arXiv:2310.18220v1 [cs.DC] 27 Oct 2023

Approaches to Conflict-free Replicated Data Types

PAULO SÉRGIO ALMEIDA, INESC TEC & University of Minho, 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 as a tutorial for prospective CRDT researchers and designers, it provides solid coverage of the essential concepts, clarifying some misconceptions which frequently occur, but also presents some novel insights gained from considerable experience in designing both specific CRDTs and approaches to CRDTs.

CCS Concepts: • **Computer systems organization** → *Availability*; • **Theory of computation** → **Distributed algorithms**; • **Computing methodologies** → *Distributed computing methodologies.*

Additional Key Words and Phrases: Eventual Consistency, Replicated Data Types, CRDT

1 INTRODUCTION

Classic distributed systems aim for strong consistency (e.g., linearizability [33]), through the state-machine replication approach, proposed by Lamport [40]. 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 [14, 30].

The importance of always-on availability for real businesses motivated relaxing consistency. A seminal work, which popularized the NoSQL movement, was Amazon's Dynamo [25], which stresses the importance of availability: "even the slightest outage has significant financial consequences and impacts customer trust". But programming such NoSQL data stores is significantly difficult and error prone, given a low level read-write based API, and the need for ad hoc reconciliation of concurrent updates.

Since their appearance [58], 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 paper 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 [2], aim to achieve, in a way, "the best of both worlds", while pure operation-based CRDTs [3, 4] 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.

Author's address: Paulo Sérgio Almeida, psa@di.uminho.pt, INESC TEC & University of Minho, Portugal.

This paper also clarifies some misconceptions regarding CRDTs, which frequently occur, in papers or presentations, and shows novel depictions, never presented before, 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 [38] is the support for data types with non-commutative operations. We also clarify the role of commutativity, and what exactly needs to be commutative, by presenting a better (than the usual) graph 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 paper 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 present a novel *observed-remove set* which 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 these approaches and with approaches to replicated data types based on totally-ordered histories. Along the paper 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 [52], RGA [54]), which need considerably more involved algorithms.

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, such as 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 [55] (then named *plexes*) and adopted for Algol by Wirth and Hoare [64]; the combination of procedures and records. What actually happened [46] was the generalization of Algol blocks to lifetime not restricted to stack allocation in Simula [24] processes (and later classes). The final ingredient was hiding the representation from client code, in CLU [43], 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 [34], involving the notions of preconditions, postconditions and invariants, and its adaptation to data types [36]. The pervading occurrence of aliasing in languages poses a problem, but new languages better at avoiding mutable state sharing, like Rust, give us hope.

While easier to reason about, they are harder to implement efficiently in functional languages, due to the absence of destructive updates and the need for persistence. Amortization of cost can be achieved with lazy evaluation [47], but still not matching what is possible in imperative implementations, and causing performance unpredictability.

Concurrent data types. It was only natural to extend the concept of data types in sequential imperative languages to shared memory concurrency, to obtain objects usable by concurrent threads. Inspired by Simula, Hansen [31] introduced monitors, with the construct of *shared classes*, and Hoare [35] 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, as we will see, aim to be always available locally.

Lock-free and wait-free data structures. Towards achieving performance in shared-memory multiprocessors and improve fault tolerance, 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 [32], 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 arise: how to express correctness criteria when several actions are happening concurrently. This is relevant for distributed systems.

2.2 Strongly consistent replication

Linearizability. Linearizability [33] is the more widely used correctness criteria 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.

> "Linearizability provides the illusion that each **operation** applied by concurrent processes takes effect **instantaneously** at some point **between** its **invocation** and its **response**"

Consider the example of a register object, with read and write operations, being accessed by two processes, as shown in Figure 1. 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.

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 [40] 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;

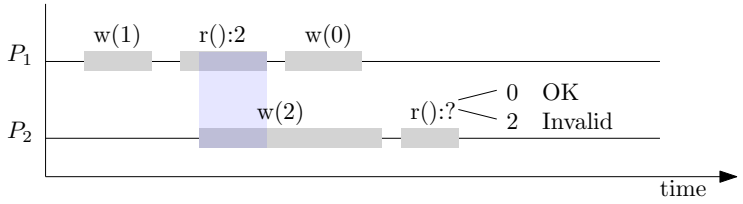

Fig. 1. A register object with read and write operations. To be linearizable, last read must return 0.

(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 [49]. The extreme case of tolerating byzantine faults is applied in the now popular blockchains, many of which use some variant of the PBFT algorithm [22].

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 2000 conference, Brewer [14] stated a conjecture that in a distributed system we can only achieve, simultaneously, at most two the three guarantees:

- strong Consistency
- Availability
- Partition tolerance

This conjecture was proved [30], 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 there is more than the yes/no guarantee. Even when there are no network partitions, the design choice has also implication 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.

Regarding AP systems, forgoing linearizability, an important question is: what consistency model can/do they aim for?

Consistency models. The definition of consistency models, either for programming languages or distributed systems, 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,

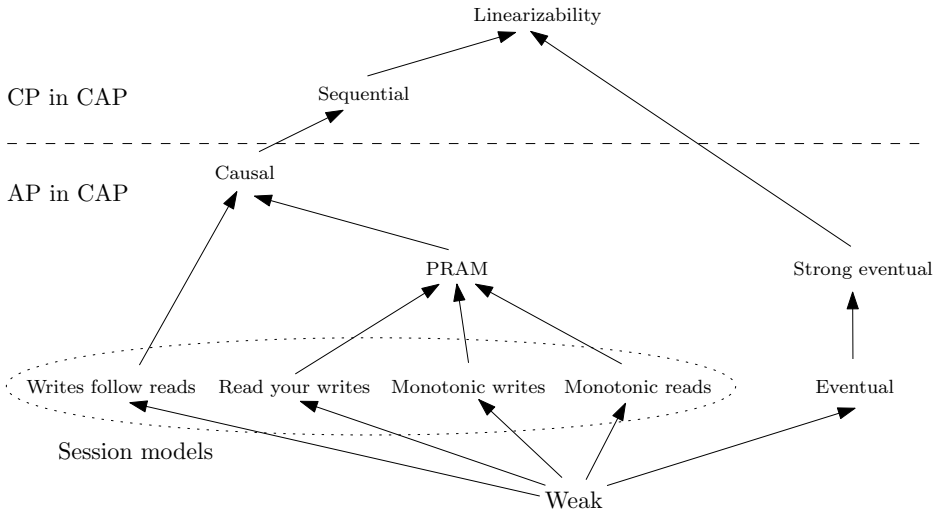

Fig. 2. Some consistency models for distributed systems.

programmers), in matters such as efficient use of "the machine", ease of implementability, ease of reasoning, or useful guarantees.

