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 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 $m : \mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S}$ 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 kth 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 kth 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 cth *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).

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

Algorithm 1

A Naive OR-set implementation without ordering constraints on operations, 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'
```

Fig. 1. A Naive OR-Set implementation

involving m may overtake an *add* update for m. For example in Figure 2, 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 2, 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.

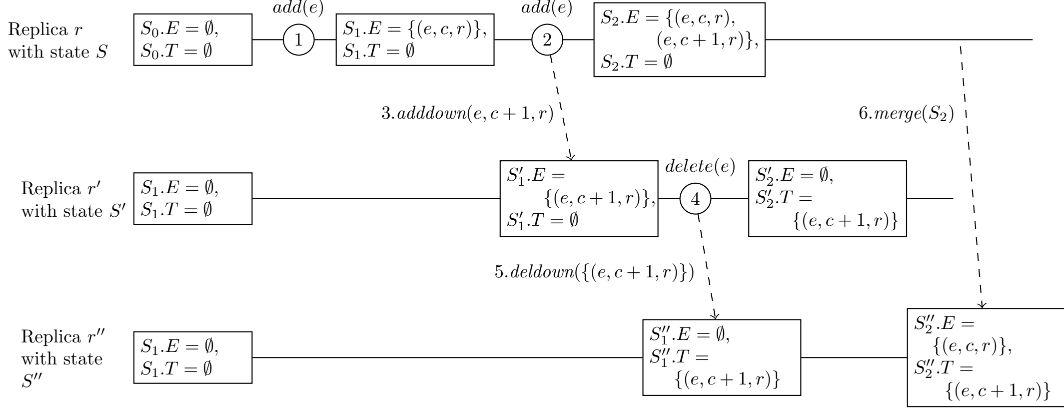

Fig. 2. Non-transitivity of the happened-before relation.

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

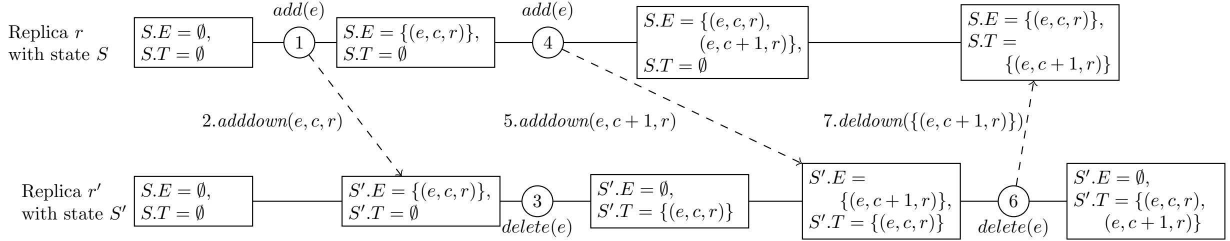

Fig. 3. Non-intuitive behaviour of *deletes* in the absence of causal delivery.

Example 2 *In the absence of causal delivery, sometimes a* delete-downstream(e) *may not remove all copies of* e *from the set—even copies corresponding to* $\mathrm{add}(e)$ *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 3.*

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 ℓ. 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 $\textsc{pack}(X)$ collapses a set of numbers X 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 $X \subseteq \mathbb{N}$, $A, B \in \mathcal{I}$, and $n \in \mathbb{N}$:

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

Algorithm 2 An optimized OR-Set implementation

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

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

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 and Space Complexity

Let S_r denote the state of a replica $r \in [0 \ldots N{-}1]$. Let n_ℓ be the number of *adddown* operations whose source is ℓ. 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 jth 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 cth *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, $S_r.V[r']$ consists of one interval corresponding to the first $(c - k)$ add-downstream operations from r' and at most $k/2$ intervals for the remaining k add-downstream operations from r'. Since this is true for every r', we can conclude that n_{int} is bounded by $O(k)$. Hence, the space-complexity of the state S_r of an optimized-OR-Set in the presence of *k-causal-delivery* is $O((n_p+Nk)\log(n_t))$. If we assume causal delivery of updates ($k = 1$), the space complexity is bounded by $O((n_p + N)\log(n_t))$.

Coalescing adds: An Optimization

With k-causal delivery, we can also enforce a bound on the size of the E set at every replica by coalescing the adds of the same element originating from the same replica. The algorithm 3 captures this optimization. Whenever a replica r receives a downstream add-request of an element (e, c, r') from a replica r' , it could evict all the triples (e, c', r') from its E set for in which $c' \leq c - k$ (lines 10-11 in Algorithm 3). Every replica can uniformly do this add-coalescing on receiving (e, c, r') since k-causal-delivery ensures that any replica which sees (e, c, r') would have seen all the triple (f, c', r'). Similarly, whenever a replica sees a triple (e, c, r') as a part of the downstream delete, it knows that the source-replica of that delete operation would have seen all the triples (f, c', r') with $c' \leq c - k$, and would have performed add-coalescing of all such triples with $f = e$. So this replica on receiving the downstream delete can also evict all triples (e, c', r') with $c' \leq c - k$ (lines 26-27) even though c' may not feature in the interval version vector $V'[r']$ sent as an argument of the downstream-delete operation.

These optimisations ensures that every replica r stores at most k triples corresponding to a visible element $e \in \mathcal{U}$ added by the source replica r'. Thus if the number of visible elements is denoted by n_a, the size of the E set is bounded

by $O(n_a k N \log(n_t))$. Thus the total space complexity with add-coalescing is $O((n_a+1)kN\log(n_t))$. The message complexity of the downstream delete operation is bounded by $O(Nk\log(n_t))$. If we assume causal delivery of updates ($k=1$), the space complexity and the message complexity of our optimized solution matches the space complexity and the message complexity of the solution in [6].

It should also be noted that if the messages are delivered in FIFO order, we can coalesce the adds in a manner similar to the case when $k=1$ in Algorithm 3. Thus, the space complexity of the E set at every replica would be $O(n_a N \log(n_t))$. However, FIFO does not guarantee any bound on the the number of intervals in each component of the interval-version-vector as illustrated in example 3.

Example 3 *Consider a OR-Set with three replicas r, r' and r''. Suppose r gets unboundedly many add requests of which every alternate add request corresponds to that of element e and every other pair of add requests correspond to that of distinct elements of the universe. Thus the E set at r would be of the form $\{(e,0,r),(f,1,r),\ (e,2,r),(g,3,r),(e,4,r),\ldots\}$. Suppose further that all the downstream add operations from r get delivered at replica r', but none of the downstream are delivered yet at replica r''. This is allowed in FIFO ordering. Now, replica r' gets delete requests corresponding to all the non-e elements that have been added by replica r. Suppose further that the downstream operations of these deletes are delivered at r'' in FIFO order. Thus at r'', $V[r]=\{[1,1],[3,3],[5,5],\ldots\}$. Hence the number of intervals can be unbounded.*

In the case where message delivery is constrained by FIFO ordering, the worst case space complexity in would be $O((n_a+n_{int})N\log(n_t))$ which is comparable to space complexity of the generalized algorithm without ordering constraints since $O(n_{int})$ can be as large as $O(n_t)$ as highlighted in the example above.

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 the appendix. Our aim is show that the solution satisfies the specification of OR-Sets and is a CvRDT as well as a CmRDT. In the subsequent section we provide a detailed proof of the equivalence between the naïve implementation and our optimized implementation.

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).

Algorithm 3 An optimized OR-Set implementation with k-causal-delivery

Optimized OR-set implementation for the replica r with k-causal delivery constraint

```
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          M = {m′ | data(m′) = data(m) ∧
                      rep(m′) = rep(m) ∧
                      ts(m′) ≤ ts(m) − k}
11          E = E \ M
12          if (ts(m) ∉ V[rep(m)])
13              E := E ∪ {m}
14              V[rep(m)] :=
                    add(V[rep(m)], {ts(m)})
15          if (rep(m) = r)
16              c = ts(m) + 1
17
18  DELETE(e ∈ U):
19      DELETE.PREPARE(e ∈ U):
20          Let V′ : Reps → I = [0, . . . , 0]
21          for m ∈ E with data(m) = e
22              add(V′[rep(m)], {ts(m)})
23          Broadcast downstream(e, V′)
24      DELETE.DOWNSTREAM(e ∈ U,
25                        V′ : Reps → I):
26          Let M = {m ∈ E |
                    ts(m) ∈ V′[rep(m)]∨
                    (data(m) = e ∧
                    ∃c ∈ V′[rep(m)] ts(m) ≤ c − k)}
27          E := E \ M
28          for i ∈ Reps
29              V[i] := V[i] ∪ V′[i]
30
31  Boolean COMPARE(S′, S′′ ∈ S):
32      Assume that S′ = (E′, V′)
33      Assume that S′′ = (E′′, V′′)
34      b_seen := ∀i(V′[i] ⊆ V′′[i])
35      b_deletes := ∀m ∈ E′′ \ E′
                (ts(m) ∉ V′[rep(m)])
36          // If m is deleted from E′ then
37          // it is also deleted in E′′.
38          // So anything in E′′ \ E′
39          // is not even visible in S′.
