Making Operation-Based CRDTs Operation-Based

Carlos Baquero, Paulo Almeida, Ali Shoker

Making Operation-based CRDTs Operation-based

Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker

HASLab/INESC TEC and Universidade do Minho, Portugal

Abstract. Conflict-free Replicated Datatypes (CRDT) are usually classified as either state-based or operation-based. However, the standard definition of op-based CRDTs is very encompassing, allowing even sending the full-state, blurring the distinction. We introduce *pure* op-based CRDTs, that can only send operations to other replicas, drawing a clear distinction from state-based ones. Datatypes with commutative operations can be trivially implemented as pure op-based CRDTs using standard reliable causal delivery. We propose an extended API – *tagged reliable causal broadcast* – that provides causality information upon delivery, and show how it can be used to also implement other datatypes having non-commutative operations, through the use of a *PO-Log* – a partially ordered log of operations – inside the datatype. A semantically-based PO-Log compaction framework, using both causality and what we denote by *causal stability*, allows obtaining very compact replica state for pure op-based CRDTs, while also benefiting from small message sizes.

1 Introduction

Eventual consistency [1] is a relaxed consistency model that is often adopted by large-scale distributed systems [2–5] where losing availability is normally not an option, whereas delayed consistency is acceptable. In eventually consistent systems, data replicas are allowed to temporarily diverge, provided that they can eventually be reconciled into a common consistent state. Reconciliation (or merging) used to be error-prone, being application-dependent, until new datatype-dependent models like the Conflict-free Replicated DataTypes (CRDTs) [6, 7] were recently introduced. CRDTs allow both researchers and practitioners to design correct replicated datatypes that are always available, and are guaranteed to eventually converge once all operations are known to all replicas. Though CRDTs have been successfully deployed in practice [2], a lot of work is still required to improve their designs and performance.

CRDTs support two complementary designs: operation-based (or simply, op-based) and state-based. In principle, op-based designs are supposed to disseminate operations, while state-based designs disseminate object states. In op-based designs [8, 7], the execution of an operation is done in two phases: *prepare* and *effect*. The former is performed only on the local replica and looks at the operation and current state to produce a message that aims to represent the operation, which is then shipped to all replicas. Once received, the representation of the

operation is applied remotely using *effect*. Different replicas are guaranteed to converge as long as messages are disseminated through a reliable causal broadcast messaging middleware, and *effect* is designed to be commutative for concurrent operations. On the other hand, in a state-based design [9, 7], an operation is only executed on the local replica state. A replica propagates its local changes to other replicas through shipping its entire state. A received state is incorporated with the local state via a *merge* function that, deterministically, reconciles the *merged* states. To maintain convergence, *merge* is defined as a join: a least upper bound over a join-semilattice [9, 7].

Typically, state-based CRDTs support ad hoc dissemination of states and can handle duplicate and out-of-order delivery of messages without breaking causal consistency; however, they impose complex state designs and store extra meta-data. On the other hand, in the systems where the message dissemination layer guarantees reliable causal broadcast, operation-based CRDTs have more advantages as they can allow for simpler implementations, concise replica state, and smaller messages.

In standard op-based CRDTs the designer is given much freedom in defining *prepare*, namely using the state in an arbitrary way. This is needed to have the *effect*s of concurrently invoked data-type operations commute, and thus provide replica convergence despite the absence of causality information in current causal delivery APIs. This forces current op-based designs to include causality information in the state to be used in *prepare*, sent in messages, and subsequently used in *effect*. The designer ends up intervening in many components (the state, *prepare*, *effect*, and *query* functions) in an ad hoc way. This can result in large complex state structures and also large messages.

Currently, a *prepare* not only builds messages that duplicate the information already present in the middleware (even if it is not currently made available), but causality meta-data is often incorporated in the object state, hence, reusing design choices similar to those used in state-based approaches. Such designs, made to work assuming little messaging guarantees, impose larger state size and do not fully exploit causal delivery guarantees. This freedom in current op-based designs is against the spirit of 'sending operations', and leads to confusion with the state-based approach. Indeed, in the current op-based framework, a *prepare* can return the full state, and an *effect* can do a full state-merge (which mimics a state-based CRDT) [9, 7].

We believe that the above weaknesses can be avoided if the causality meta-data can be provided by the messaging middleware. Causal broadcast implementations already posses that information internally, but it is not exposed to clients. In this paper we propose and exploit such an extended API to achieve both simplicity and efficiency in defining op-based CRDTs.

We introduce a *Pure* Op-Based CRDT framework, in which *prepare* cannot inspect the state, being limited to returning the operation (including potential parameters). The entire logic of executing the operation in each replica is delegated to *effect*, which is also made generic (i.e., not datatype dependent). For pure op-based CRDTs, we propose that the object state is a *partially ordered log*

of operations – a PO-Log . Causality information is provided by an extended messaging API: *tagged reliable causal broadcast* (TRCB). We use this information to preserve convergence and also design compact and efficient CRDTs through a *semantically based PO-Log compaction* framework, which makes use of a datatype-specific *obsolescence* relation, defined over timestamp-operation pairs.

