Optimistic Operations for Replicated Abstract Data Types

Hyun-Gul Roh[*]
KAIST

Jin-Soo Kim[†] Joonwon Lee[‡]
Sungkyunkwan University

Seungryoul Maeng[§]
KAIST

CS/TR-2009-318

September 30, 2009

K A I S T
Department of Computer Science

[*] hgroh@camars.kaist.ac.kr
[†] jinsookim@skku.edu
[‡] joonwon@skku.edu
[§] maeng@camars.kaist.ac.kr

Optimistic Operations for Replicated Abstract Data Types

Hyun-Gul Roh[a,1], Jin-Soo Kim[b], Joonwon Lee[b], Seungryoul Maeng[a]

[a]*Department of Computer Science, KAIST, Daejeon 305-701, Republic of Korea*
[b]*School of Information and Communication Engineering, Sungkyunkwan University (SKKU), Suwon, Republic of Korea*

Abstract

For distributed applications requiring collaboration, responsive and transparent interactivity is highly desired. Though such interactivity can be achieved with optimistic replication, maintaining consistency among replicas is troublesome. To support efficient implementations of collaborative applications, this paper extends a few representative abstract data types (ADTs), such as array, hash table, and growable array (or linked list), into replicated abstract data types (RADTs). In RADTs, a shared ADT is replicated and modified with optimistic operations, allowing each site to execute operations in a different order. Operation commutativity and precedence transitivity are two theoretical principles that enable RADTs to maintain consistency for optimistic operations, especially for the insertion/deletion/update operations into replicated growable arrays (RGAs). RGAs comply with the two principles and show significant improvement in performance, scalability, and reliability over the effects of the OT methods.

Key words: Distributed data structures, Optimistic replication, Replicated abstract data types, Optimistic algorithm, Collaboration

1. Introduction

Optimistic replication is an essential technique for collaborative applications [1, 2]. This paper deals with consistency issues of the optimistic replication that supports implementations of interactive collaborative applications. To illustrate replication issues, assume the following scenario in an editorial office publishing a daily newspaper.

> *To meet a deadline, a number of newsmen are editing a newspaper using computerized editing tools. For efficient overall editing, they can concurrently see all pages consisting of news items such as text, pictures, and tables. Even when both a writer and a photographer collaborate on editing the same article with their own PCs, each should be allowed to modify the article as if working with a local editing tool. Obviously, after their interactions, all the copies of the newspaper must be eventually consistent.*

Human users, the subjects of these applications prefer high responsiveness and transparent interactivity to strict consistency [1, 2, 3]. Responsiveness is how quickly the effect of an operation is delivered to users, and interactivity means how freely operations can perform on replicas. Replication with optimistic operations can enhance responsiveness and interactivity by allowing local and remote operations to be executed immediately, but may lead to inconsistency among replicas since sites executes operations in different orders.

This optimism makes pessimistic concurrency control protocols, such as serialization [4, 5] or locking [6, 7], incompatible with the optimistic replication [2]. Even if some global locking protocols allow optimistic operations [3], they not only require state rollback mechanisms, but also damage interactivity due to the restrictive nature of the locking protocol. There have been various researches for genuine optimistic replications, which were oriented to specific services, such as replicated database [8, 9], Usenet [10, 11, 12], and collaborative textual or graphical editors [1, 13, 14, 15]. However, they are time-consuming to modify or might be inflexible for complicated and various functions of up-to-date editing applications, such as Microsoft Office and Google Docs.

In the sense that developers generally make use of various abstract data types (ADTs) according to the characteristics of data, we suggest *replicated abstract data types* (RADTs), a novel type of ADTs that can be used as building blocks for collaborative applications. RADTs are multiple copies of a shared ADT replicated over distributed sites. They provide a set of primitive operation types corresponding to that of normal ADTs, concealing details for consistency maintenance. RADTs ensure *eventual consistency* [2], a weaker model for achieving responsiveness and interactivity. To accommodate RADT deployment in general environments, operation delivery in RADTs is free from any constraint except causal dependency. Though this allows sites to execute remote operations immediately, *sites may execute operations in different orders*. We model such executions and explore principles to achieve consistency in the model.

This paper suggests two principles that lead to successful designs of complicated RADT operations. First, *operation commutativity* (OC) is a condition that every pair of concurrent operations commutes. Though the concept of commutativity was discussed in many distributed systems [16, 17, 13], it was not fully assimilated. We formally prove that OC guarantees eventual consistency for all possible execution orders, so *mandate*

URL: `hgroh@calab.kaist.ac.kr` (Hyun-Gul Roh)

RADT operations to satisfy OC. Second, *precedence transitivity* (PT) is a guideline on how to design remote operations in order to make operations commute and preserve their intentions. RADTs require precedence rules to reconcile the conflicting intentions. PT is a condition that all precedence rules are transitive. PT explains how concurrent relations have to be designed against happened-before relations while OC is a general principle only on concurrent operations.

With causality preservation, OC and PT constitute the theoretical part of the RADT framework. Technically, efficient implementations of three RADTs are presented: replicated fixed-size array (RFA), replicated hash table (RHT), and replicated growable array (RGA). Although key ideas for RFAs and RHTs were already presented, we introduce them because they exemplify the concepts of RADTs, and above all, because their problems and ideas are inherited by RGAs.

RGAs are another main contribution of this paper, which solves the time-worn problem on *optimistic insertions and deletions* into a replicated ordered list. Though these operations have been highly desired in collaborative applications [1, 3], the operational transformation (OT) framework, which is a mainstream approach for these operations [18, 13, 19, 14, 20, 21], has difficulty in verifying algorithms based on their correctness properties. Besides, the recent study reports poor performance and scalability of OT methods are non-negligible [22]. Thanks to OC and PT, RGAs provide full correctness verification for not only the two operations but also the update operation. Above all, RGAs show better performance than previous works dealing with the insertion and deletion. Especially, compared to the time complexity of the remote operations of OT methods, which is generally quadratic, RGAs provide the remote operations of $O(1)$ by proposing the s4vector index (SVI) scheme. Due to these optimal remote operations and the fixed-size s4vectors, RGAs scale. Moreover, the SVI scheme can enhance reliability by autonomous causality validations.

This paper is organized as follows. The next section defines three RADTs and describes their inconsistency problems. Sections 3 and 4 formalize OC and PT, respectively. Section 5 proposes concrete algorithms of three RADTs. We introduce the related work in Section 6, and contrast RGAs with the previous works in complexity, scalability, and reliability in Section 7. Section 8 concludes this paper.

2. Problem Definition

2.1. Preliminary: Causality preservation among operations

The replication system discussed in this paper can be characterized by a set of distributed sites and operations as shown in the time-space diagram of Fig. 1 which describes the propagation and execution of operations. Lamport presented two definitions for the causality relationship between two distinct operations [23]: happened-before relation ('$\rightarrow$') and concurrent relation ('$\|$'). Note that, given a time-space diagram consisting of n operations, all $\frac{n(n-1)}{2}$ relations are obtained because every pair of distinct operations is in either of the two relations.

While no uniquely correct order is defined for concurrent operations, partial orders defined by happened-before relations

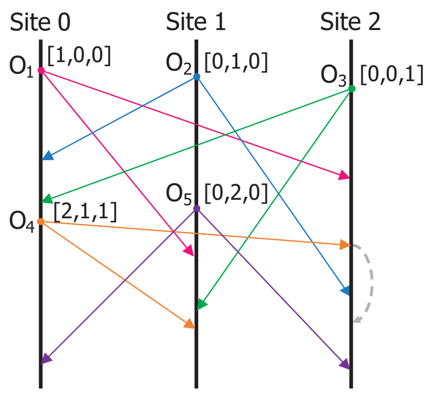


Figure 1: A time-space diagram in which three sites participate. A vector on the right of each operation is its vector clock.

Algorithm 1 The main control loop of RADTs at site i

```
1  MainLoop():
2     ∀k: $\vec{v}_i[k]$ := 0;
3     i := this site ID;
4     initialize queue Q;
5     initialize RADT;
6     while(not aborted)
7        if(O is a local operation and not Read)
8           $\vec{v}_i[i] := \vec{v}_i[i] + 1$;
9           tf := RADT.localAlgorithm(O);
10          if(tf = true) broadcast (O, $\vec{v}_i$, i);
11          else $\vec{v}_i[i] := \vec{v}_i[i] - 1$;
12       if(O is a local Read) RADT.localAlgorithm(O);
13       if(an operation O arrives with $\vec{v}_O$ from site j)
14          enqueue the set (O, $\vec{v}_O$, j) into Q;
15          while(there is a causally ready set in Q)
16             (O, $\vec{v}_O$, j) := dequeue the set from Q;
17             ∀k: $\vec{v}_i[k]$ := max($\vec{v}_i[k], \vec{v}_O[k]$);
18             RADT.remoteAlgorithm(O);
```

need preserving at every site [13, 19]. The reason is owing to causality that might exist between operations; e.g., imagine O_4 is to delete the object inserted by O_2 in Fig. 1. Causality has been detected and preserved by vector clock timestamping [4, 19]. Following Birman et al.'s CBCAST scheme [4], in RADTs of N sites, site i updates its own N-tuple vector clock $\vec{v}_i$ as lines 2, 8, 11, and 17 in Algorithm 1. To preserve causality, causally unready operations are delayed using a queue (lines 13–15). When an operation O issued at site j ($i \neq j$) arrives with its timestamp $\vec{v}_O$, O is causally ready if $\vec{v}_O[j] = \vec{v}_i[j] + 1$ and $\vec{v}_O[k] \leq \vec{v}_i[k]$ for $0 \leq k \leq N-1$. To illustrate, consider site 2 in Fig. 1. After O_1's execution, site 2 has $\vec{v}_2$=[1,0,1]. When O_4 arrives at site 2 with $\vec{v}_{O_4}$=[2,1,1], it is causally unready; thus, it is delayed until executing O_2. According to Birman et al.[24], CBCAST is 3 to 5 times faster than ABCAST that supports the total ordering. Nevertheless, this causality preservation scheme is so strict that it might incur a chain of inessential delays when a site fails to broadcast operations; in Section 7, we discuss relaxing of this scheme.

2.2. System model of RADTs

A replicated abstract data type (RADT) is extended from a normal ADT. The system model of RADTs can be summarized below, and the main control loop is presented in Algorithm 1.

- An RADT is a particular data structure with a definite set of operation types (OPTYPE).
- RADTs are multiple copies of an RADT, each of which is replicated at one of the distributed sites.
- At a site, a local operation is the one issued locally while a remote operation is the one received from a remote site.
- At a site, every local operation is immediately executed on the RADT of the site according to the local algorithm of its operation type.
- Every local operation modifying its local RADT is broadcast to the other sites in the form of the remote operation.
- At a site, every remote operation is immediately executed according to the remote algorithm of the operation type if it is causally ready by its vector clock.

For the operation types modifying RADTs, two kinds of algorithms are given: local and remote. In RADTs, *consistency is maintained by remote algorithms* while local algorithms are almost the same as those of the corresponding normal ADT. Since every operation is executed first at its local site, different sites execute operations in different orders. Section 3 will go into detail on the features of operation executions.

On the other hand, though local *Read* operations are allowed without restriction, they are not propagated to remote sites. A *Read* issued at a site, therefore, never globally performs. Even if every site executes a *Read* on the same location at the exact same time, sites might read different objects. Thus, consistency is not defined for *Read*s. Instead, RADTs guarantee an *eventual consistency model* which is defined only for the operation types modifying replica states as follows.

DEFINITION 1. *Eventual consistency*
All copies of RADT data structure eventually converge only when every RADT has executed the same set of modifying operations from the same initial data structure.

For scientific calculations, this model may lead to undesired results. As in the scenario of editing newspapers, however, if consistency of shared objects, such as pictures or textual strings, displayed to human users, accords with this model, momentary inconsistency of human users' views is acceptable [18, 1, 3]. In particular, eventual consistency is most suitable for achieving high responsiveness and transparent interactivity since it allows sites to execute operations in different orders.

2.3. Definitions of RADTs and inconsistency problems

This paper chooses three kinds of representative ADTs and extends them into RADTs: fixed-size array, hash table, and growable array (or linked list). Their functionality is prevalently demanded in such applications as in the newspaper editing scenario. For example, a page of newspapers might be divided into a fixed number of blocks, which can be managed by a fixed-size array. A hash table enables news items to be rapidly accessed with unique keys. A page needs inserting into or deleting from a growable array of pages, respecting the order of existing pages. Three types of RADTs can fulfil such functionality with the local algorithms of given operation types. However, if remote algorithms are not properly designed, RADTs suffer pathological inconsistency problems. Below, we define each RADT and show its potential inconsistency problems.

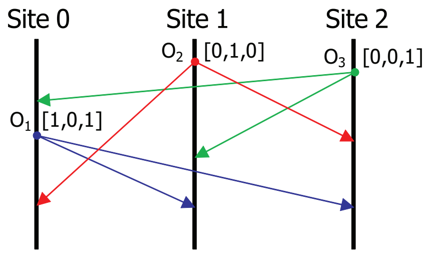


Figure 2: A simple example of a time-space diagram.

