# Optimized OR-Sets Without Ordering Constraints

Madhavan Mukund [⋆], Gautham Shenoy R[⋆⋆], and S P Suresh

Chennai Mathematical Institute, India
{madhavan,gautshen,spsuresh}@cmi.ac.in

**Abstract.** Eventual consistency is a relaxation of strong consistency that guarantees that if no new updates are made to a replicated data object, then all replicas will converge. The *conflict free replicated datatypes (CRDTs)* of Shapiro et al. are data structures whose inherent mathematical structure guarantees eventual consistency. We investigate a fundamental CRDT called *Observed-Remove Set (OR-Set)* that robustly implements sets with distributed add and delete operations. Existing CRDT implementations of OR-Sets either require maintaining a permanent set of "tombstones" for deleted elements, or imposing strong constraints such as causal order on message delivery. We formalize a concurrent specification for OR-Sets without ordering constraints and propose a generalized implementation of OR-sets without tombstones that provably satisfies strong eventual consistency. We introduce *Interval Version Vectors* to succinctly keep track of distributed time-stamps in systems that allow out-of-order delivery of messages. The space complexity of our generalized implementation is competitive with respect to earlier solutions with causal ordering. We also formulate *k-causal delivery*, a generalization of causal delivery, that provides better complexity bounds.

## 1 Introduction

The Internet hosts many services that maintain replicated copies of data across distributed servers with support for local updates and queries. An early example is the Domain Name Service (DNS) that maintains a distributed mapping of Internet domain names to numeric IP addresses. More recently, the virtual shopping carts of online merchants such as Amazon also follow this paradigm.

The users of these services demand high availability. On the other hand, the underlying network is inherently prone to local failures, so the systems hosting these replicated data objects must be partition tolerant. By Brewer's CAP theorem, one has to then forego strong consistency, where local queries about distributed objects return answers consistent with the most recent update [1].

*Eventual consistency* is a popular relaxation of consistency for distributed systems that require high availability alongside partition tolerance [2–4]. In such

[⋆] Partially supported by Indo-French CNRS LIA Informel
[⋆⋆] Supported by TCS Research Scholarship

systems, the local states of nodes are allowed to diverge for finite, not necessarily bounded, durations. Assuming that all update messages are reliably delivered, eventual consistency guarantees that the states of all the replicas will converge if there is a sufficiently long period of quiescence [2]. However, convergence involves detecting and resolving conflicts, which can be problematic.

*Conflict free replicated datatypes (CRDTs)* are a class of data structures that satisfy strong eventual consistency by construction [5]. This class includes widely used datatypes such as replicated counters, sets, and certain kinds of graphs.

Sets are basic mathematical structures that underlie many other datatypes such as containers, maps and graphs. To ensure conflict-freeness, a robust distributed implementation of sets must resolve the inherent non-serializability of add and delete operations of the same element in a set. One such variant is known as an *Observed-Remove Set* (*OR-Set*), in which adds have priority over deletes of the same element, when applied concurrently.

The naïve implementation of OR-sets maintains all elements that have ever been deleted as a set of *tombstones* [5]. Consider a sequence of add and delete operations in a system with $N$ replicas, in which $t$ elements are added to the set, but only $p$ are still present at the end of the sequence because of intervening deletes. The space complexity of the naïve implementation is $O(t \log t)$, which is clearly not ideal. If we enforce causal ordering on the delivery of updates, then the space complexity can be reduced to $O((p + N) \log t)$ [6].

On the other hand, causal ordering imposes unnecessary constraints: even independent actions involving separate elements are forced to occur in the same order on all replicas. Unfortunately, there appears to be no obvious relaxation of causal ordering that retains enough structure to permit the simplified algorithm of [6]. Instead, we propose a generalized implementation that does not make any assumptions about message ordering but reduces to the algorithm of [6] in the presence of causal ordering. We also describe a weakening of causal ordering that allows efficient special cases of our implementation.

The main contributions of this paper are as follows:

- We identify some gaps in the existing concurrent specification of OR-Sets [6], which assumes causal delivery of updates. We propose a new concurrent specification for the general case without assumptions on message ordering.
- We present a generalized implementation of OR-sets whose worst-case space complexity is $O((p+Nm) \log t)$, where $m$ is the maximum number of updates at any one replica. We introduce *Interval Version Vectors* to succinctly keep track of distributed-time stamps in the presence of out-of-order messages.
- We formally prove the correctness of our generalized solution, from which the correctness of all earlier implementations follows.
- We introduce *$k$-causal delivery*, a delivery constraint that generalizes causal delivery. When updates are delivered in $k$-causal order, the worst-case space complexity of our generalized implementation is $O((p + Nk) \log t)$. Since 1-causal delivery is the same as causal delivery, the solution presented in [6] is a special case of our generalized solution.

The paper is organized as follows. In Section 2, we give a brief overview of *strong eventual consistency* and *conflict free replicated datatypes (CRDTs)*. In the next section, we describe the naïve implementation of OR-Sets and the existing concurrent specification that assumes causal delivery. In Section 4, we propose a generalized specification of OR-sets along with an optimized implementation, neither of which require any assumption about delivery constraints. In the next section, we introduce $k$-causal delivery and analyze the space-complexity of the generalized algorithm. In Section 6, we provide a proof of correctness for the generalized solution. We conclude with a discussion about future work.

## 2 Strong Eventual Consistency and CRDTs

We restrict ourselves to distributed systems with a fixed number of nodes (or replicas). We allow both nodes and network connections to fail and recover infinitely often, but we do not consider Byzantine faults. We assume that when a node recovers, it starts from the state in which it crashed.

In general, concurrent updates to replicas may conflict with each other. The replicas need to detect and resolve these conflicts to maintain eventual consistency. For instance, consider two replicas $r_1$ and $r_2$ of an integer with value 1. Suppose $r_1$ receives an update *multiply*(3) concurrently with an update *add*(2) at $r_2$. If each replica processes its local update before the update passed on by the other replica, the copies at $r_1$ and $r_2$ would have values 5 and 9, respectively.