Furthermore, we propose an extension that improves the design and implementation of op-based CRDTs through decomposing the state into two components: a PO-Log (as before), and a causality-stripped-component which, in many cases, will be simply a standard sequential datatype. The idea is that operations are kept only transiently in the PO-Log, but once they become *causally stable*, causality meta-data is stripped, and the operations are stored in the sequential datatype. This reduces the storage overhead to a level that was never achieved before in CRDTs, neither state-based nor op-based.

2 System Model and Notations

2.1 System and Fault Models

The system is composed of a fixed set of nodes, each with a globally unique identifier in a set I. Nodes execute operations at different speeds and communicate using asynchronous message passing, abstracted by reliable causal broadcast (or gossip in the brief discussion about state-based CRDTs). Messages can be lost, reordered or duplicated, and the system can experience arbitrary, but transient, partitions. A node can fail by crashing and can recover later on; upon recovery, the last durable state of a node is assumed to be intact (not destroyed). We do not consider Byzantine faults. A fixed membership is assumed for causal broadcast: messages towards a node that is temporarily crashed or partitioned are buffered until it becomes reachable.

For presentation purposes, and without loss of generality, we consider a single object that is replicated at each node; each replica initially starts with the same state. Once a datatype operation is locally applied on a replica, the latter can diverge from the other replicas, but it may eventually convergence as new operations arrive. A local operation is applied atomically on a given replica.

2.2 Definitions and Notations

Σ denotes the type of the state. $\mathcal{P}(V)$ denotes a *power set* (the set of all subsets of V). The initial state of a replica i is denoted by $\sigma_i^0 \in \Sigma$. Operations are taken from a set O and can include arguments (in which case they are surrounded by brackets, e.g., inc and $[\mathsf{add}, v]$). We use total functions $K \to V$ and maps (partial functions) $K \hookrightarrow V$ from keys to values, both represented as sets of pairs (k, v). Given a function m, the notation $m\{k \mapsto v\}$ maps k to v, and behaves like m on other keys, e.g., Fig. 1a.

$$\Sigma = I \to \mathbb{N} \quad \sigma_i^0 = \{(r, 0) \mid r \in I\}$$
$$\mathsf{apply}_i(\mathsf{inc}, m) = m\{i \mapsto m(i) + 1\}$$
$$\mathsf{eval}_i(\mathsf{rd}, m) = \sum_{r \in I} m(r)$$
$$\mathsf{merge}_i(m, m') = \{(r, \max(m(r), m'(r))) \mid r \in I\}$$

(a) State-based counter

$$\Sigma = \mathbb{N} \quad \sigma_i^0 = 0$$
$$\mathsf{prepare}_i(\mathsf{inc}, n) = \mathsf{inc}$$
$$\mathsf{effect}_i(\mathsf{inc}, n) = n + 1$$
$$\mathsf{eval}_i(\mathsf{rd}, n) = n$$

(b) Op-based counter

Fig. 1: Counter CRDT in both state-based and op-based approaches

2.3 Conflict-free Replicated Data Types Approaches

State-Based CRDTs. These CRDTs maintain a *state* representation of an object, which evolves according to a well defined partial order. A state evolves via executing datatype operations or through applying a *join* operation, which merges any two states, thus resolving conflicting states. State-based replicas of an object converge by always shipping the entire local state, and applying the join operation on received states. State-based CRDTs are costly as the entire replica state must be shipped, but they demand less guarantees from the network because joins are designed to be commutative, idempotent, and associative. Figure 1a represents a state-based increment-only counter. In this paper we do not address state-based CRDTs.

Operation-Based CRDTs. In op-based CRDTs, representations of operations issued at each node are reliably broadcast to all replicas. Once all replicas receive all issued operations (on all nodes), they eventually converge to a single state, if: (a) operations are broadcast via a reliable causal broadcast, and (b) 'applying' representations of concurrently issued operations is commutative. Op-based CRDTs can often have a simple and compact state since they can rely on the exactly-once delivery properties of the broadcast service, and thus do not have to explicitly handle non-idempotent operations. Figure 1b represents an op-based increment-only counter. The state contains a simple integer counter that is incremented for each inc operation that is delivered.