A replicated fixed-size array (RFA) is an array of a fixed number of elements with OPTYPE={*Write*, *Read*}. In ALGORITHM 2, their local algorithms are presented. An element is an object container of RFAs. A local *Write*(int i, Object o) replaces the object at the ith element of the array with the new object o. In RFAs, different execution orders lead to inconsistency. For example, if three operations of Fig. 2 are given as O_1: *Write*(1, o_1), O_2: *Write*(1, o_2), and O_3: *Write*(1, o_3) in RFAs, the element of index 1 lastly contains o_1 at sites 1 and 2, but o_2 at site 0.

Algorithm 2 The local algorithms for RFA operation types

```
Write(int i, Object o):
    if(RFA[i] exists)     //RFA[i]: the ith element
        RFA[i].obj := o;  //replaces the ith object with o
        return true;
    else return false;
Read(int i):
    if(RFA[i] exists) return RFA[i].obj;
    else return nil;
```

Hash tables are extended into replicated hash tables (RHTs), which access shared objects in slots by hashing unique keys with OPTYPE={*Put*, *Remove*, *Read*}, as shown in ALGORITHM 3. This paper assumes that an RHT resolves key collisions by separate chaining scheme. If a *Put* performs on an existing slot, it updates the slot with its new object. RHTs have an additional source of inconsistency because *Put*s and *Remove*s dynamically create and destroy slots. This necessitates the idea of tombstones, which are invisible object container left behind after *Remove* [10, 11, 12]. Despite the tombstone, if the remote algorithms are the same as the local ones, RHTs might diverge. Consider the scenario of Fig. 2 again, where the operations are given as O_1: *Remove*(k_1), O_2: *Put*(k_1, o_2), and O_3: *Put*(k_1, o_3). Having executed the two *Put*s, sites 1 and 2 have different objects for k_1. After executing all the three, sites 1 and 2 have the tombstone for k_1 while site 0 has o_2 for k_1.

A replicate growable array (RGA) is of main concern to this paper. An RGA supports OPTYPE={*Insert*, *Delete*, *Update*, *Read*}, each of which accesses an object with *an integer index*. The local algorithms of RGAs are presented in ALGORITHM 4.

Algorithm 3 The local algorithms for RHT operation types

```
Put(Key k, Object o):
    s := RHT[hash(k)]; // RHT[hash(k)]: the slot where k is mapped;
    if(s != nil) s.obj := o;   // if slot exists;
    else  new_s := make a new slot;
          new_s.obj := o;
          RHT[hash(k)] := new_s; // link new_s to RHT;
    return true;
Remove(Key k):
    s := RHT[hash(k)];
    if(s = nil) return false; // if no slot exists,
    s.obj := nil;             // make s tombstone;
    return true;
Read(Key k):
    s := RHT[hash(k)];
    if(s = nil or s.obj = nil) return nil; // no slot or tombstone
    return s.obj;
```

Since nodes, the object containers, are frequently inserted and deleted, an RGA adopts a linked list internally for efficiency. Explicit *Update* is also required because *Insert* cannot update a node and modifications should be propagated. RGAs, therefore, inherit all the problems of RFAs and RHTs. In order to enhance user interactions, such as carets or cursors, it is also possible to supplement the OPTYPE with the linked list operations. They are parameterized with node pointers instead of integers and use *findlink* in ALGORITHM 4. This paper, however, deals with the operations of integer indices because their semantics have been more frequently studied in collaborative applications [18, 13, 19, 14, 20, 21]. Note that, due to *findlist* and *findlink*, the local RGA operations are not allowed on tombstones and not propagated to remote sites in ALGORITHM 1.

Since *the order among nodes matters*, RGAs have additional inconsistency problems. Suppose that the operations in Fig. 2 are given as O_1: *Update*(2, o_x), O_2: *Insert*(1, o_y), and O_3: *Insert*(1, o_z) and that they are executed on an initial RGA $[o_1 o_2]$ by the local algorithms.[1] After executing both O_2 and O_3, two results are available according to the execution order: $[o_1 \boldsymbol{o}_z \boldsymbol{o}_y o_2]$ at site 1, and $[o_1 \boldsymbol{o}_y \boldsymbol{o}_z o_2]$ at site 2. Though both results are not wrong, only one must be chosen for consistency.

When executing O_1, its remote sites might violate the intention of O_1, which is what O_1 intends to do at its local site. We formally define the intention of an operation as follows.

DEFINITION 2. *Intention of an operation*
Given an operation on an RADT with parameter(s), its intention is the effect of the local algorithm of its type on the RADT.

Obviously, the local algorithms accomplish all the intentions of local operations. In RGAs, intentions can be violated at remote sites because *Insert*s or *Delete*s change integer indices of some nodes located behind their target nodes. This intention violation problem was first addressed by Sun et al. [19]. In the example, though O_1 intends to replace o_z on $[o_1 \boldsymbol{o}_z o_2]$ with o_x at its local site, O_1 of site 2 may update o_y on $[o_1 \boldsymbol{o}_y o_z o_2]$, which is not the intention of O_1. The operations of RGAs incurs many other puzzling problems [20]. We show that RGAs can solve them with concrete examples in Section 5.

[1]The first object is referred to by 1. An *Insert* adds a new node next to its reference. To insert at the head, we use *Insert*(0, o_x).

Algorithm 4 The local algorithms for RGA operation types

```
findlist(int i):
    n := head of the linked list;
    int k := 0;
    while(k < i)
        n := n.link;              // next node in the linked list;
        if(n = tail) return nil;  // i is wrong;
        if(n.obj != nil) k++;     // skip tombstones;
    return n;
findlink(node n):
    if(n.obj = nil) return nil;   // if n is tombstone;
    else return n;
Insert(int i, Object o):
    if((refer_n := findlist(i)) = nil) return false;
    new_n := make a new node;
    new_n.obj := o;
    link new_n next to refer_n in the RGA structure;
    return true;
Delete(int i):
    if((target_n := findlist(i)) = nil) return false;
    target_n.obj := nil;         // make target_n tombstone;
    return true;
Update(int i, Object o):
    if((target_n := findlist(i)) = nil) return false;
    target_n.obj := o;
    return true;
Read(int i):
    if((target_n := findlist(i)) = nil) return nil;
    if(target_n = tail) return nil;
    return target_n.obj;
```

Note that local ALGORITHMS 2–4 are *incomplete* since we do not present exact details of the data structures yet. However, the local RADT algorithms ensure the same responsiveness and interactivity as the corresponding normal ADTs. After introducing the two principles, the remote algorithms, which mandate consistency maintenance, will be presented with the details of the data structure in Section 5.

3. Operation Commutativity

RADTs allow different sites to execute operations in different orders. To denote an execution order between two operations, '$\mapsto$' is used; e.g., $O_a \mapsto O_b$ if O_a is executed before O_b. In addition, we use '$\Rightarrow$' to express changes of replica states caused by the execution of an operation or a sequence of operations; e.g., $RS_0 \overset{O_a}{\Rightarrow} RS_1 \overset{O_b}{\Rightarrow} RS_2$ means O_a and O_b change a replica state RS_0 into RS_1 and RS_2 one after the other and can be shorten by $RS_0 \overset{O_a \mapsto O_b}{\Rightarrow} RS_2$. Though time-space diagrams are intuitive and illustrative, we present a better definition for formal analysis as follows.

DEFINITION 3. *Causally executable graph (CEG)*
Given a time-space diagram TS, a graph G=(V, E) is a causally executable graph, iff*: V is a set of vertices corresponding to all the operations in TS, and $E \subset V \times V$ is a set of edges corresponding to all the relations between every pair of distinct operations in V, where a happened-before relation $O_a \rightarrow O_b$ corresponds to a directed edge in E from O_a to O_b, and a concurrent relation to an undirected edge in E, respectively.*

Fig. 3 shows the CEG obtained from the time-space diagram in Fig. 1. Every CEG essentially has the following properties.

LEMMA 1. *A CEG G has no cycle with its directed edges and is a complete graph.*

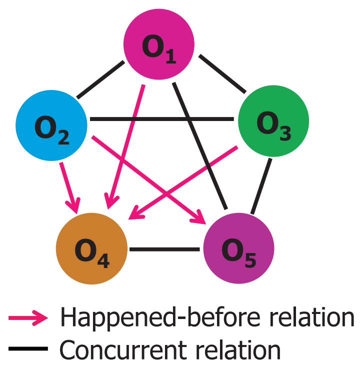


Figure 3: CEG of the time-space diagram of Fig. 1

Proof. According to the definitions of happened-before and concurrent relations [23], they are not defined reflexively and happened-before relations are all transitive; thus, G has no cycle. Unless every pair of two distinct operations is in happened-before relation, it is concurrent; hence, G is complete.

For a given CEG, if all the vertices can be traveled without going against directed edges, casuality can be preserved in the execution sequence. In Fig. 3, at site 0, the execution sequence of $O_1 \mapsto O_2 \mapsto O_3 \mapsto O_4 \mapsto O_5$ does not go against the direction of any directed edges, but at site 2, $O_3 \mapsto O_1 \mapsto O_4 \mapsto O_2 \mapsto O_5$ might violate causality because O_4 is executed before O_2, whose order is the reverse of edge $O_2 \rightarrow O_4$. A causality-preservable sequence encompassing all the operations of a CEG satisfies the conditions in the following definition.

Definition 4. *Causally executable sequence (CES)*
Given a CEG G=(V, E), where $|V|=n$*, an execution sequence* $\mathbf{s}$*:* $O_1 \mapsto \ldots \mapsto O_n$ *is a* causally executable sequence*,* iff*: all the operations in V participate only once in* $\mathbf{s}$*, and* not $O_j \rightarrow O_i$ *for* $1 \leq i < j \leq n$*.*

Unless all the edges in E are directed ones, a CEG has more than one CES. A set of all possible CESes is defined as follows.

Definition 5. *Causally executable sequence set (CESS)*
Given a CEG G=(V, E), a set of CESes is a causally executable sequence set *of G, denoted as* $S(G)$*,* iff: $S(G)$ *contains* all the possible *CESes derived from G.*

According to the system model, RADTs permit the execution of every CES in a CESS. Eventual consistency, therefore, can be achieved if all the CESes produce the same replica state. To observe the relationship among CESes of a CEG, consider a CES $\mathbf{s}_1$: $O_1 \mapsto O_2 \mapsto O_3 \mapsto O_4 \mapsto O_5$ in the CEG of Fig. 3. If a pair of adjacent operations on $\mathbf{s}_1$ is concurrent, another CES can be derived by swapping the order of the pair. If the execution order of $O_1 \| O_2$ is swapped, another CES $\mathbf{s}_2$: $O_2 \mapsto O_1 \mapsto O_3 \mapsto O_4 \mapsto O_5$ can be obtained. Only if both $O_1 \mapsto O_2$ and $O_2 \mapsto O_1$ yield the same result from an identical replica state, will $\mathbf{s}_1$ and $\mathbf{s}_2$ produce a consistent result. In this regard, given a CES $\mathbf{s}$ from a CEG G, if we show that the other CESes in $S(G)$ can be derived from $\mathbf{s}$ and find the condition that they yield the same result, eventual consistency can be ensured. This is the basic concept of operation commutativity (OC), which is developed from commutative relation as follows:

Definition 6. *Commutative relation '$\leftrightarrow$'*
Given two concurrent operations O_a *and* O_b*, they are in* commutative relation*, expressed as* $O_a \leftrightarrow O_b$*,* iff*: for any identical replica state* RS_0*, when* $RS_0 \overset{O_a \mapsto O_b}{\Rightarrow} RS_1$ *and* $RS_0 \overset{O_b \mapsto O_a}{\Rightarrow} RS_2$*,* RS_1 *is equal to* RS_2 *(*$RS_1 = RS_2$*).*

To illustrate the effect of a commutative relation in CESes, consider two CESes in Fig. 1: $\mathbf{s}_1$: $\boldsymbol{O_1} \mapsto O_2 \mapsto O_3 \mapsto O_4 \mapsto \boldsymbol{O_5}$ at site 0 and $\mathbf{s}_3$: $O_2 \mapsto \boldsymbol{O_5} \mapsto \boldsymbol{O_1} \mapsto O_3 \mapsto O_4$ at site 1. Even though $O_1 \| O_5$ is $O_1 \leftrightarrow O_5$, we are not sure if this commutative relation helps $\mathbf{s}_1$ and $\mathbf{s}_3$ to be consistent because the initial states of $O_1 \| O_5$ are different and because other operations may or may not intervene between them. Indeed, to make all the CESes of a CEG consistent, the following condition is necessary.

Definition 7. *Operation commutativity (OC)*
Given a CEG G=(V, E), operation commutativity *is established in G,* iff*:* $O_a \leftrightarrow O_b$ *for* $\forall (O_a \| O_b) \in E$*.*

OC is the condition in which every pair of concurrent operations commutes. For example, consider $\mathbf{s}_1$ and $\mathbf{s}_3$ again. If OC holds in the CEG of Fig. 3, $O_1 \leftrightarrow O_2$, $O_4 \leftrightarrow O_5$, $O_3 \leftrightarrow O_5$, and $O_1 \leftrightarrow O_5$ are ensured. By applying the properties of those commutative relations in sequence, $\mathbf{s}_1$ can be transformed into $\mathbf{s}_3$. For completeness, we present the following theorem and prove it in *Appendix* A:

Theorem 1. *If OC holds in a given CEG G=(V, E), all the CESes in* $S(G)$ *executed on* RS_0 *eventually produce the same state.*

This theorem implies that OC is a sufficient condition for eventual consistency. We, therefore, mandate every pair of operation types to be commutative when they are concurrent. Besides, OC will be used as a proof methodology. To prove if a kind of RADTs is consistent or not, it is sufficient to show that each pair of concurrent operations actually commutes on all the replica states defined exhaustively. In this way, we proved the correctness of each kind of RADTs in *Appendix* B. However, OC suggests no guideline to achieve itself. In the next section, precedence transitivity explains how to make concurrent operations commute against happened before operations.