Conflict resolution requires the replicas to agree on the order in which to apply the set of updates received from the clients. In general, it is impossible to solve the consensus problem in the presence of failures [7]. However, there are several eventually consistent data structures whose design ensures that they are conflict free. To characterize their behaviour, a slightly stronger notion of eventual consistency called *strong eventual consistency (SEC)* has been proposed [8].

Strong eventual consistency is characterized by the following principles

- **Eventual delivery:** An update delivered at some correct replica will eventually be delivered to all correct replicas.
- **Termination:** All delivered methods are eventually enabled (their preconditions are satisfied) and method executions terminate.
- **Strong Convergence:** Correct replicas that have been delivered the same updates have equivalent state.

Strong convergence ensures that systems are spared the task of performing conflict-detection and resolution. Datatypes that satisfy strong eventual consistency are called *conflict-free replicated datatypes (CRDTs)*.

#### Conflict-free Replicated DataTypes (CRDTs)

In a replicated datatype, a client can send an update operation to any replica. The replica that receives the update request from the client is called the *source*

*replica* for that update. The source replica typically applies the update locally and then propagates information about the update to all the other replicas. On receiving this update, each of these replicas applies it in its current state.

Replicated datatypes come in two flavours, based on how replicas exchange information about updates. In a *state-based replicated data object*, the source replica propagates its entire updated state to the other replicas. State-based replicated objects need to specify a *merge* operation to combine the current local state of a replica with an updated state received from another replica to compute an updated local state. Formally, a state-based replicated datatype is a tuple $O = (\mathcal{S}, S_\perp, Q, U, m)$ where $\mathcal{S}$ is the set of all possible states of the replicated object, $S_\perp$ is the initial state, $Q$ is the set of all side-effect free query operations, $U$ is the set of all update-operations that source replicas apply locally, and $m : \mathcal{S} \times \mathcal{S} \to \mathcal{S}$ is the merge function.

Propagating states may not be practical—for instance, the payload may be prohibitively large. In such cases replicas can, instead, propagate update operations. These are called *operation based (op-based) replicated datatypes.* When a source replica receives an update request, it first computes the arguments required to perform the actual update and sends a confirmation to the client. This is called the *update-prepare* phase and is free of side-effects. It then sends the arguments prepared in the update-prepare phase to all other replicas, including itself, to perform the actual update. This phase modifies the local state of each replica and is called the *update-downstream phase.* At the source replica, the prepare and downstream phases are applied *atomically.* Formally, an op-based replicated datatype is a tuple $O = (\mathcal{S}, S_\perp, Q, V, P)$ where $\mathcal{S}$, $S_\perp$, and $Q$ are as in state-based replicated datatypes, $V$ is the set of updates of the form $(p, u)$ where $p$ is the side-effect free update-prepare method and $u$ is the update-downstream method, and $P$ is a set of delivery preconditions that control when an update-downstream message can be delivered at a particular replica.

We denote the $k^{th}$ operation at a replica $r$ of a state-based or object-based datatype by $f_r^k$ and the state of replica $r$ after applying the $k^{th}$ operation by $S_r^k$. Note that for all replicas $r$, $S_r^0 = S_\perp$. The notation $S \circ f$ is used to denote the result of applying operation $f$ on state $S$. The notation $S \xrightarrow{f} S'$ is used to denote that $S' = S \circ f$. The argument of the operation $f$ is denoted $\arg(f)$.

A reachable state of a replica is obtained by a sequence of operations $S_r^0 \xrightarrow{f_r^1} S_r^1 \xrightarrow{f_r^2} \cdots \xrightarrow{f_r^k} S_r^k$. The *causal history* of a reachable state $S_r^k$, denoted by $\mathcal{H}(S_r^k)$, is the set of updates (for state-based objects) or update-downstream operations (for op-based objects) received so far. This is defined inductively as follows:

- $\mathcal{H}(S_r^0) = \emptyset$.
- $\mathcal{H}(S_r^k) = \mathcal{H}(S_r^{k-1})$ if $f_r^k$ is a query operation or an update-prepare method.
- $\mathcal{H}(S_r^k) = \mathcal{H}(S_r^{k-1}) \cup \{f_r^k\}$ if $f_r^k$ is an update or update-downstream operation.
- $\mathcal{H}(S_r^k) = \mathcal{H}(S_r^{k-1}) \cup \mathcal{H}(S_{r'}^\ell)$ if $f_r^k$ is a merge operation and $\arg(f_r^k) = (S_r^{k-1}, S_{r'}^\ell)$.

An update $(p, u)$ at source replica $r$ is said to have *happened before* an update $(p', u')$ at source replica $r'$ if $\exists k : p' = f_{r'}^k \wedge u \in \mathcal{H}(S_{r'}^{k-1})$. We denote this by

$(p, u) \xrightarrow{hb} (p', u')$ or simply $u \xrightarrow{hb} u'$. Any pair of updates that are not comparable through this relation are said to be *concurrent updates*. A pair of states $S$ and $S'$ are said to be *query-equivalent*, or simply *equivalent*, if for all query operations $q \in Q$, the result of applying $q$ at $S$ and $S'$ is the same. A collection of updates is said to be *commutative* if at any state $S \in \mathcal{S}$, applying any permutation of these updates leads to equivalent states. We say that the delivery subsystem satisfies *causal delivery* if for any two update operations $(p, u)$ and $(p', u')$,

$$(p, u) \xrightarrow{hb} (p', u') \implies \forall r, k : (u' \in \mathcal{H}(S_r^k) \implies u \in \mathcal{H}(S_r^{k-1})).$$

That is, whenever $(p, u)$ has happened before $(p', u')$, at all other replicas, the downstream method $u$ is delivered before $u'$.

A state-based replicated object that satisfies strong eventual consistency is called a *Convergent Replicated DataType (CvRDT)*. A sufficient condition for this is that there exists a partial order $\leq$ on its set of states $\mathcal{S}$ such that: i) $(\mathcal{S}, \leq)$ forms a join-semilattice, ii) whenever $S \xrightarrow{u} S'$ for an update $u$, $S \leq S'$, and iii) all merges compute least upper bounds [8].