40      return b_seen ∧ b_deletes
41
42  MERGE(S′ ∈ S):
43      Assume that S′ = (E′, V′)
44      E := {m ∈ E ∪ E′ |
                m ∈ E ∩ E′∨
                ts(m) ∉ V[rep(m)] ∩ V′[rep(m)]}
45  // You retain m if it is either
46  // in the intersection, or if it is fresh
47  // (so one of the states has not seen it).
48  ∀i.(V[i] := V[i] ∪ V′[i])
```

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.

We first prove that the *add* and *delete* downstream operations can be simulated using the *merge* operation with appropriate arguments.

Proposition 1. *Let S be a state of some replica. Let $u(a)$ be some downstream operation where a is the argument of the downstream operation. If* $S_{op} = S \circ u(a)$ *then* $S_{op} \equiv S \circ merge(S_\perp \circ u(a))$.

Proof. Let $S' = S_\perp \circ u(a)$ and $S_{merge} = S \circ merge(S')$. We have to consider the following two cases:

Case $u(a) = \boldsymbol{adddown}(m)$: In this case, from the code for ADD.DOWNSTREAM, it is seen that

$$S_{op}.E = \begin{cases} S.E \cup \{m\} & \text{if } ts(m) \notin S.V[rep(m)] \\ S.E & \text{otherwise} \end{cases}$$

Also,

$$S_{op}.V[r] = \begin{cases} S.V[r] \cup \{ts(m)\} & \text{if } r = rep(m) \text{ and } ts(m) \notin S.V[r] \\ S.V[r] & \text{otherwise} \end{cases}$$

Simplifying, we get

$$S_{op}.V[r] = \begin{cases} \mathbf{add}(S.V[r], \{ts(m)\}) & \text{if } r = rep(m) \\ S.V[r] & \text{otherwise} \end{cases}$$

From the code of MERGE, it is immediately seen that $S'.E = \{m\}$,

$$S'.V[r] = \begin{cases} \{(ts(m), ts(m))\} & \text{if } r = rep(m) \\ \emptyset & \text{otherwise} \end{cases}$$

Again from the code of MERGE, it follows that $m' \in S_{merge}.E$ iff

$(m' \neq m$ and $m' \in S.E)$ or $\{m' = m$ and $(m \in S.E$ or $ts(m) \notin S.V[rep(m)])\}$.

Simplifying, we see that $m' \in S_{merge}.E$ iff

$$m' \in S.E \text{ or } (m' = m \text{ and } ts(m) \notin S.V[rep(m)]).$$

Thus it follows that $S_{merge}.E = S_{op}.E$.
Again from the code of MERGE, we see that

$$S_{merge}.V[r] = \begin{cases} \mathbf{add}(S.V[r], \{ts(m)\}) & \text{if } r = rep(m) \\ S.V[r] & \text{otherwise} \end{cases}$$

Thus $S_{merge}.V = S_{op}.V$. Therefore $S_{op} = S_{merge}$.

Case $u(a) = \boldsymbol{deldown}(V')$: From the code V' is the interval version vectors containing timestamps of triples m that have to be deleted. Now, it is easy to see that $S'.E = \emptyset$ and $S'.V = V'$. From the code for the merge, for any r', $S_{merge}.V[r'] = S.V[r'] \cup S'.V[r']$. Similarly from the code for *deldown*, $S_{merge}.V[r'] = S.V[r'] \cup V'[r']$. Hence $S_{merge}.V = S_{op}.V$. Since $S'.E = \emptyset$, $S_{merge}.E = S.E \setminus \{m' \mid ts(m') \in S'.V[rep(m')]\} = S.E \setminus \{m' \mid ts(m') \in V'[rep(m')]\} = S_{op}.E$. From this and the fact that $S_{op}.V = S_{merge}.V$, we can conclude that $S_{op} = S_{merge}$.

Any reachable state of the OR-Set is obtained by applying some sequence of *adddown*, *deldown* and *merge* operations to the initial state $S_\perp$. From Proposition 1, since any *adddown* or *deldown* operation can be simulated using the *merge* operation, the structure of the reachable states can be reasoned about using the properties of the merge operations. We prove two important properties of merges, *commutativity* and *idempotence*.

Proposition 2. *If S_1, S_2, S_3 are three reachable states of some run then $(S_1 \circ merge(S_2)) \circ merge(S_3) \equiv S_1 \circ merge(S_2 \circ merge(S_3)) \equiv (S_1 \circ merge(S_3)) \circ merge(S_2)$*

Proof. Let

- $S_{12} = S_1 \circ merge(S_2)$, $S_{23} = S_2 \circ merge(S_3)$ and $S_{13} = S_1 \circ merge(S_3)$,
- $S_{(12)3} = S_{12} \circ merge(S_3)$, $S_{1(23)} = S_1 \circ merge(S_{23})$ and $S_{(13)2} = S_{13} \circ merge(S_2)$.

For any r, $S_{(12)3}.V[r] = S_{1(23)}.V[r] = S_{1(32)}.V[r] = S_1.V[r] \cup S_2.V[r] \cup S_3.V[r]$.

For any distinct $i, j \in \{1,2,3\}$ one can observe from the code of *merge* that for any triple m, $m \in S_{ij}.E$ iff $(m \in S_i.E \cap S_j.E) \vee (m \in S_i.E \wedge ts(m) \notin S_j.V[rep(m)]) \vee (m \in S_j.E \wedge ts(m) \notin S_i.V[rep(m)])$. Using repeated application of this principle, we can show that $m \in S_{(12)3}.E$ iff one of the following is satisfied:

- $(m \in S_1.E \cap S_2.E \cap S_3.E)$
- $\bigvee_{i,j,k \in \{1,2,3\} \wedge i \neq j \neq k} (m \in (S_i.E \cap S_j.E) \wedge ts(m) \notin S_k.V[rep(m)])$
- $\bigvee_{i,j,k \in \{1,2,3\} \wedge i \neq j \neq k} (m \in S_i.E \wedge ts(m) \notin (S_j.V[rep(m)] \cup S_k.V[rep(m)]))$.

One can show that these are also the conditions for m to belong to $S_{1(23)}.E$ or $S_{1(32)}.E$ Thus $S_{(12)3}.E = S_{1(23)}.E = S_{1(32)}.E$. Thus $S_{(12)3} = S_{1(23)} = S_{1(32)}$.

Proposition 3. *If S_1 and S_2 are two reachable states then $(S_1 \circ merge(S_2)) \circ merge(S_2) \equiv S_1 \circ merge(S_2)$.*

Proof. Let $S = S_1 \circ merge(S_2)$ and $S' = S \circ merge(S_2)$. For any r, $S'.V[r] = S.V[r] \cup S_2.V[r] = (S_1.V[r] \cup S_2.V[r]) \cup S_2.V[r] = S_1.V[r] \cup S_2.V[r] = S.V[r]$.

For any element m, $m \in S'.E$ iff $m \in S.E \cap S_2.E$ or $m \in S.E \wedge ts(m) \notin S_2.V[rep(m)]$ or $m \in S_2.E \wedge m \notin S.V[rep(m)]$. Again, we have $m \in S.E$ iff $m \in S_1.E \cap S_2.E$ or $m \in S_1.E \wedge ts(m) \notin S_2.V[rep(m)]$ or $m \in S_2.E \wedge ts(m) \notin S_1.V[rep(m)]$. On combining and simplifying these two conditions we can see that $m \in S'.E$ iff $m \in S.E$.

Thus $S \equiv S'$ and hence $(S_1 \circ merge(S_2)) \circ merge(S_2) \equiv S_1 \circ merge(S_2)$.

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 π_1 and π_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. We prove the result by induction on $|\mathcal{O}|$. If $|\mathcal{O}| = 1$, the result follows trivially. Assume that the result holds for all $\mathcal{O}$ of size smaller than n. Now consider $\mathcal{O} = \{u_1, u_2, \ldots, u_n\}$. Let π_1 and π_2 be any two permutations of $[1 \ldots n]$. Let $i, j \in [1 \ldots n]$ such that $\pi_1(i) = \pi_2(n-1)$ and $\pi_1(j) = \pi_2(n)$.

