_i \in S$, delta-buffer; initially $D_i = \bot$
on operation$_i(m^\delta)$
let $d = m^\delta(X_i)$
$X_i \leftarrow X_i \sqcup d$
$D_i \leftarrow D_i \sqcup d$

on receive$_{j,i}(d)$
$X_i \leftarrow X_i \sqcup d$
$D_i \leftarrow D_i \sqcup d$
periodically
let m = choose$_i(X_i, D_i)$
for j **in** neighbors$_i$ **do**
send$_{i,j}(m)$
$D_i \leftarrow \bot$

Fig. 21. Naive propagation algorithm for delta CRDTs, for replica i.

Join-irreducible	Join decomposition
$x = \bigsqcup F \Rightarrow x \in F$	$D \subseteq \mathcal{J}(L) \wedge \bigsqcup D = x$
Irredundant join decomposition (IJD)	**Unique IJD for distributive lattices**
$D' \subset D \Rightarrow \bigsqcup D' \sqsubset \bigsqcup D$	$\Downarrow x = \max\{r \in \mathcal{J}(L) \mid r \sqsubseteq x\}$
Difference	**Optimal delta-mutator**
$\Delta(a,b) = \bigsqcup\{y \in \Downarrow a \mid y \not\sqsubseteq b\}$	$\mathrm{m}^\delta(x) = \Delta(\mathrm{m}(x), x)$

Fig. 22. Join decompositions and optimal deltas.

delta-mutator to both CRDT state and delta-buffer; a delta-group received in a message is joined to both CRDT state and delta-buffer; periodically, it broadcasts the delta-buffer to neighbors and resets it to bottom. This algorithm would even ensure causal consistency under reliable FIFO messaging, but it fails to ensure it under the weaker messaging guarantees desired for state-based CRDTs. Unfortunately, this algorithm is too naive: The delta-buffer, even being periodically reset, easily tends to grow and become the full state.

Problems with Naive Delta Propagation. Unfortunately, the naive transitive propagation causes much redundancy, which makes messages and buffers converge to the full state. Resetting the buffer does not help if messages become large and get joined subsequently.

There are two main problems [32]: (1) Deltas are re-propagated back to where they came from; (2) a delta-group received is wholly joined to the local delta-buffer even if it is already mostly reflected in the state. The solution is a more sophisticated algorithm that: (1) avoids back-propagation of delta-groups by tagging the origin of each message; (2) removes redundant state in received delta-groups (already reflected in the local state). But how can this redundant state be identified?

Optimal Deltas and Smart Propagation through Join Decompositions. The solution to the problem of how to extract "new information" from a received delta-group and also to the fundamental problem of how to define optimal delta-mutators (that produce the smallest delta possible) can be found in results in lattice theory developed by Birkhoff [12], namely, the concept of *join decompositions*.

The main concepts and results are summarized in Figure 22. An element is said to be join-irreducible if it cannot result from the join of a finite number of elements not containing itself. A join decomposition of some element x is a set of join-irreducible elements whose join produces x. An **irredundant join decomposition (IJD)** is when no element is redundant (removing any element produces a smaller result when joined).

Birkhoff's representation theorem establishes a correspondence between an element of a finite distributive lattice and the downward closed set of the join irreducibles below it. This down set

Table 2. Comparison between CRDT Approaches

	operation-based		state-based	
	general	pure-operation	standard	delta-state
reuse sequential ADTs	commutative ops		idempotent inflations	
open vs. closed systems	fixed set of participants		dynamic independent groups	
partition tolerance	unsuitable for long partitions		good	
state size	independent of replicas		component linear with replicas	
metadata amortization	transparent (middleware)		explicit (container)	
message size	normally small	small	large	possibly small
messaging (usual)	causal broadcast		epidemic at-least-once	
causal consistency	from messaging		for free	needs care
causal stability	useful	important	not available in general	

is isomorphic to the set of its maximals, which is the unique irreducible join decomposition. (A distributive lattice is a poset with not only the join but also its dual *meet* and where one distributes over the other.)

For most CRDTs, the state is not merely a join-semilattice but a distributive lattice. Even if the state normally belongs to infinite lattices, we can apply the result to CRDTs [32] applying it to finite quotient sublattices. This allows obtaining the unique IJD of x as the maximals of the set of irreducibles below x.

Having unique IJDs, we can define the difference between two states a and b: It is the join of the elements in the unique IJD of a not below b. (As an example, the difference becomes simply set difference when the lattice is a powerset.) The difference to the current state can be used to extract "new" information from a received delta-group, and the optimal delta-mutator can be defined as the difference between what the original mutator produces, $m(x)$, and the current state x.

Inter-datacenter Delta Propagation in Geo-replicated Deployments. Delta propagation algorithms for the general case of a network of nodes require considerable care to be efficient. But practical deployments of CRDTs may be in more restricted scenarios, which can be exploited to simplify communication. One such case is having just a few replicas, one in each of a small number of nodes (e.g., datacenters) connected in a full mesh.

In this case, the system can run, by default, without transitive propagation: Each replica will only send its own deltas to all others. A few simple ingredients will allow an efficient causal delta-propagation algorithm: merging only the self-issued deltas in delta-groups; starting a new delta-group after applying a received remote one; keeping track (with a version vector) of the logical time when each delta-group starts to send it together with the delta-group; and delay applying received delta-groups until causal dependencies are met. This avoids the need to perform join-decompositions to achieve efficiency, as in a more general delta-propagation algorithm. It amounts to performing a kind of causal broadcast, where the unit of propagation is a delta-group, with the advantage of (1) allowing compression due to delta-merging; (2) the possibility of, given a long network partition, discarding the delta-groups and falling back to sending the full-state when the partition heals.

7 Comparison of the Approaches

A comparison of the approaches is now made regarding issues such as open vs. closed systems, partition tolerance, and state size (including the issue of amortizing metadata overhead over several objects). A summary of these aspects (as well as others already discussed) is presented in Table 2. We start by a taxonomy, discussing the dual nature of some CRDTs.