Assuming termination and causal delivery of updates, a sufficient condition for an op-based replicated datatype to satisfy strong eventual consistency is that concurrent updates should commute and all delivery preconditions are compatible with causal delivery. Such a replicated datatype is called a *Commutative Replicated DataType (CmRDT)* [8].

In this paper we look at a conflict-free *Set* datatype that supports features of both state-based and op-based datatypes.

## 3 Observed-Remove Sets

Consider a replicated set of elements over a universe $\mathcal{U}$ across $N$ replicas $Reps = [0..N-1]$. Clients interact with the set through a query method *contains* and two update methods *add* and *delete*. The set also provides an internal method *compare* that induces a partial order on the states of the replicas and a method *merge* to combine the local state of a replica with the state of another replica. Let $i$ be the source replica for one of these methods *op*. Let $S_i$ and $S_i'$ be the states at $i$ before and after applying the operation *op*, respectively. The sequential specification of the set is the natural one:

- $S_i \circ contains(e)$ returns true iff $e \in S_i$.
- $S_i \xrightarrow{contains(e)} S_i'$ iff $S_i' = S_i$.
- $S_i \xrightarrow{add(e)} S_i' \implies S_i' = S_i \cup \{e\}$.
- $S_i \xrightarrow{delete(e)} S_i' \implies S_i' = S_i \setminus \{e\}$.

Thus, in the sequential specification, two states $S, S'$ are query-equivalent if for every element $e \in \mathcal{U}$, $S \circ contains(e)$ returns true iff $S' \circ contains(e)$. Notice that the state of a replica gets updated not only when it acts as a source replica for some update operation, but also when it applies updates, possibly concurrent

ones, propagated by other replicas, either through downstream operations or through a *merge* request.

Defining a concurrent specification for sets is a challenge because *add(e)* and *delete(e)*, for the same element $e$, do not commute. If these updates are concurrent, the order in which they are applied determines the final state.

An *Observed-Remove Set* (OR-Set) is a replicated set where the conflict between concurrent *add(e)* and *delete(e)* operations is resolved by giving precedence to the *add(e)* operation so that $e$ is eventually present in all the replicas [5]. An OR-Set implements the operations *add*, *delete*, *adddown*, *deldown*, *merge*, and *compare*, where *adddown* and *deldown* are the downstream operations corresponding to the *add* and *delete* operations, respectively.

A concurrent specification for OR-sets is provided in [6]. Let $S$ be the abstract state of an OR-set, $e \in \mathcal{U}$ and $u_1 \parallel u_2 \parallel \cdots \parallel u_n$ be a set of concurrent update operations. Then, the following conditions express the fact that if even a single update adds $e$, $e$ must be present after the concurrent updates.

- $(\exists i : u_i = \mathit{delete}(e) \wedge \forall i : u_i \neq \mathit{add}(e)) \implies e \notin S$ after $u_1 \parallel u_2 \parallel \cdots \parallel u_n$
- $(\exists i : u_i = \mathit{add}(e)) \implies e \in S$ after $u_1 \parallel u_2 \parallel \cdots \parallel u_n$

As we shall see, this concurrent specification is incomplete unless we assume causal delivery of updates.

**Naïve implementation** Algorithm 1 is a variant of the naïve implementation of this specification given in [5]. Let $\mathcal{M} = \mathcal{U} \times \mathbb{N} \times [0 \ldots N{-}1]$. For a triple $m = (e, c, r)$ in $\mathcal{M}$, we say $\mathit{data}(m) = e$ (the data or payload), $\mathit{ts}(m) = c$ (the timestamp), and $\mathit{rep}(m) = r$ (the source replica). Each replica maintains a local set $E \subseteq \mathcal{M}$. When replica $r$ receives an $\mathit{add}(e)$ operation, it tags $e$ with a unique identifier $(c, r)$ (line 8), where this $\mathit{add}(e)$ operation is the $c^{th}$ *add* operation overall at $r$, and propagates $(e, c, r)$ downstream to be added to $E$. Symmetrically, deleting an element $e$ involves removing every triple $m$ from $E$ with $\mathit{data}(m) = e$. In this case, the source replica propagates the set $M \subseteq E$ of elements matching $e$ downstream to be deleted at all replicas (lines 16–17).

A Naive OR-set implementation for replica $r$

```
1   E ⊆ M, T ⊆ M, c ∈ ℕ: initially ∅, ∅, 0.
2
3   Boolean CONTAINS(e ∈ U):
4         return (∃m : m ∈ E ∧ data(m) = e)
5
6   ADD(e ∈ U):
7         ADD.PREPARE(e ∈ U):
8               Broadcast downstream((e, c, r))
9         ADD.DOWNSTREAM(m ∈ M):
10              E := (E ∪ {m}) \ T
11              if (rep(m) = r)
12                    c = ts(m) + 1
13
14  DELETE(e ∈ U):
15        DELETE.PREPARE(e ∈ U):
16              Let M := {m ∈ E | data(m) = e}
17              Broadcast downstream(M)
18        DELETE.DOWNSTREAM(M ⊆ M):
19              E := E \ M
20              T := T ∪ M
21
22  Boolean COMPARE(S′, S′′ ∈ S):
23        Assume that S′ = (E′, T′, c′)
24        Assume that S′′ = (E′′, T′′, c′′)
25        Let b_seen := (E′ ∪ T′) ⊆ (E′′ ∪ T′′)
26        Let b_deletes := T′ ⊆ T′′
27        return b_seen ∧ b_deletes
28
29  MERGE(S′ ∈ S):
30        Assume that S′ = (E′, T′, c′)
31        E := (E \ T′) ∪ (E′ \ T)
32        T := T ∪ T′
```

Algorithm 1: Naïve implementation

For an add operation, each replica downstream should add the triple $m$ to its local copy of $E$. However, with

no constraints on the delivery of messages, a *delete* operation involving $m$ may overtake an *add* update for $m$. For example in Figure 1, replica $r''$ receives $deldown(\{(e, c+1, r)\})$ before it receives $adddown(e, c+1, r)$. Alternatively, after applying a *delete*, a replica may merge its state with another replica that has not performed this *delete*, but has performed the corresponding *add*. For instance, in Figure 1, replica $r''$ merges its state with replica $r$ when $r''$ has applied $deldown(\{(e, c+1, r)\})$ but $r$ has not. To ensure that $m$ is not accidentally added back in $E$ in such cases, each replica maintains a set $T$ of *tombstones*, containing every triple $m$ ever deleted (lines 19–20). Before adding $m$ to $E$, a replica first checks that it is not in $T$ (line 10).