The API of the underlying middleware at each node i provides an interface method $\mathsf{cbcast}_i(m)$ that sends a message m using causal broadcast. When applying an operation o at some node i with state σ, function $\mathsf{prepare}_i(o, \sigma)$ is called returning a message m. This message is then broadcast by calling $\mathsf{cbcast}_i(m)$. Once m is delivered to each destination node j, $\mathsf{effect}_j(m, \sigma)$ is called, returning the new replica state σ'. For each node that broadcasts a given operation, the broadcast, the corresponding local delivery, and the *effect* on the local state are executed atomically. When a query operation q is performed, $\mathsf{eval}_i(q, \sigma)$ is invoked. eval takes the query and the state as input and may return a result (leaving the state unchanged).

$$
\begin{aligned}
\Sigma = \mathbb{N} \times \mathcal{P}(I \times \mathbb{N} \times V) \qquad & \sigma_i^0 = (0, \{\}) \\
\mathsf{prepare}_i([\mathsf{add}, v], (n, s)) &= [\mathsf{add}, v, i, n+1] \\
\mathsf{effect}_i([\mathsf{add}, v, i', n'], (n, s)) &= (n' \textbf{ if } i = i' \textbf{ otherwise } n, s \cup \{(v, i', n')\}) \\
\mathsf{prepare}_i[\mathsf{rmv}, v], (n, s)) &= [\mathsf{rmv}, \{(v', i', n') \in s \mid v' = v\}] \\
\mathsf{effect}_i([\mathsf{rmv}, r], (n, s)) &= (n, s \setminus r) \\
\mathsf{eval}(\mathsf{rd}, (n, s)) &= \{v \mid (v, i', n') \in s\}
\end{aligned}
$$

Fig. 2: Standard op-based observed-remove add-wins set

3 Pure Op-based CRDTs

In this section we introduce *pure op-based CRDTs* and discuss what datatypes can be implemented as pure using standard causal broadcast.

Definition 1 (Pure op-based CRDT). *An op-based CRDT is* pure *if messages contain only the operation (including arguments, if any). Given operation o and state σ,* prepare *is always defined as:*

$$\mathsf{prepare}(o, \sigma) = o.$$

This means that prepare cannot build an arbitrary message depending on the current state; in fact, in pure op-based CRDTs the operation can be immediately broadcast without even reading the replica state. As an example, the counter in Fig. 1b is pure op-based, while the observed-remove set implementation (from [6]) in Fig. 2 is not, because in a remove operation prepare builds a set of triples present in the current state, to be removed from the state at each replica when performing effect.

3.1 Pure Implementations of Commutative Datatypes

As we discuss now, the pure model of op-based CRDTs can be directly applied, using standard reliable causal broadcast [10], to implement datatypes whose operations are commutative.

Definition 2 (Commutative datatype). *A concurrent datatype is commutative if (a) for any operations f and g, their (sequential) invocation commutes: $f(g(\sigma)) = g(f(\sigma))$, and (b) concurrent invocations are defined as equivalent to some linearization.*

Commutative datatypes reflect a *principle of permutation equivalence* [11] stating that "If all sequential permutations of updates lead to equivalent states, then it should also hold that concurrent executions of the updates lead to equivalent states".

As the extension to concurrent scenarios follows directly from their sequential definition, with no room for design choices, commutative datatypes can have a

$$\Sigma = \mathbb{N} \quad \sigma_i^0 = 0$$
$$\mathsf{prepare}_i(o, \sigma) = o$$
$$\mathsf{effect}_i(\mathsf{inc}, n) = n + 1$$
$$\mathsf{effect}_i(\mathsf{dec}, n) = n - 1$$
$$\mathsf{eval}_i(\mathsf{rd}, n) = n$$

(a) Pure PN-counter

$$\Sigma = \mathcal{P}(V) \quad \sigma_i^0 = \{\}$$
$$\mathsf{prepare}_i(o, \sigma) = o$$
$$\mathsf{effect}_i([\mathsf{add}, v], s) = s \cup \{v\}$$
$$\mathsf{eval}_i(\mathsf{rd}, s) = s$$

(b) Pure grow-only set

Fig. 3: Pure op-based CRDTs for commutative datatypes

standard sequential specification and implementation. As such, a pure op-based CRDT implementation is trivial: as when using the standard causal broadcast, the message returned from prepare, containing the operation, will arrive exactly once at each replica, it is enough to make effect consist simply in applying the received operation to the state, over a standard sequential datatype, i.e., defining for any datatype operation o:

$$\mathsf{effect}_i(o, \sigma) = o(\sigma).$$

Two examples of commutative datatypes, presented in Fig. 3, are: a PN-counter with inc and dec operations; a grow-only set (G-set) with add operation. Both cases use a standard sequential datatype for the replica state, and applying effect is just invoking the corresponding operation in the sequential datatype. Both these examples explore commutativity and rely on the exactly-once delivery, leading to a trivial pure implementation.

3.2 Non-commutative Datatypes

In the case where datatype operations are not commutative, such as a set with add and rmv operations, where $\mathsf{add}(v, \mathsf{rmv}(v, s)) \neq \mathsf{rmv}(v, \mathsf{add}(v, s))$, we have two reasons that prevent effect from being simply applying the operation over a sequential datatype.