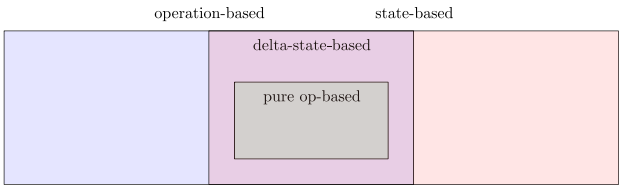

Fig. 23. Taxonomy of CRDTs, showing the dual nature of some CRDTs as both op-based and delta-state-based, including pure op-based CRDTs.

Taxonomy of CRDTs. Some CRDTs cannot be state-based, as the state is not a join-semilattice, not supporting merging states, or effect is not an inflation. However, any state-based CRDT could be considered, vacuously, op-based, by defining prepare as the state mutator and effect as the join operation. But this is not useful for taxonomy purposes and would make no sense in practice. Therefore, we should classify as op-based CRDTs only those in which prepare does not return the full state. Interestingly, delta-state CRDT implementations can be repurposed to the op-based model, fitting this restriction, and can be considered to have a dual nature as both state- and op-based. Given operation o, corresponding delta mutator m^δ_o, and current state s, we can define $\text{prepare}(o, s) \doteq m^\delta_o(s)$ and effect $\doteq \sqcup$. This means that we can use op-based propagation for delta-state CRDTs. It would be, however, typically worse, as no delta-merging before propagation could be exploited, and delta-CRDTs do not need the stronger delivery guarantees used for op-based propagation.

Conversely, for some op-based CRDTs the state happens to be a join-semilattice, prepare returns an element of that lattice, and effect is the same for all operations and amounts to the lattice join. Such CRDTs, even if originally defined under the op-based model, can switch to use a delta-based propagation model (possibly requiring causal delta-propagation, depending on the case), gaining flexibility and possibly higher efficiency, defining $m^\delta_o(s) \doteq \text{prepare}(o, s)$ and $\sqcup \doteq \text{effect}$. Going in this direction is practically useful, not just a taxonomic curiosity. Some List CRDTs, discussed in Section 9.2, have this dual nature and can choose either model.

Pure op-based CRDTs are a special case. In the naive form, without PO-Log compaction, the state is a lattice (set of causally tagged operations) and the result of prepare, tagged with causality information, can be considered a delta. We can define $m^\delta_o(s) \doteq \{(t, o)\}$ and $\sqcup \doteq \cup$, where t is the causal timestamp assigned to operation o. For this, we need to expose the causal context present in the TCB middleware to the CRDT to generate timestamps at the source. This means that naive pure-op-based CRDTs can use a more flexible delta-state-based propagation. They would typically need causal delta propagation for semantic reasons and also to make causal tagging simple. PO-Log compaction can also be achieved in this model but requires some care. Namely, we cannot simply use the set of tagged operations for lattice, as compaction would cause a deflation. This would cause removed operations to reappear, incorrectly, upon a merge. But we can make the set of tagged operations a Dot Store and, together with the exposed causal context, obtain a causal CRDT, where compactions become inflations and merge works correctly.

Given these observations, we obtain the taxonomy of CRDTs presented in Figure 23.

Open vs. Closed Systems. Operation-based CRDTs, using causal broadcast, assume a known set of participants. Dynamic joining or retiring of replicas can possibly be done, but it is a complex operation, needing some care. But two independent groups of replicas that have evolved independently cannot merge. (For realistic implementations, not considering naive implementations that keep all history of operations, or those that are effectively state-based CRDTs.) However, state-based CRDTs suit an unknown set of participants and allow trivial dynamic joining, leaving, or merging

of independent groups of replicas. The only requisite, for named CRDTs, is to have globally unique replica identifiers, a reasonable assumption.

Partition Tolerance. Both kinds of CRDTs are network-partition-tolerant in terms of operations being always available, locally, but a partition has significantly different consequences for each case. In operation-based CRDTs there is a transient cost that is linear with the partition duration, having to do with the need to store operations not yet delivered to all. Thus, they may be unsuitable for large weakly connected systems prone to long partitions, which will cause unbounded memory consumption. State-based CRDTs are better in this regard, as each replica is completely autonomous, with no need to track individual operations. Each operation is immediately incorporated in the state, and the approach works well even under long network partitions. State-based CRDTs are thus more suitable for autonomous operation over unstructured networks with poor connectivity.

State Size. Operation-based CRDTs can frequently have smaller states, independent of the number of replicas, even having non-idempotent operations (e.g., a counter). This comes from the power from delegating to a reliable message delivery mechanism. Even though the size can be, in theory, in the worse case, linear with the number of replicas, in practice, that is not an issue; e.g., an ORSet with replicas issuing concurrent adds on the same elements could have size temporarily linear with the degree of replication, but in practice that would be unlikely to happen for a substantial proportion of elements in the set at the same time; moreover, subsequent operations would reduce state size, bringing it to an effectively constant small size per element.

State-based CRDTs assume the worst case of unreliable messaging. This leads normally to some state component that grows linearly with the number of replicas. That may be very relevant, e.g., for a counter, where having a map from replica IDs to integers is much more space than a single integer; or less relevant, e.g., for a large set, implemented as a causal CRDT with a DotMap, where the cost of the causal context pales in comparison with the large number of elements in the set and where the meta-data cost per set element (a set of dots, frequently a singleton) is small. In general, we can say that for small data types (scalar-like, e.g., counters, or small container-like, e.g., sets), state-based CRDTs have significant overhead (linear with the number of replicas), while for large container-like data types (sets, maps, lists), the overhead for state-based CRDTs becomes less of a problem.