State $S$ of replica $r$ is more up-to-date than state $S'$ of replica $r'$ if $r$ has seen all the triples present in $S'$ (either through an add or a delete) and $r$ has deleted all the triples that $r'$ has deleted. This is checked by *compare* (lines 22–27). Finally, the *merge* function of states $S$ and $S'$ retains only those triples from $S.E \cup S'.E$ that have not been deleted in either $S$ or $S'$ (line 31). The *merge* function also combines the triples that have been deleted in $S$ and $S'$ (line 32).

**Eliminating Tombstones [6]** Since $T$ is never purged, $E \cup T$ contains every element that was ever added to the set. To avoid keeping an unbounded set of tombstones, a solution is proposed in [6] that requires all updates to be delivered in causal order. The solution uses a version vector [9] at each replica to keep track of the latest add operation that it has received from every other replica.

Causal delivery imposes unnecessary restrictions on the delivery of independent updates. For example, updates at a source replica of the form $add(e)$ and $delete(f)$, for distinct elements $e$ and $f$, need not be delivered downstream in the same order to all other replicas. For the concurrent specification presented earlier to be valid, it is sufficient to have causal delivery of updates involving the same element $e$. While this is weaker than causal delivery across all updates, it puts an additional burden on the underlying delivery subsystem to keep track of the partial order of updates separately for each element in the universe. A weaker delivery constraint is FIFO, which delivers updates originating at the same source replica in the order seen by the source. However, this is no better than out-of-order delivery since causally related operations on the same element that originate at different sources can still be delivered out-of-order.

On the other hand, the naïve implementation works even when updates are delivered out-of-order. However, reasoning about the state of the replicas is non-trivial in the absence of any delivery guarantees. We illustrate the challenges posed by out-of-order delivery before formalizing a concurrent specification for OR-Sets that is independent of delivery guarantees.

### Life without causal delivery: challenges

If we assume causal delivery of updates, then it is easy to see that all replicas apply non-concurrent operations in the same order. Hence it is sufficient

for the specification to only talk about concurrent operations. However, without causal delivery, even non-concurrent operations can exhibit counter-intuitive behaviours. We identify a couple of them in Examples 1 and 2.

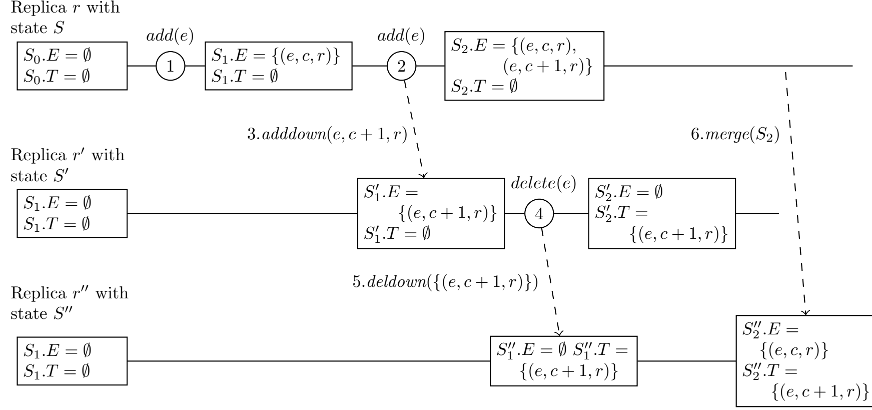


**Fig. 1.** Non-transitivity of the happened-before relation.

**Example 1** *In the absence of causal delivery, the happened-before relation need not be transitive. For instance, in Figure 1, if we denote the add operations at* 1 *and* 2 *as* $\text{add}_1(e)$ *and* $\text{add}_2(e)$*, respectively, then we can observe that* $\text{add}_1(e) \xrightarrow{hb} \text{add}_2(e)$ *and* $\text{add}_2(e) \xrightarrow{hb} \text{delete}(e)$*. However, it is not the case that* $\text{add}_1(e) \xrightarrow{hb} \text{delete}(e)$ *since the source replica of* $\text{delete}(e)$*, which is* $r'$*, has not processed the downstream of* $\text{add}_1(e)$ *before processing the prepare method of* $\text{delete}(e)$*.*

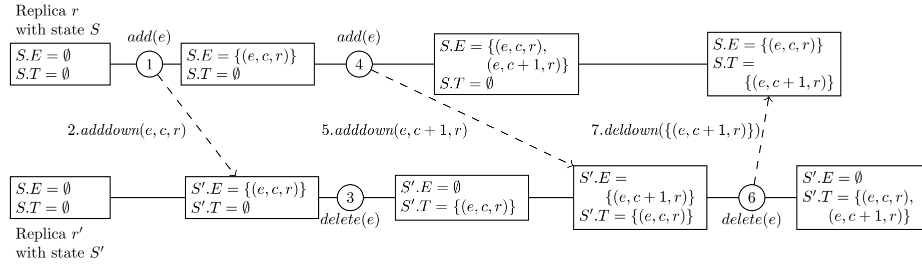


**Fig. 2.** Non-intuitive behaviour of *deletes* in the absence of causal delivery.

**Example 2** *In the absence of causal delivery, a* $\text{delete-downstream}(e)$ *may not remove all copies of* $e$ *from the set—even copies corresponding to* $\text{add}(e)$ *operations that happened before. Say* $(e,c,r)$ *is added at* $r$*, propagated to* $r'$*, and subsequently deleted at* $r'$*. Suppose* $(e,c+1,r)$ *is later added at* $r$*, propagated to* $r'$*, and subsequently deleted at* $r'$*. If the second delete is propagated from* $r'$ *to* $r$ *before the first one,* $r$ *removes only* $(e,c+1,r)$ *while retaining* $(e,c,r)$*, as illustrated in Figure 2.*

To address these issues, we present a more precise formulation of the concurrent specification that captures the intent of [5] and allows us to uniformly reason about the states of the replicas of OR-Sets independent of the order of delivery of updates.