$S_1 = (S \circ u_{\pi_1(1)} \circ u_{\pi_1(2)} \cdots u_{\pi_1(n-1)}) \circ u_{\pi_1(n)}$. From the induction hypothesis, this is the same as $(S \circ u_{\pi_1(1)} \circ u_{\pi_1(2)} \cdots u_{\pi_1(j-1)} \circ u_{\pi_1(j+1)} \cdots u_{\pi_1(n-1)} \circ u_{\pi_1(j)}) \circ u_{\pi_1(n)}$. From Proposition 1, we can be write this $((S \circ u_{\pi_1(1)} \circ u_{\pi_1(2)} \cdots u_{\pi_1(j-1)} \circ u_{\pi_1(j+1)} \cdots u_{\pi_1(n-1)}) \circ merge(S_\perp \circ u_{\pi_1(j)})) \circ merge(S_\perp \circ u_{\pi_1(n)})$. From Proposition 2, since merges commute, using Proposition 1 we can rewrite this as $(S \circ u_{\pi_1(1)} \circ u_{\pi_1(2)} \cdots u_{\pi_1(j-1)} \circ u_{\pi_1(j+1)} \cdots u_{\pi_1(n-1)} \circ u_{\pi_1(n)}) \circ u_{\pi_1(j)}$. By the induction hypothesis, this is the same as $(S \circ u_{\pi_1(1)} \circ u_{\pi_1(2)} \cdots u_{\pi_1(i-1)} \circ u_{\pi_1(i+1)} \cdots u_{\pi_1(j-1)} \circ u_{\pi_1(j+1)} \cdots u_{\pi_1(n-1)} \circ u_{\pi_1(n)} \circ u_{\pi_1(i)}) \circ u_{\pi_1(j)}$. Since $\pi_1(i) = \pi_2(n-1)$ and $\pi_1(j) = \pi_2(n)$, and $\{\pi_1(1), \ldots \pi_1(i-1), \pi_1(i+1), \ldots, \pi_1(j-1), \pi_1(j+1), \ldots \pi_1(n)\} = \{\pi_2(1), \ldots \pi_2(n-2)\}$ by the induction hypothesis, S_1 is the same as $S \circ u_{\pi_2(1)} \circ u_{\pi_2(2)} \cdots u_{\pi_2(n-1)}) \circ u_{\pi_2(n)}$, which is the same as S_2.

The following lemma shows the relationship between any reachable state and its causal history. From this result we can also conclude that any two reachable states with the same causal history are identical.

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

Proof. The proof follows by induction over the size of $\mathcal{H}(S)$. If $\mathcal{H}(S) = \emptyset$ then by definition, $S = S_\perp$ and the result follows. On the other hand suppose $\mathcal{H}(S) \neq \emptyset$. Since S is a reachable state, we can find a reachable state $S' \neq S$ and an operation $u \in \{adddown, deldown, merge\}$ such that $S = S' \circ u$. By definition, $\mathcal{H}(S') \subsetneq \mathcal{H}(S)$. Hence, by the induction hypothesis, $S' = S_\perp \circ \mathcal{H}(S')$. Consider

the case when $u = merge(S'')$ where $S'' \neq S_\perp$ and $S'' \neq S$. By the induction hypothesis $S'' = S_\perp \circ \mathcal{H}(S'')$. Let $\mathcal{H}(S'') = \{u''_1, u''_2, \ldots, u''_k\}$. Now $S = S' \circ merge(S'')$ which can be written as $S' \circ merge(S_\perp \circ u''_1 \circ u''_2 \cdots u''_m)$. By appealing to Propositions 1 and 2, this is the same as $S' \circ merge(S_\perp \circ u''_1) \circ merge(S_\perp \circ u''_2) \cdots merge(S_\perp \circ u''_k) \circ merge(S_\perp)$. Appealing to Proposition 1 once again, we can rewrite this as $S' \circ u''_1 \circ u''_2 \cdots u''_k. \circ merge(S_\perp)$. Since for any state S''', $S''' \circ merge(S_\perp) = S_\perp \circ merge(S''') = S'''$, and the induction hypothesis, we can write $S = S_\perp \circ \mathcal{H}(S') \circ u''_1 \circ u''_2 \cdots u''_k = S_\perp \circ \mathcal{H}(S') \circ \mathcal{H}(S'')$. From Propositions 2 and 3, this is the same as $S_\perp \circ (\mathcal{H}(S') \cup \mathcal{H}(S''))$. By definition, $\mathcal{H}(S) = \mathcal{H}(S') \cup \mathcal{H}(S'')$.

If u were an *adddown* or a *deldown* operation, appealing to Proposition 1, we can reduce it to the merge case.

We now analyse the structure of the state of the optimized implementation. In particular, we state the necessary and sufficient conditions for a certain integer to be present in the integer version vector of the state.

Proposition 4. *Let S be any reachable state. $c \in S.V[r]$ iff there exists an update operation $u \in \mathcal{H}(S)$ such that u is an adddown operation with* $\arg(u) = m$ *with $ts(m) = c$ and $rep(m) = r$ or u is a deldown operation with* $\arg(u) = V'$ *and $c \in V'[r]$.*

Proof. Proof follows by induction over $|\mathcal{H}(S)|$. The base case when $S = S_\perp$ is trivial. If $|\mathcal{H}(S)| = 1$, then from Lemma 2, $S = S_\perp.u$. If u is an *adddown* operation with $\arg(u) = m$, such that $ts(m) = c$ and $rep(m) = r$, then $c' \in S.V[r']$ iff $(c' = c)$ and $(r' = r)$. If $u =$ *deldown* operation with $\arg(u) = V'$ then, $c \in S.V[r]$ iff $c \in V'[r]$. Thus the result is true for all S with $|\mathcal{H}(S)| = 1$. Assume that the result holds for all states S with $|\mathcal{H}(S)| < n$. Now consider a state S such that $|\mathcal{H}(S)| = n$.

From proposition 1 we can write $S = S'.merge(S'')$ with $\mathcal{H}(S') \subsetneq \mathcal{H}(S)$ and $\mathcal{H}(S'') \subsetneq \mathcal{H}(S)$ for appropriate S' and S''. Now $c \in S.V[r]$ iff $c \in S'.V[r] \cup S''.V[r]$. Since $|\mathcal{H}(S')| < n$ and $|\mathcal{H}(S'')| < n$, by induction hypothesis, this happens iff there exists such an downstream operation $u \in \mathcal{H}(S') \cup \mathcal{H}(S'')$ such that u is an *adddown* operation with $\arg(u) = m$ for which $ts(m) = c$ and $rep(m) = r$ or u is a *deldown* operation with $\arg(u) = V'$ and $c \in V'[r]$. By definition, $\mathcal{H}(S) = \mathcal{H}(S') \cup \mathcal{H}(S'')$. Hence $u \in \mathcal{H}(S)$.

We now establish the necessary and sufficient conditions for a *deldown* operation to be a *nearest-delete* of an *adddown* operation by inspecting their respective arguments. This result along with Proposition 4 yields the necessary and sufficient condition characterising the structure of the Interval Version Vectors of any reachable state through the *nearest-delete* relations between the *adddown* and *deldown* operations present in the state-history.

Proposition 5. *If u is an adddown operation with* $\arg(u) = m$ *and u' is a deldown operation with* $\arg(u') = V'$ *then $u' \in NearestDel(u)$ iff $ts(m) \in V'[rep(m)]$.*

Proof. Suppose $data(m) = e$. Let p and p' be the corresponding *del prepare* methods of u and u' respectively. Let r' be the source replica of the update

(p', u'). Let S be the state of r' before applying u and let S' be the state of r' before applying p'. Let $\rightarrow$ be a total order on the set of all downstream operations of OR-Set such that $\xrightarrow{hb} \subseteq \rightarrow$. Let u' be the ith *e-deldown* operation in $\rightarrow$. The proof follows from induction over i.

The base case occurs when $i = 1$. Suppose $ts(m) \in V'[rep(m)]$. Since V' is prepared by p', from the code of *delprepare*, we know that $m \in S'.E$. This is possible only if $u \in \mathcal{H}(S')$, as every *adddown* operation has a unique argument. Hence $u \xrightarrow{hb} u'$. Since u is the earliest *e-deldown* operation in $\rightarrow$ and since $\rightarrow$ is consistent with $\xrightarrow{hb}$, there is no other *e-deldown* operation u'' such that $u \xrightarrow{hb} u'' \xrightarrow{hb} u'$. Hence by definition, $u' \in NearestDel(u)$. Conversely suppose $u' \in NearestDel(u)$. Then since $\mathcal{H}(S')$ contains u and cannot not contain any *e-deldown* operations without contradicting the minimality of u' in $\rightarrow$, we can conclude that $m \in S'.E$. From the code of *delprepare*, $ts(m) \in V'[rep(m)]$.

Assume that the result holds for all $i < n$. Suppose u' is the nth *e-deldown* operation in $\rightarrow$. If $ts(m) \in V'[rep(m)]$ then we know that $m \in S'.E$ which implies that $u \in \mathcal{H}(S')$ and hence $u \xrightarrow{hb} u'$. If $u' \notin NearestDel(u)$ then there exists an *e-deldown* operation u'' such that $u \xrightarrow{hb} u'' \xrightarrow{hb} u$ and $u'' \in NearestDel(u)$. Since $\rightarrow$ is consistent with $\xrightarrow{hb}$, u'' occurs before u' in $\rightarrow$. Let $V'' = \arg(u'')$. By induction hypothesis, $ts(m) \in V''[rep(m)]$. If S'' is the state of r' after applying u'' then, from the code of *deldown*, we know that $m \notin S''.E$. No *adddown* operation applied by r' to reach S' from S'' adds m to the E set of r' since $ts(m) \in S''.V[rep(m)]$ (from proposition 4). Hence $m \notin S'.E$ which implies that $ts(m) \notin V'[rep(m)]$ which is a contradiction. Hence $u' \in NearestDel(u)$.