Figure 2 presents some important consistency models/guarantees for distributed systems, the strongest on top. Viotti and Vukolic [61] present over 40 models, not even including transactional models. The horizontal line separates what is achievable in AP systems and in CP systems. Of these, causal consistency [1] is of special importance, being (broadly) the strongest achievable while not losing availability [45].

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 [15] (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. Depending on the abstraction, 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 (some variation of) causal consistency together with what has become known as *strong eventual consistency* (SEC) [58], 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*). This means that different replicas keep converging as operations become visible, and if update operations stop being issued replicas will eventually converge.

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. [60] 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. [59], 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. [54]. The informal meaning of EC "eventual convergence when updates stop being issued", since it became popular due to Vogels [62], is just a consequence of the above properties which 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.

Remark: It would be difficult to achieve convergence otherwise. If that was not the case, *when* would they converge? I.e., as result of processing *which* other events, namely if there were no more events to be acted upon? So, SEC is not something that implies some extra cost over some baseline, but the natural condition for EC systems, and the term SEC itself should be deprecated in favor of using EC, as originally intended, together with the terms eventual visibility and strong convergence for clarification.

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 on 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 that it can suit the hardware, and run efficiently in the underlying machine. The term came from Jackie Stewart, Formula One pilot, referring to a driver having intimate knowledge of how the car responds.)

On the contrary, linearizability can be said to be, in a sense, *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 own 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., it would be completely unsuitable for a system encompassing Earth and a future Mars colony.

Similar observations about spatial scalability were made by Lipton and Sandberg [42], when introducing a scalable memory (Pipelined RAM), regarding *coherency*, a weaker model than linearizability: "Thus, no matter how clever or complex a protocol is, if it implements a coherent shared memory or CRAM, it must be 'slow.' If a shared memory system must be consistent, then it must take time proportional to τ_{global} for reading and writing."

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* [56] 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 [38], relax ordering, but only for commutative operations.
- In Bayou [59], 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 Conflict-free Replicated Data Types (CRDTs) [58] 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 any kind of synchronization. This makes a CRDT object highly available, even under partitions, with essentially zero response time for its operations (only local computation).

As previous optimistic replication approaches, information is propagated to other replicas 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. The 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. The evolution of object-oriented languages lead to the need to relate several concepts, like abstract data type, parametric polymorphism, object, class, inheritance and subtyping, as described in the classic by Cardelli and Wegner [21], with tools such as existential and bounded universal quantification. As well summarized by Cook [23], 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 paper, 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 convergence, for some CRDTs no ordering guarantees at all would be needed. But towards ensuring, in addition to convergence, the strongest possible consistency model while remaining available under partitions, i.e., causal consistency, a causal broadcast [11] 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 (in two replicas), 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 3, divides the execution of an update operation in two phases: *prepare* and *effect*.

(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.

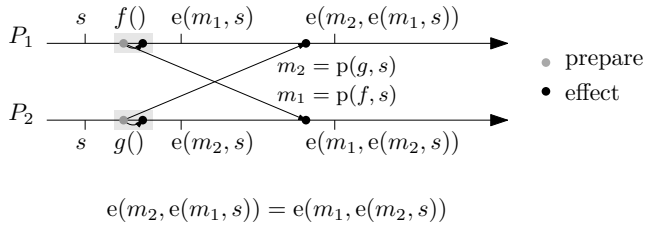

Fig. 3. Execution model for op-based CRDTs, stressing the commutativity of effect for concurrently invoked operations.

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

This last point, immediate self-delivery, is important to ensure "Read Your Writes" [60], making the state-change immediately reflected locally, 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* [8]: 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 4: 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, 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 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*. (This is how pure op-based CRDTs, below, are defined, whose unoptimized versions are essentially runnable specifications.)

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 which 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 which have already taken effect on the state, but 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" which CRDTs aim to avoid. Two examples are the observed-remove set [7]:

- 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 [63]:

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

CRDT state:
$s : \mathcal{P}(E \times \ldots) = \emptyset$

query elements() : E
return $\{e \mid (e, \_) \in s\}$

query contains($e : E$) : $\mathbb{B}$
return $\exists x \cdot (e, x) \in s$

update add($e : E$)
prepare
let u = [some unique id]
return (add, e, u)
effect (add, e, u)
$s \leftarrow s \cup \{(e, u)\}$

update remove($e : E$)
prepare
let $r = \{(x, u) \in s \mid x = e\}$
return (remove, r)
effect (remove, r)
$s \leftarrow s \setminus r$

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

The appeal of observed-cancel semantics can be seen in the observed-reset counter, as it allows a sample-and-reset pattern, in which a process periodically samples the counter value and resets it. This allows grouping increments in a sequence of values, without losing any increment, even the ones concurrently issued, which will be accounted for in the next sample-and-reset. An alternative semantics, in which a reset cancels concurrent increments, would not allow such 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" [7]. An even more optimized version, not previously published, is presented in Figure 6.

Many CRDTs start indeed from a naive version, being further optimized. The observed-remove set is a good example, to show possible improvements. A vanilla, naive, version is shown in Figure 5. 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 achieve them. This is a frequent need, and can be achieved by using unique replica ids and a counter per replica, incremented at each operation, with unique ids obtained as pairs (replica id, counter), sometimes called a "dot" (from Dotted Version Vectors [53]). 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 the 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 6, 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 the 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

types:
$\mathbb{I}$, set of replica identifiers

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 $(\text{add}, e, (i, c), m[e])$
effect (add, e, d, r)
$m[e] \leftarrow m[e] \setminus r \cup \{d\}$

update remove($e : E$)
prepare
return $(\text{remove}, e, m[e])$
effect (remove, e, r)
$m[e] \leftarrow m[e] \setminus r$

Fig. 6. Op-based observed-remove set, ORSet$\langle E \rangle$, optimized implementation. Algorithm for replica i.

implementation. Currently published versions which do not make this distinction end up being 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 which the map 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 [7]: 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 for concurrently issued add/remove is commutative.