Amortizing Metadata Overhead over Many Objects. As several objects of a data type are frequently used together, another question is whether the overhead in meta-data can be amortized over several objects. In this respect, there is some difference in the approaches. In operation-based CRDTs, the cost (e.g., a version vector in some implementations of causal broadcast) is independent of the number of CRDT objects and is amortized by all objects (even of different CRDT types) that use the middleware. This cost can be almost none for operation-based CRDTs that do not require knowledge about happens-before (like for counters or that extract relevant information from the CRDT state itself in prepare), e.g., using a causal broadcast based on a tree topology with FIFO channels [16], which requires negligible metadata even for a large number of participants. Pure op-based CRDTs cannot use this strategy, as they need knowledge about happens-before to be provided by the causal broadcast middleware (prepare is unable to extract information from the CRDT state). State-based CRDTs not only have non-amortizable metadata cost per object in some cases (like counters) but cannot have a transparent amortization as op-based CRDTs can. For causal CRDTs (like an ORSet), it is possible to amortize metadata cost over many objects, but only by explicitly putting those objects inside a container; e.g., if we have many ORSets, then it is possible to place them inside a map CRDT, thereby sharing a single causal context over those

many objects. But this is not transparent, needs a refactoring effort, being undesirable from the software engineering perspective.

8 Identity Management towards Scalability

Apart from simple, anonymous CRDTs such as GSets, most CRDTs make use of unique replica identifiers, with each replica using its own identifier to generate unique IDs, in the form of dots, and updating part of the state that "belongs to it." Causal CRDTs use a causal context in the form of a version vector, whose size is linear with the number of replicas. State-based counters have a similar state. This makes the growth of maps where the keys are replica identifiers the main obstacle to scaling CRDTs to a large number of participants. We discuss here some techniques that can be used to address it.

8.1 CRDT-based Identity Management

Identity Containment and Handoff. For CRDTs such as small ORSets or for counters, the size of the set of IDs determines the number of map entries and, therefore, the state size. Scaling a state-based counter CRDT to many participants is a difficult problem. It was addressed in Handoff Counters [3] by using a hierarchical structure, in which only the entries of the small number of replicas at tier 0 (e.g., datacenters) are propagated to all replicas. Any other state generated using the ID of replicas from other tiers is just temporary, being propagated under peer-to-peer interactions but remaining contained, not propagated to the whole system, and being eventually garbage collected through local interactions. The more intricate aspect is the handoff, which migrates accounted values towards lower tiers, making all increments eventually be accounted in the entries for tier 0 IDs. This approach is more generally applicable to other data types with associative and commutative operations.

Identity Borrowing. Handoff Counters are considerably complex, and the generality of allowing many tiers may not be needed. Borrow Counter [33] is a simpler proposal, being a Causal CRDT that only distinguishes permanent replicas (typically servers) and transient replicas. A transient replica i may ask a permanent replica j to create a dot, based on the permanent replica ID j, for i to use as key in a map where to store increments within values. The transient replica may retire by flagging borrowed dots as inactive, upon which they are able to be acquired by j and garbage collected. The permanent state, namely, the causal context, grows only with the number of permanent replicas.

Renaming Operation Unique Identifiers. While the two previous approaches exploit the fungibility of increments, causal CRDTs such as ORSets identify individual operations using dots. For these, a possibility towards scaling, assuming a simple classification of replicas as clients or servers, is to: (1) rename dots based on client IDs to become based on server IDs; (2) only propagate globally the server-based dots. The causal context will only depend on the number of servers, achieving scalability to many clients. This approach is still under research; Marubayashi and Baquero [58] provide an example of a preliminary development.

8.2 Server-based Identity Management

Identity Reuse and Collapsing Identities. Another possibility is to exploit system assumptions, externally to CRDTs themselves. An example: Clients obtain replicas from a central server/service to use in a session involving peer-to-peer interactions; a client uses the replica during the session and then relinquishes the right to use it, handing it back to the server. A strongly consistent service can control replica use by clients, assign IDs to clients, keep track of IDs in use and the ones that became inactive (when clients end sessions), and only assign new IDs when no previously

generated ID is inactive. This allows the size of the ID set to grow only with the number of clients that concurrently interact and not the number of clients that have ever used the CRDT. This keeps the benefits of CRDTs, namely, allowing local peer-to-peer interactions. Partitions may only delay the possibility of reusing an identity, which in the worst case will cause a new identity to be generated. Moreover, if no session is active, for most state-based CRDTs that we discussed up to now, then the server could collapse all identities and dots into a single one. This would require an extra collapse operation to be provided in the CRDTs that, e.g., for a counter would sum all the entries and return a map with a single entry. We are not aware of these techniques having been described or if they have been used.

9 Practical Applications

CRDTs started being used in the industry for scenarios involving large scale or partition tolerance in a distributed setting. Early examples were in the League of Legends [41] game by Riot Games, using Riak CRDTs and implementing an Ejabberd CRDT library, and in SoundCloud's Roshi [79], a time-series event storage via a LWW-element-set CRDT. CRDTs also became supported in libraries or frameworks. Akka is a popular framework for building distributed applications, with emphasis on supporting the actor model, having added support [55] for state-based CRDTs, both standard and delta-state. An area that motivated much research and resulted in successful products was NoSQL databases. Pre-CRDT NoSQL data stores, such as Amazon's Dynamo, were difficult to use due to the need for ad hoc reconciliation of concurrent writes. Another area of application where CRDTs were successful has been collaborative editing. We now present examples in these two areas.

9.1 Databases

Following Amazon's Dynamo, Riak [49] was one of the first successful data stores that added support for CRDTs through Riak DT, which introduced several CRDTs, such as the Riak DT Map [18], possibly the first map where values themselves could be state-based CRDTs. It also provided the ORSet. Other applications were designed on top of Riak Core, such as Riak PG [59], a process group registry for Erlang using ORSets. Riak DT ORSets were used successfully by bet365 [66], an online gambling site, to scale their application. Redis is a very popular in-memory database with an emphasis on being a data structure store. It added support for active-active geo-distribution and CRDT-enabled databases [72]. AntidoteDB [1] is a database that served as reference platform and applies research from two European Projects, SyncFree and LightKone. It provides highly available transactions and geo-replication together with CRDTs. A related database, SwiftCloud [71], also supports transactions over CRDTs while supporting client-side partial replication and local execution. These systems allow transactions that read from causally consistent snapshots and apply updates atomically, exploiting CRDTs to allow merging updates from different transactions. This allows committing transactions that would have to abort under strong consistency requirements.