Conversely suppose $u' \in NearestDel(u)$. Suppose there is an *e-deldown* operation $u'' \in \mathcal{H}(S')$ such that $\arg(u'') = V''$ and $ts(m) \in V''[rep(m)]$. Since $u'' \xrightarrow{hb} u'$, u'' appears before u' in the total order $\rightarrow$. By induction hypothesis, $u'' \in NearestDel(u)$ which implies $u \xrightarrow{hb} u'' \xrightarrow{hb} u'$ which contradicts the fact that $u' \in NearestDel(u)$. Hence no such u'' exists. Since $u \in \mathcal{H}(S')$, $m \in S'.E$. From the code of *delprepare*, m is in the argument set prepared by p'. Hence $ts(m) \in V'[rep(m)]$.

Once a replica has applied a *deldown* operation, any subsequent *adddown* operation for which the earlier *deldown* operation was a *nearest-delete* has no effect on the state of the replica. We formally prove this through the following proposition.

Proposition 6. *Let S be a reachable state, u be a adddown operation and u' a deldown operation such that $u' \in NearestDel(u)$. Then $S \circ u' \circ u \equiv S \circ u'$.*

Proof. Let $S' = S \circ u'$. We need to show that $S' \circ u \equiv S'$. Let $m = \arg(u), V' = \arg(u')$. From proposition 5, $ts(m) \in V'[rep(m)]$. From proposition 4, $ts(m) \in S'.V[rep(m)]$. Hence from the code of *adddown*, the operation u will not affect any change in the state. Hence $S' \circ u \equiv S'$.

We now prove the necessary and sufficient conditions characterising the structure of the E set of any reachable state. With this result we are closer to showing that the optimized OR-Set implementation satisfies the concurrent-specification.

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 $\mathit{NearestDel}(u) \cap \mathcal{H}(S) = \emptyset$.*

Proof. Suppose $\mathit{NearestDel}(u) \cap \mathcal{H}(S) = \emptyset$.

From Proposition 5, for any u', we have $u' \in \mathit{NearestDel}(u)$ iff $m = \arg(u), V' = \arg(u') \implies \mathit{ts}(m) \in V'[\mathit{rep}(m)]$. Also it is given that $u \notin \mathcal{H}(S)$. Since $\mathit{NearestDel}(u) \cap \mathcal{H}(S) = \emptyset$, from proposition 4, we know that $\mathit{ts}(m) \notin S.V[\mathit{rep}(m)]$. Hence from the code of the *adddown* operation, $m \in S'.E$. Conversely suppose $m \in S.E$. If $\mathit{NearestDel}(u) \cap \mathcal{H}(S) \neq \emptyset$. Let $u' \in \mathit{NearestDel}(u) \cap \mathcal{H}(S)$. Let $S' = S_\perp \circ (\mathcal{H}(S) \setminus \{u, u'\})$. Clearly $m \notin S''.E$ since $\mathcal{H}(S'')$ does not contain u. $S = S'' \circ u' \circ u$. From Proposition 6, we know that $S'' \circ u' \circ u \equiv S' \circ u' = S'$. From the code of *deldown* we can see that $S'.E \subseteq S''.E$. Hence $m \notin S'.E$. Since $S = S'$, $m \notin S.E$ which is a contradiction. Therefore $\mathit{NearestDel}(u) \cap \mathcal{H}(S) = \emptyset$.

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

Proof. 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 $\mathit{NearestDel}(u) \cap \mathcal{H}(S) = \emptyset$. Let $\mathcal{O}_{others} = \mathcal{H}(S) \setminus \mathcal{O}_{add}$ and $S' = S_\perp \circ \mathcal{O}_{others}$. Then, $S = S' \circ \mathcal{O}_{add}$.

Since $\mathcal{O}_{others}$ contains no e-add-downstream operation u such that $\mathit{NearestDel}(u) \cap \mathcal{H}(S) = \emptyset$, from Lemma 3, we can conclude that for all $m \in S'.E$, $\mathit{ts}(m) \neq e$. Hence $e \notin S'$. 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.

To show that the optimized OR-Set implementation is a CvRDT we need to define a partial order on the set of reachable states of the implementation. We can observe that for states S_1 and S_2, if $\mathit{Seen}(S_1) = \mathit{Seen}(S_2)$ and $\mathit{Deletes}(S_1) = \mathit{Deletes}(S_2)$ then $S_1 = S_2$. Thus the $(\mathit{Seen}(S), \mathit{Deletes}(S))$ pair uniquely identifies a state S. We use this definition to show that there exists a well-defined partial order on the set of states of the optimized OR-Set implementation through the following Proposition.

Proposition 7. *For states S_1 and S_2 , $S_1 \leq_{compares} S_2$ iff $\mathit{Seen}(S_1) \subseteq \mathit{Seen}(S_2)$ and $\mathit{Deletes}(S_1) \subseteq \mathit{Deletes}(S_2)$. $\leq_{compare}$ induces a partial order on $\mathcal{S}$.*

Proof. Suppose $S_1 \leq_{compares} S_2$. We know that for every r', $S_1.V[r'] \subseteq S_2.V[r']$. Hence $\mathit{Seen}(S_1) \subseteq \mathit{Seen}(S_2)$. Also, for every $m' \in S_2.E \setminus S_1.E$, $\mathit{ts}(m') \notin S_1.V[\mathit{rep}(m')]$. Suppose $(c, r) \in \mathit{Deletes}(S_1)$. By definition, $c \in S_1.V[r]$ and $\forall m \in S_1.E$, $(\mathit{ts}(m), \mathit{rep}(m)) \neq$

(c, r). Now suppose $(c, r) \notin \mathit{Deletes}(S_2)$. Since, $S_1.V[r] \subseteq S_2.V[r]$, $c \in S_2.V[r]$. Hence it is the case that $\exists m \in S_2.E$ with $(\mathit{ts}(m), \mathit{rep}(m)) = (c, r)$. But from the code of *compare*, if $m \in S_2.E \setminus S_1.E$, then $\mathit{ts}(m) \notin S_1.V[\mathit{rep}(m)]$, which is a contradiction. Hence $(c, r) \in \mathit{Deletes}(S_2)$. Thus $\mathit{Deletes}(S_1) \subseteq \mathit{Deletes}(S_2)$.

Conversely, suppose $\mathit{Seen}(S_1) \subseteq \mathit{Seen}(S_2)$ and $\mathit{Deletes}(S_1) \subseteq \mathit{Deletes}(S_2)$. By definition of *Seen*(), for all r', $S_1.V[r'] \subseteq S_2.V[r']$. Also, if $m \in S_2.E \setminus S_1.E$. Now if $\mathit{ts}(m) \in S_1.V[\mathit{rep}(m)]$ then, $(\mathit{ts}(m), \mathit{rep}(m)) \in \mathit{Deletes}(S_1)$. However, Since $m \in S_2.E$, $(\mathit{ts}(m), \mathit{rep}(m)) \notin \mathit{Deletes}(S_2)$ which contradicts the fact that $\mathit{Deletes}(S_1) \subseteq \mathit{Deletes}(S_2)$. Hence $\mathit{ts}(m) \notin S_1.V[\mathit{rep}(m)]$. From this we can conclude that $S_1 \leq_{compare} S_2$.

For any S, $S \leq_{compare} S$. For S_1 and S_2, $S_1 \leq_{compare} S_2$ and $S_2 \leq_{compare} S_1$ implies that $\mathit{Seen}(S_1) = \mathit{Seen}(S_2)$ and $\mathit{Deletes}(S_1) = \mathit{Deletes}(S_2)$. This implies that $S_1 = S_2$. Finally $S_1 \leq_{compare} S_2$ and $S_2 \leq_{compare} S_3$ implies $\mathit{Seen}(S_1) \subseteq \mathit{Seen}(S_2) \subseteq \mathit{Seen}(S_3)$ and $\mathit{Deletes}(S_1) \subseteq \mathit{Deletes}(S_2) \subseteq \mathit{Deletes}(S_3)$. Hence from the previous part $S_1 \leq_{compare} S_3$. Thus $\leq_{compare}$ is a partial order on the set of states $\mathcal{S}$.

We first show that the state computed *merge* operation is the upper bound of the two states in the partial order defined by $\leq_{compare}$.

Proposition 8. *For states S_1, S_2 and S_3 we have $S_3 = S_1 \circ \mathit{merge}(S_2)$ iff $\mathit{Seen}(S_3) = \mathit{Seen}(S_1) \cup \mathit{Seen}(S_2)$ and $\mathit{Deletes}(S_3) = \mathit{Deletes}(S_1) \cup \mathit{Deletes}(S_2)$.*

Proof. From the code, for any r', $S_3.V[r'] = S_1.V[r'] \cup S_2.V[r']$. Hence, by definition, $\mathit{Seen}(S_3) = \mathit{Seen}(S_1) \cup \mathit{Seen}(S_2)$. $(c, r) \in \mathit{Deletes}(S_3)$ iff $(c, r) \in \mathit{Seen}(S_3)$ and $\forall m \in S_3.E$, $(\mathit{ts}(m), \mathit{rep}(m)) \neq (c, r)$. Now, $m \in S_3.E$ iff $m \in S_1.E \cap S_2.E \vee (m \in S_1.E \wedge \mathit{ts}(m) \notin S_2.V[\mathit{rep}(m)]) \vee (m \in S_2.E \wedge \mathit{ts}(m) \notin S_1.V[\mathit{rep}(m)])$. From this and the definition of *Deletes*() we can conclude that $(c, r) \in \mathit{Deletes}(S_3)$ iff $(c, r) \in \mathit{Deletes}(S_1)$ or $(c, r) \in \mathit{Deletes}(S_2)$ iff $(c, r) \in \mathit{Deletes}(S_1) \cup \mathit{Deletes}(S_2)$. Thus $\mathit{Deletes}(S_3) = \mathit{Deletes}(S_1) \cup \mathit{Deletes}(S_2)$.