## 4 Optimized OR-Sets

**Revised specification** For an $add(e)$ operation $op$, the set of *nearest delete* operations, $NearestDel(op)$, is defined to be the following set:

$$\{op' \mid op' = delete(e) \wedge op \xrightarrow{hb} op' \wedge \neg(\exists op''. op'' = delete(e) \wedge op \xrightarrow{hb} op'' \xrightarrow{hb} op')\}$$

If $u$ is the downstream operation of $op$ and $u'$ is the downstream operation of $op' \in NearestDel(op)$ then we extend this notation to write $u' \in NearestDel(u)$. Our new concurrent specification for OR-Sets is as follows.

> For any reachable state $S$ and element $e$, $e \in S$ iff $\mathcal{H}(S)$ contains a downstream operation $u$ of an $add(e)$ operation such that $NearestDel(u) \cap \mathcal{H}(S) = \emptyset$.

The specification ensures that a delete operation at a replica removes only those elements whose add operations the replica has *observed*. Thus, whenever a replica encounters concurrent add and delete operations with the same argument, the add wins, since the delete has not seen the element added by that particular add. The specification also ensures that any two states that have the same causal history are query-equivalent. Hence the order in which the update operations in the causal history were applied to arrive at these two states is not relevant. Since there are no delivery preconditions in the specification, any implementation of this specification is a *CmRDT*, as all the operations commute.

The revised specification generalizes the concurrent OR-set specification from [6]. Suppose $u_1 \parallel u_2 \parallel \cdots \parallel u_n$ is performed at a replica with state $S$. Let $S' = S \circ (u_1 \parallel u_2 \parallel \cdots \parallel u_n)$. If one of the $u_i$'s is $add(e)$, it is clear that $NearestDel(u_i) \cap \mathcal{H}(S') = \emptyset$. Thus, $e \in S'$. On the other hand, if at least one of the $u_i$'s is $del(e)$ and none of the $u_i$'s is an $add(e)$, then, *assuming causal delivery*, for every $u_i$ of the form $delete(e)$, if $e \in S$, there is an $add(e)$ operation $u \in \mathcal{H}(S)$ such that $u_i \in NearestDel(u)$. Thus $e \notin S'$, as expected.

The new specification also explains Examples 1 and 2. In Example 1, the $add(e)$ operation at 1 does not have a nearest delete in $\mathcal{H}(S_2'')$, which explains why $e \in S_2''$. Similarly in the other example, the $add(e)$ operation at 1 does not have a nearest delete in the history of replica $r$, but it has a nearest delete (operation 3) in the history of replica $r'$. This explains why $e$ is in the final state of $r$ but does not belong to the final state of $r'$.

**Generalized implementation** Algorithm 2 describes our optimized implementation of OR-sets that does not require causal ordering and yet uses space

comparable to the solution provided in [6]. Our main observation is that tombstones are only required to track *delete*($e$) operations that overtake *add*($e$) operations from the same replica. Since a source replica attaches a timestamp $(c, r)$ with each *add*($e$) operation, all we need is a succinct way to keep track of those timestamps that are "already known".

For a pair of integers $s \leq \ell$, $[s, \ell]$ denotes the *interval* consisting of all integers from $s$ to $\ell$. A finite set of intervals $\{[s_1, \ell_1], \ldots, [s_n, \ell_n]\}$ is *nonoverlapping* if for all distinct $i, j \leq n$, either $s_i > \ell_j$ or $s_j > \ell_i$. An *interval sequence* is a finite set of nonoverlapping intervals. We denote by $\mathcal{I}$ the set of all interval sequences.

The basic operations on interval sequences are given below. The function $\textsc{pack}(N)$ collapses a set of numbers $N$ into an interval sequence. The function $\textsc{unpack}(A)$ expands an interval sequence to the corresponding set of integers. Fundamental set operations can be performed on interval sequences by first unpacking and then packing. For $N \subseteq \mathbb{N}$, $A, B \in \mathcal{I}$, and $n \in \mathbb{N}$:

- $\textsc{pack}(N) = \{[i, j] \mid \{i, i+1, \ldots, j\} \subseteq N, i-1 \notin N, j+1 \notin N\}$.
- $\textsc{unpack}(A) = \{n \mid \exists [n_1, n_2] \in A \land n_1 \leq n \leq n_2\}$.
- $n \in A$ iff $n \in \textsc{unpack}(A)$.
- $\mathbf{add}(A, N) = \textsc{pack}(\textsc{unpack}(A) \cup N)$.
- $\mathbf{delete}(A, N) = \textsc{pack}(\textsc{unpack}(A) \setminus N)$.
- $\mathbf{max}(A) = \max(\textsc{unpack}(A))$.
- $A \cup B = \textsc{pack}(\textsc{unpack}(A) \cup \textsc{unpack}(B))$.
- $A \cap B = \textsc{pack}(\textsc{unpack}(A) \cap \textsc{unpack}(B))$.
- $A \subseteq B$ iff $\textsc{unpack}(A) \subseteq \textsc{unpack}(B)$.

As in the algorithm of [6], when replica $r$ receives an *add*($e$) operation, it tags $e$ with a unique identifier $(c, r)$ and propagates $(e, c, r)$ downstream to be added to $E$. In addition, each replica $r$ maintains the set of all timestamps $c$ received from every other replica $r'$ as an interval sequence $V[r']$. The vector $V$ of interval sequences is called an *Interval Version Vector*. Since all the downstream operations with source replica $r$ are applied at $r$ in causal order, $V[r]$ contains a single interval $[1, c_r]$ where $c_r$ is the index of the latest add operation received by $r$ from a client. Notice that if *delete*($e$) at a source replica $r'$ is a *nearest delete* for an *add*($e$) operation, then the unique identifier $(ts(m), rep(m))$ of the triple $m$ generated by the *add* operation will be included in the interval version vector propagated downstream by the *delete* operation. When this vector arrives at a replica $r$ downstream, $r$ updates the interval sequence $V[rep(m)]$ to record the missing *add* operation (lines 25–26) so that, when $m$ eventually arrives to be added through the add-downstream operation, it can be ignored (lines 10–12).