3.3 Beyond sequential semantics and API

Lack of equivalence to sequential executions. CRDTs aim to preserve the sequential semantics for sequential executions, but what concerns concurrent invocations not always is it possible to achieve such equivalence. 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 which 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., $\text{add}(a); \text{remove}(b) \parallel \text{add}(b); \text{remove}(a)$. 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.

types:
$\mathbb{I}$, set of replica identifiers
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 (write, $(e, (i, c)), r$)
effect (write, v, r)
$s \leftarrow \{(e, d) \in s \mid d \notin r\} \cup \{v\}$

Fig. 7. Op-based multi-value register, MVReg$\langle E \rangle$. Algorithm for replica i.

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 [25] 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 7.

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. When effect is applied, it keeps the pairs 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. The op-based model is overly broad. We can pick any state-based CRDT, define prepare as applying the operation and returning the resulting state, and define effect as merging states. Thus, any state-based CRDT implementation can fit the op-based model.

Pure op-based CRDTs [3], are a special kind of op-based CRDTs, to restrict them to the most pure form of op-based – "send only operations" – unlike the general op-based model, that can have CRDT implementations that depart too much from the op-based spirit, in the extreme being, for all practical purposes, state-based. This is not to say that only pure op-based "fit the op-based spirit", and pure op-based, being a limited subset of op-based, may be less efficient or demand more support to regain efficiency than general op-based. 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:

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

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 on 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 info, but normal APIs do not expose it to clients. *Tagged Causal Broadcast* [3] provides two kinds of information:

- partial order information, more precisely 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 which 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 [12].

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 message 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, specially 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 over TCB. Having a TCB middleware, the distributed algorithm for pure op-based CRDTs becomes simply reacting to the middleware callbacks and invoking either prepare, effect, or a function *stable* to make use of causal stability information, as shown in Figure 8.

Table 1. Some usages of "stability" and their meaning.

Term	Provenance	Meaning
Self stabilization	Dijkstra [27]	returns to valid state
Stable storage	Lampson and Sturgis [41]	durable storage, survives crashes
Multicast stability	Birman, Schiper, and Stephenson [12]	message was received by all nodes
Write stability	Terry et al. [59]	a tentative write commits
Causal stability	Baquero, Almeida, and Shoker [3]	concurrent messages were delivered

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

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 to have a universal definition. Only queries are data type dependent, being defined over the PO-Log. This approach is shown in Figure 9. It is very naive, but a starting point for subsequent optimization.

PO-Log based observed-remove (add-wins) set. An example of a PO-Log based CRDT, an observed-remove set is shown in Figure 10. 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 [17, 20], and how it fits directly the pure op-based model, as opposed to the totally ordered history of operations used for sequential specifications.

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

Fig. 10. PO-Log based observed-remove (add-wins) set CRDT.

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 dependent, 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 will be compared as in the past relative to new operations that will arrive.

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;
- any remove is made obsolete by any other (timestamp, op) pair, i.e., after having made other operations obsolete, the remove itself can be discarded from the PO-Log when newer operations arrive.

So, a data type specific obsolete relation can be defined and used to compact the PO-Log:

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

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

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

A detailed description of how causality and causal stability can be used to achieve PO-Log compaction is beyond the scope of this paper. More details can be found in Baquero, Almeida, and Shoker [3, 4].

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, 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, 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, they must be merged (reconciled) into a new state, which reflects the conflict resolution. In classic optimistic replication this was performed in some ad hoc way, through a user-defined merge procedure.

Contrary to operation-based, 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 [26] 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 same result when merging in different orders;
- monotonic: merging with a "newer" state produces a "newer" state;

The solution to these goals is to adopt for the replica state the mathematical concept of *join-semilattices* [9]. 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 partially ordered set (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), 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 pair 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 $\mathbb{N}$: $\sqsubseteq = \leq$; $\sqcup = \max$; $\bot = 0$;
- booleans $\mathbb{B}$: $\mathsf{False} \sqsubset \mathsf{True}$; $\sqcup = \lor$; $\bot = \mathsf{False}$;
- any totally ordered set (called a *chain*).

Some posets which are not join-semilattices are:

- unordered posets (called *antichains*), where all elements are incomparable;

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. 11. Some classic lattice compositions.

- 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 11: 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, 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 specially useful. These and other examples are presented by Baquero et al. [5], where it is shown that many update functions can also be defined by composition.

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 immediately looking at the local state. The essential difference is how knowledge about operations is propagated between replicas. 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 to 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.

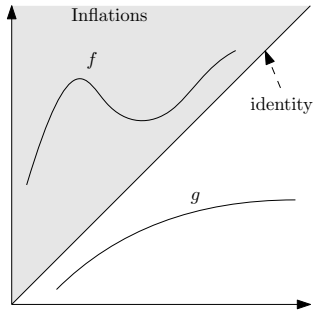

Fig. 12. Inflations vs monotonic functions.

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 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, a newer one will subsume it; an old message can arrive as a duplicate causing no harm.

Inflations vs monotonic functions. Sometimes there is, however, some confusion about what is monotonic. Erroneously, many places say mutators need to be monotonic, when in fact state mutators need to be inflations. Being monotonic is not necessary nor sufficient, as illustrated in Figure 12: 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 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 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. This is not true. A counter-example is a sequential data type, presented in Figure 13, whose state is a map from some unspecified set of keys to integer values; with a single operation 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 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):

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

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

Fig. 13. Sequential "advancer" data type, reusable as state-based CRDT.

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

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

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. All updates being inflations is obviously necessary for reuse. 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 $o(s) \mid\mid o(s)$ is $o(o(s))$, but

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

So, the reuse of a non-idempotent operation from the sequential data type would violate permutation equivalence. The simplest example of impossibility to reuse 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 14: 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 only updated by a single process, has long been used to obtain elegant designs, such as the Bakery algorithm [39], and is specially relevant to the design of state-based CRDTs [28]:

- the state is partitioned in several parts;

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

Fig. 14. State-based grow only set $\mathsf{GSet}\langle E\rangle$

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

Fig. 15. State-based GCounter

- each node updates a part dedicated exclusively to itself;
- the state is joined by joining respective parts.