Conversely suppose $\mathit{Seen}(S_3) = \mathit{Seen}(S_1) \cup \mathit{Seen}(S_2)$ and $\mathit{Deletes}(S_3) = \mathit{Deletes}(S_1) \cup \mathit{Deletes}(S_2)$. Let $S_3' = S_1.\mathit{merge}(S_2)$. From the previous part, $\mathit{Seen}(S_3') = \mathit{Seen}(S_1) \cup \mathit{Seen}(S_2)$ and $\mathit{Deletes}(S_3') = \mathit{Deletes}(S_1) \cup \mathit{Deletes}(S_2)$. Since $(\mathit{Seen}(S), \mathit{Deletes}(S))$ pair uniquely identifies a state, $S_3 = S_3'$.

Using this result we show that *merge* operation indeed computes the least-upper-bound.

Proposition 9. *If $S_3 = S_1 \circ \mathit{merge}(S_2)$ then S_3 is the least upper bound of S_1 and S_2 in the partial order defined by $\leq_{compare}$.*

Proof. From propositions 8 and 7 it is clear that $S_1 \leq_{compare} S_3$ and $S_2 \leq_{compare} S_3$. Hence S_3 is an upper bound of S_1 and S_2. If S_4 is any other upper bound of S_1 and S_2 then by lemma 7, $\mathit{Seen}(S_1) \subseteq \mathit{Seen}(S_4)$ and $\mathit{Seen}(S_2) \subseteq \mathit{Seen}(S_4)$. This implies that $\mathit{Seen}(S_1) \cup \mathit{Seen}(S_2) \subseteq \mathit{Seen}(S_4)$. Similarly, $\mathit{Deletes}(S_1) \cup \mathit{Deletes}(S_2) \subseteq \mathit{Deletes}(S_4)$. From lemma 8, $\mathit{Seen}(S_3) \subseteq \mathit{Seen}(S_4)$ and $\mathit{Deletes}(S_3) \subseteq \mathit{Deletes}(S_4)$. From lemma 7 this implies that $S_3 \leq_{compare} S_4$. Hence S_3 is the least upper bound of S_1 and S_2 in the partial order defined by $\leq_{compare}$.

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

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

Proof. Follows from Propositions 7, 8 and 9.

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

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

Proof. From Lemma 1, we know that all the downstream update operations of OR-Set commute. Furthermore, the OR-Set specification does not have any delivery preconditions. Thus, the implementation is a CmRDT since it satisfies the sufficient condition for a replicated datatype to be a CmRDT.

From Lemma 7, we know that $\leq_{compare}$ induces a partial order on the set of states $\mathcal{S}$ such that $(\mathcal{S}, \leq_{compare})$ is a join-semilattice. From propositions 1 and 8, we know that the states are monotonically non-decreasing across update operations. Finally, from proposition 9 we know that the merge of any pair of states yields the least upper bound of the two states in the join semi-lattice $(\mathcal{S}, \leq_{compare})$. Hence the implementation is a CvRDT.

7 Equivalence of the näive and the optimised implementations

In this section, we set up the formal semantics for both the naïve implementation (Algorithm 1) and our optimized implementation (Algorithm 2)) and establish a strong correspondence between the two in terms of bisimulation.

A naïve state describes the local state of one replica during the execution of Algorithm 1.

Definition 1. *A naïve state is a tuple* $(E, T, count)$ *where* $E, T \subseteq \mathcal{M}$ *such that* $E \cup T$ *is valid,* $E \cap T = \emptyset$ *and* $count \in \mathbb{N}$. $\mathcal{S}^{naive}$ *denotes the set of naïve states. The initial naïve state is the state* $(\emptyset, \emptyset, 0)$. *For a valid set* M, *a naïve state* $(E, T, count)$ *is called* M*-compatible if* $E \cup T \subseteq M$.

An opt state describes the local state of one replica during the execution of Algorithm 2.

Definition 2. *An opt state is a tuple* $(E, V, count)$ *where* $E \subseteq \mathcal{M}$ *is a valid set,* $V : Reps \rightarrow \mathcal{I}$ *is a vector of interval sequences and* $count \in \mathbb{N}$. $\mathcal{S}^{opt}$ *denotes the set of opt states. In the initial opt state,* $E = \emptyset$, $V[i] = \emptyset$ *for each* $i \in Reps$, $count = 0$. *For a valid set* M, *an opt state* $(E, V, count)$ *is called* M*-compatible if* $E \subseteq M$ *and for all* $r \in Reps$ *and* $c \in V[r]$, *there is some* e *such that* $(e, c, r) \in M$.

Let $\mathcal{S} = \mathcal{S}^{naive} \cup \mathcal{S}^{opt}$ be the set of all states. For a state $S \in \mathcal{S}$, we use the notation $S.E$, $S.T$, $S.V$, and $S.count$, to refer to the appropriate components of S.

To show the correspondence between the behaviour of the two implemenations, we have to define when a naïve state is equivalent to an opt state. The intuition is that $(E, T, count)$ is equivalent to $(E, V, count)$, if the elements seen so far in $E \cup T$ correspond to the timestamps represented in V—this exploits the fact that for each element $(e, c, r) \in E \cup T$ the pair (c, r) is unique.

Definition 3. *Let $S = (E, T, count)$ be a naïve state, $S' = (E', V', count')$ be an opt state, and M be a valid set of labelled elements. We say that S is M-equivalent to S' (denoted $S \equiv_M S'$) if $E = E'$, $E \cup T \subseteq M$ and for all $m \in M$, $m \in E \cup T$ iff $(m) \in V[rep(m)]$.*

The following observation is immediate for any valid set M of labelled elements.

Observation 3
1. *For every M-compatible naïve state $S = (E, T, count)$, there is exactly one opt state S' such that $S \equiv_M S'$.*
2. *For every M-compatible opt state $S' = (E', V', count')$, there is exactly one naïve state S with $S \equiv_M S'$.*

As usual, global configurations consists of tuples of local states.

Definition 4. *A naïve (respectively, opt) configuration is a tuple $(S_0, \ldots, S_{N-1})$ of naïve (respectivelly, opt) states, one for each replica. The collection of naïve and opt configurations are denoted, respectively, by $\mathcal{C}^{naive}$ and $\mathcal{C}^{opt}$. A naïve (respectively, opt) configuration $C = (S_0, \ldots, S_{N-1})$ is an initial configuration if each S_i is an initial naïve (repectively, opt) state.*

Given a valid set of labelled elements M, a naïve configuration $C = (S_0, \ldots, S_{N-1})$ is M-equivalent to an opt configuration $C' = (S'_0, \ldots, S'_{N-1})$ iff $S_i \equiv_M S'_i$ for all $i \in \{0, 1, \ldots, N-1\}$.

The effect of the local and downstream updates in Algorithms 1 and 2 is captured by defining corresponding abstract operations on the states of our formal model.

Definition 5. *The set of operations is given by*

$$\{r.query(e), r.add(e), r.adddown(m), r.del(e), r.deldown(M), r.merge(S)\}$$

where $r \in Reps, e \in \mathcal{U}, m \in \mathcal{M},\ M, M' \subseteq \mathcal{M},\ V' : Reps \to \mathcal{I}$ and $S \in \mathcal{S}$.

We say that an operation is naïve (respectively, opt) if $S \in \mathcal{S}^{naive}$ (respectively, $S \in \mathcal{S}^{opt}$). The sets of naïve and opt operations are denoted $\mathcal{O}^{naive}$ and $\mathcal{O}^{opt}$, respectively.

For an operation o, $Site(o) = r$ where $o = r.query(e), r.add(e), r.merge(S)$, etc. This is the replica at which the operation is applied.

Given a valid sets of labelled elements M, a subset $M' \subseteq M$ and an interval version vector V' we say that M' is M-equivalent to V' (written as $M' \equiv_M V'$)

iff $\forall m \in M, m \in M' \iff (m) \in V'[rep(m)]$. *If* $M = M'$ *then we drop the subscript* M *and write* $M' \equiv V'$.

Given a valid set of labelled elements M, *a naïve operation* o *is* M*-equivalent to an opt operation* o' *iff either* $o = o'$ *or* ($o = r.deldown(M')$, $o' = r.deldown(V')$, *and* $M' \equiv_M V'$) *or* ($o = r.merge(S)$, $o' = r.merge(S')$, *and* $S \equiv_M S'$).