Thus, we avoid maintaining tombstones altogether. The price we pay is maintaining a collection of interval sequences, but these interval sequences will eventually get merged once the replica receives all the pending updates, collapsing the representation to contain at most one interval per replica.

In [6], the authors suggest a solution in the absence of causal delivery using *version vectors with exceptions (VVwE)*, proposed in [10]. A VVwE is an array each of whose entries is a pair consisting of a timestamp and an *exception set*, and is used to handle out-of-order message delivery. For instance,

Optimized OR-set implementation for the replica $r$

```
1   E ⊆ M, V : Reps → I, c ∈ ℕ: initially ∅, [∅, ..., ∅], 0
2
3   Boolean CONTAINS(e ∈ U):
4       return (∃m : m ∈ E ∧ data(m) = e)
5
6   ADD(e ∈ U):
7       ADD.PREPARE(e ∈ U):
8           Broadcast downstream((e, c, r))
9       ADD.DOWNSTREAM(m ∈ M):
10          if (ts(m) ∉ V[rep(m)])
11              E := E ∪ {m}
12              V[rep(m)] :=
                    add(V[rep(m)], {ts(m)})
13          if (rep(m) = r)
14              c = ts(m) + 1
15
16  DELETE(e ∈ U):
17      DELETE.PREPARE(e ∈ U):
18          Let V' : Reps → I = [0, ..., 0]
19          for m ∈ E with data(m) = e
20              add(V'[rep(m)], {ts(m)})
21          Broadcast downstream(V')
22      DELETE.DOWNSTREAM(V' : Reps → I):
23          Let M = {m ∈ E |
                        ts(m) ∈ V'[rep(m)]}
24          E := E \ M
25          for i ∈ Reps
26              V[i] := V[i] ∪ V'[i]
27

28  Boolean COMPARE(S', S'' ∈ S):
29      Assume that S' = (E', V')
30      Assume that S'' = (E'', V'')
31      b_seen := ∀i(V'[i] ⊆ V''[i])
32      b_deletes := ∀m ∈ E'' \ E'
                (ts(m) ∉ V'[rep(m)])
33          // If m is deleted from E' then
34          // it is also deleted in E''.
35          // So anything in E'' \ E'
36          // is not even visible in S'.
37      return b_seen ∧ b_deletes
38
39  MERGE(S' ∈ S):
40      Assume that S' = (E', V')
41      E := {m ∈ E ∪ E' |
              m ∈ E ∩ E' ∨
              ts(m) ∉ V[rep(m)] ∩ V'[rep(m)]}
42  // You retain m if it is either
43  // in the intersection, or if it is fresh
44  // (so one of the states has not seen it).
45  ∀i.(V[i] := V[i] ∪ V'[i])
```

Algorithm 2: An optimized OR-Set implementation

if replica $r$ sees operations of $r'$ with timestamps $1, 2$, and $10$, then it will store $(10, \{3, 4, 5, 6, 7, 8, 9\})$, signifying that 10 is the latest timestamp of an $r'$-operation seen by $r$, and that $\{3, 4, \ldots, 9\}$ is the set of operations that are yet to be seen. The same set of timestamps would be represented by the interval sequence $\{[1, 2], [10, 10]\}$. In general, it is easy to see that interval sequences are a more succinct way of representing timestamps in systems that allow out-of-order delivery.

## 5 *k*-causal delivery, add-coalescing and space complexity

Let $S_r$ denote the state of a replica $r \in [0 \ldots N{-}1]$. Let $n_\ell$ be the number of *adddown* operations whose source is $\ell$. The space required to store $S_r$ in the naïve implementation is bounded by $O(n_t \log(n_t))$, where $n_t = \sum_{\ell=0}^{N-1} n_\ell$.

Let $n_p$ denote the number all *adddown* operations $u \in \mathcal{H}(S_r)$ such that $\mathit{NearestDel}(u) \cap \mathcal{H}(S_r) = \emptyset$. Clearly $n_p \leq n_t$. Let $n_m = \max(n_0, \ldots, n_{N-1})$ and let $n_{int}$ denote the maximum number of intervals across any index $r'$ in $V_r[r']$. In our optimized implementation, the space required to store $S_r.V$ is bounded by $N n_{int} \log(n_t)$ and the space required to store $S_r.E$ is bounded by $n_p \log(n_t)$. The space required to store $S_r$ is thus bounded by $O((n_p + N n_{int}) \log(n_t))$. In the worst case, $n_{int}$ is bounded by $n_m/2$, which happens when $r$ sees only the

alternate elements generated by any replica. Thus the worst case complexity is $O((n_p + Nn_m)\log(n_t))$. Note that the factor that is responsible for increasing the space complexity is the number of intervals $n_{int}$. We propose a reasonable way of bounding this value below.

Let $(p, u)$ be an update operation whose source is replica $r$. For a given $k \geq 1$, we say that the delivery of updates satisfies *k-causal delivery* iff

$$\forall r, r' : (p = f_r^j \wedge u = f_{r'}^{j'}) \implies \forall u' \in \mathcal{H}(S_r^{j-k}), u' \in \mathcal{H}(S_{r'}^{j'-1}).$$

Intuitively it means that when a replica $r'$ sees the $j^{th}$ add operation originating at replica $r$, it should have already seen all the operations from $r$ with index smaller than $j - k$. Note that when $k = 1$, *k-causal delivery* is the same as *causal delivery*. Thus *k-causal delivery* ensures that the out of order delivery of updates is restricted to a bounded suffix of the sequence of operations submitted to the replicated datatype.

In particular, if the latest add-downstream operation $u$ received by a replica $r$ from a replica $r'$ corresponds to the $c^{th}$ *add* operation at $r'$, then $k$-causal delivery ensures that $r$ would have received all the add-downstream operations from $r'$ whose index is less than or equal to $(c-k)$. Thus, the number of intervals in $S_r.V[r']$ is bounded by $k$ and hence $n_{int}$ is bounded by $O(k)$.