Unique node identifiers can be used to partition the state, which becomes a map from ids to parts. This technique is behind the well known version vectors [48]. CRDTs that use node 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 15, 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.

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 16. (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 being a set with add and remove operations, but we must always use inflations for state mutators. This prevents us, in this example, from using a single set for the state and removing elements 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 [57] (two-phase set) CRDT: it allows add and remove, but once removed, an element cannot

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

Fig. 16. State-based PNCounter

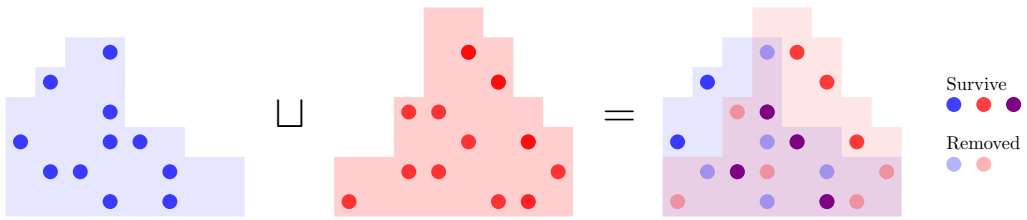

Fig. 17. Joining causal CRDTs.

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 removal of information while avoiding tombstones is used in *Causal CRDTs* [2], which draw inspiration from *Dotted Version Vectors* [53]. The state has 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*, in the form of a pair (replica-identifier × counter). The causal context (CC) represents the causal history: the ids of all visible updates, normally encoded by a compact version vector.

Remark: a dot is not merely a unique identifier. That would be enough for the op-based ORSet CRDT, and could have different forms, such as a pair (Lamport-timestamp × replica-identifier). A dot is also 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 in the 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 17. 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.

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

Fig. 18. Join operator for causal CRDTs, considering each kind of dot store.

Some dot stores. Being a dot store a container, we can have different types of such containers. Three such dot stores are a DotSet – a set of dots; a $\mathsf{DotFun}\langle V\rangle$ – a map from dots to values in some lattice V; and a $\mathsf{DotMap}\langle K, V\rangle$ – a map from keys in some arbitrary set K to (recursively) 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 which embed CRDTs, including DotMaps themselves, e.g.:

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

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 18, simply returns the dot store containing the appropriate surviving items, as illustrated in Figure 17, paired with the join of the causal contexts.

Observed-remove set $\mathsf{ORSet}\langle E\rangle$. To illustrate the power of causal CRDTs, we give the example of a state-based observed-remove set, shown in Figure 19, 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 context is in the form of a version vector. The join, not shown, is the previously defined for the DotMap (and DotSet) case:

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

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

Fig. 19. State-based observed-remove set $\mathsf{ORSet}\langle E\rangle$.

State-based	Delta-state based
$X_i \leftarrow m(X_i)$	$X_i \leftarrow X_i \sqcup m^\delta(X_i)$
$X_i \leftarrow X_i \sqcup X_j$	$X_i \leftarrow X_i \sqcup d$

Fig. 20. Normal state-based vs delta state-based approaches

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. 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. On the other hand, state-based approaches work under weak messaging assumptions (at-least-once is enough), 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 [2] 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 current state; 2) joined with other delta states, forming *delta-groups* d, to send in messages. For each mutator m of a state-based CRDT we can define a delta-mutator m^δ such that:

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

The two approaches are summarized in Figure 20. 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 21 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

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$\langle E \rangle$

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

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

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

Fig. 21. State- vs delta-state examples: GCounter and ORSet.

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 $\mathsf{operation}_i(m^\delta)$
let $d = m^\delta(X_i)$
$X_i \leftarrow X_i \sqcup d$
$D_i \leftarrow D_i \sqcup d$

on $\mathsf{receive}_{j,i}(d)$
$X_i \leftarrow X_i \sqcup d$
$D_i \leftarrow D_i \sqcup d$

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

Fig. 22. Naive propagation algorithm for delta CRDTs.

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 normal state-based, delta-state aims for allowing arbitrary topologies. Therefore, a basic propagation mechanism aims for transitive propagation of information. Such an algorithm is presented in Figure 22. It simply keeps a single *delta-buffer* to broadcast to neighbors; upon some update joins the delta produced by the 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. Moreover, and unfortunately, this algorithm is too naive: the delta-buffer, even being periodically reset, easily tends to grow and become the full state.

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. 23. Join decompositions and optimal deltas.

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 [29]: 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 which: 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 both 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 [10], namely the concept of *join decompositions*.

The main concepts and results are summarized in Figure 23. 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 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 [29] 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.

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	

7 COMPARISON OF THE APPROACHES

A comparison of the approaches is now made, regarding issues such as open vs closed systems, partition tolerance, state size (including the issue of amortizing metadata overhead over several objects). A summary of these aspects (as well as other already discussed) is presented in Table 2.

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 without becoming effectively state-based.)

On the other hand, 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 which 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 lived partitions, which will cause unbounded memory consumption.

State-based CRDTs are better in this regards, 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 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 [13], that requires negligible metadata even for a large number of participants. In this respect, pure op-based cannot use this strategy, as it needs 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 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, 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, which may be problematic, being undesirable from the software engineering perspective.

8 COMPARISON WITH TOTAL-ORDER BASED APPROACHES

Some approaches to replicated data types are based on converging to a total order of update operations at all replicas. Some examples are: Eventually Consistent Transactions [19] and Cloud types [18], based on concurrent revisions [16], analogous to branches, where forking a new revision allows obtaining a new replica, and joining with a revision appends all updates of the joinee revision to the joiner revision. OpSets [37], where the set of visible updates is totally ordered using Lamport timestamps and replica ids. SECROs [51] also build a common total order, but in a smart way, trying different permutations compatible with happens-before in a deterministic way until operation pre- and post-conditions are met; to avoid unbounded operation history, a replica may *commit*, which clears the history and keeps the current state; unfortunately, operations concurrent with the commit will be discarded. ECROs [50] uses static analysis involving dependencies and commutativity to reorder operations, not restricted to being compatible with causality when unrelated commutative operations are involved; as in other cases, updates can be (conceptually) re-executed, and occasionally some operations may be discarded.