9.2 Collaborative Editing—the List Data Type

Collaborative editing is possibly one of the best examples of a topic involving a non-trivial data structure (List), where CRDTs achieved considerable success. The long line of research before CRDTs faced many difficulties and failed to overcome problems for the non-centralized scenario in a satisfactory way. It is well worth contrasting pre-CRDT and CRDT solutions for the List [5] data type. A possible API contains two update operations: $\mathsf{ins}(e, k)$ inserts an element e at position k (index, starting from 0), and $\mathsf{del}(k)$ deletes element at position k.

Before CRDTs—Position Transformation. For the List data type, given two operations concurrently issued at different replicas, applying them in different serial orders in different replicas does

not lead to convergence nor the intended result. The first approach to address this problem was to transform an operation before applying it remotely to obtain convergence and also the intended effect as when it was issued at the originating replica. This approach became known as the **OT (Operational Transformation)** approach, from the ***Distributed Operational Transformation (dOPT)*** [30] algorithm. Transformations modify positions (indexes) of operations; e.g., an $\mathsf{ins}(x, 5)$ issued at some replica i could be transformed to, say, $\mathsf{ins}(x, 7)$ before being applied at another replica j, to maintain the intended effect, given concurrent modifications at j that inserted two elements before position 5.

Unfortunately, transforming operation positions appropriately turned out to be much more difficult, complex, and error-prone than expected. While it is relatively manageable under centralized coordination [61], e.g., in Google Docs, it becomes dauntingly complex in the general case of non-centralized peer-to-peer replica synchronization, which is important for *local-first* applications [48].

Problems with the original dOPT algorithm were identified by Ressel et al. [73]. They proposed a framework involving two *transformation properties* (TP_1 and TP_2), to be satisfied by *transformation functions* to ensure correctness. TP_2 turned out very difficult to satisfy, and Imine et al. [45] showed that all initially proposed transformation functions did not satisfy it and proposed different transformation functions. Unfortunately, these were found to violate TP_2 and intention preservation [54]. Another proposal [64] enforces both TP_1 and TP_2 by relying on maintaining a *tombstone* when an element is deleted, defining *Tombstone Transformation Functions*.

CRDTs Approaches—Use Immutable Globally Unique Element Identifiers. To avoid the complexity and fragility of transforming positions, CRDT approaches assign globally unique identifiers to elements and use them in operations, instead of the position. These IDs can be sent in messages and used in other replicas to refer to elements, even if their position has changed. These IDs can be an explicit part of the API ("insert after element with ID x"), or they can be encapsulated in the data type representation and retrieved before being sent in messages. In a position-based API, an "insert at position k" can be implemented as "insert after ID of element at position $k-1$," or "before ID of element at position k," or "between the IDs of elements at positions $k-1$ and k."

CRDT approaches to Lists use basically a growing set of operations involving unique IDs for state (even if in practice performance optimizations are done, such as having hash tables and linked lists). They can be seen as both state-based CRDTs (state is a set) or operation-based CRDTs (propagate operations and apply effect). As adding an operation to a set is idempotent, the operations can also be seen as deltas, and exactly-once delivery is not needed. But as operations may refer to IDs in other operations in their causal past, practical implementations may need causal delivery. Nevertheless, this dual state- and op-based nature allows these CRDTs to be used with either kind of propagation; e.g., normally op-based propagation to have low latency of visibility, but also allowing long partitions and offline editing by updating local state and doing a state merge when the partition heals.

Collaborative editing based on CRDTs has achieved notable success and is being used in popular libraries, such as Automerge [24], based on RGA [74], and Yjs [46], based on a modified version of YATA [62], with a list of over 40 applications using it. The two main variants of these approaches use either some pre-existing dense total order or build a total order as operations are issued.

CRDTs with a Dense Total Order of Globally Unique Identifiers. A particular approach generates IDs from a totally ordered set. An insert becomes essentially generating an ID between the IDs of the "before" and "after" elements. To make an insert always possible, this approach uses a *dense total order*, where for any two identifiers $x < y$, there exists a z such that $x < z < y$. A way to have a dense domain is using lists for IDs, representing paths in a tree. To ensure globally unique ID

generation, element IDs also contain replica IDs to tie-break when two replicas insert concurrently at the same position. Some CRDTs that use this approach are Treedoc [70], Logoot [86], and LSEQ [60]. A problem with these approaches is that the size of generated IDs keeps growing over time.

CRDTs that Explicitly Order Unique Identifiers. To avoid ever-growing element IDs, some approaches do not assign IDs from a pre-existing dense total order, but: (1) use simple small globally unique IDs, such as pairs (replica ID, counter), what we call a dot; (2) explicitly add ordering constraints (rules), such as "$(i, 4)$ is before $(j, 6)$." Ordering constraints are only added but not changed or removed. A delete operation does not remove the element or constraints but creates a tombstone. This allows small sizes and simple convergence, considering the state as an ever-growing set of operations and immutable constraints. Some CRDTs that follow this approach are Woot [65], Causal Trees [36], RGA [74], and YATA [62]. A problem with these approaches is the accumulation of tombstones in the state. Tombstone removal can be achieved resorting to causal stability. This is in fact done algorithmically in RGA, without the authors using a term for it, while citing Golding [35] as already introducing a safe condition for removal. In fact, that thesis discusses log pruning using classic message stability (message received everywhere) and not causal stability (no more concurrent messages delivered). This is another example of the confusion between these concepts.