The operations $r.add(e)$ and $r.del(e)$ take an element as argument and prepare the corresponding set of labelled elements to be added or deleted. This information is propagated through the network. The actual addition or deletion is handled by $r.adddown(m)$ and $r.deldown(M)$, respectively, which add or delete the labelled elements provided as an argument. Formally, the effect of these operations on naïve and opt configurations is captured through the transition relations described below.

Definition 6. *Given two naïve configurations* $C = (S_0, \ldots, S_{N-1})$ *and* $C' = (S'_0, \ldots, S'_{N-1})$ *from* $\mathcal{C}^{naive}$, *and a naïve operation* $o \in \mathcal{O}^{naive}$ *with* $Site(o) = r$, *we say that* $C \xrightarrow{o}_{\text{naive}} C'$ *iff:*

- *for* $i \neq r$, $S_i = S'_i$.
- *if* $o = r.query(e)$ *or* $o = r.add(e)$ *or* $o = r.del(e)$ *then* $E' = E$, $T = T'$, *and* $count = count'$.
- *if* $o = r.adddown(m)$ *then* $E' = E \cup \{m\}$, $T = T'$, *and* $count' = \begin{cases} (m)+1 & \text{if} \quad rep(m) = r \\ count & \text{otherwise} \end{cases}$
- *if* $o = r.deldown(M)$ *then* $E' = E \setminus M$, $T' = T \cup M$, *and* $count' = count$.
- *if* $o = r.merge(S)$ *and* $S = (E_1, T_1, count_1)$ *then* $E' = (E \setminus T_1) \cup (E_1 \setminus T)$, $T' = T \cup T_1$ *and* $count' = count$.

Definition 7. *Given two opt configurations* $C = (S_0, \ldots, S_{N-1})$ *and* $C' = (S'_0, \ldots, S'_{N-1})$ *from* $\mathcal{C}^{opt}$, *and an opt operation* $o \in \mathcal{O}^{opt}$ *with* $Site(o) = r$, *we say that* $C \xrightarrow{o}_{\text{opt}} C'$ *iff:*

- *for* $i \neq r$, $S_i = S'_i$
- *with* $S_r = (E, V, count)$ *and* $S'_r = (E', V', count')$, *the following conditions are satisfied:*
  - *if* $o = r.query(e)$ *or* $o = r.add(e)$ *or* $o = r.del(e)$ *then* $E' = E$ *and* $V' = V$.
  - *if* $o = r.adddown(m)$ *then*
    - $E' = E \cup \{m\}$,
    - $V'[rep(m)] = \boldsymbol{add}(V[rep(m)], \{ (m)\})$ *and* $V'[i] = V[i]$ *for* $i \neq rep(m)$, *and*
    - $count' = \begin{cases} (m)+1 & \text{if} \quad rep(m) = r \\ count & \text{otherwise} \end{cases}$
  - *if* $o = r.deldown(V'')$ *then*
    - $E' = E \setminus M$ *where* $M = \{m \in E \mid (m) \in V''[rep(m)]\}$.
    - *for all* $i \in Reps$: $V'[i] = V[i] \cup V''[i]$, *and*
    - $count' = count$.

- *if* $o = r.merge(S)$ *and* $S = (E_1, V_1, count_1)$ *then*
  - $E' = \{m \in E \cup E_1 \mid m \in E \cap E_1 \vee (m) \notin V[rep(m)] \cap V_1[rep(m)]\}$.
  - *for all* $i \in Reps$: $V'[i] = V[i] \cup V_1[i]$.
  - $count' = count$.

As usual, a run of the system is a sequence of configurations starting from the initial configuration that respects the transition relation. In addition, we have to ensure that each downstream operation has a corresponding prepare operation preceding it in the sequence.

Definition 8. *A naïve run (respectively, opt run) of an OR-set implementation is a sequence* $C_0 o_1 C_1 \cdots C_{n-1} o_n C_n$ *where (letting each* $C_i = (S_0^i, \ldots, S_{N-1}^i)$*):*

- *each* C_i *is a naïve (respectively, opt) configuration*
- C_0 *is an initial naïve (respectively, opt) configuration*
- *each* o_i *is a naïve (respectively, opt) operation*
- *for all* $i < n$, $C_i \xrightarrow{o_{i+1}}_{\text{naïve}} C_{i+1}$ *(respectively,* $C_i \xrightarrow{o_{i+1}}_{\text{opt}} C_{i+1}$ *)*
- *if* o_i *is an* $r.add(e)$ *operation, then* $i < n$ *and* o_{i+1} *is an* $r.adddown((e, S_r^{i-1}.count, r))$ *operation.*
- *if* o_i *is an* $r.del(e)$ *operation, then* $i < n$ *and* o_{i+1} *is a* $r.deldown(M)$ *operation for some* $M \subseteq \mathcal{M}$ *such that* $data(m) = e$ *for every* $m \in M$ *(respectively,* o_{i+1} *is a* $r.deldown(V')$ *operation where* $V' : Reps \to \mathcal{I}$ *such that for all* $m \in S_r^{i-1}.E$ *with* $data(m) = e$, $(m) \in V'[rep(m)]$*).*
- *if* o_i *is an* $r'.adddown((e, c, r))$ *operation, then there exists* $j < i$ *such that* o_j *is an* $r.add(e)$ *operation and* $c = S_r^{j-1}.count$.
- *if* o_i *is a* $r'.deldown(M)$ *operation (respectively,* $r'.deldown(V')$ *operation), then there exists* $j < i, r \in Reps$ *such that* o_j *is a* $r.del(e)$ *operation and* $M = \{m \in S_r^{j-1}.E \mid data(m) = e\}$ *(respectively,* $V' : Reps \to \mathcal{I}$ *such that for all* $m \in S_r^{j-1}.E$ *with* $data(m) = e$, $(m) \in V'[rep(m)]$*).*
- *if* o_i *is a* $r.merge(S)$ *operation, then* S *is a state in some earlier configuration* C_j.

By a run (without any qualifiers) we mean either a naïve run or an opt run. For a run $\alpha = C_0 o_1 C_1 \cdots C_{n-1} o_n C_n$, define $C(\alpha)$ to be C_n.

We can now define what it means for a naïve run to be equivalent to an opt run. We begin with the following preliminary definition of the set of labelled elements generated during a run.

Definition 9. *If* $\alpha = C_0 o_1 C_1 \ldots C_{n-1} o_n C_n$ *is a run, then* $\mathcal{M}(\alpha) = \{m \mid \exists i \leq n, r \in Reps$ *such that* o_i *is an* $r.adddown(m)$ *operation*$\}$.

Proposition 10. *For any run* α, $\mathcal{M}(\alpha)$ *is a valid set of labelled elements.*

Proof. We prove by induction on the length of α the following property (letting $C(\alpha) = (S_0, \ldots, S_{N-1})$ and $\mathcal{M}(\alpha) = M$):

$$[\forall m \in M. (m) < S_{rep(m)}.count] \wedge M \text{ is valid.}$$

The base case is a run with zero operations. In this case $M = \emptyset$ and all *count* values are zero. So the proposition is true.

Suppose now that $\alpha = \alpha' \cdot oC$ for some operation o and configuration C. Let $M' = \mathcal{M}(\alpha')$ and let $C(\alpha') = (S'_0, \ldots, S'_{N-1})$. We observe that the $M = M'$ and $S'_i.count = S_i.count$ for all $i \in Reps$, except when $o = r.adddown(m)$ with $rep(m) = r$. In this case, $M = M' \cup \{m\}$ and $(m) = S'_r.count$ and therefore $S_r.count = S'_r.count + 1$.

By induction hypothesis, for every $m' \in M'$, $(m') < S'_{rep(m')}.count$. And $(m) < S_r.count$. And hence the first part of the statement is true. Also, since every $m' \in M'$ with $rep(m) = r$, $(m') < S'_r.count = (m)$. Thus m is distinct from all other labelled elements in M. And the statement of the proposition is proved.

Definition 10. *For a naïve run $\alpha = C_0 o_1 C_1 \ldots C_{n-1} o_n C_n$ and an opt run $\alpha' = C'_0 o'_1 C'_1 \ldots C'_{n-1} o'_n C'_n$, we say that α is equivalent to α' (denoted $\alpha \equiv \alpha'$) iff for every $i \leq n$, $C_i \equiv_{\mathcal{M}(\alpha)} C'_i$ and $o_i \equiv_{\mathcal{M}(\alpha)} o'_i$.*

Remark Though it is not obvious from the definition above, the relation $\alpha \equiv \alpha'$ is symmetric. We will show that if α is equivalent to α', then $\mathcal{M}(\alpha) = \mathcal{M}(\alpha')$.

We now show that the optimized OR-set implementation behaves the same as the naïve implementation. The strategy is to set up a correspondence between the configurations of the naïve and the optimized implementations through a bisimulation.

In the following, we use $S_1 \leq_{naive} S_2$ to denote that $\textsc{compare}_{naive}(S_1, S_2)$ returns $\textsc{true}$, for two naïve states S_1 and S_2. We define $\leq_{opt}$ similarly.