Regardless of the more or less smart way in which operations are ordered, in these approaches based on building a totally ordered update history, there is a mismatch between converged and observed history: even though states are converging to something resulting from a total order of

operations, queries cannot (in general) be placed at any point of that totally ordered history, as they do not observe (in general) a state resulting from the prefix of the totally ordered history up to that point. This is because there will be gaps in the observed history: the concurrent operations that happen to be placed as before some operations, which are themselves before the query point will not be seen by the query. One can say that "the past can be rewritten after the query has been performed". And even though update operations are (conceptually) re-executed to achieve convergence, queries are not re-executed.

Query re-execution does not make sense if the aim is to have immediate availability (to have an AP system), where we may want to use the query result, e.g., for an output to the external world. So, query results that may end up used to define subsequent operations (as arguments to those operations) and therefore, to define history itself, do not match what the query would result given the totally ordered history up to the query point. So, the totally ordered (converged) history does not match what the program which generated the history observed, and cannot alone be used to explain itself, given the program.

Essentially, all these approaches suffer from a fundamental problem of, to achieve availability, allowing "tentative writes" to be used for queries, before write stability [59], while building a totally ordered history that may depend on those tentative writes. Only by waiting for write stability before allowing queries would a fully self consistent totally-ordered history (including queries) be generated, but that imposes latency and is not partition tolerant, i.e., it resembles a classic strongly consistent CP system.

On the other hand, CRDTs embrace concurrency, normally having partial-order based specifications (something clear in pure op-based CRDTs). In CRDTs the observations are consistent with the partially ordered update history: queries can be placed in points of that history, with query results matching that history (i.e., resulting from the downward closed set of operations, given by the partial order, up to the observation point). History is not rewritten, updates are not reapplied, queries use the best knowledge up to the query moment. The whole history can be said to be consistent with the program: future operations that depend on previous query results (e.g., using a query result as an argument) are therefore consistent with observed history.

So, while for CRDTs the partially ordered history fully describes the outcome, including query results, while matching the program, in these other approaches the totally ordered update history cannot alone explain query results nor the history itself, given the program.

Auction example. To make this somewhat abstract discussion more concrete, consider an auction datatype similar to the one in Porre et al. [51], with two update operations (bid and close), and a query operation (winner). A bid is not allowed after a close. Given a bid issued concurrently with a close, approaches that build total orders "solve" the problem by reordering a concurrent bid that arrives later to become before the close. Consider however:

$$\mathsf{bid}(\mathit{Alice}, 50) \quad ; \quad [\quad \mathsf{close}() \,;\, \mathsf{winner}() \quad || \quad \mathsf{bid}(\mathit{Bob}, 60) \quad]$$

There are three possible options, all problematic:

(1) The first is for the system to aim for AP and zero latency, allowing queries to be executed immediately, in which case the winner query returns Alice. The concurrent Bob bid is reordered before close when it arrives later, which makes the converged state "winner is Bob with 60" to be different from what the query returned.
(2) A second option, to solve the above problem, is for the system to wait until close becomes causally stable before winner is allowed to execute, so that the Bob bid, reordered before close, makes the winner query return Bob, as in the converged state. But this system would not tolerate network partitions, as a partition prevents causal stability to be reached.

(3) A third (stranger) possibility, to remain partition tolerant, is for queries to be executed immediately, but over the state resulting from already stabilized updates, ignoring tentative updates. But in this case, the winner query would fail if invoked before close becomes causally stable. This third option seems to be the less interesting, as it leads to strange outcomes. The following run:

$$\mathsf{close}() \,;\, \mathsf{bid}(\mathit{Carol}, 70) \,;\, \mathsf{winner}() \quad || \quad \mathsf{bid}(\mathit{Bob}, 60)$$

would lead to (if close had not yet stabilized) the Carol bid to fail (auction already closed), which is the desired outcome, but the subsequent winner query to also fail (auction not yet closed), when these are sequential operations by the same process.

There seems to be no way out if all that can be done is to reorder operations and mimic a sequential execution. What would the "CRDT way" for this auction problem be like? Instead of trying to keep the original sequential API, the close operation could be replaced by a pair closing and closed update operations:

- closing would prevent subsequent bids (allowing concurrent bids to be accepted) but keep the auction open (not allowing yet a subsequent winner query);
- only the closed operation (to be invoked by a single auction administrator replica) would close the auction, choosing the winner, and allow a subsequent winner query to be invoked. Concurrent bids could afterwards be collected, to be reported as late.

This CRDT overcomes the above problems, allowing the auction application to choose what to do given a network partition. Normally, if causal stability information is available (e.g., an op-based CRDT over a TCD middleware), the administrator would only invoke closed when closing had become causally stable, thereby seeing all bids that had been issued. But there could be some auction policy about what to do under partitions, such as "after closing, the administrator will wait for bids up to one minute; replicas that are partitioned over one minute at the closing time may see their bids ignored". The CRDT, with a richer API, does not restrict the outcome, leaving such policies open, to be chosen by the application.

It is possible to have CRDTs that use a totally ordered arbitration to achieve convergence. A last-writer-wins (LWW) register is technically a CRDT, but not very useful. And achieving convergence is not difficult nor the main goal itself. COPS [44], the work that introduced the *causal+* term (causal consistency plus convergence) adopted LWW arbitration through Lamport timestamps. But doing so on a read-write API leads to lost updates. This is exactly the problem that motivated Amazon's Dynamo. Avoiding silently lost updates is indeed one of the main purposes of CRDTs. While higher level abstractions make this easier, restricting to totally ordered histories is not always expressive enough. Sometimes, as a CRDT designer, the expression "if I resort to arbitration I have already lost" comes to mind, meaning that using some artificial arbitration (like LWW) just for the sake of achieving convergence should be a last resort.