4. Precedence Transitivity

In RADTs, operations relate to object containers, i.e., elements, slots, and nodes. The relation would be clarified by the following two definitions.

Definition 8. *Causal object for an operation (cobject)*
Given an operation O modifying RADTs, an object container is a causal object (cobject) *for O, if it has existed in its local RADT at the issued time of O and affects the execution of the local and remote algorithms of O.*

Definition 9. *Effective operation on a container (eoperation)*
An operation is an effective operation (eoperation) *on an object container, if its local or remote algorithm succeeds in creating/destroying/updating the container.*

Except *Insert* type, a local operation executed on an existing object container adopts the container as its cobject (cf. a *Put* on no slot has no cobject) while it becomes an eoperation on the cobject. In the case of an *Insert*, the node referred to by its local *Insert* and the next one are the cobjects, but the *Insert* becomes an eoperation only on its new node. Using these two definitions, intention preservation at a remote site can be defined as follows.

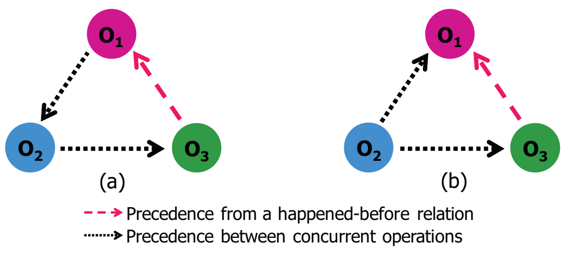
Definition 10. *Intention preservation at a remote site*
The intention of an operation is preserved at a remote site, iff *(1) for Write, Put, Remove, Delete, and Update types, the operation becomes an eoperation on its cobject or on the slot of its key (for Put on no slot), or (2) for Insert type, its new node is placed between its two cobjects.*

If different operations are supposed to be eoperations on a common object container or their cobjects overlap, their intentions might be in conflict; thus, a precedence rule has to determine the effect of an operation in the remote algorithms. Enacting precedence rules is, however, complicated since the rules should not conflict with each other. We, therefore, suggest another principle, *precedence transitivity* that makes precedence rules consistent with each other. Initially, the precedence relation is defined as an order between two operations as follows.

Definition 11. *Precedence relation '$\dashrightarrow$'*
Given two operations O_a and O_b, O_b takes precedence over O_a, denoted as $O_a \dashrightarrow O_b$, iff*: (1) $O_a \rightarrow O_b$ or (2) for $O_a \parallel O_b$, the intention of O_b is more preservable than that of O_a.*

For $O_a \rightarrow O_b$, it is evident that the intention of O_b should be preserved even if that of O_a is impeded or canceled; hence, $O_a \dashrightarrow O_b$. When the intentions of concurrent operations conflict, they need to be reconciled. For example, suppose $O_a \dashrightarrow O_b$ for $O_a \parallel O_b$. If they are both *Write*s on the same element, O_b overwrites the element where O_a has performed, but O_a does nothing on the element where O_b has been executed in order for the intention of O_b to be more preservable than that of O_a. If they are both *Insert*s next to the same node, O_b should insert its new node closer to their reference node than O_a because that makes the effect similar to the effect of $O_a \dashrightarrow O_b$ derived from $O_a \rightarrow O_b$. As a matter of course, the intentions of no conflict are preserved at once.

Precedence relations for a set of operations should be carefully enacted so that they will not conflict with each other. To illustrate, suppose the operations in Fig. 2 are given as O_1: *Write*$(1, o_1)$, O_2: *Write*$(1, o_2)$, and O_3: *Write*$(1, o_3)$. For each pair of operations, assume the following precedence relations: $O_1 \dashrightarrow O_2$, $O_3 \dashrightarrow O_1$ (from $O_3 \rightarrow O_1$), and $O_2 \dashrightarrow O_3$. These precedence relations on an element are expressed with a graph called *precedence relation graph* (PRG). A PRG can be derived from a CEG by keeping the directed edges intact and by choosing a direction for each undirected edge. Such a directed complete graph is called a tournament in graph theory [25]. The PRG of the above precedence relations is shown in Fig. 4-(a). Assuming that three operations are executed according to this PRG, the element of index 1 at each site will be as follows.



O1
O2
O3
(a)
O1
O2
O3
(b)
Precedence from a happened-before relation
Precedence between concurrent operations

Figure 4: Two PRGs of the time-space diagram of Fig. 2

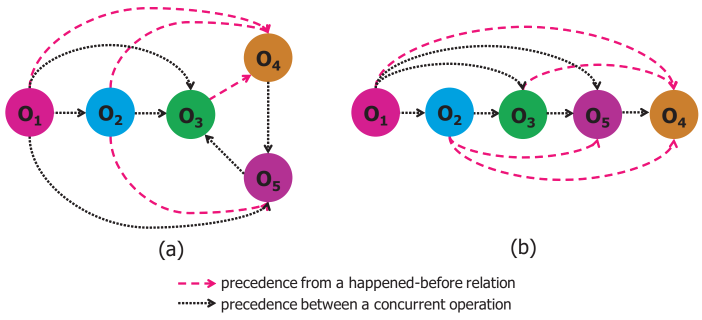
$$\text{site 0: } o_? \overset{O_3}{\Rightarrow} o_3 \overset{O_1}{\Rightarrow} o_1 \overset{O_2}{\Rightarrow} o_x,$$
$$\text{site 1: } o_? \overset{O_2}{\Rightarrow} o_2 \overset{O_3}{\Rightarrow} o_3 \overset{O_1}{\Rightarrow} o_y,$$
$$\text{site 2: } o_? \overset{O_3}{\Rightarrow} o_3 \overset{O_2}{\Rightarrow} o_3 \overset{O_1}{\Rightarrow} o_z,$$

The first operations of sites 1 and 2 are local ones, which are effectively executed by the local algorithms; i.e., O_2 and O_3 become the eoperations on RFA[1], respectively. At site 0, we assume that the remote operation O_3 is effectively executed. At each site, the second operation is effectively executed if it takes precedence over the first one, otherwise it does nothing. Thus, the elements of index 1 become o_1 at site 0 due to $O_3 \dashrightarrow O_1$ and o_3 at sites 1 and 2 due to $O_2 \dashrightarrow O_3$, respectively.

When the third operation arrives at each site, its execution must obey the precedence relations with the previous two operations. For example, at site 1, the execution of O_1 should obey both $O_3 \dashrightarrow O_1$ and $O_1 \dashrightarrow O_2$. However, O_1 cannot satisfy both; if O_1 does nothing according to $O_1 \dashrightarrow O_2$, it violates $O_3 \dashrightarrow O_1$, but otherwise $O_1 \dashrightarrow O_2$ is disobeyed. We can find the reason from the PRG of Fig. 4-(a). Note that PRG (a) has a cycle, that is, the precedence relations are intransitive such that $O_1 \dashrightarrow O_2$ and $O_2 \dashrightarrow O_3$, but not $O_1 \dashrightarrow O_3$. Hence, obeying two precedence relations among the three inevitably leads to violating the rest in this PRG. On the other hand, another PRG shown in Fig. 4-(b) is an acyclic tournament. Since all the edges in an acyclic tournament are transitive (see theorem 3.11 in [25]), the third operation at each site can be applied while obeying all the precedence relations; thus, o_x, o_y, and o_z become o_1. In the final analysis, we suggest the following condition as a key principle to realize OC.

Definition 12. *Precedence transitivity (PT)*
Given a CEG G=(V, E), precedence transitivity *holds in G,* iff*: if $O_a \dashrightarrow O_b$ and $O_b \dashrightarrow O_c$ for $\forall(O_a \neq O_b \neq O_c) \in V$, $O_a \dashrightarrow O_c$.*

PT is a condition in which all precedence relations are transitive. Since an acyclic tournament has a unique Hamiltonian path (see theorem 3.12 in [25]), which visits all the vertices of the graph once, precedence relations are totally ordered; e.g., the PRG of Fig. 4-(b) are ordered as $O_2 \dashrightarrow O_3 \dashrightarrow O_1$. Note that PT is not a principle that regulates operation executions and operations need never be executed in this order. Instead,



O1
O2
O3
O4
O5
(a)
(b)
precedence from a happened-before relation
precedence between a concurrent operation

Figure 5: Two PRGs of Fig. 3. (a) is the PRG based on static priorities, and (b) is the PRG based on vector clock orders.

each object container has only to store a few hints for its *last eoperation(s)*, and any remote operation reconciles its intention with that of the last eoperation based on the hints. In this way, PT enables RADT operations to commute without serialization and state rollback mechanisms.

While OC is a principle only on concurrent operations, PT explains how concurrent relations are designed against happened-before ones; indeed, precedence relations between concurrent operations (concurrent precedence relations) must accord with precedence relations inherent in happened-before relations. If static priorities are used to determine concurrent precedence relations, a derived PRG might have cycles. For example, suppose an operation issued at a higher site ID takes precedence over an operation issued at a lower site ID. The graph shown in Fig. 5-(a) is the PRG derived from the time-space diagram of Fig. 1. As those static priorities never take happened-before relations into account, PRG (a) has a cycle with O_3, O_4, and O_5. To accord concurrent precedence relations with happened-before ones, any logical clock that arranges distributed operations in a particular total order, such as Lamport clock or vector clock, can be used. For instance, since RADTs are in need of vector clocks for the causality preservation, we can use the condition deriving a total order of vector clock [19]; then, the PRG of Fig. 5-(b) can be obtained by making the precedence relations comply with their vector clock orders. Note that RADTs *never* serialize operations and *never* undo/do/redo operations, but all sites obtain the same effect as serialization by reconciling operation intentions. Instead of using original vector clocks, in Section 5.1, we introduce a fixed-size (quadruple) vector named *s4vector* that is derived from a vector clock for scalability. Based on the s4vectors, we define a transitive s4vector order.

In RADTs, precedence relations are mostly determined on the basis of s4vector orders and will be realized in remote algorithms with considering data structures and operation semantics. However, all precedence relations do not depend on only the s4vector orders. In RGAs, the precedence relation between concurrent *Update* and *Delete* is *Update*$\dashrightarrow$*Delete*, i.e., *Delete* always succeeds in removing its target container regardless of the s4vector orders. Nevertheless, since no operations happening after the *Delete* can arrive to the container, PT holds for the operations arriving to the object container.

Since precedence relations, in which PT is grounded, are differently implemented according to specific operation types, it is difficult to prove that, without loss of generality, PT guarantees the eventual consistency. In this paper, we apply PT to the implementations of operation types, and thus make pairs of operation types commute. Hence, in *Appendix* B, we prove OC for every pair of operation types to which PT is applied. As the proofs show, PT is a successful guideline to achieve OC. Although this paper uses PT as a means of achieving OC, PT itself could accomplish the eventual consistency for the implementations of RFAs or RHTs. Furthermore, unlike OC, PT might be able to ensure consistency for the execution sequences that are not CESes. We will discuss this issue further in Section 7.

However, for some operation types, defining precedence is unavailable. To illustrate, consider the four binary arithmetic operation types, i.e., addition, subtraction, multiplication, and integer division, which are allowed on replicated integer variables. Since some pairs of these arithmetic operation types are not commutative, this data type does not spontaneously ensure OC. Unlike the RADT operations, the intentions of these operation types are realized depending on the previous value as an operand. Therefore, the precedence relation defined for RADT operations is hard to be applied to those arithmetic operations. Nevertheless, OC is still available. In this example, if multiplications or integer divisions are transformed into appropriate additions or subtractions as the corresponding remote operations, OC can be achieved due to the commutative laws for additions and subtractions. To illustrate, suppose that, in Fig. 2, O_1 is to multiply 3, O_2 to add 2, and O_3 to subtract 5 on the initial value of the shared variable 10, respectively. If O_1 is transformed into the addition of 10, i.e., an increased value by the multiplication, as the remote operation of O_1, the replicated variables will converge into 17.

5. RADT Implementations

5.1. The S4Vector

For optimization purpose, we define a quadruple vector type.

```
typedef S4Vector[int ssn, int sid, int sum, int seq];
```

Let $\vec{v}_O$ be the vector clock of an operation issued at site i. Then, $\vec{s}_O$ of the `S4Vector` type can be derived from $\vec{v}_O$ as follows: (1) $\vec{s}_O$[`ssn`] is the global session number that increases monotonically, (2) $\vec{s}_O$[`sid`] is the site ID which is unique to the site, (3) $\vec{s}_O$[`sum`] is $sum(\vec{v}_O) := \sum_{\forall i} \vec{v}_O[i]$, and (4) $\vec{s}_O$[`seq`] is $\vec{v}_O[i]$, which is reserved for purging tombstones (see Section 5.6). To illustrate, suppose $\vec{v}_O = [1, 2, 3]$ is the vector clock of an operation that is issued at site 0 at session 4. Then, the s4vector of $\vec{v}_O$ is $\vec{s}_O = [4, 0, 6, 1]$. As a unit of collaboration, a session begins with the same initial vector clocks and the same RADTs at all sites. When a copy of RADT is reloaded overall sites and a new editing begins, the session number $\vec{s}_O$[`ssn`] increases. The s4vector of an operation is globally unique because $sum(\vec{v}_O)$ is unique to every operation issued at a site. We define an order between two s4vectors as follows.