Our first claim is that $\equiv_M$ guarantees query equivalence. We use the notation $S.\textsc{merge}_{naive}(S')$ to denote the resulting state of a replica r on invoking $r.\textsc{merge}_{naive}(S')$ with r in state S. $S.\textsc{merge}_{opt}(S')$ has a similar meaning.

Lemma 5. *Suppose M is a valid set of labelled elements. Suppose S_1, S_2 are M-compatible naïve states, and S'_1, S'_2 are M-compatible opt states such that $S_1 \equiv_M S'_1$ and $S_2 \equiv_M S'_2$. Then*

1. *For any $e \in \mathcal{U}$, $S_1.\textsc{query}_{\text{naive}}(e)$ returns $\textsc{true}$ iff $S'_1.\textsc{query}_{\text{opt}}(e)$ returns $\textsc{true}$.*
2. *$S_1 \leq_{\text{naive}} S_2$ iff $S'_1 \leq_{\text{opt}} S'_2$.*
3. *$S_1.\textsc{merge}_{\text{naive}}(S_2) \equiv_M S'_1.\textsc{merge}_{\text{opt}}(S'_2)$.*
4. *If $S_1.\textsc{merge}_{\text{naive}}(S_2)$ is the least upper bound of S_1 and S_2, then $S'_1.\textsc{merge}_{\text{opt}}(S'_2)$ is the least upper bound of S'_1 and S'_2.*

Proof. Let $S_i = (E_i, T_i, count_i)$ and $S'_i = (E'_i, V'_i, count'_i)$, for $i \in \{1, 2\}$. Given the assumptions, the following statements hold for $i \in \{1, 2\}$:

- $E_i = E'_i$,
- for $m \in M$, $m \in E_i \cup T_i$ iff $(m) \in V_i[rep(m)]$.
- for all $r \in Reps$, $V'_i[r] = \{\, (m) \mid m \in E_i \cup T_i, rep(m) = r\}$.
- $T_i = \{m \in M \setminus E'_i \mid (m) \in V'_i[rep(m)]\}$.

1. S_1.QUERY $_{naive}(e)$ returns TRUE iff there is $m \in E_1$ such that $data(m) = e$ iff there is $m \in E'_1$ such that $data(m) = e$ iff S'_1.QUERY $_{opt}(e)$ returns TRUE.
2. $S_1 \leq_{naive} S_2$ iff (going by the code of the naïve implementation) $E_1 \cup T_1 \subseteq E_2 \cup T_2$ and $T_1 \subseteq T_2$. The first condition is equivalent to saying that for all $r \in Reps : V'_1[r] \subseteq V'_2[r]$.
   For the second condition, note that $E_1 \cup T_1 \subseteq E_2 \cup T_2$ and $E_1 \cap T_1 = E_2 \cap T_2 = \emptyset$. Thus $T_1 \subseteq T_2$ iff $T_1 \cap E_2 = \emptyset$ iff $E_2 \setminus E_1 \cap T_1 = \emptyset$ iff $E'_2 \setminus E'_1 \cap T_1 = \emptyset$. In other words, for all $m \in E'_2 \setminus E'_1$, $m \notin T_1$. But for $m \in M$, $m \in T_1$ iff $m \notin E'_1$ and $(m) \in V'_1[rep(m)]$. So for $m \in E'_2 \setminus E'_1$, $m \notin T_1$ iff $(m) \notin V'_1[rep(m)]$. And this is precisely what COMPARE$_{opt}$ checks, as can be seen from the code in Algorithm 2. Thus $S_1 \leq_{naive} S_2$ iff $S'_1 \leq_{opt} S'_2$.
3. Suppose $S = S_1$.MERGE$_{naive}(S_2) = (E, T, count)$ and $S' = S'_1$.MERGE$_{opt}(S'_2) = (E', V', count')$. Then $count = count_1 = count'_1 = count'$. Further, $T = T_1 \cup T_2$ and $E = (E_1 \setminus T_2) \cup (E_2 \setminus T_1)$. So

   $$E \cup T = (E_1 \cup T_1) \cup (E_2 \cup T_2) = \{m \in M \mid (m) \in V'_1[rep(m)] \cup V'_2[rep(m)] = V'[rep(m)]\}.$$

   Further, since $E_i \cap T_i = \emptyset$, for $i \in \{1, 2\}$,

   $$E = [(E_1 \cup E_2) \setminus T_1] \cup [(E_1 \cup E_2) \setminus T_2] = (E_1 \cup E_2) \setminus (T_1 \cap T_2).$$

   So for $m \in M$, $m \in E$ iff

   $$(m \in E_1 \cup E_2 = E'_1 \cup E'_2) \wedge (m \in E'_1 \vee (m) \notin V'_1[rep(m)]) \wedge (m \in E'_2 \vee (m) \notin V'_2[rep(m)]).$$

   But since $m \in E'_i \implies (m) \in V'_i[rep(m)]$ for $i \in \{1, 2\}$,

   for $m \in (E'_1 \cup E'_2) \setminus E'_i$, $(m) \in V'_i[rep(m)]$ iff $(m) \in V'_1[rep(m)] \cap V'_2[rep(m)]$.

   Thus $m \in E$ iff

   $$(m \in E'_1 \cup E'_2) \wedge ((m \in E'_1 \cap E'_2) \vee (m) \notin V'_1[rep(m)] \cap V'_2[rep(m)]) .$$

   This is precisely when $m \in E'$. Thus $S \equiv_M S'$, as all the required conditions are fulfilled.
4. Suppose $S = S_1$.MERGE$_{naive}(S_1, S_2) = (E, T, count)$ is the least upper bound of S_1 and S_2, and let S' denote S'_1.MERGE$_{opt}(S'_2) = (E', V', count')$. It easily follows from part 2 that S' is an upper bound of S'_1 and S'_2. Suppose $S'_3 = (E'_3, V'_3, count'_3)$ is an upper bound of S'_1 and S'_2. From this it follows that for all $r \in Reps$, $V'_1[r] \cup V'_2[r] \subseteq V'_3[r]$, and also that for all $m \in E'_3 \setminus (E'_1 \cup E'_2)$: $(m) \notin V'_1[r] \cup V'_2[r]$. This means that if $m \in E'_3$ and $(m) \in V'_1[r] \cup V'_2[r]$ then $m \in E'_1 \cup E'_2$. Thus S'_3 does not reuse labels of elements from $E'_1 \cup E'_2$ for the new elements that are added in E'_3. Therefore one can find an appropriate valid set of labelled elements $N \supseteq M_0$, where $M_0 = E_1 \cup E_2 \cup T_1 \cup T_2$ such that all the S's and S''s are N-compatible. Further, since $N \supseteq M_0$, $S_1 \equiv_N S'_1$ and $S_2 \equiv_N S'_2$ and hence $S \equiv_N S'$. Now since S'_3 is N-compatible, there is a unique naïve state S_3 such that $S_3 \equiv_N S'_3$. It follows from part 2 that S_3 is an upper bound of S_1 and S_2. And since S is the least upper bound of S_1 and S_2, $S \leq_{naive} S_3$. And hence it again follows from part 2 that $S' \leq_{opt} S'_3$. Thus S' is the least upper bound of S'_1 and S'_2.

Our second claim is that equivalent runs reach configurations that are strongly equivalent to each other. We use the following useful observation which simplifies the proof. A bit of notation first: for a naïve (respectively, opt) state S, we define $\mathsf{incr}(S)$ to be the naïve (respectively, opt) state S' which is the same as S except that $S'.count = S.count + 1$.