Interleaving Anomalies. The List data type also illustrates that achieving convergence may be the easiest part of a CRDT. Achieving, and even defining, the intended outcome given concurrent updates can be far more problematic. Different List CRDTs can produce strange unintended outcomes given concurrent updates. An example from Kleppmann et al. [47] is concurrent insertions at the same positions: given an initial list "`Hello!`," a user inserts "`Alice`" before "`!`" (6 inserts) and another user concurrently inserts "`Charlie`" (7 inserts), also before "`!`." Many CRDTs can produce an outcome after converging, such as "`Hello Al Ciharcliee!`," exhibiting an *interleaving anomaly*, when a more desirable outcome would be "`Hello Alice Charlie!`." Moreover, this anomaly is allowed by the specification in Attiya et al. [5], which Kleppmann et al. [47] argue is too weak for collaborative editing and propose a stronger specification. The anomaly manifests in Logoot, LSEQ, Treedoc and Woot; a lesser anomaly occurs in RGA if typing backwards. Weidner and Kleppmann [85] present a detailed study of the interleaving problem as well as a CRDT (Fugue) that solves it.

10 Final Remarks

We came a long way from ad hoc optimistic replication. CRDTs offer a principled approach to optimistic replication, achieving semantically meaningful convergence and causal consistency, two main goals for partition tolerant systems. They avoid "losing concurrent updates" without requiring concurrency control or transactions and allow some desirable outcomes not possible if restricting to sequential histories. Like sequential data types, they provide general useful abstractions, but their usage may be more difficult, as they may embrace concurrency in their specification and force programmers to think in terms of concurrency. Nevertheless, CRDTs have been successfully applied in the industry, allowing systems with low response times even for large spatial spans.

The operation-based and state-based are substantially different approaches. The former is more suitable to a known group of participants under good connectivity, and the latter is good for long partitions, disconnected operation, and independent groups of participants that eventually meet. The op-based approach allows small messages, better reuse of sequential data types (with commutative operations), and tends to have smaller state. The state-based approach results in typically larger states and messages, leading to less-frequent propagation and less "fresh" data. This is partially solved by the delta-state approach that, if using smart delta propagation, can potentially result in smaller messages, allowing more "freshness." The pure-op approach is a special point in

the design space of op-based CRDTs, demanding causality information from the middleware. It is limited compared with the general op-based approach, as it cannot extract information from the state when operations are invoked. Causal stability is an important ingredient in pure-op CRDTs to optimize state size, but nothing prevents it from being used in the general op-based approach, where it may be useful, e.g., to discard tombstones, as it provides semantically useful information.

One frequent criticism is that it is difficult to obtain correct and efficient implementations. The answer to such criticism is that, like for sequential data types, they need only to be implemented by a few experts, while many practitioners can reap the benefits. This is the reason why, 70 years after being invented, research still goes on in hash tables [9]. If CRDTs keep revealing themselves as useful, then the need for research and implementation effort will not be the problem.

Acknowledgments

I would like to thank the anonymous reviewers for their comments, which helped improve the article.

References

[1] 2013. AntidoteDB: A planet scale, highly available, transactional database built on CRDT technology. Retrieved from https://github.com/AntidoteDB/antidote

[2] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. 1995. Causal memory: Definitions, implementation, and programming. *Distrib. Comput.* 9, 1 (1995), 37–49. DOI: https://doi.org/10.1007/BF01784241

[3] Paulo Sérgio Almeida and Carlos Baquero. 2019. Scalable eventually consistent counters over unreliable networks. *Distrib. Comput.* 32, 1 (2019), 69–89. DOI: https://doi.org/10.1007/s00446-017-0322-2

[4] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. 2018. Delta state replicated data types. *J. Parallel Distrib. Comput.* 111 (2018), 162–173. DOI: https://doi.org/10.1016/j.jpdc.2017.08.003

[5] Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang, and Marek Zawirski. 2016. Specification and complexity of collaborative text editing. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC'16)*, George Giakkoupis (Ed.). ACM, 259–268. DOI: https://doi.org/10.1145/2933057.2933090

[6] Carlos Baquero, Paulo Sérgio Almeida, Alcino Cunha, and Carla Ferreira. 2017. Composition in state-based replicated data types. *Bull. EATCS* 123 (2017). Retrieved from http://eatcs.org/beatcs/index.php/beatcs/article/view/507

[7] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Making operation-based CRDTs operation-based. In *Proceedings of the 14th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems (DAIS'14)*, Kostas Magoutis and Peter R. Pietzuch (Eds.).

[8] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. 2017. Pure operation-based replicated data types. Retrieved from http://arxiv.org/abs/1710.04469

[9] Michael A. Bender, Alex Conway, Martín Farach-Colton, William Kuszmaul, and Guido Tagliavini. 2023. Iceberg hashing: Optimizing many hash-table criteria at once. *J. ACM* 70, 6 (November 2023). DOI: https://doi.org/10.1145/3625817

[10] Annette Bieniusa, Marek Zawirski, Nuno M. Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. 2012. Brief announcement: Semantics of eventually consistent replicated sets. In *Proceedings of the 26th International Symposium on Distributed Computing (DISC'12), Lecture Notes in Computer Science,* Vol. 7611, Marcos K. Aguilera (Ed.). Springer, 441–442. DOI: https://doi.org/10.1007/978-3-642-33651-5_48

[11] Annette Bieniusa, Marek Zawirski, Nuno M. Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. 2012. An optimized conflict-free replicated set. Retrieved from http://arxiv.org/abs/1210.3368

[12] Garrett Birkhoff. 1937. Rings of sets. *Duke Mathematical Journal* 3, 3 (1937), 443–454. DOI: https://doi.org/10.1215/S0012-7094-37-00334-X

[13] G. Birkhoff. 1940. *Lattice Theory*. American Mathematical Society. Retrieved from https://books.google.pt/books?id=o4bu3ex9BdkC

[14] Kenneth P. Birman and Thomas A. Joseph. 1987. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.* 5, 1 (1987), 47–76. DOI: https://doi.org/10.1145/7351.7478