Definition 13. *S4vector order '≺'*
Given two s4vectors $\vec{s}_a$ and $\vec{s}_b$, $\vec{s}_a$ precedes $\vec{s}_b$, or $\vec{s}_b$ succeeds $\vec{s}_a$, denoted as $\vec{s}_a \prec \vec{s}_b$, iff*: (1) $\vec{s}_a[ssn] < \vec{s}_b[ssn]$, or (2) $(\vec{s}_a[ssn] = \vec{s}_b[ssn]) \wedge (\vec{s}_a[sum] < \vec{s}_b[sum])$, or (3) $(\vec{s}_a[ssn] = \vec{s}_b[ssn]) \wedge (\vec{s}_a[sum] = \vec{s}_b[sum]) \wedge (\vec{s}_a[sid] < \vec{s}_b[sid])$.*

Lemma 2. *The s4vector orders are transitive.*

Proof. The s4vectors issued at a site are totally ordered because $\vec{s}$[sum] increases monotonously. If $\vec{s}$[sum]s are equal, they must be issued at different sites and are ordered by unique $\vec{s}$[sid]s. The s4vectors of different sessions are ordered by monotonous $\vec{s}$[ssn]s. Since all s4vectors are totally ordered by the three conditions, s4vector orders are transitive.

In fact, conditions (2) and (3) are the same ones as in the total ordering of vector clocks [19]. In this Section below, $\vec{s}_O$ denotes the s4vector of the current operation derived from $\vec{v}_O$ in Algorithm 1.

5.2. Replicated Fixed-Size Arrays

To inform a *Write* executing on an element of the last eoperation, a single s4vector is encapsulated with an object. Hence, an element of an RFA can be constructed using C/C++ language conventions as follows.

```
struct Element {
   Object*   obj;
   S4Vector  $\vec{s}_p$;
};
Element RFA[ARRAY_SIZE];
```

An RFA is an array of a fixed number of Elements. Based on this data structure, Algorithm 5 describes the remote algorithm of the *Write* operation type, where $\vec{s}_O$ is the s4vector of the current remote operation.

Algorithm 5 The remote algorithm for *Write* type

```
1  Write(int i, Object* o)
2      if(RFA[i].$\vec{s}_p \prec \vec{s}_O$) // $\vec{s}_O$: s4vector of this Write;
3          RFA[i].obj := o;
4          RFA[i].$\vec{s}_p$ := $\vec{s}_O$;
5          return true;
6      else return false;
```

In RFAs, a remote *Write(int i, Object* o)* replaces *obj* and $\vec{s}_p$ of the *i*th Element with a new object *o* and $\vec{s}_O$, respectively, only when $\vec{s}_O$ succeeds $\vec{s}_p$, i.e., the s4vector of the last eoperation on the Element, as line 2. After that, this *Write* becomes the new last eoperation of the Element. The local algorithm for *Write* also replaces $\vec{s}_p$ of Element with its s4vector. When a site generates a local operation, its s4vector is up-to-date and thus necessarily succeeds the s4vector of any last eoperation; thus, PT is applied to *Write* type in the local and remote algorithms. Formal proofs of consistency, i.e., OC, is presented in *Appendix* B.

5.3. Replicated Hash Tables

An RHT is defined as an array of pointers to Slots as follows.

```
struct Slot {
    Object*   obj;
    S4Vector  $\vec{s}_p$;
    Key       k;
    Slot*     next;
};
Slot* RHT[HASH_SIZE];
```

For the separate chaining, a Slot has a key (*k*) and a pointer to another Slot (*next*) to link a chain. Algorithms 6 and 7 show the remote algorithms for *Put* and *Remove*, respectively.

Algorithm 6 The remote algorithm for *Put* type

```
1  Put(Key k, Object* o)
2      Slot *pre_s := nil;
3      Slot *s := RHT[hash(k)];
4      while(s != nil and s.k != k) // find slot in the chain;
5          pre_s := s;
6          s := s.next;
7      if(s != nil and $\vec{s}_O \prec$ s.$\vec{s}_p$) return false;
8      else if(s != nil and s is a tombstone) Cemetery.withdraw(s);
9      else if(s = nil)
10         s := make new Slot;
11         if(pre_s != nil) pre_s.next := s;
12         s.k := k;
13         s.next := nil;
14     s.obj := o;
15     s.$\vec{s}_p$ := $\vec{s}_O$;
16     return true;
```

A *Put* first examines if the Slot where its key *k* is mapped by a hash function *hash* already exists (lines 3–6). When the last eoperation on the Slot takes precedence over a *Put*, i.e., $\vec{s}_O \prec s.\vec{s}_p$, the *Put* is ignored (lines 7). In the case of no Slot, a new Slot is created and connected to the chain (lines 9–13). Finally, it allocates a new object and records the s4vector in the Slot (lines 14–15), only when $s.\vec{s}_p \prec \vec{s}_O$ or no Slot exists for the key.

A *Remove* first finds its target Slot addressed by its key *k* (lines 2–3). Although a local *Remove* can be invoked on a non-existent Slot, it is not propagated to remote sites as shown in Algorithms 1 and 3. Therefore, if there is no Slot for a key, a remote *Remove* throws an exception and does nothing (lines 4–6). In line 7, a *Remove* is ignored if its s4vector precedes the last eoperation's, or otherwise it demotes its target Slot into a tombstone by assigning *nil* and $\vec{s}_O$ to *obj* and $\vec{s}_p$ of the tombstone, respectively, as a mark of a *tombstone* (lines 9–10). Local *Read*s, therefore, regard tombstones as no Slots. Unless a tombstone remains for a key, any concurrent operation

Algorithm 7 The remote algorithm for *Remove* type

```
1  Remove(Key k)
2      Slot *s := RHT[hash(k)];
3      while(s != nil and s.k != k) s := s.next;
4      if(s = nil)
5          throw NoSlotException;
6          return false;
7      if($\vec{s}_O \prec$ s.$\vec{s}_p$) return false;
8      if(s is not tombstone) Cemetery.enrol(s);
9      s.obj := nil;
10     s.$\vec{s}_p$ := $\vec{s}_O$;
11     return true;
```

will miss its cobject, and thus precedence relation with the last *Remove* is lost. Hence, if an operation arrives to a tombstone, its effectiveness can be decided based also on s4vector orders. Recall the example of RHTs in Section 2.3. Here, O_1 becomes the last eoperation of the tombstone for k_1 while O_2 is ignored at site 0.

*Remove*s enrol tombstones in `Cemetery`, which is a list of tombstones sorted in the s4vector order for purging tombstones (line 8). In Section 5.6, we discuss their purging condition using the s4vector. If a tombstone receives an operation whose $\vec{s}_O$ succeeds, its $\vec{s}_p$ is replaced with $\vec{s}_O$. When a succeeding *Put* is executed on a tombstone, it is withdrawn from `Cemetery` as line 8 in Algorithm 6 since it must be visible again to local operations using the key and thus need not be purged.

5.4. The S4Vector Index (SVI) scheme for RGAs

In RGAs, *Insert*s and *Delete*s induce the intention violation problem due to integer indices used in the local RGA operations, as stated in Section 2.3; that is, the nodes indicated by integer indices might be different at remote sites. If alternative indices help remote sites to find intended nodes, the problem becomes easier to handle. Consequently, this paper introduces an *s4vector index (SVI) scheme* for the remote RGA operations. A local operation issued with an integer index is transformed into a remote one parameterized with an s4vector before it is broadcast. The SVI scheme is implemented with a hash table which associates an s4vector with a pointer to a node. Note that the s4vector of every operation is globally unique; thus, it can be used as a unique index to find a node in the hash table. As mentioned in Section 2.3, RGAs adopt a linked list to represent the order of objects. After an *Insert* adds a new node into the linked list, the pointer to the node is placed into the hash table by using the s4vector of the *Insert* as the hash key. The following shows the overall data structure of an RGA.

```
struct Node {
    Object*   obj;
    S4Vector  $\vec{s}_k$;   // for a hash key and precedence of Inserts
    S4Vector  $\vec{s}_p$;   // for precedence of Deletes and Updates
    Node*     next;    // for the hash table
    Node*     link;    // for the linked list
};
Node* RGA[HASH_SIZE];
Node  head;  // the starting point of the linked list
```

A `Node` of an RGA has five variables. $\vec{s}_k$ is the s4vector index as a hash key, and is used for precedence of *Insert*s. For precedence of *Delete*s and *Update*s, $\vec{s}_p$ is prepared . Two pointers to `Node`s, i.e., `next` and `link`, are for the separate chaining in the hash table and for the linked list, respectively. An RGA is defined as an array of pointers to `Node`s like an RHT, and `head` is a starting point of the linked list.

Fig. 6 shows examples of an RGA data structure which is constructed with a linked list combined with a hash table. The local algorithms also maintain and operate on such structures. To illustrate, assume that, at session 1, site 2 invokes *Insert*(3, o_x) with a vector clock $\vec{v}_O := [3, 1, 2]$ on the RGA structure of Fig. 6-(a), which can be denoted as $[o_1 o_2 o_3 \tau_4 o_5]$. As shown in Algorithm 4, the local *Insert* algorithm first finds its reference `Node`, i.e., o_3, from the linked list, then creates a new

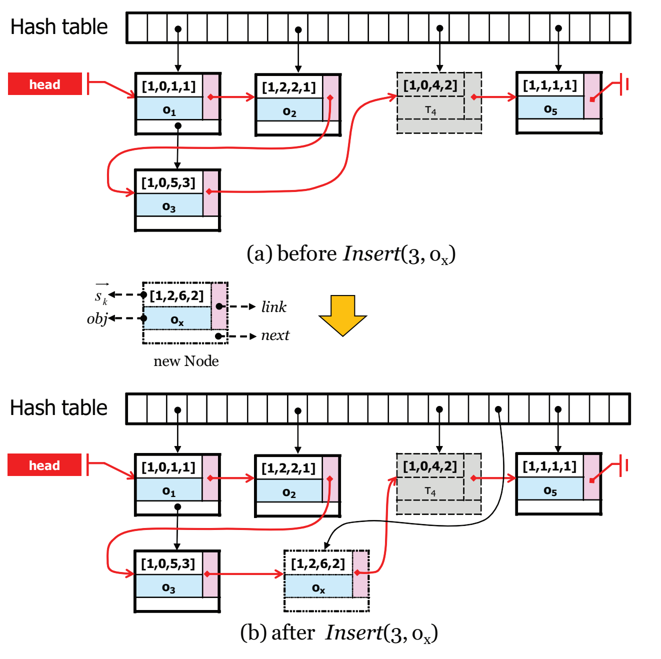


Figure 6: An example data structure of an RGA. The `Node` of τ_4 is a tombstone

`Node`. This `Node` contains the new object o_x, and the s4vector $\vec{s}_O = [1, 2, 6, 2]$ is assigned into both $\vec{s}_k$ and $\vec{s}_p$. Then, this `Node` is placed in the hash table by hashing $\vec{s}_O$ as a key and is connected to the linked list as shown in Fig. 6-(b); thus, $[o_1 o_2 o_3 \boldsymbol{o_x} \tau_4 o_5]$. We assume line 16 of Algorithm 4 does this.

Once $\vec{s}_k$ of a `Node` is set, it is immutable, thereby being adopted as an s4vector index in the remote operation into which a local operation is transformed. For example, a local *Insert*(3, o_x) will be transformed into *Insert*([1, 0, 5, 3], o_x) before it is broadcast because $\vec{s}_k$ of the third `Node` is [1, 0, 5, 3] in Fig. 6-(a). In this way, three local RGA operations are broadcast to remote sites in the following forms: *Insert(S4Vector* $\vec{i}$*, Object* o)*, *Delete(S4Vector* $\vec{i}$*)*, and *Update(S4Vector* $\vec{i}$*, Object* o)*, where *o* is a new object, and where $\vec{i}$ is the s4vector index, i.e., $\vec{s}_k$ of the reference `Node` that the local *Insert* refers to or the target `Node` that the local *Delete* or the local *Update* is executed on. If an *Insert* adds its object at the head of an RGA, $\vec{i}$ will be *nil*.

5.5. Three remote operations for RGAs

Algorithm 8 shows the remote algorithm for *Insert* type. At the beginning, a remote *Insert* looks for its reference `Node` in the hash table with the s4vector index $\vec{i}$ (lines 5–6). The SVI scheme ensures this `Node` is always the same one that the corresponding local *Insert* referred to. For non-*nil* $\vec{i}$, the reference `Node` always exists in the remote RGA structures because the `Node` where a *Delete* has performed also remains as a tombstone. To this end, an *Insert* throws an exception, unless finding its reference (lines 7–9). An *Insert* creates a new `Node` with $\vec{s}_O$ as a hash key and connects it to the beginning of the chain in the hash table (lines 10–15).