9 FINAL REMARKS

We came a long way from ad hoc optimistic replication with user-provided merge functions. CRDTs offer a principled approach to optimistic replication, allowing achieving causal consistency and strong convergence, the goals for partition tolerant, low latency systems. They solved the "losing concurrent updates" problem without requiring concurrency control or transactions. This goes beyond just mere convergence and allows desirable outcomes not possible if restricting to sequential histories.

Like sequential data types, they provide general abstractions useful to write distributed applications. Unlike sequential data types, its usage may be more difficult. They may embrace concurrency

in their specification and may force programmers to think in terms of concurrency. Nevertheless, CRDTs have been successfully applied in the industry, allowing systems not only "available" but with low response times even for large spatial spans.

The operation-based and state-based are substantially different approaches. The former more suitable to a known group of participants under good connectivity, and the later good for long partitions, disconnected operation and independent groups of participants to meet. Op-based allows small messages, better reuse of sequential data types (with commutative operations), and tends to have smaller state. State-based has typically larger states and messages, leading to less frequent propagation, and less "fresh" data. This is partially solved by the delta-state approach which, if using smart delta propagation, can potentially result in smaller messages, allowing more "freshness". Pure-op based is a special point in the design space for op-based, demanding causality information from the middleware to be efficient. Even though, it is limited compared with general op-based. Causal stability is an important ingredient in pure-op, to optimize state size, but nothing prevents it to be used in the general op-based, and 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 goes on in Hash Tables [6]. If CRDTs keep revealing themselves useful, research effort will not be the problem.

REFERENCES

[1] Mustaque Ahamad et al. "Causal Memory: Definitions, Implementation, and Programming". In: *Distributed Comput.* 9.1 (1995), pp. 37–49. DOI: 10.1007/BF01784241. URL: https://doi.org/10.1007/BF01784241.

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

[3] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. "Making Operation-Based CRDTs Operation-Based". In: *Distributed Applications and Interoperable Systems - 14th IFIP WG 6.1 International Conference, DAIS 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proceedings*. Ed. by Kostas Magoutis and Peter R. Pietzuch. Vol. 8460. Lecture Notes in Computer Science. Springer, 2014, pp. 126–140. DOI: 10.1007/978-3-662-43352-2_11. URL: https://doi.org/10.1007/978-3-662-43352-2_11.

[4] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. "Pure Operation-Based Replicated Data Types". In: *CoRR* abs/1710.04469 (2017). arXiv: 1710.04469. URL: http://arxiv.org/abs/1710.04469.

[5] Carlos Baquero et al. "Composition in State-based Replicated Data Types". In: *Bull. EATCS* 123 (2017). URL: http://eatcs.org/beatcs/index.php/beatcs/article/view/507.

[6] Michael A. Bender et al. "Iceberg Hashing: Optimizing Many Hash-Table Criteria at Once". In: *J. ACM* (Oct. 2023). ISSN: 0004-5411. DOI: 10.1145/3625817. URL: https://doi.org/10.1145/3625817.

[7] Annette Bieniusa et al. "An optimized conflict-free replicated set". In: *CoRR* abs/1210.3368 (2012). arXiv: 1210.3368. URL: http://arxiv.org/abs/1210.3368.

[8] Annette Bieniusa et al. "Brief Announcement: Semantics of Eventually Consistent Replicated Sets". In: *Distributed Computing - 26th International Symposium, DISC 2012, Salvador, Brazil, October 16-18, 2012. Proceedings*. Ed. by Marcos K. Aguilera. Vol. 7611. Lecture Notes in

Computer Science. Springer, 2012, pp. 441–442. DOI: 10.1007/978-3-642-33651-5_48. URL: https://doi.org/10.1007/978-3-642-33651-5%5C_48.

[9] G. Birkhoff. *Lattice Theory*. Colloquium publications. American Mathematical Society, 1940. URL: https://books.google.pt/books?id=o4bu3ex9BdkC.

[10] Garrett Birkhoff. "Rings of sets". In: *Duke Mathematical Journal*. 1937.

[11] Kenneth P. Birman and Thomas A. Joseph. "Reliable Communication in the Presence of Failures". In: *ACM Trans. Comput. Syst.* 5.1 (1987), pp. 47–76. DOI: 10.1145/7351.7478. URL: https://doi.org/10.1145/7351.7478.

[12] Kenneth P. Birman, André Schiper, and Pat Stephenson. "Lightweigt Causal and Atomic Group Multicast". In: *ACM Trans. Comput. Syst.* 9.3 (1991), pp. 272–314. DOI: 10.1145/128738.128742. URL: https://doi.org/10.1145/128738.128742.

[13] Manuel Bravo, Luís E. T. Rodrigues, and Peter Van Roy. "Saturn: a Distributed Metadata Service for Causal Consistency". In: *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*. Ed. by Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic. ACM, 2017, pp. 111–126. DOI: 10.1145/3064176.3064210. URL: https://doi.org/10.1145/3064176.3064210.

[14] Eric A. Brewer. "Towards robust distributed systems (abstract)". In: *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon, USA*. Ed. by Gil Neiger. ACM, 2000, p. 7. DOI: 10.1145/343477.343502. URL: https://doi.org/10.1145/343477.343502.

[15] Sebastian Burckhardt. "Principles of Eventual Consistency". In: *Found. Trends Program. Lang.* 1.1-2 (2014), pp. 1–150. DOI: 10.1561/2500000011. URL: https://doi.org/10.1561/2500000011.

[16] Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. "Concurrent programming with revisions and isolation types". In: *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*. Ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard. ACM, 2010, pp. 691–707. DOI: 10.1145/1869459.1869515. URL: https://doi.org/10.1145/1869459.1869515.

[17] Sebastian Burckhardt, Alexey Gotsman, and Hongseok Yang. *Understanding Eventual Consistency*. Tech. rep. MSR-TR-2013-39. 2013. URL: https://www.microsoft.com/en-us/research/publication/understanding-eventual-consistency/.

[18] Sebastian Burckhardt et al. "Cloud Types for Eventual Consistency". In: *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*. Ed. by James Noble. Vol. 7313. Lecture Notes in Computer Science. Springer, 2012, pp. 283–307. DOI: 10.1007/978-3-642-31057-7_14. URL: https://doi.org/10.1007/978-3-642-31057-7%5C_14.