[15] Kenneth P. Birman, André Schiper, and Pat Stephenson. 1991. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.* 9, 3 (1991), 272–314. DOI: https://doi.org/10.1145/128738.128742

[16] Manuel Bravo, Luís E. T. Rodrigues, and Peter Van Roy. 2017. Saturn: A distributed metadata service for causal consistency. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys'17)*, Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic (Eds.). ACM, 111–126. DOI: https://doi.org/10.1145/3064176.3064210

[17] Eric A. Brewer. 2000. Towards robust distributed systems (abstract). In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, Gil Neiger (Ed.). ACM, 7. DOI: https://doi.org/10.1145/343477.343502

[18] Russell Brown, Sean Cribbs, Christopher Meiklejohn, and Sam Elliott. 2014. Riak DT map: a composable, convergent replicated dictionary. In *Proceedings of the 1st Workshop on the Principles and Practice of Eventual Consistency (PaPEC@EuroSys'14)*, Marc Shapiro (Ed.). ACM, 1:1. DOI: https://doi.org/10.1145/2596631.2596633

[19] Sebastian Burckhardt. 2014. Principles of eventual consistency. *Found. Trends Program. Lang.* 1, 1-2 (2014), 1–150. DOI: https://doi.org/10.1561/2500000011

[20] Sebastian Burckhardt, Alexey Gotsman, and Hongseok Yang. 2013. *Understanding Eventual Consistency. Microsoft Research.* Retrieved from https://www.microsoft.com/en-us/research/publication/understanding-eventual-consistency/

[21] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated data types: Specification, verification, optimality. In *Proceedings of the 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'14)*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 271–284. DOI: https://doi.org/10.1145/2535838.2535848

[22] Luca Cardelli and Peter Wegner. 1985. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.* 17, 4 (1985), 471–522. DOI: https://doi.org/10.1145/6041.6042

[23] Miguel Castro and Barbara Liskov. 1999. Practical byzantine fault tolerance. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI'99)*, Margo I. Seltzer and Paul J. Leach (Eds.). USENIX Association, 173–186. Retrieved from https://dl.acm.org/citation.cfm?id=296824

[24] Automerge contributors. 2019. Automerge: A JSON-like data structure (a CRDT) that can be modified concurrently by different users, and merged again automatically. Retrieved from https://github.com/automerge/automerge

[25] William R. Cook. 2009. On understanding data abstraction, revisited. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'09)*, Shail Arora and Gary T. Leavens (Eds.). ACM, 557–572. DOI: https://doi.org/10.1145/1640089.1640133

[26] Ole-Johan Dahl and Kristen Nygaard. 1966. SIMULA—An ALGOL-based simulation language. *Commun. ACM* 9, 9 (1966), 671–678. DOI: https://doi.org/10.1145/365813.365819

[27] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, Thomas C. Bressoud and M. Frans Kaashoek (Eds.). ACM, 205–220. DOI: https://doi.org/10.1145/1294261.1294281

[28] Alan J. Demers, Daniel H. Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard E. Sturgis, Daniel C. Swinehart, and Douglas B. Terry. 1987. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, Fred B. Schneider (Ed.). ACM, 1–12. DOI: https://doi.org/10.1145/41840.41841

[29] Edsger W. Dijkstra. 1974. Self-stabilizing systems in spite of distributed control. *Commun. ACM* 17, 11 (1974), 643–644. DOI: https://doi.org/10.1145/361179.361202

[30] Clarence A. Ellis and Simon J. Gibbs. 1989. Concurrency control in groupware systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, James Clifford, Bruce G. Lindsay, and David Maier (Eds.). ACM Press, 399–407. DOI: https://doi.org/10.1145/67544.66963

[31] Vitor Enes, Paulo Sérgio Almeida, and Carlos Baquero. 2017. The single-writer principle in CRDT composition. In *Proceedings of the Workshop on Programming Models and Languages for Distributed Computing*, Christopher Meiklejohn and Heather Miller (Eds.). ACM, 4:1–4:3. DOI: https://doi.org/10.1145/3166089.3168733

[32] Vitor Enes, Paulo Sérgio Almeida, Carlos Baquero, and João Leitão. 2019. Efficient synchronization of state-based CRDTs. In *Proceedings of the 35th IEEE International Conference on Data Engineering (ICDE'19)*. IEEE, 148–159. DOI: https://doi.org/10.1109/ICDE.2019.00022

[33] Vitor Enes, Carlos Baquero, Paulo Sérgio Almeida, and João Leitão. 2017. Borrowing an identity for a distributed counter: Work in progress report. In *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC@EuroSys'17)*, Annette Bieniusa and Alexey Gotsman (Eds.). ACM, 4:1–4:3. DOI: https://doi.org/10.1145/3064889.3064894

[34] Seth Gilbert and Nancy A. Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* 33, 2 (2002), 51–59. DOI: https://doi.org/10.1145/564585.564601

[35] Richard A. Golding. 1992. *Weak-consistency Group Communication and Membership.* Ph.D. Dissertation. University of California Santa Cruz.

[36] Victor S. Grishchenko. 2010. Deep hypertext with embedded revision control implemented in regular expressions. In *Proceedings of the 6th International Symposium on Wikis and Open Collaboration*, Phoebe Ayers and Felipe Ortega (Eds.). ACM. DOI: https://doi.org/10.1145/1832772.1832777

[37] Frédéric Guidec, Yves Mahéo, and Camille Noûs. 2023. Supporting conflict-free replicated data types in opportunistic networks. *Peer-to-Peer Netw. Appl.* 16, 1 (2023), 395–419. DOI: https://doi.org/10.1007/S12083-022-01404-6

[38] Per Brinch Hansen. 1973. *Operating System Principles*. Prentice-Hall, Inc., USA.