Algorithm 8 The remote algorithm for *Insert* type

```
1  Insert(S4Vector $\vec{i}$, Object* $o$)
2      Node* ins;
3      Node* ref;
4      if($\vec{i}$ != nil) // find reference node in hash table;
5          ref := RGA[hash($\vec{i}$)];
6          while(ref!=nil and ref.$\vec{s}_k$!=$\vec{i}$) ref := ref.next;
7          if(ref = nil)
8              throw NoRefObjException;
9              return false;
10     ins := make new Node;
11     ins.$\vec{s}_k$ := $\vec{s}_O$;
12     ins.$\vec{s}_p$ := $\vec{s}_O$;
13     ins.obj := $o$;
14     ins.next := RGA[hash($\vec{s}_O$)]; //place the new node
15     RGA[hash($\vec{s}_O$)] := ins;      // into the hash table;
16     if($\vec{i}$ = nil)
17         if(head.link = tail or head.$\vec{s}_k$ $\prec$ $\vec{s}_O$)
18             ins.link := head.link;
19             head.link := ins;
20             return true;
21         else ref := head;
22     while(ref.link != tail and $\vec{s}_O$ $\prec$ ref.link.$\vec{s}_k$) ref := ref.link;
23     ins.link := ref.link;
24     ref.link := ins;
25     return true;
```

A remote *Insert* might not add its new Node on the exact right of its reference Node in order to preserve the intentions of some other concurrent *Insert*s that have already inserted their Nodes next to the same reference. If an s4vector of an *Insert* succeeds, the intention of the *Insert* is more preservable; thus, it places its new Node nearer the reference Node. Accordingly, a remote *Insert* scans the Nodes next to its reference and places its new Node in front of the first encountered Node whose $\vec{s}_k$ precedes $\vec{s}_O$ (or at the tail) according to lines 22–24. As lines 16–20 are needed for inserting a new object at the head, the conditions of line 17 are the converse of line 22; if not inserted at the head, the comparison continues again from line 22. The following example, as known as the dOPT puzzle [14] (see Section 6), illustrates how *Insert*s work.

Example 1. *[Fig. 2] dOPT puzzle, on initial RGAs=∅,*

I_1 : *Insert*$(0 = nil, o_{i_1})$ *with* $[1,0,1] \rightsquigarrow \vec{i}_1 = [1,0,2,1]$,

I_2 : *Insert*$(0 = nil, o_{i_2})$ *with* $[0,1,0] \rightsquigarrow \vec{i}_2 = [1,1,1,1]$,

I_3 : *Insert*$(0 = nil, o_{i_3})$ *with* $[0,0,1] \rightsquigarrow \vec{i}_3 = [1,2,1,1]$.

We assume the initial RGAs are empty (∅), and the operations are propagated as shown in Fig. 2; thus, I_1, I_2, and I_3 correspond to O_1, O_2, and O_3 of Fig. 2, respectively. All their remote forms have *nil*s as the s4vector indices because their intentions are to insert new nodes at the head. In this example, $\vec{i}_1$, $\vec{i}_2$, and $\vec{i}_3$ are the s4vectors derived from the left vector clocks on the assumption of session 1. Due to $\vec{i}_2 \prec \vec{i}_3 \prec \vec{i}_1$, $I_2 \dashrightarrow I_3 \dashrightarrow I_1$; the intention of I_1 is most preservable, then I_3, and that of I_2 is least. PT is realized in *Insert*s as follows.

At site 0, when I_3 arrives in the form of *Insert*(nil, o_{i_3}), o_{i_3} is placed at the head by lines 16–20. Then, I_1 is executed as $[o_{i_1}o_{i_3}]$ by the local *Insert* algorithm. Finally, when I_2 arrives, *head.*$\vec{s}_k$ $(= \vec{i}_1)$ succeeds $\vec{s}_O$ $(= \vec{i}_2)$; i.e., $\vec{i}_2 \prec \vec{i}_1$. Next, in line 22, $\vec{s}_O$ $(= \vec{i}_2)$ is compared with $\vec{s}_k$ of o_{i_3} $(= \vec{i}_3)$. Due to $\vec{i}_2 \prec \vec{i}_3$, I_2 inserts o_{i_2} at the tail as $[o_{i_1}o_{i_3}o_{i_2}]$

At sites 1 and 2, likewise, concurrent I_2 and I_3 commute; thus, $[o_{i_3}o_{i_2}]$. At these sites, the remote I_1 compares $\vec{i}_1$ with $\vec{i}_3$, i.e., $\vec{s}_k$ of o_{i_3}. Note that $I_3 \rightarrow I_1$, so $\vec{i}_3 \dashrightarrow \vec{i}_1$. Hence, I_1 puts o_{i_1} in front of o_{i_3} because *head.*$\vec{s}_k \prec \vec{s}_O$ (i.e., $\vec{i}_3 \prec \vec{i}_1$) in line 22, and eventually the RGA states converge as follows.

Execution 1. *site 0:* $\varnothing \overset{I_3}{\Rightarrow} [o_{i_3}] \overset{I_1}{\Rightarrow} [o_{i_1}o_{i_3}] \overset{I_2}{\Rightarrow} [o_{i_1}o_{i_3}o_{i_2}]$,

site 1: $\varnothing \overset{I_2}{\Rightarrow} [o_{i_2}] \overset{I_3}{\Rightarrow} [o_{i_3}o_{i_2}] \overset{I_1}{\Rightarrow} [o_{i_1}o_{i_3}o_{i_2}]$,

site 2: $\varnothing \overset{I_3}{\Rightarrow} [o_{i_3}] \overset{I_2}{\Rightarrow} [o_{i_3}o_{i_2}] \overset{I_1}{\Rightarrow} [o_{i_1}o_{i_3}o_{i_2}]$.

It is worth noting that consistency is achieved without comparing the s4vector of I_1 with the effect of concurrent I_2 at sites 1 and 2. This is due to PT that harmonizes concurrent precedence relations with happened-before ones.

Algorithm 9 The remote algorithm for *Delete* type

```
1  Delete(S4Vector $\vec{i}$)
2      Node* n := RGA[hash($\vec{i}$)];
3      while(n != nil and n.$\vec{s}_k$ != $\vec{i}$) n := n.next;
4      if(n = nil)
5          throw NoTargetObjException;
6          return false;
7      if(n is not a tombstone)
8          Cemetery.enrol(n);
9          n.obj := nil;
10         n.$\vec{s}_p$ := $\vec{s}_O$;
11     return true;
```

The local and remote *Delete* algorithms also leave a tombstone behind. In Algorithm 9, a remote *Delete* finds its target Node with $\vec{i}$ via the hash table (lines 2–3); otherwise, it throws an exception (lines 4–6). Regardless of s4vector orders, *Delete*s assign *nil* and $\vec{s}_O$ into *obj* and $\vec{s}_p$ (not $\vec{s}_k$), respectively, as a mark of a tombstone, and enrols it into Cemetery (lines 8–10). Unlike RHTs, an RGA tombstone never revives. In Algorithm 4, *findlist* and *findlink* functions exclude tombstones from counting. Being invisible to local operations, a tombstone is never employed as a target or a reference Node with an integer index. For example, in Fig. 6-(a), local *Insert*$(4, o_y)$ refers to the Node of o_5 instead of the tombstone of τ_4, and is transformed into remote *Insert*$([1,1,1,1], o_y)$.

Algorithm 10 The remote algorithm for *Update* type

```
1  Update(S4Vector $\vec{i}$, Object* $o$)
2      Node* n := RGA[hash($\vec{i}$)];
3      while(n != nil and n.$\vec{s}_k$ != $\vec{i}$) n := n.next;
4      if(n = nil)
5          throw NoTargetObjException;
6          return false;
7      if(n is a tombstone) return false;
8      if($\vec{s}_O$ $\prec$ n.$\vec{s}_p$) return false;
9      n.obj := $o$;
10     n.$\vec{s}_p$ := $\vec{s}_O$;
11     return true;
```

In Algorithm 10, a remote *Update* operates in the same way as a remote *Delete* until finding its target Node. An *Update* replaces *obj* and $\vec{s}_p$ (but not $\vec{s}_k$) of its target Node with its owns, if $\vec{s}_O$ succeeds $\vec{s}_p$ (lines 8–10). Unlike *Put* of RHTs, an *Update* does nothing on a tombstone as in line 7; thus, always *Update*$\dashrightarrow$*Delete*. This prevents an *Update* on a tombstone from being translated into the semantic of an *Insert* and makes its purging condition simple (see Section 5.6).

Example 2 illustrates how RGA operations interact with each other when they are propagated as shown in Fig. 1.

Example 2. *[Fig. 1] Initially, RGAs=[o_{i_a}] with* $\vec{i}_a$=[1, 0, 1, 1],
$U_1(O_1)$: *Update(1=*$\vec{i}_a$*,* $\dot{o}_{i_a}$*) with* [1, 0, 0] $\rightsquigarrow$ $\vec{i}_1$ = [2, 0, 1, 1],
$U_2(O_2)$: *Update(1=*$\vec{i}_a$*,* $\ddot{o}_{i_a}$*) with* [0, 1, 0] $\rightsquigarrow$ $\vec{i}_2$ = [2, 1, 1, 1],
$D_3(O_3)$: *Delete(1=*$\vec{i}_a$*) with* [0, 0, 1] $\rightsquigarrow$ $\vec{i}_3$ = [2, 2, 1, 1],
$I_4(O_4)$: *Insert(0=nil,* o_{i_4}*) with* [2, 1, 1] $\rightsquigarrow$ $\vec{i}_4$ = [2, 0, 4, 2],
$I_5(O_5)$: *Insert(1=*$\vec{i}_a$*,* o_{i_5}*) with* [0, 2, 0] $\rightsquigarrow$ $\vec{i}_5$ = [2, 1, 2, 2].

For $U_1 \parallel U_2$ being in conflict, the intention of U_2 is more preservable than that of U_1 due to $\vec{i}_1 \prec \vec{i}_2$; thus, as shown in Execution 2, U_1 is ignored at site 1 by line 8 of Algorithm 10. When D_3 conflicts with U_1 and U_2, D_3 always succeeds in leaving the tombstone of o_{i_a}, i.e., τ_{i_a}, regardless of $\vec{s}_p$ of o_{i_a}, but the *Update*s do nothing on the tombstone by line 7 of Algorithm 10. Due to τ_{i_a}, I_5, which is concurrent with D_3, can also find its reference `Node` at sites 0 and 2. In addition, when I_4 performs at sites 1 and 2, it places o_{i_4} in front of τ_{i_a} since Algorithm 8 regards τ_{i_a} as a normal `Node` whose $\vec{s}_k$ is $\vec{i}_a \prec \vec{i}_4$ in lines 16–20. In concludsion, RGAs eventually converge at all the sites as follows.

Execution 2. *At each site of Fig. 1,*
site 0: $[o_{i_a}]\overset{U_1}{\Rightarrow}[\dot{o}_{i_a}]\overset{U_2}{\Rightarrow}[\ddot{o}_{i_a}]\overset{D_3}{\Rightarrow}[\tau_{i_a}]\overset{I_4}{\Rightarrow}[o_{i_4}\tau_{i_a}]\overset{I_5}{\Rightarrow}[o_{i_4}\tau_{i_a}o_{i_5}]$,
site 1: $[o_{i_a}]\overset{U_2}{\Rightarrow}[\ddot{o}_{i_a}]\overset{I_5}{\Rightarrow}[\ddot{o}_{i_a}o_{i_5}]\overset{U_1}{\Rightarrow}[\ddot{o}_{i_a}o_{i_5}]\overset{D_3}{\Rightarrow}[\tau_{i_a}o_{i_5}]\overset{I_4}{\Rightarrow}[o_{i_4}\tau_{i_a}o_{i_5}]$,
site 2: $[o_{i_a}]\overset{D_3}{\Rightarrow}[\tau_{i_a}]\overset{U_1}{\Rightarrow}[\tau_{i_a}]\overset{U_2}{\Rightarrow}[\tau_{i_a}]\overset{I_4}{\Rightarrow}[o_{i_4}\tau_{i_a}]\overset{I_5}{\Rightarrow}[o_{i_4}\tau_{i_a}o_{i_5}]$.

To summarize, the SVI scheme enables the remote RGA operations always to find their intended `Node`s, i.e., cobjects, correctly using the hash table. For this purpose, $\vec{s}_k$ is prepared in a `Node` as an s4vector index, which is immutable once being set by an *Insert*. Also, $\vec{s}_k$ is used to realize PT among *Insert*s. Since tombstones remain, no remote operations miss their cobjects. Another s4vector of a `Node`, $\vec{s}_p$ is renewed by *Update*s and *Delete*s. The effectiveness of *Update*s is decided by the s4vector orders of $\vec{s}_p$, but *Delete*s are always successful; i.e., always *Update* $\dashrightarrow$ *Delete*. Nevertheless, OC can hold for the operations on tombstones because no operations happening after a *Delete* target or refer to the tombstone. Separation of $\vec{s}_k$ and $\vec{s}_p$ means *Insert*s never conflict with any *Update*s or *Delete*s if cobjects are preserved, which will be explained in the following Section.