With $k$-causal delivery, we can also coalesce the adds of the same elements that originate from the same replica. This is discussed in detail in the full version of the paper [11]. The key idea is that whenever a replica $r$ sees a triple $(e, c, r')$ , it can evict all the triples of the form $(e, c', r')$ with $c' \leq c - k$. Thus each replica keeps at most $k$ triples corresponding to each visible element originating from a replica. If there are $n_a$ visible elements, then $n_p$ is bounded by $O(n_a N k)$. With add-coalescing and $k$-causal delivery, the total space complexity at a replica is is $O((n_a + 1)Nk\log(n_t))$. Furthermore, the message complexity of a *deldown* operation with this optimization would be bounded by $O(Nk\log(n_t))$. If we assume causal delivery of updates ($k = 1$), the space complexity is bounded by $O((n_a + 1)N\log(n_t))$ and the message complexity of *deldown* is $O(N\log(n_t))$ which matches the complexity measures in [6]. If the delivery guarantee is FIFO delivery, then with add-coalescing, $n_p$ can be bounded by $O(n_a NK)$. However, there is no natural way of bounding the number of intervals $n_{int}$ with FIFO order, and hence the space complexity with FIFO ordering would be $O((n_a k + n_{int})N\log(n_t))$ as discussed in [11].

# 6 Correctness of the Optimized Implementation

In this section we list down the main lemmas and theorems, with proof sketches, to show the correctness of our optimized solution. The complete proofs can be found in [11]. Our aim is show that the solution satisfies the specification of OR-Sets and is a CvRDT as well as a CmRDT. For a detailed proof of the equivalence between the naïve implementation and our optimized implementation, the reader is encouraged to refer to [11].

Recall that $\mathcal{U}$ is the universe from which elements are added to the OR-Set and $Reps = [0 \ldots N{-}1]$ is the set of replicas. We let $\mathcal{M} = \mathcal{U} \times \mathbb{N} \times Reps$ denote the set of *labelled elements*. We use $r$ to denote replicas, $e$ to denote elements of $\mathcal{U}$, and $m$ to denote elements of $\mathcal{M}$, with superscripts and subscripts as needed. For $m = (e, c, r) \in \mathcal{M}$, we set $data(m) = e$ (the data or payload), $ts(m) = c$ (the timestamp), and $rep(m) = r$ (the source replica).

A set of labelled elements $M \subseteq \mathcal{M}$ is said to be *valid* if it does not contain distinct items from the same replica with the same timestamp. Formally,

$$\forall m, m' \in M : (ts(m) = ts(m') \wedge rep(m) = rep(m')) \implies m = m'$$

A downstream operation $u$ is said to be an $e$-add-downstream operation (respectively, $e$-delete-downstream operation) if it is a downstream operation of an $add(e)$ (respectively, $delete(e)$) operation. If $\mathcal{O}$ is a collection of commutative update operations then for any state $S$, $S \circ \mathcal{O}$ denotes the state obtained by applying these operations to $S$ in any order.

We say that two states $S$ and $S'$ are *equivalent* and write $S \equiv S'$ iff $S.E = S'.E$ and $S.V = S'.V$. It is easy to see that if $S$ and $S'$ are equivalent then they are also query-equivalent.

**Lemma 1.** *Let $S$ be some reachable state of the* OR-Set, $\mathcal{O} = \{u_1, u_2, \ldots u_n\}$ *be a set of downstream operations and $\pi_1$ and $\pi_2$ are any two permutations of $[1 \cdots n]$. If $S_1 = S \circ u_{\pi_1(1)} \circ u_{\pi_1(2)} \ldots u_{\pi_1(n)}$ and $S_2 = S \circ u_{\pi_2(1)} \circ u_{\pi_2(3)} \ldots u_{\pi_2(n)}$ then $S_1 = S_2$.*

*Proof Sketch:* If $u$ is any downstream update and $S$ is any state it can be easily shown that $S \circ u = S \circ merge(S_\perp \circ u)$ Now for any three states $S_1$, $S_2$, $S_3$, it can be seen that $(S_1 \circ merge(S_2)) \circ merge(S_3) = (S_1 \circ merge(S_3)) \circ merge(S_2)$, thereby proving commutativity of merges. Using these two results, it follows that any collection of update operations commute. $\square$

Using the lemma above, one can conclude the following, by induction over $|\mathcal{H}(S)|$.

**Lemma 2.** *Let $S$ be any reachable state, with $\mathcal{H}(S) = \{u_1, u_2, \ldots u_n\}$. Then $S = S_\perp \circ \mathcal{H}(S)$.*

**Lemma 3.** *Let $S$ be any state and $u$ be an $e$-add downstream operation such that $u \notin \mathcal{H}(S)$ and* $\arg(u) = m$*, and let $S' = S \circ u$. Then $m \in S'.E$ iff $NearestDel(u) \cap \mathcal{H}(S) = \emptyset$.*

*Proof Sketch:* If $NearestDel(u) \cap \mathcal{H}(S) = \emptyset$ then $ts(m) \notin S.V[rep(m)]$ and, from the code of *adddown*, it follows that $m \in S'.E$. Conversely suppose $u' \in NearestDel(u) \cap \mathcal{H}(S)$. Since downstream operations commute (and prepare and query operations do not change state), we can assume that there is a state $S''$ such that $S = S'' \circ u'$. One can then show (by examining the code for the *add* and *delete* methods) that $S'' \circ u' \circ u = S'' \circ u'$. Thus $S = S'$. Since $u \notin \mathcal{H}(S)$, we have that $m \notin S.E$. Hence $m \notin S'.E$ $\square$