One reason is that, even when the semantics of concurrent invocations can be defined as equivalent to some linearization of those operations, the messages corresponding to concurrent operations will be, in general, delivered in different orders in different replicas. Therefore, as the operations do not commute, simply applying them in different orders in different replicas makes replicas diverge. Under the assumption of causal delivery and the aim of convergence, effect must always be commutative, and therefore, cannot be defined directly as operations that are not commutative themselves. It must be defined in some other way.

The other reason is that it is useful to specify concurrent datatypes in which the outcomes of concurrent executions are not equivalent to some linearization. The best example is the multi-value register, where two concurrent writes make

state:
    $\sigma_i \in \Sigma$
on $\mathsf{operation}_i(o)$:
    $\mathsf{tcbcast}_i(\mathsf{prepare}(o, \sigma_i))$
on $\mathsf{tcdeliver}_i(m, t)$:
    $\sigma_i := \mathsf{effect}(m, t, \sigma_i)$
on $\mathsf{tcstable}_i(t)$:
    $\sigma_i := \mathsf{stable}(t, \sigma_i)$

Algorithm 1: Distributed algorithm for node i using tagged causal broadcast

a read in their causal future return a set with both values written. This outcome could not arise under a sequential specification.

In general, a concurrent datatype will have a specification depending on the partial order of operations over the datatype. Given that such information about that partial order is already present in metadata in causal delivery middleware, we propose an approach for pure op-based CRDTs for general non-commutative datatypes that leverages this metadata, now exposed by an extended causal delivery API, what we call *tagged reliable causal broadcast*.

4 Tagged Reliable Causal Broadcast (TRCB)

A common implementation strategy for a reliable causal broadcast service [12] is to assign a vector clock to each message broadcast and use the causality information in the vector clock to decide at each destination when a message can be delivered. If a message arrives at a given destination before causally preceding messages have been delivered, the service delays delivery of that message until those messages arrive and are delivered. Unlike totally ordered broadcast, which requires a global consensus on the delivery order, causal broadcast can progress with local decisions. For general datatypes, causal consistency is likely the strongest consistency criteria compatible with an always-available system that eventually converges [13].

By leveraging this information, we can specify a reliable causal broadcast service with an extended API, and refer to its broadcast operation at each replica i as $\mathsf{tcbcast}_i(m)$. Algorithm 1, running on each node i, shows how the events triggered by the tagged causal delivery service are used to invoke the generic functions for pure op-based CRDTs: prepare, effect and stable; these functions, in different variants, will be discussed in the following sections. This extended service provides nodes with information about two aspects.

4.1 Partial order

The first salient difference is that message delivery on each node i, given by the event $\mathsf{tcdeliver}_i(m, t)$, provides not only the message m itself, but also the vector clock timestamp t corresponding to m. When implementing pure op-based CRDTs, in which only the operations are sent in messages, we can use the

timestamp supplied by the service upon delivery in the definition of effect; i.e., we can have $\mathsf{effect}(o, t, s)$ as a function of the operation, the timestamp and the current state.

As we will see in the next section, this information about the partial order can be embedded in the state in a general way so that effect is commutative and reference implementations of general possibly non-commutative datatypes can be obtained, following their specification. Moreover, in Sect. 6 we will see how realistic efficient pure CRDTs can be obtained, in which the use of this causality information, together with the semantics of the datatype operations is essential.

4.2 Causal stability

Definition 3 (Causal Stability). *A clock t, and corresponding message, is* causally stable *at node i when all messages subsequently delivered at i will have timestamp $t' \geq t$;*

This implies that no message with a timestamp t' concurrent with t can be delivered at i when t is causally stable at i. This notion differs from classic message stability [10] in which a message is stable if it has been received by all nodes. Here we not only need this to happen but also that no further concurrent messages may be delivered. Therefore, causal stability is a stronger notion, implying classic message stability.

The extended API will provide an event $\mathsf{tcstable}_i(t)$ which will be triggered when it is determined that t is stable at i. The middleware at node i can check if timestamp t is causally stable at i by checking if a message with timestamp $t' \geq t$ has already been delivered at i from every other other node j, i.e.:

$$\mathsf{tcstable}_i(t) \textbf{ when } \forall j \in I \setminus \{i\} \cdot \exists t' \in \mathsf{delivered}_i() \cdot \mathsf{origin}(t') = j \wedge t \leq t',$$

where $\mathsf{delivered}_i()$ returns the set of messages that have been delivered at node i, while $\mathsf{origin}(t)$ denotes the node from where the message corresponding to t has been sent. To evaluate this clause efficiently, the middleware only needs to keep the most recently delivered timestamp from each origin [14].

We will see in Sect. 6 how causal stability can be used to reduce CRDT state size, by stripping causality information from causally stable operations.

5 Pure CRDTs Based on a Partially Ordered Log

Having a tagged causal broadcast service, it is now possible to obtain a universal mechanism for obtaining pure reference implementations for any (possibly non-commutative) concurrent datatype in which semantics are defined over the partial order of operations.