5.6. Cobject preservation

Definition 10 implies that cobjects should be preserved for intentions of remote operations. Since the effect of a local operation is caused by its cobject(s), if such causality is not preserved at remote sites, remote operations might cause different effects. Tombstones enable a remote operation to manifest its intention by retaining cobject(s), but need purging. However, if the tombstone purging algorithm is not properly designed, RADTs might suffer inconsistency problem. Especially, the operational transformation (OT) framework supporting optimistic *Insert* and *Delete* types have failed to achieve consistency because cobjects are not preserved at remote sites (see Section 6). To illustrate, consider Example 3 where three operations are executed as in Fig. 7.

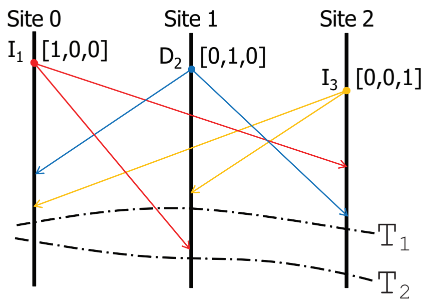


Figure 7: A time-space diagram of *Example* 3.

Example 3. *[Fig. 7] Initially, RGAs*=[o_{i_a}] *with* $\vec{i}_a$=[1, 0, 1, 1],
I_1 : *Insert(0 = nil,* o_{i_1}*) with* [1, 0, 0] $\rightsquigarrow$ $\vec{i}_1$ = [2, 0, 1, 1],
D_2 : *Delete(1 =* $\vec{i}_a$*) with* [0, 1, 0] $\rightsquigarrow$ $\vec{i}_2$ = [2, 1, 1, 1],
I_3 : *Insert(1 =* $\vec{i}_a$*,* o_{i_3}*) with* [0, 0, 1] $\rightsquigarrow$ $\vec{i}_3$ = [2, 2, 1, 1].

Two cobjects of I_1 are the head and o_{i_a} while o_{i_a} and the tail are the cobjects of I_3. The former cobject looks indispensable to the execution of an *Insert* because it prescribes the position where an *Insert* has to be executed at all sites. However, the necessity of the latter cobject might be overlooked, though it is used in the conditions to terminate the comparisons, i.e., lines 17 or 22 in Algorithm 8. In our RGA implementation, if the latter cobject of an *Insert* is purged before the remote execution of the *Insert*, *Insert*s of different intentions can be in conflict. To illustrate, see Execution 3 where τ_{i_b} is assumed to be purged at the time $\mathtt{T}_1$ of Fig. 7 ($\overset{P}{\Rightarrow}$ means purging).

Execution 3. *If purging* τ_{i_b} *at* T_1*,*
site 0: $[o_{i_a}]\overset{I_1}{\Rightarrow}[o_{i_1}o_{i_a}]\overset{D_2}{\Rightarrow}[o_{i_1}\tau_{i_a}]\overset{I_3}{\Rightarrow}[o_{i_1}\tau_{i_a}o_{i_3}]\overset{P}{\Rightarrow}[o_{i_1}o_{i_3}]$,
site 1: $[o_{i_a}]\overset{D_2}{\Rightarrow}[\tau_{i_a}]\overset{I_3}{\Rightarrow}[\tau_{i_a}o_{i_3}]\overset{P}{\Rightarrow}[o_{i_3}]\overset{I_1}{\Rightarrow}[o_{i_3}o_{i_1}]$,
site 2: $[o_{i_a}]\overset{I_3}{\Rightarrow}[o_{i_a}o_{i_3}]\overset{I_1}{\Rightarrow}[o_{i_1}o_{i_a}o_{i_3}]\overset{D_2}{\Rightarrow}[o_{i_1}\tau_{i_a}o_{i_3}]\overset{P}{\Rightarrow}[o_{i_1}o_{i_3}]$.

In this execution, observe the effect of I_1 according to the existence of its latter cobject, i.e., o_{i_a} or τ_{i_a}. At site 0, the local algorithm of I_1 places o_{i_1} at the head without considering o_{i_a}. Being indispensable to I_3 as the former cobject, τ_{i_a} should be retained. At site 2, I_1 has to insert o_{i_1} in front of o_{i_a} by line 17 in Algorithm 8, and then D_2 performs. Hence, in the final results at sites 0 and 2, $[o_{i_1}o_{i_3}]$ is a correct order between o_{i_1} and o_{i_2}. At site 1, however, if τ_{i_a} is purged, I_1 compares $\vec{i}_1$ with $\vec{s}_k$ of o_{i_3} (= $\vec{i}_3$) despite I_3 having a different reference; thus, $[o_{i_3}o_{i_1}]$. Consequently, the loss of the latter cobject can lead to the different effect of I_1. Instead, if the latter cobject is purged at $\mathtt{T}_2$ of Fig. 7, the effect of I_1 is consistent with sites 0 and 1 as shown in Execution 4.

Execution 4. *At site 1, if purging τ_{i_b} at T_2,*

site 1: $[o_{i_a}]\overset{D_2}{\Rightarrow}[\tau_{i_a}]\overset{I_3}{\Rightarrow}[\tau_{i_a}o_{i_3}]\overset{I_1}{\Rightarrow}[o_{i_1}\tau_{i_a}o_{i_3}]\overset{P}{\Rightarrow}[o_{i_1}o_{i_3}]$.

In this respect, tombstones must be preserved as far as they could be cobjects of some operations in order to ensure consistent intentions. However, in RGAs, tombstones impeding search for `Node`s in the linked list need purging as soon as possible. We, therefore, introduce a safe tombstone purging condition using s4vectors.

To begin with, let D_i be a *Delete* issued at site i and let τ_i be the tombstone caused by D_i, respectively. Recall that D_i assigns its s4vector into $\tau_i.\overrightarrow{s}_p$ and that RGAs guarantee two properties for a tombstone: (1) a tombstone is never targeted or referred to by any subsequent local operations, and (2) a tombstone never revives. Hence, only for the operations concurrent with D_i, can τ_i be a cobject. Although another *Delete* (say D_j) might perform on τ_i, it must be $D_i \parallel D_j$. In this case, τ_i can be targeted or referred to by only the operations concurrent with both D_i and D_j. By retaining τ_i as far as any concurrent operations with either one of the *Delete*s might arrive, we can preserve the cobject of those concurrent operations. Golding already introduced a safe condition to ensure this [26]. The existing condition enables RHTs and RGAs to preserve the cbjects of their operations, except the latter cobjects of *Insert*s.

To preserve the latter cobjects of *Insert*s, an additional condition is needed. Note that, at site 1 in Execution 3, the loss of τ_{i_a} causes problems only when the s4vector of I_1 precedes $\overrightarrow{s}_k$ of o_{i_3}, i.e., the next `Node` of τ_{i_a}. In other words, if it is ensured that the s4vector of a new *Insert* succeeds $\overrightarrow{s}_k$ of every `Node` in RGAs, the next `Node` of a tombstone can substitute the tombstone as a latter cobject. To this end, an RGA needs to maintain VC_{last} which is a set of vector clocks including as many vector clocks as the number of sites N, i.e., $VC_{last} = \{\overrightarrow{v}_{last_0}, \ldots, \overrightarrow{v}_{last_{N-1}}\}$; $\overrightarrow{v}_{last_j} \in VC_{last}$ is the vector clock of the operation that was lastly issued at site j and successfully executed in this site. Using VC_{last}, a tombstone τ_i, demoted by D_i, can be safely purged if satisfying both of the following conditions.

(1) $\tau_i.\overrightarrow{s}_p[\mathtt{seq}] \leq \min\limits_{\forall \overrightarrow{v} \in VC_{last}} \overrightarrow{v}[i]$ for $i = \tau_i.\overrightarrow{s}_p[\mathtt{sid}]$,

(2) $\tau_i.link.\overrightarrow{s}_k[\mathtt{sum}] < \min\limits_{\forall \overrightarrow{v} \in VC_{last}} sum(\overrightarrow{v})$ or $\tau_i.link = \text{tail}$.

Condition (1) is the similar one to Golding's [26], which means that every site had executed D_i, so hereafter only the operations happening after D_i will arrive. Condition (2) means the s4vector of any new operation succeeds than that of the `Node` next to the purged tombstone in the linked list. However, these two conditions rely on the assumption that every site is alive. If a site stops issuing operations, the other sites cannot purge tombstones. In this case, a site can request the paused sites to send their vector clocks back and renews VC_{last} with the received ones. Thus, the site can continue purging.

6. Related Work

The concept of commutativity was first introduced in distributed database systems [17, 16]. It was, however, applied to concurrency control for a centralized resource, but not to consistency maintenance among replicas. In other words, to grant more concurrency for some transactions in a locking scheme, transaction schedulers allow innately commutative operations, e.g., write operations on different objects, to be executed concurrently. Thus, noncommutative operations still have to be locked. On the other hand, the works of [5] and [10] considered commutativity to maintain replica consistency. However, they also allow only the execution order of innately commutative operations to be changed.

The behavior of RFAs is similar to that of *Thomas's writing rule*, which was introduced in multiple copy databases [8, 9]. The rule prescribes that a *Write*-like operation can take effect only when it is newer than the previous one on its target position [23]. To represent the newness, Lamport clocks are adopted. In fact, unless tombstones are purged, Lamport clocks can be used in RADTs on behalf of the `sum` component of s4vectors because they also provide a transitive total order. In this paper, however, we use the s4vectors derived from vector clocks to ensure strict causality preservation not only among operations but also among operations and their cnodes. The idea of tombstones was also introduced in replicated directory services [10, 11, 12], which are similar to RHTs.

With respect to RGAs, the operational transformation (OT) framework is one of a few relevant approaches that allows sites to execute *insertions and deletions* in different orders on ordered characters. In this framework, an integration algorithm calls a series of transformation functions (TFs) to transform the integer index of every remote operation against each of its concurrent operations in the history buffer. A function $O'_a = tf(O_a, O_b)$ obtains a transformed operation of O_a against O_b, that is, O'_a, which is mandated to satisfy the following transformation properties, called TP_1 and TP_2.

Property 1. *Transformation property 1 (TP_1)*
For $O_1 \parallel O_2$ issued on the same replica state, tf satisfies TP_1 iff: $O_1 \mapsto tf(O_2, O_1) \equiv O_2 \mapsto tf(O_1, O_2)$.

Property 2. *Transformation property 2 (TP_2)*
For $O_1 \parallel O_2$ and $O_2 \parallel O_3$ and $O_1 \parallel O_3$ issued on the same replica state, tf satisfies TP_2 iff: $tf(tf(O_3, O_1), tf(O_2, O_1)) = tf(tf(O_3, O_2), tf(O_1, O_2))$.

TP_1, introduced in the dOPT algorithm by Ellis and Gibbs [18], is another expression of Definition 6 in terms of the OT framework. However, since it is not sufficient, a counterexample, called as dOPT puzzle (see Example 1), was found. Ressel et al. proposed TP_2, which means consecutive TFs along different paths must result in a unique transformed operation. It was proven that TP_1 and TP_2 are the sufficient conditions that ensure consistency of some OT integration algorithms such as adOPTed [27, 28]. In the sense that TP_1 and TP_2 specify the properties of TFs only for concurrent operations, it is worth comparing them with OC that also specifies the design of concurrent operations. Though we are not sure that they are equivalent, TP_2 is clearly a property for TFs on *sequential* concurrent operations. Therefore, if happened-before operations intervene among concurrent operations, TP_2 explains nothing. In our RADT framework, PT explains how concurrent operations are designed in consideration of happened-before relations.

Various OT methods, such as adOPTed [13], GOT [19], GOTO [14], SOCT2 [28], SOCT4 [29], SDT [30] and TTF [21] have been introduced, which proposed different TFs or integration algorithms. However, having presented no guidelines on how to develop the TFs that satisfy TP_2, in most of OT algorithms, i.e., adOPTed, GOTO, SOCT2, and SDT, counterexamples have been found. Though GOT or SOCT4 fixed up the transformation path by an undo-do-redo scheme or a global sequencer, respectively, in order to avoid TP_2, but responsiveness is significantly degraded. In addition, the intention preservation, addressed in Sun el al. [19], also has gone through the lack of effective guidelines. In our opinion, PT could be an effective guideline for satisfying TP_2 and preserving intentions. For example, when breaking ties among insertions in various causality relations in TFs, or when writing TFs about *Update*-like operation, PT can be a clue to compromise their conflicting intentions.

Another reason why OT methods have failed not only in convergence but also in the intention preservation is the loss of cobjects (see Section 5.6) if the reason is illustrated in terms of our RADT framework. Similar to Execution 3, once a character is removed, TFs are difficult to consider it. To solve this problem, Li et al. have suggested a series of algorithms, such as SDT [30], ABT [31], and LBT [20]. The authors said these algorithms free TP_2 by relying on the effects relation, which is an order between a pair of characters in the document. However, deriving effects relations incurs additional computational or space overheads because it requires to transpose operations in the history buffer or to reserve the effects relations in an additional table.

Oster et al. introduced the TTF approach that first invites tombstones to the OT framework [21]. Making a deleted character invisible, TTF achieves the TFs satisfying TP_2 based on a document growing indefinitely. However, purging tombstones in TTF is more restrictive than in RGAs because it makes integer indices of a site incomparable at the other sites. Hence, some optimizations, such as caret operations or D-TTF, are provided. TTF must be combined with an existing OT integration algorithm, such as adOPTed or SOCT2, so inherits the features of OT in complexity, scalability and reliability (see Section 7).