**Theorem 1.** *The optimized OR-set implementation satisfies the specification of* OR-Sets.

*Proof Sketch:* Given a state $S$ and an element $e$, let $\mathcal{O}_{add}$ be the set of all $e$-add-downstream operations $u$ in $\mathcal{H}(S)$ such that $\mathit{NearestDel}(u) \cap \mathcal{H}(S) = \emptyset$. If $\mathcal{O}_{others} = \mathcal{H}(S) \setminus \mathcal{O}_{add}$ then, $S = S_\perp \circ \mathcal{O}_{others} \circ \mathcal{O}_{add}$ (since the downstream operations commute). Using Lemma 3 we can show that $e \notin S_\perp \circ \mathcal{O}_{others}$. Again from Lemma 3, $e \in S$ iff $\mathcal{O}_{add}$ is non-empty iff there exists an $e$-add-downstream operation $u$ such that $\mathit{NearestDel}(u) \cap \mathcal{H}(S) = \emptyset$. □

Given a reachable state $S$ we define the set of timestamps of all the elements added and deleted, $\mathit{Seen}(S)$, as $\{(c, r) \mid c \in S.V[r]\}$, and the set of timestamps of elements deleted in $S$, $\mathit{Deletes}(S)$, as $\mathit{Seen}(S) \setminus \{(\mathit{ts}(m), \mathit{rep}(m)) \mid m \in S.E\}$. For states $S, S'$ we say $S \leq_{compare} S'$ to mean that $\mathit{compare}(S, S')$ returns true.

**Lemma 4.** *For states $S_1$,$S_2$ and $S_3$ ,*

1. *$S_1 \leq_{compare} S_2$ iff $\mathit{Seen}(S_1) \subseteq \mathit{Seen}(S_2)$ and $\mathit{Deletes}(S_1) \subseteq \mathit{Deletes}(S_2)$. Therefore $\leq_{compare}$ defines a partial order on $\mathcal{S}$.*
2. *$S_3 = S_1 \circ \mathit{merge}(S_2)$ iff $\mathit{Seen}(S_3) = \mathit{Seen}(S_1) \cup \mathit{Seen}(S_2)$ and $\mathit{Deletes}(S_3) = \mathit{Deletes}(S_1) \cup \mathit{Deletes}(S_2)$ iff $S_3$ is the least upper bound of $S_1$ and $S_2$ in the partial order defined by $\leq_{compare}$.*

*Proof Sketch:*

1. Follows from the definitions of *Seen*, *Deletes* and the code of *compare* in the optimized solution.
2. The first equivalence follows from the code of *merge*, and the definitions of *Seen*, *Deletes*. The second equivalence follows from the first part, and the fact that $A \cup B$ is the least upper bound of sets $A$ and $B$ over the partial order defined by $\subseteq$.

□

From Lemma 1 and Lemma 4, we have the following.

**Theorem 2.** *The optimized* OR-Set *implementation is a CmRDT and CvRDT.*

## 7 Conclusion and Future work

In this paper, we have presented an optimized OR-Set implementation that does not depend on the order in which updates are delivered. The worst-case space complexity is comparable to the naïve implementation [5] and the best-case complexity is the same as that of the solution proposed in [6].

The solution in [6] requires causal ordering over all updates. As we have argued, this is an unreasonably strong requirement. On the other hand, there seems to be no simple relaxation of causal ordering that retains the structure required by the simpler algorithm of [6]. Our new generalized algorithm can accommodate any specific ordering constraint that is guaranteed by the delivery subsystem. Moreover, our solution has led us to identify $k$-causal ordering as a natural generalization of causal ordering, where the parameter $k$ directly captures the impact of out-of-order delivery on the space requirement for bookkeeping.

Our optimized algorithm uses interval version vectors to keep track of the elements that have already been seen. It is known that regular version vectors have a bounded representation when the replicas communicate using pairwise synchronization [9]. An alternative proof of this in [12] is based on the solution to the *gossip problem* for synchronous communication [13], which has also been generalized to message-passing systems [14]. It would be interesting to see if these ideas can be used to maintain interval version vectors using a bounded representation. This is not obvious because intervals rely on the linear order between timestamps and reusing timestamps typically disrupts this linear order.

Another direction to be explored is to characterize the class of datatypes with noncommutative operations for which a CRDT implementation can be obtained using interval version vectors.

## References


1. Gilbert, S., Lynch, N.A.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News **33**(2) (2002) 51–59
2. Shapiro, M., Kemme, B.: Eventual consistency. In: Encyclopedia of Database Systems. (2009) 1071–1072
3. Saito, Y., Shapiro, M.: Optimistic replication. ACM Comput. Surv. **37**(1) (2005) 42–81
4. Vogels, W.: Eventually consistent. ACM Queue **6**(6) (2008) 14–19
5. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de recherche RR-7506, INRIA (January 2011) `http://hal.inria.fr/inria-00555588/PDF/techreport.pdf`.
6. Bieniusa, A., Zawirski, M., Preguiça, N.M., Shapiro, M., Baquero, C., Balegas, V., Duarte, S.: An optimized conflict-free replicated set. CoRR **abs/1210.3368** (2012)
7. Fischer, M.J., Lynch, N.A., Paterson, M.: Impossibility of distributed consensus with one faulty process. J. ACM **32**(2) (1985) 374–382
8. Shapiro, M., Preguiça, N.M., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: SSS. (2011) 386–400
9. Almeida, J.B., Almeida, P.S., Baquero, C.: Bounded version vectors. In: DISC. (2004) 102–116
10. Malkhi, D., Terry, D.B.: Concise version vectors in WinFS. Distributed Computing **20**(3) (2007) 209–219
11. Mukund, M., Shenoy R, G., Suresh, S.P.: Optimized or-sets without ordering constraints. Technical report, Chennai Mathematical Institute (2013) `http://www.cmi.ac.in/~madhavan/papers/pdf/mss-tr-2013.pdf`.
12. Mukund, M., Shenoy R, G., Suresh, S.P.: On bounded version vectors. Technical report, Chennai Mathematical Institute (2012) `http://www.cmi.ac.in/~gautshen/pubs/BVV/on_bounded_version_vectors.pdf`.
13. Mukund, M., Sohoni, M.A.: Keeping track of the latest gossip in a distributed system. Distributed Computing **10**(3) (1997) 137–148
14. Mukund, M., Narayan Kumar, K., Sohoni, M.A.: Bounded time-stamping in message-passing systems. Theor. Comput. Sci. **290**(1) (2003) 221–239