The reference mechanism, presented in Fig. 4, uses a uniform notion of state for a replica: a *partially ordered log of operations*, what we call a PO-Log. This uses the ordering information offered by the messaging middleware to keep information about concurrent operations, not trying to impose a local total-order over them, contrary to a classic sequential log.

$$\Sigma = T \hookrightarrow O \quad \sigma_i^0 = \{\}$$
$$\mathsf{prepare}(o, s) = o$$
$$\mathsf{effect}(o, t, s) = s \cup \{(t, o)\}$$
$$\mathsf{eval}(q, s) = [\text{datatype-specific query function over partial order}]$$

Fig. 4: PO-Log based reference implementation for pure op-based CRDTs

$$\Sigma = T \hookrightarrow O \quad \sigma_i^0 = \{\}$$
$$\mathsf{prepare}(o, s) = o \qquad (\text{with } o \text{ either } [\mathsf{add}, v] \text{ or } [\mathsf{rmv}, v])$$
$$\mathsf{effect}(o, t, s) = s \cup \{(t, o)\}$$
$$\mathsf{eval}(\mathsf{rd}, s) = \{v \mid (t, [\mathsf{add}, v]) \in s \wedge \nexists (t', [\mathsf{rmv}, v]) \in s \cdot t < t'\}$$

Fig. 5: PO-Log based observed-remove add-wins set

The PO-Log can be defined as a map (a partial function) $T \hookrightarrow O$ from message timestamps (as given by the tagged causal broadcast service) to the corresponding operation. Here we have a universal datatype-independent definition of effect as:

$$\mathsf{effect}(o, t, s) = s \cup \{(t, o)\},$$

which is trivially commutative, as needed. Only the query functions will need datatype-specific definitions according to desired semantics. Their definition over the PO-Log will typically be a direct transposition of their specification.

Figure 5 shows a pure PO-Log based implementation of an *add-wins* observed-remove set over the new tcbcast service. The *add-wins* semantic is defined in the rd query function: the set of values reported to be in the set are those values that have been added with no rmv causally in the future of the add. Another example, shown in Fig. 6, is a multi-value register. Here a read reports the set of all concurrently written values that have not been subsequently overwritten.

These reference implementations are not realistic to be actually used, namely because state size in each replica is linear with the number of operations. They are a starting point from which actual efficient implementations can be derived, by semantically based PO-Log compaction, as we show in the next section. But they are relevant, as they provide a clear description of the concurrent semantics of the replicated datatype. This is possible since we are capturing the partial ordered set of all operations delivered to each replica. (A similar approach to express the semantics is found in [15] when relating to the *visibility* relation.)

6 Semantically Based PO-Log Compaction

We now show how PO-Log based CRDTs can be made efficient by performing PO-Log compaction. There are two ingredients that we explore. The first one is

$$\Sigma = T \hookrightarrow O \quad \sigma_i^0 = \{\}$$
$$\mathsf{prepare}([\mathsf{wr}, v], s) = [\mathsf{wr}, v]$$
$$\mathsf{effect}([\mathsf{wr}, v], t, s) = s \cup \{(t, [\mathsf{wr}, v])\}$$
$$\mathsf{eval}(\mathsf{rd}, s) = \{v \mid (t, [\mathsf{wr}, v]) \in s \wedge \nexists (t', [\mathsf{wr}, v']) \in s \cdot t < t'\}$$

Fig. 6: PO-Log based multi-value register

$$\Sigma = T \hookrightarrow O \quad \sigma_i^0 = \{\}$$
$$\mathsf{prepare}(o, s) = o$$
$$\mathsf{effect}(o, t, s) = \{x \in s \mid \neg\, \mathsf{obsolete}(x, (t, o))\} \cup \{(t, o) \mid x \in s \Rightarrow \neg\, \mathsf{obsolete}((t, o), x)\}$$
$$\mathsf{obsolete}() = [\text{datatype-specific relation to identify obsolete operations}]$$
$$\mathsf{eval}(q, s) = [\text{datatype-specific query function over partial order}]$$

Fig. 7: Reference implementation for PO-Log compaction

to prune the PO-Log after each operation is delivered in the effect, so as to keep the minimum number of operations such that all queries return the same result as when the full PO-Log is present. The second one is to explore causal stability information, to discard timestamp information for elements once they become stable, possibly merging some elements.

6.1 Exploring Causality Information

As the possibility of discarding operations while preserving semantics is datatype dependent, we propose a unified framework which includes the PO-Log, prepare, and a more sophisticated effect which makes use of a datatype-specific relation to discard operations made irrelevant by newer arrivals, according to both operation content and corresponding timestamp, as shown in Fig. 7.