[39] Maurice Herlihy. 1991. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.* 13, 1 (1991), 124–149. DOI: https://doi.org/10.1145/114005.102808

[40] Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492. DOI: https://doi.org/10.1145/78969.78972

[41] High Scalability. 2014. How League of Legends Scaled Chat to 70 million Players. Retrieved from https://highscalability.com/how-league-of-legends-scaled-chat-to-70-million-players-it-t

[42] C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580. DOI: https://doi.org/10.1145/363235.363259

[43] C. A. R. Hoare. 1972. Proof of correctness of data representations. *Acta Inform.* 1 (1972), 271–281. DOI: https://doi.org/10.1007/BF00289507

[44] C. A. R. Hoare. 1974. Monitors: An operating system structuring concept. *Commun. ACM* 17, 10 (1974), 549–557. DOI: https://doi.org/10.1145/355620.361161

[45] Abdessamad Imine, Pascal Molli, Gérald Oster, and Michaël Rusinowitch. 2003. Proving correctness of transformation functions in real-time groupware. In *Proceedings of the 8th European Conference on Computer Supported Cooperative Work*, Kari Kuutti, Eija Helena Karsten, Geraldine Fitzpatrick, Paul Dourish, and Kjeld Schmidt (Eds.). Springer, 277–293. DOI: https://doi.org/10.1007/978-94-010-0068-0_15

[46] Kevin Jahns. 2023. Yjs: Shared data types for building collaborative software. Retrieved from https://github.com/yjs/yjs

[47] Martin Kleppmann, Victor B. F. Gomes, Dominic P. Mulligan, and Alastair R. Beresford. 2019. Interleaving anomalies in collaborative text editors. In *Proceedings of the 6th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC@EuroSys'19)*. ACM, 6:1–6:7. DOI: https://doi.org/10.1145/3301419.3323972

[48] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. 2019. Local-first software: You own your data, in spite of the cloud. In *Proceedings of the ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!'19)*, Hidehiko Masuhara and Tomas Petricek (Eds.). ACM, 154–178. DOI: https://doi.org/10.1145/3359591.3359737

[49] Rusty Klophaus. 2010. Riak core: Building distributed applications without shared state. In *Proceedings of the ACM SIGPLAN Conference on Commercial Users of Functional Programming (CUFP'10)*. DOI: https://doi.org/10.1145/1900160.1900176

[50] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. 1992. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.* 10, 4 (1992), 360–391. DOI: https://doi.org/10.1145/138873.138877

[51] Leslie Lamport. 1974. A new solution of Dijkstra's Concurrent programming problem. *Commun. ACM* 17, 8 (1974), 453–455. DOI: https://doi.org/10.1145/361082.361093

[52] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978), 558–565. DOI: https://doi.org/10.1145/359545.359563

[53] Butler W. Lampson and Howard E. Sturgis. 1979. *Crash Recovery in a Distributed Data Storage System*. Technical Report. Xerox Palo Alto Research Center.

[54] Du Li and Rui Li. 2004. Preserving operation effects relation in group editors. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'04)*, James D. Herbsleb and Gary M. Olson (Eds.). ACM, 457–466. DOI: https://doi.org/10.1145/1031607.1031683

[55] Lightbend. 2015. Concurrent sharing of "data in motion" across clusters with CRDTs in Akka Distributed Data. Retrieved from https://www.lightbend.com/blog/concurrent-sharing-of-data-in-motion-across-clusters-with-crdts-in-akka-distributed-data

[56] Barbara H. Liskov and Stephen N. Zilles. 1974. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, Burt M. Leavenworth (Ed.). ACM, 50–59. DOI: https://doi.org/10.1145/800233.807045

[57] Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin. 2011. *Consistency, Availability, and Convergence*. Technical Report UTCS TR-11-22. Department of Computer Science, The University of Texas at Austin.

[58] Juliane Marubayashi and Carlos Baquero. 2023. Making Causal Based CRDTs Scalable. Retrieved from https://inforum2023-ylfd.vercel.app/Atas/paper_366/366-CM.pdf

[59] Christopher Meiklejohn. 2013. Riak PG: Distributed process groups on dynamo-style distributed storage. In *Proceedings of the 12fth ACM SIGPLAN Erlang Workshop*, Steve Vinoski and Laura M. Castro (Eds.). ACM, 27–32. DOI: https://doi.org/10.1145/2505305.2505309

[60] Brice Nédelec, Pascal Molli, Achour Mostéfaoui, and Emmanuel Desmontils. 2013. LSEQ: An adaptive structure for sequences in distributed collaborative editing. In *Proceedings of the ACM Symposium on Document Engineering*, Simone Marinai and Kim Marriott (Eds.). ACM, 37–46. DOI: https://doi.org/10.1145/2494266.2494278

[61] David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. 1995. High-latency, low-bandwidth windowing in the Jupiter collaboration system. In *Proceedings of the 8th Annual ACM Symposium on User Interface Software and Technology (UIST'95)*, George G. Robertson (Ed.). ACM, 111–120. DOI: https://doi.org/10.1145/215585.215706

[62] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. 2016. Near real-time peer-to-peer shared editing on extensible data types. In *Proceedings of the 19th International Conference on Supporting Group Work*, Stephan G. Lukosch, Aleksandra Sarcevic, Myriam Lewkowicz, and Michael J. Muller (Eds.). ACM, 39–49. DOI: https://doi.org/10.1145/2957276.2957310

[63] Kristen Nygaard and Ole-Johan Dahl. 1978. The development of the SIMULA languages. *ACM SIGPLAN Not.* 13, 8 (1978), 245–272. DOI: https://doi.org/10.1145/960118.808391

[64] Gérald Oster, Pascal Molli, Pascal Urso, and Abdessamad Imine. 2006. Tombstone transformation functions for ensuring consistency in collaborative editing systems. In *Proceedings of the 2nd International ICST Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom'06)*, Enrico Blanzieri and Tao Zhang (Eds.). IEEE Computer Society/ICST. DOI: https://doi.org/10.1109/COLCOM.2006.361867