Recently, for the optimistic insertion and deletion types, several prototypes that adopt unique indices were introduced. Oster et al. proposed WOOT framework that is free from vector clocks for scalability [32]. Instead of vector clocks, causality is regarded to be preserved if parameter characters of operations exist at remote sites; also in RGAs, preserving causality in this way can be in use due to the SVI scheme (see Section 7). In the WOOT framework, an inserted character has a unique index of the pair $<$ *site ID*, *logical clock* $>$ and includes the indices of the previous and next characters at the insertion time. An insertion is parameterized with two indices of both sides where a new character will be inserted. Thus, whenever a character is inserted, a total order among existing characters is derived with considering both sides of characters, but this makes the insertion of WOOT an order of magnitude slower than that of RGAs. A deletion leaves a tombstone behind, but its purging algorithm is not presented.

Meanwhile, inspired by our early work [33], Shapiro et. al proposed a commutative replicated data type (CRDT) called *treedoc* that adopts an ingenious index scheme for the two types [34, 35]. Treedoc is a binary tree whose paths to nodes are unique indices and ordered totally in infix order. Using paths as unique indices, treedoc can avoid storing indices separately and provide a new index to a new node continuously, thereby ensuring index density. When insertions conflict, a pair like the WOOT index should be supplemented as a special index, and their new characters are placed in the same node. Besides, even if a deletion performs on a node that is not a leaf, its tombstone should be preserved; otherwise, indices of its child nodes would change. Thus, as editing ages, treedoc structures becomes unbalanced and might contain many tombstones. To clean up treedoc, the authors suggest two structural operations, i.e., *flatten* and *explode*, which obtain a character string from a treedoc and vise versa, respectively. However, since *flatten* operations require a distributed commitment protocol, the clean-up process is costly and not scalable.

For scalability purpose, Weiss et al. suggested another CRDT *logoot* which is a sparse n-ary tree in order to provide the index density and total order like treedoc [36]. However, unlike treedoc, logoot explicitly specify unique indices that are unbounded sequences of the pair $<$ *pos*, *site ID* $>$, where *pos* is the position in a depth of a logoot tree. The explicit indices allow logoot trees to be sparse, and thus no tombstone is needed for a deletion, incurring no overheads. Though an individual logoot index is larger than that of treedoc, the overall data overheads could be reduced for numerous operations abounding in a large-scale replication due to the absence of tombstones. However, logoot also constrains causality preservation, and thus the authors suggest the causal barrier introduced by [37] on behalf of vector clocks for scalability. Although, compared to the vector clocks, causal barriers might reduce the data overheads for transmitting logical clocks, sites should manage their local clock data structures against membership changes in the same manner as with vector clocks. In fact, causality preservation relates to the reliability issue, which is discussed in Section 7.

Above all, none of the above three approaches could derive an underlying principle, such as PT, to make operations commute. Therefore, including the effects relations of Li et al. [20], their approaches are bounded only to consistency of the insertion and deletion types. In other words, for the *Update*-like operation, they present no solutions. Though an update can be emulated with consecutive deletion and insertion, if multiple users concurrently update the same object, multiple objects will be obtained. To our knowledge, also in the OT framework, *the update operation* has not been discussed. Instead, independently of the OT framework, Sun et. al proposed a multi-version approach for the update in collaborative graphi-

Algorithms	local operations	remote operations
RGAs	$O(N)$ or $^{\dagger}O(1)$	$O(1)$
ABT	$O(\|H\|)$	$O(\|H\|^2)$
SDT	$O(1)$	$O(\|H\|^2)$ or $^{\S}O(\|H\|^3)$
D-TTF w/ adOPTed	$O(N)$ or $^{\ddagger}O(1)$	$O(\|H\|^2 + N)$
WOOT	$^{\sharp}O(N^2)$ and $^{\P}O(1)$	$^{\sharp}O(N^3)$ and $^{\P}O(N)$

Table 1: N: the number of objects or characters, $|H|$: the number of operations in history buffer, $\dagger$: the linked list operations, $\ddagger$: the caret operations, $\S$: worst-case complexity, $\sharp$: WOOT insertion operation, $\P$: WOOT deletion operation

cal editors, where the order of graphical objects does not matter [15]. In this approach, when the operations that update some attributes of graphical objects, such as position or color, are in conflict, multiple object versions are shown to users in order not to lose any intentions. Although RADTs need to behave deterministically as building blocks, they can mimic the multi-version approach. For example, if remote *Put*s or *Update*s return false, their effects can be shown as auxiliary information by using local ADTs.

7. Complexity, Scalability, and Reliability

The time complexity of RADTs is decisive for the performance and quality of collaborative applications as RADTs are designed as their building blocks. Especially, a user can issue block editing operations, or a single user's interaction may lead to the modifications of various data in complex collaborative applications. The time complexity of the local RADT operations is the same as that of the operations of the corresponding normal ADTs. In RFAs and RHTs, the remote operations perform in the same complexity as the corresponding local ADTs based on the same data structures; thus, *Write*, *Put*, and *Remove* work optimally in $O(1)$. Only when the hash functions malfunction, the theoretical worst-case of *Put* and *Remove* is $O(N)$ for the number of objects N due to the separate chaining.

The local RGA operations work in $O(N)$ since $findlist$ function of Algorithm 4 searches the intended `Node` via the linked list from the head. As mentioned in Section 2.3, RGAs can also bring in the local linked list operations using $findlink$ function; thus, complexity is constant. Meanwhile, the remote RGA operations can perform in $O(1)$ as a `Node` is searched via the hash table. The worst-case complexity of a remote *Insert* can be $O(N)$ in case that all the existing `Node`s have been inserted by the concurrent *Insert*s on the same reference `Node`.

We compare RGAs with the recent OT methods, such as ABT, SDT, and TTF, and WOOT in performance, as shown in `Table` 1. About the complexity of ABT and SDT, we consult [22], while considering D-TTF combined with the adOPTed integration algorithms for the complexity of TTF [21, 13]. According to [22], the performance of the remote ABT and SDT operations fluctuates depending on the number of operations and the operation types in the history buffer. The history buffer must include an operation as long as it could be concurrent with any remote operations arriving in the future. Therefore, as the membership size grows, an operation in a history buffer is more likely to be concurrent with any remote operations for a long time, and thus the history buffer must maintain more operations. Meantime, according to [38], WOOT presents different time complexity for insertion and deletion on the assumption that a local and a remote operation access a character in $O(1)$ and $O(N)$, respectively. `Table` 1 shows RGAs overwhelm the others in the performance of the remote operations. Besides, the remote RGA operations can perform without fluctuation and thus guarantee stable responsiveness in the collaboration.

This paper considers scalability in two aspects: membership size and the number of objects in a replication. As the membership size or the number of objects scales up, performance may degrade, or space complexity might rise. In a group communication like the RADT system model, the more sites participate in a group, the more remote operations a site must have. For example, suppose each of total s=10 sites evenly issues n=50 operations. Then, each site will execute n=50 local operations but $n \times (s - 1)$=450 remote operations. Consequently, the performance of remote operations are more critical to scalability than that of local ones. As aforementioned, OT methods are not scalable because their remote operations are inefficient and the history buffer grows as the membership size grows. Thus, RGAs are more scalable than the OT methods in performance.

In meantime, vector clocks, adopted by most of optimistic replications requiring causality detection and preservation, can affect scalability because the clock size must be proportional to the membership size. In the OT framework, intact vector clocks should be stored in the history buffer for causality detection. The space complexity to maintain the history buffer is $O(s \times |H|)$ where s is the number of sites, and wherein $|H|$ has a tendency to grow in relation to s. In addition, OT methods also demand at least more than $O(N)$ space complexity in order to store the document or the effects relations. However, RGAs can achieve consistency with only two s4vectors per object since causality detection requiring vector clocks is no longer needed due to PT; thus, the space complexity is $O(N)$. Hence, the s4vector, which is independent of the membership size, enhances the scalability of RGAs over the OT methods with respect to the space overheads. Nevertheless, if we compare the `Node` size with the index size of treedoc or WOOT, RGAs may incur higher overheads. The space complexity of WOOT and treedoc is also $O(N)$ including tombstones, while the unbounded index of logoot might incur a slightly higher complexity with no tombstone. In fact, the overheads incurred by tombstones can not be ignored in a scalable collaborative applications, and thus the tombstone purging algorithms determine the overheads. Therefore, we plan the experiments on the overheads incurred by tombstones as future work.

As most of optimistic replication systems, RADTs also constrain themselves to preserve the causality defined by Lamport, but this relates to the reliability issue. When a site crashes, some operations issued at this site might be delivered to a part of the other sites. Though such a fault as an operation loss is detected by the causality preservation scheme using vector clocks, it leads to a chain of delays in executing the operations happening after the partially delivered operations at some other sites. In the sense that a fault might result in the failures of other sites, the reliability could be significantly exacerbated in scalable collaborative applications. In Example 2, under the vector clock scheme, if a site misses U_2, then I_4 and I_5 happening after U_2 should be delayed. However, I_4 and I_5 can perform while U_2 is being retransmitted because U_2 has no *essential causality* with I_4 and I_5; in other words, I_4 and I_5 can achieve their intentions without U_2 in RGAs. Therefore, the causality preservation scheme can be relaxed, and this might enhance reliability. However, relaxing causality preservation means to allow sites to execute some operation sequences that preserve only essential causality (say eCES), which belong to a superset of CESS. Since RGA consistency is verified based on OC, it is theoretically not guaranteed that eCESes converge. In fact, though WOOT is the only approach that allows eCESes to be executed, consistency was verified by the model-checker TLC on a specification modeled on the TLA+ specification languages [39]. Since the model checker exhaustively verifies all the states produced by possible operation executions, the states must explode; thus, the verification is made only up to four sites and five characters in WOOT [38]. In any case, for the insertion and deletion types which are executed in eCES order, there is no generalized proof methodology of consistency yet.

In RFAs and RHTs, to which PT is applied, it is simple to show the eventual consistency is guaranteed even for eCESes, though the implementation of *Remove* type should be slightly modified (lines 4–6 in Algorithm 7); if the same set of operations is executed on an object container, we can make its last eoperation always the same one regardless of their execution orders due to PT. Necessarily, to achieve consistency for eCESes, the current implementations of RGAs need be modified, and the proof should be conducted on the implementations. We leave them as future work, and believe OC and PT can be also clues to make eCESes converge and to prove consistency of RGAs. Notwithstanding, compared to the OT framework, RGAs have much room for improving reliability due to the SVI scheme. In the OT framework, all happened-before operations of a remote operation are indispensable to satisfying *the precondition*, addressed by Sun. et al [19], because the integer index of an operation is dependent on the executions of the previous happened-before operations. Meanwhile, in RGAs, the SVI scheme can validate cobjects of remote operations by the s4vector indices autonomously, i.e., independently of most of the other operations, thereby checking the essential causality without causality detection using vector clocks. Even if the value of Lamport clock is substituted for $\vec{s}$[sum], the s4vector order holds. Hence, RGAs would be free from vector clocks, which could improve the reliability.

8. Conclusions

When developing applications, programmers are used to using various ADTs. Providing the same semantics of ADTs to programmers, RADTs can support efficient implementations of collaborative applications. Operation commutativity and precedence transitivity make it possible to design the complicated optimistic RGA operations without serialization/locking protocols/state rollback scheme/undo-do-redo scheme/OT methods. Especially, in performance, RGAs provide the remote operations of $O(1)$ with the SVI scheme using s4vectors. This is a significant achievement over previous works and makes RGAs scalable. Furthermore, since the SVI scheme autonomously validates the causality and intention of an RGA operation, reliability would be enhanced. The work presented here has profound implication for future studies of other RADTs such as various tree data types.

A. Proof of Operation Commutativity

To prove Theorem 1, additional definitions are needed.

Definition 14. *Happened-before graph (HBG)*
Given a CEG G=(V, E) and $O_k \in V$, HB(G, O_k)=(V', E') is a happened-before graph *of G for O_k,* iff*: $V' \subset V$ consists of operations happened before O_k , i.e., V'=$\{O_x \mid (O_x \rightarrow O_k) \in E$ and $O_x \in V\}$, and $E' \subset E$ consists of all relations composed of operations in V', i.e., E'=$\{e_x$=$(O_a \rightarrow O_b) \mid e_x \in E$ and O_a, $O_b \in V'\} \cup \{e_y$=$(O_c \| O_d) \mid e_y \in E$ and O_c, $O_d \in V'\}$.*

An HBG is a subgraph of a CEG. Lemma 3 shows that an HBG is also a CEG.

Lemma 3. *For a CEG $G = (V, E)$, $G_{HB} = (V_{HB}, E_{HB}) = HB(G$, $O_k)$ for $O_k \in V$ is also a CEG.*

Proof. By Definition 14, G_{HB} inherits all the edges belonging to V_{HB} from G, so has either a directed or an undirected edge for every pair of vertices; thus, G_{HB} is a CEG.