This relation between pairs timestamp-operation – $\mathsf{obsolete}((t, o), (t', o'))$ – is used by effect in the following way: when a new pair (t, o) is delivered to a replica, effect discards from the PO-Log all elements x such that $\mathsf{obsolete}(x, (t, o))$ holds; also, the delivered pair (t, o) is only inserted into the PO-Log if it is not redundant itself, according to the current elements, i.e., if for any current x in the PO-Log $\mathsf{obsolete}((t, o), x)$ is false. This relation is not restricted to be a partial-order, but can be a more general relation, allowing, e.g., a newly arrived operation to discard others in the PO-Log without necessarily being itself added.

It is easy to see by simple induction that this execution mechanism provides the invariant that for any two different pairs p_1 and p_2 in the PO-Log, $\mathsf{obsolete}(p_1, p_2)$ is false. This invariant allows reasoning about the datatype, namely to be able to write simplified query functions that give the same result over the compact PO-Log as the original query functions over the full PO-Log.

$$\begin{aligned}
\mathsf{obsolete}((t,[\mathsf{add},v]),(t',[\mathsf{add},v'])) &= t < t' \wedge v = v' \\
\mathsf{obsolete}((t,[\mathsf{add},v]),(t',[\mathsf{rmv},v'])) &= t < t' \wedge v = v' \\
\mathsf{obsolete}((t,[\mathsf{rmv},v]),x) &= \mathrm{true} \\
\mathsf{eval}(\mathsf{rd},s) &= \{v \mid (t,[\mathsf{add},v]) \in s\}
\end{aligned}$$

Fig. 8: Observed-remove add-wins set with PO-Log compaction

$$\begin{aligned}
\mathsf{obsolete}((t,[\mathsf{wr},v]),(t',[\mathsf{wr},v'])) &= t < t' \\
\mathsf{eval}(\mathsf{rd},s) &= \{v \mid (t,[\mathsf{wr},v]) \in s\}
\end{aligned}$$

Fig. 9: Multi-value register with PO-Log compaction

An observed-remove add-wins set using the PO-Log compaction framework can be seen in Fig. 8 (which presents only the datatype-specific functions). Here it can be clearly seen that obsolete was defined directly according to the essence of the datatype: a subsequent add obsoletes a previous add of the same value; a rmv obsoletes an add of the same value. The more interesting rule is that any rmv is made obsolete by any other timestamp-operation pair; this implies that a rmv can only exist as the single element of a PO-Log (if it was inserted into an empty PO-Log), being discarded once other operation arrives (including another rmv), and never being inserted into a non-empty PO-Log. This reflects the *add-wins* nature, in which the role of a rmv is basically to discard same-value additions in its causal past. Under the now compacted PO-Log, the query function rd can be defined in a simple way, and clearly seen to give the same result as the original one over the full PO-Log (in Fig. 5).

Another example where PO-Log compaction leads to an efficient datatype is the multi-value register, in Fig. 9, where it is obvious the effect of a write in making all writes in its causal past obsolete, regardless of value written. The set of concurrent writes that have not been made obsolete will be returned in a read, which is equivalent to the original definition in Fig. 6 over the full PO-Log.

6.2 Exploring Causal Stability Information

The second component of PO-Log compaction involves using causal stability to strip logical clocks from the PO-Log. From the definition of causal stability, if some pair (t, o) is in the PO-Log, with t causally stable, all future deliveries (t', o') used in effect will be causally in the future, i.e., $t' > t$.

Because effect only compares, through obsolete, new arrivals and PO-Log elements – but never PO-Log elements among themselves – and for a stable t, all future deliveries will be causally in its future, the value of t is no longer needed, and it can be replaced by any timestamp that is less than all other

$$
\begin{aligned}
\Sigma = \mathcal{P}(O) \times (T \hookrightarrow O) \quad & \sigma_i^0 = (\{\}, \{\}) \\
\mathsf{prepare}(o, (s,p)) &= o \\
\mathsf{effect}(o, t, (s,p)) &= (s', p'), \text{ where} \\
s' &= \{x \in s \mid \neg\, \mathsf{obsolete}((\bot, x), (t, o))\} \\
p' &= \{x \in p \mid \neg\, \mathsf{obsolete}(x, (t, o))\} \\
&\quad \cup \{(t,o) \mid x \in p \Rightarrow \neg\, \mathsf{obsolete}((t,o), x)\} \\
\mathsf{stable}(t, (s,p)) &= (s \cup p(t), p \setminus \{(t, p(t))\}) \\
\mathsf{obsolete}() &= \text{[datatype-specific relation to identify obsolete operations]} \\
\mathsf{eval}(q, (s,p)) &= \text{[datatype-specific query function over PO-Log]}
\end{aligned}
$$

Fig. 10: Full PO-Log compaction framework exploring causal stability

timestamps: the bottom ($\bot$) of the timestamp domain; e.g., a null map $\{\}$ for a vector-clock timestamp.