Observation 4 *Let $o_1 = r.adddown(m)$, $o_2 = r.deldown(M)$, $o_2' = r.deldown(V)$. Let $T_1 = (\{m\}, \emptyset, 0)$, $T_2 = (\emptyset, M, 0)$, $T_1' = (\{m\}, V_1', 0)$, and $T_2' = (\emptyset, V_2', 0)$ where*

- *$V_1'[rep(m)] = \{ (m)\}$, and $V_1'[i] = \emptyset$ for $i \neq rep(m)$.*
- *$V_2'[i] = V[i]$ for all $i \in Reps$.*

Let $C = (S_0, \ldots, S_r, \ldots, S_{N-1})$ and $C' = (S_0, \ldots, S_r', \ldots, S_{N-1})$ be two configurations. Then

1. *If $C \xrightarrow{o_1}_{\text{naive}} C'$ then $S_r' = \begin{cases} \mathsf{incr}(S_r.\text{MERGE}_{\text{naive}}(T_1)) & \text{if} \quad rep(m) = r \\ S_r.\text{MERGE}_{\text{naive}}(T_1) & \text{otherwise} \end{cases}$*
2. *If $C \xrightarrow{o_1}_{\text{opt}} C'$ then $S_r' = \begin{cases} \mathsf{incr}(S_r.\text{MERGE}_{\text{opt}}(T_1')) & \text{if} \quad rep(m) = r \\ S_r.\text{MERGE}_{\text{opt}}(T_1') & \text{otherwise} \end{cases}$*
3. *If $C \xrightarrow{o_2}_{\text{naive}} C'$ then $S_r' = S_r.\text{MERGE}_{\text{naive}}(T_2)$.*
4. *If $C \xrightarrow{o_2'}_{\text{opt}} C'$ then $S_r' = S_r.\text{MERGE}_{\text{opt}}(T_2')$.*

Lemma 6. *Suppose α is a naïve run and α' is an opt run such that $\alpha \equiv \alpha'$. Then*

1. *$\mathcal{M}(\alpha) = \mathcal{M}(\alpha')$.*
2. *For any naïve operation o and naïve configuration C such that $\alpha \cdot oC$ is a naïve run, there exists an opt operation o' and an opt configuration C' such that $\alpha' \cdot o'C'$ is an opt run and $\alpha \cdot oC \equiv \alpha' \cdot o'C'$.*
3. *For any opt operation o' and opt configuration C' such that $\alpha' \cdot o'C'$ is an opt run, there exists a naïve operation o and a naïve configuration C such that $\alpha \cdot oC$ is a naïve run and $\alpha \cdot oC \equiv \alpha' \cdot o'C'$.*

Proof. We prove the lemma by induction on the number of operations in α (and hence in α'). Assume that the result holds for all equivalent runs α and α', each having strictly fewer than n operations. Now let $\alpha = C_0 o_1 C_1 \cdots C_{n-1} o_n C_n$ be a naive run and $\alpha' = C_0' o_1' C_1' \cdots C_{n-1}' o_n' C_n'$ be an opt run such that $\alpha \equiv \alpha'$. Let $M_n = \mathcal{M}(\alpha)$ and $M_n' = \mathcal{M}(\alpha')$. When $n > 0$, let $M_{n-1} = \mathcal{M}(C_0 o_1 C_1 \cdots C_{n-1})$ and $M_{n-1}' = \mathcal{M}(C_0' o_1' C_1' \cdots C_{n-1}')$.

1. If $n = 0$ then $M_n = M_n' = \emptyset$. Hence the statement holds. Assume that $n > 0$. Then by induction hypothesis $M_{n-1} = M_{n-1}'$. The following cases need to be considered.
   $o_n = r.\boldsymbol{query}(e)$ **or** $o_n = r.\boldsymbol{add}(e)$ **or** $o_n = r.\boldsymbol{del}(e)$**:** In this case $o_n = o_n'$, $M_n = M_{n-1}$, and $M_n' = M_{n-1}'$. Therefore $M_n = M_n'$.
   $o_n = r.\boldsymbol{adddown}(m)$**:** In this case also $o_n = o_n'$. If $rep(m) \neq r$, then $M_n = M_{n-1}$ and $M_n' = M_{n-1}'$. If $rep(m) = r$, then $M_n = M_{n-1} \cup \{m\}$ and $M_n' = M_{n-1}' \cup \{m\}$. Hence $M_n = M_n'$.

$o_n = r.\textbf{\textit{deldown}}(M)$**:** Then, $o'_n = r.deldown(V)$ such that $M \equiv_{M_n} V$. Further, $M \subseteq M_{n-1} = M'_{n-1}$. Therefore $M_n = M_{n-1}$. and importantly, $M'_n = M'_{n-1}$. Hence $M_n = M'_n$.

$o_n = r.\textbf{\textit{merge}}(S)$**:** In this case $o'_n = r.merge(S')$ such that S and S' are states from configurations C_i and C'_j respectively, $i, j < n$. Thus both the states are M_{n-1}-compatible. But then $M_n = M_{n-1}$ and $M'_n = M'_{n-1}$. Hence $M_n = M'_n$.

2. Let o and C be a naïve operation and configuration, respectively, such that $\alpha \cdot oC$ is a run. Let $M = \mathcal{M}(\alpha \cdot oC)$. Note that since $C_n \equiv_{M_n} C'_n$, it is also the case $C_n \equiv_M C'_n$. We aim to show that there is an opt operation o' and an opt configuration C' such that $\alpha' \cdot o'C'$ is a run and $\alpha \cdot oC \equiv \alpha' \cdot o'C'$. Let $Site(o) = r$. Then the only state change happens in replica r. We denote the state of r in C_n and C'_n by S_1 and S'_1 respectively and the state of r in C and C' by S_2 and S'_2 respectively.

   The following cases need to be considered.

   $o = r.\textbf{\textit{query}}(e)$ **or** $o = r.\textbf{\textit{add}}(e)$ **or** $o = r.\textbf{\textit{del}}(e)$**:** In this case $C = C_n$. We choose o' to be o and $C' = C'_n$. From the fact that $\alpha \cdot oC$ is a naïve run, it easily follows that $\alpha' \cdot o'C'$ is an opt run. Furthermore, $o \equiv_M o'$ and $C = C_n \equiv_M C'_n = C'$. Therefore $\alpha \cdot oC \equiv \alpha' \cdot o'C'$.

   $o = r.\textbf{\textit{adddown}}(m)$**:** In this case, either $M = M_n$ or $M = M_n \cup \{m\}$. We choose o' to be o and C' such that $C'_n \xrightarrow{o'}_{opt} C'$. Since $\alpha \cdot oC$ is a naïve run, $\alpha' \cdot o'C'$ is an opt run. By the observation preceding this lemma, there is a naïve state S and an opt state S' such that $S \equiv_M S'$, $S_2 = S_1.\text{MERGE}_{naive}(S)$, and $S'_2 = S'_1.\text{MERGE}_{opt}(S')$. Thus it follows that $S_2 \equiv_M S'_2$ (from part 3 of Lemma 5). Therefore $C_n \equiv_M C'_n$ and hence $\alpha \cdot oC \equiv \alpha' \cdot o'C'$.

   $o_n = r.\textbf{\textit{deldown}}(M')$**:** In this case, $M = M_n$. Choose o' to be $r.deldown(V')$ with $M' \equiv_M V'$ and C' such that $C'_n \xrightarrow{o'}_{opt} C'$. Since $\alpha \cdot oC$ is a naïve run, $\alpha' \cdot o'C'$ is an opt run. By the observation preceding this lemma, there is a naïve state S and an opt state S' such that $S \equiv_M S'$, $S_2 = S_1.\text{MERGE}_{naive}(S)$, and $S'_2 = S'_1.\text{MERGE}_{opt}(S')$. Thus it follows that $S_2 \equiv_M S'_2$ (from part 3 of Lemma 5). Therefore $C_n \equiv_M C'_n$ and hence $\alpha \cdot oC \equiv \alpha' \cdot o'C'$.

   $o_n = r.\textbf{\textit{merge}}(S)$**:** In this case too, $M = M_n$. Since $\alpha \cdot oC$ is a naïve run, there exists a replica r' and an index $i \leq n$ such that S is the local state of r' in C_i. Choose $o' = r.merge(S')$ where S' is the local state of r' in C'_i, and choose C' such that $C'_n \xrightarrow{o'}_{opt} C'$. It is easily seen that $\alpha' \cdot o'C'$ is an opt run. Since $\alpha \equiv \alpha'$, $S \equiv_M S'$ and hence $o \equiv_M o'$. Since $S \equiv_M S'$, $S_2 = S_1.\text{MERGE}_{naive}(S)$, and $S'_2 = S'_1.\text{MERGE}_{opt}(S')$, it follows that $S_2 \equiv_M S'_2$ (from part 3 of Lemma 5). Therefore $C \equiv_M C'$ and hence $\alpha \cdot oC \equiv \alpha' \cdot o'C'$.

3. Similar to the proof of previous item by swapping the roles of o and o', α and α', and C and C'.

We can now describe the correspondence we seek between naïve runs and opt runs. We match a naïve configuration C with an opt configuration C' if they can be reached by equivalent runs

Definition 11. *Let $\mathcal{B}$ be a binary relation on $\mathcal{C}^{naive} \times \mathcal{C}^{opt}$ defined by*

$$\mathcal{B} \stackrel{def}{=} \{(C, C') \mid \exists \text{ a naïve run } \alpha \text{ and an opt run } \alpha' \text{ such that } C = C(\alpha), C' = C(\alpha'), \alpha \equiv \alpha'\}.$$

Lemma 7. *$\mathcal{B}$ is a nontrivial bisimulation.*

Proof. Follows from Lemmas 5, 6, and the fact that $(C_0, C'_0) \in \mathcal{B}$, where C_0 and C'_0 are the initial naïve and initial opt configurations, respectively.

Having established a bisimulation between the two systems, we can assert that our optimized implementation of OR-Sets inherits all the properties that have already been established for the naïve implementation in [5].

8 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 [11] is based on the solution to the *gossip problem* for synchronous communication [12], which has also been generalized to message-passing systems [13]. 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.: On bounded version vectors. Technical report, Chennai Mathematical Institute (2012) `http://www.cmi.ac.in/~gautshen/pubs/BVV/on_bounded_version_vectors.pdf`.

[12] Mukund, M., Sohoni, M.A.: Keeping track of the latest gossip in a distributed system. Distributed Computing **10**(3) (1997) 137–148

[13] Mukund, M., Narayan Kumar, K., Sohoni, M.A.: Bounded time-stamping in message-passing systems. Theor. Comput. Sci. **290**(1) (2003) 221–239