[19] Sebastian Burckhardt et al. "Eventually Consistent Transactions". In: *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. Ed. by Helmut Seidl. Vol. 7211. Lecture Notes in Computer Science. Springer, 2012, pp. 67–86. DOI: 10.1007/978-3-642-28869-2_4. URL: https://doi.org/10.1007/978-3-642-28869-2%5C_4.

[20] Sebastian Burckhardt et al. "Replicated data types: specification, verification, optimality". In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. Ed. by Suresh Jagannathan and Peter Sewell. ACM, 2014, pp. 271–284. DOI: 10.1145/2535838.2535848. URL: https://doi.org/10.1145/2535838.2535848.

[21] Luca Cardelli and Peter Wegner. "On Understanding Types, Data Abstraction, and Polymorphism". In: *ACM Comput. Surv.* 17.4 (1985), pp. 471–522. DOI: 10.1145/6041.6042. URL: https://doi.org/10.1145/6041.6042.

[22] Miguel Castro and Barbara Liskov. "Practical Byzantine Fault Tolerance". In: *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*. Ed. by Margo I. Seltzer and Paul J. Leach. USENIX Association, 1999, pp. 173–186. URL: https://dl.acm.org/citation.cfm?id=296824.

[23] William R. Cook. "On understanding data abstraction, revisited". In: *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*. Ed. by Shail Arora and Gary T. Leavens. ACM, 2009, pp. 557–572. DOI: 10.1145/1640089.1640133. URL: https://doi.org/10.1145/1640089.1640133.

[24] Ole-Johan Dahl and Kristen Nygaard. "SIMULA - an ALGOL-based simulation language". In: *Commun. ACM* 9.9 (1966), pp. 671–678. DOI: 10.1145/365813.365819. URL: https://doi.org/10.1145/365813.365819.

[25] Giuseppe DeCandia et al. "Dynamo: amazon's highly available key-value store". In: *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*. Ed. by Thomas C. Bressoud and M. Frans Kaashoek. ACM, 2007, pp. 205–220. DOI: 10.1145/1294261.1294281. URL: https://doi.org/10.1145/1294261.1294281.

[26] Alan J. Demers et al. "Epidemic Algorithms for Replicated Database Maintenance". In: *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada, August 10-12, 1987*. Ed. by Fred B. Schneider. ACM, 1987, pp. 1–12. DOI: 10.1145/41840.41841. URL: https://doi.org/10.1145/41840.41841.

[27] Edsger W. Dijkstra. "Self-stabilizing Systems in Spite of Distributed Control". In: *Commun. ACM* 17.11 (1974), pp. 643–644. DOI: 10.1145/361179.361202. URL: https://doi.org/10.1145/361179.361202.

[28] Vitor Enes, Paulo Sérgio Almeida, and Carlos Baquero. "The Single-Writer Principle in CRDT Composition". In: *Proceedings of the Workshop on Programming Models and Languages for Distributed Computing, Barcelona, Spain, June 20, 2017*. Ed. by Christopher Meiklejohn and Heather Miller. ACM, 2017, 4:1–4:3. DOI: 10.1145/3166089.3168733. URL: https://doi.org/10.1145/3166089.3168733.

[29] Vitor Enes et al. "Efficient Synchronization of State-Based CRDTs". In: *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 2019, pp. 148–159. DOI: 10.1109/ICDE.2019.00022. URL: https://doi.org/10.1109/ICDE.2019.00022.

[30] Seth Gilbert and Nancy A. Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In: *SIGACT News* 33.2 (2002), pp. 51–59. DOI: 10.1145/564585.564601. URL: https://doi.org/10.1145/564585.564601.

[31] Per Brinch Hansen. *Operating System Principles*. USA: Prentice-Hall, Inc., 1973. ISBN: 0136378439.

[32] Maurice Herlihy. "Wait-Free Synchronization". In: *ACM Trans. Program. Lang. Syst.* 13.1 (1991), pp. 124–149. DOI: 10.1145/114005.102808. URL: https://doi.org/10.1145/114005.102808.

[33] Maurice Herlihy and Jeannette M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects". In: *ACM Trans. Program. Lang. Syst.* 12.3 (1990), pp. 463–492. DOI: 10.1145/78969.78972. URL: https://doi.org/10.1145/78969.78972.

[34] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (1969), pp. 576–580. DOI: 10.1145/363235.363259. URL: https://doi.org/10.1145/363235.363259.

[35] C. A. R. Hoare. "Monitors: An Operating System Structuring Concept". In: *Commun. ACM* 17.10 (1974), pp. 549–557. DOI: 10.1145/355620.361161. URL: https://doi.org/10.1145/355620.361161.

[36] C. A. R. Hoare. "Proof of Correctness of Data Representations". In: *Acta Informatica* 1 (1972), pp. 271–281. DOI: 10.1007/BF00289507. URL: https://doi.org/10.1007/BF00289507.

[37] Martin Kleppmann et al. "OpSets: Sequential Specifications for Replicated Datatypes (Extended Version)". In: *CoRR* abs/1805.04263 (2018). arXiv: 1805.04263. URL: http://arxiv.org/abs/1805.04263.

[38] Rivka Ladin et al. "Providing High Availability Using Lazy Replication". In: *ACM Trans. Comput. Syst.* 10.4 (1992), pp. 360–391. DOI: 10.1145/138873.138877. URL: https://doi.org/10.1145/138873.138877.

[39] Leslie Lamport. "A New Solution of Dijkstra's Concurrent Programming Problem". In: *Commun. ACM* 17.8 (1974), pp. 453–455. DOI: 10.1145/361082.361093. URL: https://doi.org/10.1145/361082.361093.

[40] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (1978), pp. 558–565. DOI: 10.1145/359545.359563. URL: https://doi.org/10.1145/359545.359563.

[41] Butler W. Lampson and Howard E. Sturgis. *Crash recovery in a distributed data storage system.* Tech. rep. Xerox Palo Alto Research Center, 1979.

[42] Richard J. Lipton and Jonathan S. Sandberg. *PRAM: A scalable shared memory.* Tech. rep. CS-TR-180-88. Princeton University, Department of Computer Science, 1988.