In practice this means that, instead of having timestamps that are maps or vectors with size linear on the number of replicas, we can have a special marker denoting bottom (e.g. a null pointer). This greatly diminishes the size of a replica state for two reasons. For some CRDTs where size may be a problem, like sets of integers, values may take considerably less space than timestamps, which constitute the great percentage of state size; stripping a timestamp from an element can be a huge improvement per element. The second reason is that, again in such scenarios with large states, the percentage of elements in the PO-Log that are not yet causally stable will be quite small, with most already stable. This means that this optimization, which gives good results per element, will be applied to most elements, leading to a large overall improvement.

Instead of being a map $T \hookrightarrow O$, the PO-Log is split in $\mathcal{P}(O) \times (T \hookrightarrow O)$, detaching into a plain set all operations that have stable timestamps (i.e., making the $\bot$ timestamp implicit). A new framework using the split PO-Log is presented in Fig. 10. In effect a new delivery possibly discards elements both from the set of operations and the partially ordered set of operations; elements from the latter are used to decide on the addition of the new delivery, as before. (The possibility of a stable operation obsoleting a new arrival, in its causal future, is not considered, but this can easily be changed if some example shows its usefulness.) In stable, the operation corresponding to a stable timestamp is fetched, to be added to the set and the corresponding entry removed from the map.

The PO-Log split allows implementations tailored to specific datatypes to achieve further improvements. As an example, an or-set will have only $(t, [\mathsf{add}, v])$ entries in its PO-Log (except for the singleton $\{(t, [\mathsf{rmv}, v])\}$) which can be prevented by a special rule); when elements become causally stable, only the value v needs to be migrated and not the operation name. The set component of the split PO-Log becomes a plain set of values, allowing traditional implementations of sets to be used; e.g., a bitmap if the datatype is for a dense set of integers.

By exploiting both causality and causal stability information, made available by the proposed tagged causal delivery API, we have paved the way for these optimizations that allow pure op-based CRDTs that are much more suitable for large datatypes than current designs.

7 Related Work

Weakly Consistent Replication. The design of replicated systems that are always available and eventually converge can be traced back to historical designs in [14, 16, 17], among others. Lazy Replication [18] allows enforcing causal consistency, but may apply concurrent operations in different orders in different replicas, possibly leading to divergence if operations are not commutative; TSAE [19] also either applies concurrent operations in possibly different orders, or allows enforcing a total order compatible with causality, at the cost of delaying message delivery. Both these systems use a message log, the former with complete causality information, but the log is *pre-delivery*, unseen by the application: operations are applied sequentially to the current state and queries use only the state. In our framework the PO-Log is *post-delivery*, being part of the datatype state, maintains causality information and is used in query operations.

Conflict-Free Replicated Data Types. The formalization of the commutativity requirements for concurrent operations in replicated datatypes was introduced in [8, 20], and that of the state based semi-lattices was presented in [9]. Afterwards, the integration of the two models with many extensions was presented in Conflict-free Replicated Datatypes [6, 7]. Currently, CRDTs have made their way into the industry through designing highly available scalable systems in cloud databases like RIAK [2], and mobile gaming industry such as Rovio [21].

Message Stability. The notion of message stability was defined in [10] to represent a message that has been received by all recipients; each replica can discard any message it knows to be stable after delivering it. Similar notions are used in Lazy Replication [18] and TSAE [19]. In all these cases the aim is message garbage collection. Our definition of *causal stability* is the stronger notion that *no more concurrent messages will be delivered*; we use it inside the datatype to discard causality information while keeping the operation. Causal stability is close to what is used in the mechanics of Replicated Growable Arrays (RGA) [20], although no definition is presented there.

Message Obsolescence. Semantically reliable multicast [22] uses the concept of *message obsolescence* to avoid delivering messages made redundant by some newly arrived message, where obsolescence is a strict partial order that is a subset of causality, possibly relating messages from the same sender or totally ordered messsages from different senders. Our obsolescence relation is more general, being defined on clock-operation pairs, and can relate concurrent messages. Also, it is defined per-datatype, being used inside each datatype, post-delivery.

8 Conclusions

In this paper we improved the CRDT model by introducing the stricter notion of *pure* op-based, and establishing a clear frontier with state-based models. We have shown which pure datatypes are possible over off-the-shelf causal delivery middleware and then introduce an extended API, *tagged* reliable broadcast, that supports the remaining datatypes, those non-commutative in their sequential specifications. Supported by this API, that conveys causal information present in the middleware, we were able to define a partially ordered log, named PO-Log, that supports a clear semantic description and abstract implementation of each concurrent datatype.

To obtain efficient implementations we developed a framework for semantic compaction of a PO-Log and, in a final step, resorted to a notion of *causal stability* to determine when it is safe to strip PO-Log entries of their causal order metadata. This final step allows eventually moving all data to a standard sequential datatype, or a local database, and harvest the efficiency gains of reusing existing optimized data structures and database engines.

Having exemplified the framework with relevant non-trivial datatypes (replicated sets and registers) we expect that future research, and the existing developer community, can apply these techniques to other derived datatypes, such a maps, graphs, and sequences.