[65] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. 2006. Data consistency for P2P collaborative editing. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'06)*, Pamela J. Hinds and David Martin (Eds.). ACM, 259–268. DOI: https://doi.org/10.1145/1180875.1180916

[66] Michael Owen. 2015. Using Erlang, Riak and the ORSWOT CRDT at bet365 for Scalability and Performance. Retrieved from https://www.erlang-factory.com/euc2015/michael-owen

[67] Douglas Stott Parker, Jr., Gerald J. Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David Edwards, Stephen Kiser, and Charles Kline. 1983. Detection of mutual inconsistency in distributed systems. *IEEE Trans. Softw. Eng.* 9, 3 (1983), 240–247. DOI: https://doi.org/10.1109/TSE.1983.236733

[68] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. 1980. Reaching agreement in the presence of faults. *J. ACM* 27, 2 (1980), 228–234. DOI: https://doi.org/10.1145/322186.322188

[69] Nuno M. Preguiça, Carlos Baquero, Paulo Sérgio Almeida, Victor Fonte, and Ricardo Gonçalves. 2010. Dotted version vectors: Logical clocks for optimistic replication. Retrieved from http://arxiv.org/abs/1011.5808

[70] Nuno M. Preguiça, Joan Manuel Marquès, Marc Shapiro, and Mihai Letia. 2009. A commutative replicated data type for cooperative editing. In *Proceedings of the 29th IEEE International Conference on Distributed Computing Systems (ICDCS'09)*. IEEE Computer Society, 395–403. DOI: https://doi.org/10.1109/ICDCS.2009.20

[71] Nuno M. Preguiça, Marek Zawirski, Annette Bieniusa, Sérgio Duarte, Valter Balegas, Carlos Baquero, and Marc Shapiro. 2014. SwiftCloud: Fault-tolerant geo-replication integrated all the way to the client machine. In *Proceedings of the 33rd IEEE International Symposium on Reliable Distributed Systems Workshops (SRDS Workshops'14)*. IEEE Computer Society, 30–33. DOI: https://doi.org/10.1109/SRDSW.2014.33

[72] Redis. 2022. Diving into Conflict-Free Replicated Data Types (CRDTs). Retrieved from https://redis.io/blog/diving-into-crdts

[73] Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser. 1996. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'96)*, Mark S. Ackerman, Gary M. Olson, and Judith S. Olson (Eds.). ACM, 288–297. DOI: https://doi.org/10.1145/240080.240305

[74] Hyun-Gul Roh, Myeongjae Jeon, Jinsoo Kim, and Joonwon Lee. 2011. Replicated abstract data types: Building blocks for collaborative applications. *J. Parallel Distrib. Comput.* 71, 3 (2011), 354–368. DOI: https://doi.org/10.1016/j.jpdc.2010.12.006

[75] Douglas T. Ross and Clarence G. Feldmann. 1964. Verbal and graphical language for the AED system: A progress report. In *Proceedings of the SHARE Design Automation Workshop (DAC'64)*. ACM. DOI: https://doi.org/10.1145/800265.810743

[76] Yasushi Saito and Marc Shapiro. 2005. Optimistic replication. *ACM Comput. Surv.* 37, 1 (2005), 42–81. DOI: https://doi.org/10.1145/1057977.1057980

[77] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A Comprehensive Study of Convergent and Commutative Replicated Data Types*. Research Report RR-7506. Inria – Centre Paris-Rocquencourt; INRIA. 50 pages. Retrieved from https://inria.hal.science/inria-00555588

[78] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free replicated data types. In *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'11), Lecture Notes in Computer Science,* Vol. 6976, Xavier Défago, Franck Petit, and Vincent Villain (Eds.). Springer, 386–400. DOI: https://doi.org/10.1007/978-3-642-24550-3_29

[79] SoundCloud. 2014. Roshi: A large-scale CRDT set implementation for timestamped events. Retrieved from https://github.com/soundcloud/roshi

[80] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent B. Welch. 1994. Session guarantees for weakly consistent replicated data. In *Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems (PDIS'94)*. IEEE Computer Society, 140–149. DOI: https://doi.org/10.1109/PDIS.1994.331722

[81] Douglas B. Terry, Marvin Theimer, Karin Petersen, Alan J. Demers, Mike Spreitzer, and Carl Hauser. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP'95)*, Michael B. Jones (Ed.). ACM, 172–183. DOI: https://doi.org/10.1145/224056.224070

[82] Paolo Viotti and Marko Vukolic. 2016. Consistency in non-transactional distributed storage systems. *ACM Comput. Surv.* 49, 1 (2016), 19:1–19:34. DOI: https://doi.org/10.1145/2926965

[83] Werner Vogels. 2009. Eventually consistent. *Commun. ACM* 52, 1 (2009), 40–44. DOI: https://doi.org/10.1145/1435417.1435432

[84] Matthew Weidner and Paulo Sérgio Almeida. 2022. An oblivious observed-reset embeddable replicated counter. In *Proceedings of the 9th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC@EuroSys'22*, Adriana Szekeres and K. C. Sivaramakrishnan (Eds.). ACM, 47–52. DOI: https://doi.org/10.1145/3517209.3524084

[85] Matthew Weidner and Martin Kleppmann. 2023. The art of the fugue: Minimizing interleaving in collaborative text editing. DOI: https://doi.org/10.48550/ARXIV.2305.00583

[86] Stéphane Weiss, Pascal Urso, and Pascal Molli. 2009. Logoot: A scalable optimistic replication algorithm for collaborative editing on P2P networks. In *Proceedings of the 29th IEEE International Conference on Distributed Computing Systems (ICDCS'09)*. IEEE Computer Society, 404–412. DOI: https://doi.org/10.1109/ICDCS.2009.75

[87] Niklaus Wirth and C. A. R. Hoare. 1966. A contribution to the development of ALGOL. *Commun. ACM* 9, 6 (1966), 413–432. DOI: https://doi.org/10.1145/365696.365702

Received 27 October 2023; revised 19 July 2024; accepted 2 September 2024