[43] Barbara H. Liskov and Stephen N. Zilles. "Programming with Abstract Data Types". In: *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages, Santa Monica, California, USA, March 28-29, 1974.* Ed. by Burt M. Leavenworth. ACM, 1974, pp. 50–59. DOI: 10.1145/800233.807045. URL: https://doi.org/10.1145/800233.807045.

[44] Wyatt Lloyd et al. "Don't settle for eventual: scalable causal consistency for wide-area storage with COPS". In: *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011.* Ed. by Ted Wobber and Peter Druschel. ACM, 2011, pp. 401–416. DOI: 10.1145/2043556.2043593. URL: https://doi.org/10.1145/2043556.2043593.

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

[46] Kristen Nygaard and Ole-Johan Dahl. "The development of the SIMULA languages". In: *ACM SIGPLAN Notices* 13.8 (1978), pp. 245–272. DOI: 10.1145/960118.808391. URL: https://doi.org/10.1145/960118.808391.

[47] Chris Okasaki. *Purely functional data structures.* Cambridge University Press, 1999. ISBN: 978-0-521-66350-2.

[48] Douglas Stott Parker Jr. et al. "Detection of Mutual Inconsistency in Distributed Systems". In: *IEEE Trans. Software Eng.* 9.3 (1983), pp. 240–247. DOI: 10.1109/TSE.1983.236733. URL: https://doi.org/10.1109/TSE.1983.236733.

[49] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. "Reaching Agreement in the Presence of Faults". In: *J. ACM* 27.2 (1980), pp. 228–234. DOI: 10.1145/322186.322188. URL: http://doi.acm.org/10.1145/322186.322188.

[50] Kevin De Porre et al. "ECROs: building global scale systems from sequential code". In: *Proc. ACM Program. Lang.* 5.OOPSLA (2021), pp. 1–30. DOI: 10.1145/3485484. URL: https://doi.org/10.1145/3485484.

[51] Kevin De Porre et al. "Putting Order in Strong Eventual Consistency". In: *Distributed Applications and Interoperable Systems - 19th IFIP WG 6.1 International Conference, DAIS 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17-21, 2019, Proceedings*. Ed. by José Pereira and Laura Ricci. Vol. 11534. Lecture Notes in Computer Science. Springer, 2019, pp. 36–56. DOI: 10.1007/978-3-030-22496-7_3. URL: https://doi.org/10.1007/978-3-030-22496-7%5C_3.

[52] Nuno M. Preguiça et al. "A Commutative Replicated Data Type for Cooperative Editing". In: *29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009), 22-26 June 2009, Montreal, Québec, Canada*. IEEE Computer Society, 2009, pp. 395–403. DOI: 10.1109/ICDCS.2009.20. URL: https://doi.org/10.1109/ICDCS.2009.20.

[53] Nuno M. Preguiça et al. "Dotted Version Vectors: Logical Clocks for Optimistic Replication". In: *CoRR* abs/1011.5808 (2010). arXiv: 1011.5808. URL: http://arxiv.org/abs/1011.5808.

[54] Hyun-Gul Roh et al. "Replicated abstract data types: Building blocks for collaborative applications". In: *J. Parallel Distributed Comput.* 71.3 (2011), pp. 354–368. DOI: 10.1016/j.jpdc.2010.12.006. URL: https://doi.org/10.1016/j.jpdc.2010.12.006.

[55] Douglas T. Ross and Clarence G. Feldmann. "Verbal and graphical language for the AED system: A progress report". In: *Proceedings of the SHARE design automation workshop, DAC '64, Cambridge, Massachusetts, USA, May 6-7, 1964*. ACM, 1964. DOI: 10.1145/800265.810743. URL: https://doi.org/10.1145/800265.810743.

[56] Yasushi Saito and Marc Shapiro. "Optimistic replication". In: *ACM Comput. Surv.* 37.1 (2005), pp. 42–81. DOI: 10.1145/1057977.1057980. URL: https://doi.org/10.1145/1057977.1057980.

[57] Marc Shapiro et al. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA, Jan. 2011, p. 50. URL: https://inria.hal.science/inria-00555588.

[58] Marc Shapiro et al. "Conflict-Free Replicated Data Types". In: *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*. Ed. by Xavier Défago, Franck Petit, and Vincent Villain. Vol. 6976. Lecture Notes in Computer Science. Springer, 2011, pp. 386–400. DOI: 10.1007/978-3-642-24550-3_29. URL: https://doi.org/10.1007/978-3-642-24550-3_29.

[59] Douglas B. Terry et al. "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System". In: *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*. Ed. by Michael B. Jones. ACM, 1995, pp. 172–183. DOI: 10.1145/224056.224070. URL: https://doi.org/10.1145/224056.224070.

[60] Douglas B. Terry et al. "Session Guarantees for Weakly Consistent Replicated Data". In: *Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS 94), Austin, Texas, USA, September 28-30, 1994*. IEEE Computer Society, 1994, pp. 140–149. DOI: 10.1109/PDIS.1994.331722. URL: https://doi.org/10.1109/PDIS.1994.331722.

[61] Paolo Viotti and Marko Vukolic. "Consistency in Non-Transactional Distributed Storage Systems". In: *ACM Comput. Surv.* 49.1 (2016), 19:1–19:34. DOI: 10.1145/2926965. URL: https://doi.org/10.1145/2926965.

[62] Werner Vogels. "Eventually consistent". In: *Commun. ACM* 52.1 (2009), pp. 40–44. DOI: 10.1145/1435417.1435432. URL: https://doi.org/10.1145/1435417.1435432.

[63] Matthew Weidner and Paulo Sérgio Almeida. "An oblivious observed-reset embeddable replicated counter". In: *PaPoC@EuroSys 2022: Proceedings of the 9th Workshop on Principles and Practice of Consistency for Distributed Data, Rennes, France, April 5 - 8, 2022*. Ed. by Adriana Szekeres and K. C. Sivaramakrishnan. ACM, 2022, pp. 47–52. DOI: 10.1145/3517209.3524084. URL: https://doi.org/10.1145/3517209.3524084.

[64] Niklaus Wirth and C. A. R. Hoare. “A contribution to the development of ALGOL”. In: *Commun. ACM* 9.6 (1966), pp. 413–432. DOI: 10.1145/365696.365702. URL: https://doi.org/10.1145/365696.365702.