Acknowledgments. We thank Marek Zawirski, Ricardo Gonçalves and anonymous reviewers for comments that helped improve this work. Project Norte-01-0124-FEDER-000058 is co-financed by the North Portugal Regional Operational Programme (ON.2 O Novo Norte), under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF). Project FCOMP-01-0124-FEDER-037281 financed by National Funds by FCT, and by ERDF. Funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement 609551, SyncFree project.

References

1. Vogels, W.: Eventually consistent. ACM Queue **6**(6) (October 2008) 14–19
2. Cribbs, S., Brown, R.: Data structures in Riak. In: Riak Conference (RICON), San Francisco, CA, USA (oct 2012)
3. Bailis, P., Ghodsi, A.: Eventual consistency today: Limitations, extensions, and beyond. Queue **11**(3) (March 2013) 20:20–20:32
4. Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J., Hauser, C.H.: Managing update conflicts in Bayou, a weakly connected replicated storage system. In: Symp. on Op. Sys. Principles (SOSP), Copper Mountain, CO, USA, ACM SIGOPS, ACM Press (December 1995) 172–182
5. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: Symp. on Op. Sys. Principles (SOSP). Volume 41 of Operating Systems Review., Stevenson, Washington, USA, Assoc. for Computing Machinery (October 2007) 205–220

6. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: A comprehensive study of Convergent and Commutative Replicated Data Types. Rapp. Rech. 7506, Institut National de la Recherche en Informatique et Automatique (INRIA), Rocquencourt, France (January 2011)
7. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In Défago, X., Petit, F., Villain, V., eds.: Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS). Volume 6976 of Lecture Notes in Comp. Sc., Grenoble, France, Springer-Verlag (October 2011) 386–400
8. Letia, M., Preguiça, N., Shapiro, M.: CRDTs: Consistency without concurrency control. Rapp. Rech. RR-6956, Institut National de la Recherche en Informatique et Automatique (INRIA), Rocquencourt, France (June 2009)
9. Baquero, C., Moura, F.: Using structural characteristics for autonomous operation. Operating Systems Review **33**(4) (1999) 90–96
10. Birman, K., Schiper, A., Stephenson, P.: Lightweight causal and atomic group multicast. ACM Trans. Comput. Syst. **9**(3) (August 1991) 272–314
11. Bieniusa, A., Zawirski, M., Preguiça, N., Shapiro, M., Baquero, C., Balegas, V., Duarte, S.: Brief announcement: Semantics of eventually consistent replicated sets. In Aguilera, M.K., ed.: Int. Symp. on Dist. Comp. (DISC). Volume 7611 of Lecture Notes in Comp. Sc., Salvador, Bahia, Brazil, Springer-Verlag (October 2012) 441–442
12. Schmuck, F.B.: The use of efficient broadcast protocols in asynchronous distributed systems. Technical Report TR 88-928, Cornell University (1988)
13. Mahajan, P., Alvisi, L., Dahlin, M.: Consistency, availability, and convergence. Technical Report UTCS TR-11-22, Dept. of Comp. Sc., The U. of Texas at Austin, Austin, TX, USA (2011)
14. Wuu, G.T.J., Bernstein, A.J.: Efficient solutions to the replicated log and dictionary problems. In: Symp. on Principles of Dist. Comp. (PODC), Vancouver, BC, Canada (August 1984) 233–242
15. Burckhardt, S., Gotsman, A., Yang, H., Zawirski, M.: Replicated data types: specification, verification, optimality. In Jagannathan, S., Sewell, P., eds.: POPL, ACM (2014) 271–284
16. Johnson, P.R., Thomas, R.H.: The maintenance of duplicate databases. Internet Request for Comments RFC 677, Information Sciences Institute (January 1976)
17. Quarterman, J.S., Hoskins, J.C.: Notable computer networks. Commun. ACM **29**(10) (October 1986) 932–971
18. Ladin, R., Liskov, B., Shrira, L., Ghemawat, S.: Providing high availability using lazy replication. ACM Trans. Comput. Syst. **10**(4) (November 1992) 360–391
19. Golding, R.A.: Weak-consistency group communication and membership. PhD thesis, University of California Santa Cruz, Santa Cruz, CA, USA (December 1992) Tech. Report no. UCSC-CRL-92-52.
20. Roh, H.G., Jeon, M., Kim, J.S., Lee, J.: Replicated Abstract Data Types: Building blocks for collaborative applications. Journal of Parallel and Dist. Comp. **71**(3) (March 2011) 354–368
21. Rovio Entertainment Ltd.: Rovio gaming. http://www.rovio.com/en (2013)
22. Pereira, J., Rodrigues, L., Oliveira, R.: Semantically reliable multicast: Definition, implementation, and performance evaluation. IEEE Trans. Comput. **52**(2) (February 2003) 150–165