To illustrate, the HBG for O_4 of the CEG G of Fig. 3, G_{HB} = $HB(G, O_4)$ = (V_{HB}, E_{HB}) is also a CEG; i.e., V_{HB}=$\{O_1$, O_2, $O_3\}$ and E_{HB} = $\{O_1 \| O_2, O_1 \| O_3, O_2 \| O_3\}$. In addition, for $G_{HB} = HB(G, O_k)$, let $\overline{G_{HB}} = (\overline{V_{HB}}, \overline{E_{HB}})$ be defined as follows: $\overline{V_{HB}} = V - V_{HB} - \{O_k\}$, $\overline{E_{HB}} = \{e_x = (O_a \rightarrow O_b) \mid e_x \in E$ and O_a, $O_b \in \overline{V_{HB}}\} \cup \{e_y = (O_c \| O_d) \mid e_y \in E$ and O_c, $O_d \in \overline{V_{HB}}\}$. Note $\overline{G_{HB}}$ is also complete and transitive; that is, a CEG. Next, we present the last definition for proving Theorem 1.

Definition 15. *Preceding operation set*
*Given a CES **s**: $O_1 \mapsto \cdots \mapsto O_{k-1} \mapsto O_k \mapsto \cdots \mapsto O_n$, the preceding operation set of O_k, $P(\mathbf{s}, O_k)$, is a set of operations executed before O_k in **s**, i.e., $P(\mathbf{s}, O_k)$=$\{O_1, \cdots, O_{k-1}\}$.*

Finally, we present the proof of Theorem 1 as follows:

Proof. (Proof by contradiction)
Let $|V| = n$. Assume the contrary there are two CESes $\mathbf{s}_1$, $\mathbf{s}_2 \in S(G)$ that do not converge and are given as $\mathbf{s}_1$: $a_1 \mapsto \cdots \mapsto a_k \mapsto \cdots \mapsto a_n$, $\mathbf{s}_2$: $b_1 \mapsto \cdots \mapsto b_k \mapsto \cdots \mapsto b_n$ for a_k, $b_k \in V (1<k<n)$. If $RS_0 \overset{\mathbf{s}_1}{\Rightarrow} RS_1$, $RS_0 \overset{\mathbf{s}_2}{\Rightarrow} RS_2$, then $RS_1 \neq RS_2$.
Perform the following procedure on each of $\mathbf{s}_1$ and $\mathbf{s}_2$.
Procedure: ⌜*For $\exists O \in V$, $O = a_x = b_y (1 \leq x, y \leq n)$, let V_{1P}=$P(\mathbf{s}_1, O)$, V_{2P}=$P(\mathbf{s}_2, O)$, and G_{HB}=$HB(G, O)$, respectively. Then, V_{1P} and V_{2P} are shown as below:*

$$\mathbf{s}_1 : \underbrace{a_1 \mapsto \cdots \mapsto a_{x-1}}_{V_{1P}=\{a_1,\ldots,a_{x-1}\}} \mapsto O \mapsto a_{x+1} \mapsto \cdots\cdots \mapsto a_n$$
$$\mathbf{s}_2 : \underbrace{b_1 \mapsto \cdots\cdots \mapsto b_{y-1}}_{V_{2P}=\{b_1,\ldots,b_{y-1}\}} \mapsto O \mapsto b_{y+1} \mapsto \ \cdots \mapsto b_n$$

Move $\forall a_k \in V_{1P} - V_{HB}$ *between* O *and* a_{x+1} *on* $\mathbf{s}_1$ *and* $\forall b_j \in V_{2L} - V_{HB}$ *between* O *and* b_{y+1} *on* $\mathbf{s}_2$*, respectively, with preserving their relevant orders. Then, the transformed execution sequences* $\mathbf{s}'_1$ *and* $\mathbf{s}'_2$ *are as follows:*

$$\mathbf{s}'_1 : \overbrace{\underbrace{c_1 \mapsto \cdots \mapsto c_m}_{V_{HB}=\{c_1,\cdots,c_m\}}}^{\mathbf{s}_{1L}} \mapsto O \mapsto \overbrace{\underbrace{d_1 \mapsto \cdots \mapsto d_p}_{V_{1P}-V_{HB}=\{d_1,\cdots,d_p\}} \mapsto a_{x+1} \mapsto \cdots \mapsto a_n}^{\mathbf{s}_{1R}}$$
$$\mathbf{s}'_2 : \overbrace{\underbrace{c_1 \mapsto \cdots \mapsto c_m}_{V_{HB}=\{c_1,\cdots,c_m\}}}^{\mathbf{s}_{2L}} \mapsto O \mapsto \overbrace{\underbrace{e_1 \mapsto \cdots \mapsto e_q}_{V_{2P}-V_{HB}=\{e_1,\cdots,e_q\}} \mapsto b_{y+1} \mapsto \cdots \mapsto b_n}^{\mathbf{s}_{2R}}$$

In $\mathbf{s}'_1$ *and* $\mathbf{s}'_2$, $\forall d \in V_{1P} - V_{HB}$ *on* $\mathbf{s}'_1$ *and* $\forall e \in V_{2P} - V_{HB}$ *on* $\mathbf{s}'_2$*, which are concurrent with both* O *and* $\forall c \in V_{HB}$*, can be moved due to OC. Though* $\forall d$ *and* $\forall e$ *move between* O *and* a_{x+1} *and between* O *and* b_{y+1}*, respectively, happened-before relations can be preserved; thus, each of* $\mathbf{s}'_1$ *and* $\mathbf{s}'_2$ *is also a CES.*$\lrcorner$

Due to $\mathbf{s}_{1L}$, $\mathbf{s}_{2L} \in S(G_{HB})$ and $\mathbf{s}_{1R}$, $\mathbf{s}_{2R} \in S(\overline{G_{HB}})$, the procedure can perform on each of $\mathbf{s}_{1L}$, $\mathbf{s}_{2L}$ of G_{HB} and $\mathbf{s}_{1R}$, $\mathbf{s}_{2R}$ of $\overline{G_{HB}}$ again. If the procedure performs recursively until one or no operation remains in the result CESes, $\mathbf{s}_1$ and $\mathbf{s}_2$ are transformed into the same CES because at least one operation is located at the same position on the two CESes once the procedure performs. Thus, if $RS_0 \overset{\mathbf{s}_1}{\Rightarrow} RS_1$ and $RS_0 \overset{\mathbf{s}_2}{\Rightarrow} RS_2$, then $RS_1 = RS_2$, contradicting the assumption.

B. Proofs for Consistency of RADTs

OC of RADTs can be proven by showing every pair of the operation types always commutes when they are concurrent. To show that, we must categorize all the possible conditions in which a pair of the concurrent types can be executed. In RADTs, the conditions can be defined with the two requisites:

- the s4vector order between two concurrent operations.
- the initial RADT state where the two operations perform.

Since the effect of an RADT operation is determined by a few objects in the RADT state, we specify the state of only the relevant objects there. In this way, we can prove OC of RFAs by the following lemmas and theorems.

Lemma 4. *For* W_1*:* Write*(*i_1*,* o_1*)* $\|$ W_2*:* Write*(*i_2*,* o_2*),* $W_1 \leftrightarrow W_2$*.*

Proof. Let the s4vector order between W_1 and W_2 be $W_1 < W_2$.[2] For i_1 and i_2, the followings cover all the cases:

(1) $i_1 = i_2$: Let o_{i_1} be the initial RFA[i_1]. Since s4vector orders are transitive, a total s4vector order among o_{i_1}, W_1, and W_2 is one of the followings:

a. $o_{i_1} < W_1 < W_2$: $o_{i_1} \overset{W_1}{\Rightarrow} o_1 \overset{W_2}{\Rightarrow} o_2$ and $o_{i_1} \overset{W_2}{\Rightarrow} o_2 \overset{W_1}{\Rightarrow} o_2$

b. $W_1 < o_{i_1} < W_2$: $o_{i_1} \overset{W_1}{\Rightarrow} o_{i_1} \overset{W_2}{\Rightarrow} o_2$ and $o_{i_1} \overset{W_2}{\Rightarrow} o_2 \overset{W_1}{\Rightarrow} o_2$

c. $W_1 < W_2 < o_{i_1}$: $o_{i_1} \overset{W_1}{\Rightarrow} o_{i_1} \overset{W_2}{\Rightarrow} o_{i_1}$ and $o_{i_1} \overset{W_2}{\Rightarrow} o_{i_1} \overset{W_1}{\Rightarrow} o_{i_1}$

a, **b**, and **c** demonstrate $W_1 \leftrightarrow W_2$, even if $i_1 = i_2$.

(2) $i_1 \neq i_2$: No conflict.

According to (1) and (2), $W_1 \leftrightarrow W_2$.

Theorem 2. *RFAs guarantee OC.*

Proof. By Lemma 4 and Theorem 1, OC holds in RFAs.

The proof of OC of RHTs should be shown by *Put*$\leftrightarrow$*Put*, *Put*$\leftrightarrow$*Remove*, and *Remove*$\leftrightarrow$*Remove*.

Lemma 5. *For* P_1*:* Put*(*k_1*,* o_1*)* $\|$ P_2*:* Put*(*k_2*,* o_2*),* $P_1 \leftrightarrow P_2$*.*

Proof. A *Put* on no Slot is comparable with a *Write* on a preceding `Element`, i.e., **a** in Lemma 4. Hence, $P_1 \leftrightarrow P_2$ is established as $W_1 \leftrightarrow W_2$ in Lemma 4.

Lemma 6. *For* P_1*:* Put*(*k_1*,*o_1*)* $\|$ R_2*:* Remove*(*k_2*),* $P_1 \leftrightarrow R_2$*.*

Proof. Note the `Slot` where R_2 performs always exists.

(1) $k_1 = k_2$: Letting the initial RHT[k_1] be o_{k_1}, only six cases are available due to transitivity of s4vector orders.

a. $o_{k_1} < P_1 < R_2$: $o_{k_1} \overset{P_1}{\Rightarrow} o_1 \overset{R_2}{\Rightarrow} \tau_2$ and $o_{k_1} \overset{R_2}{\Rightarrow} \tau_2 \overset{P_1}{\Rightarrow} \tau_2$

b. $o_{k_1} < R_2 < P_1$: $o_{k_1} \overset{P_1}{\Rightarrow} o_1 \overset{R_2}{\Rightarrow} o_1$ and $o_{k_1} \overset{R_2}{\Rightarrow} \tau_2 \overset{P_1}{\Rightarrow} o_1$

c. $P_1 < o_{k_1} < R_2$: $o_{k_1} \overset{P_1}{\Rightarrow} o_{k_1} \overset{R_2}{\Rightarrow} \tau_2$ and $o_{k_1} \overset{R_2}{\Rightarrow} \tau_2 \overset{P_1}{\Rightarrow} \tau_2$

d. $R_2 < o_{k_1} < P_1$: $o_{k_1} \overset{P_1}{\Rightarrow} o_1 \overset{R_2}{\Rightarrow} o_1$ and $o_{k_1} \overset{R_2}{\Rightarrow} o_{k_1} \overset{P_1}{\Rightarrow} o_1$

e. $P_1 < R_2 < o_{k_1}$: $o_{k_1} \overset{P_1}{\Rightarrow} o_{k_1} \overset{R_2}{\Rightarrow} o_{k_1}$ and $o_{k_1} \overset{R_2}{\Rightarrow} o_{k_1} \overset{P_1}{\Rightarrow} o_{k_1}$

f. $R_2 < P_1 < o_{k_1}$: $o_{k_1} \overset{P_1}{\Rightarrow} o_{k_1} \overset{R_2}{\Rightarrow} o_{k_1}$ and $o_{k_1} \overset{R_2}{\Rightarrow} o_{k_1} \overset{P_1}{\Rightarrow} o_{k_1}$

(2) $k_1 \neq k_2$: No conflict.

According to (1) and (2), $P_1 \leftrightarrow R_2$.

Lemma 7. *For* R_1*:* Remove*(*k_1*)* $\|$ R_2*:* Remove*(*k_2*),* $R_1 \leftrightarrow R_2$*.*

Proof. Let the s4vector order between R_1 and R_2 as $R_1 < R_2$. For k_1 and k_2, the followings cover all cases:

(1) $k_1 = k_2$: Assume that o_{k_1} is the initial RHT[k_1], and τ_1 and τ_2 are the tombstones executed by R_1 and R_2, respectively. Then, the two *Remove*s work commutatively as follows:

a. $o_{k_1} < R_1 < R_2$: $o_{k_1} \overset{R_1}{\Rightarrow} \tau_1 \overset{R_2}{\Rightarrow} \tau_2$ and $o_{k_1} \overset{R_2}{\Rightarrow} \tau_2 \overset{R_1}{\Rightarrow} \tau_2$

[2]In RFAs and RHTs, if an operation or an object appears at either side of '<', it is a surrogate of its s4vector, i.e., $\vec{s}_O$ or $\vec{s}_p$ of `Element` or `Slot` defined in Section 5.

b. $R_1{<}o_{k_1}{<}R_2$: $o_{k_1}\overset{R_1}{\Rightarrow}o_{k_1}\overset{R_2}{\Rightarrow}\tau_2$ and $o_{k_1}\overset{R_2}{\Rightarrow}\tau_2\overset{R_1}{\Rightarrow}\tau_2$

c. $R_1{<}R_2{<}o_{k_1}$: $o_{k_1}\overset{R_1}{\Rightarrow}\, o_{k_1}\overset{R_2}{\Rightarrow}o_{k_1}$ and $o_{k_1}\overset{R_2}{\Rightarrow}o_{k_1}\overset{R_1}{\Rightarrow}o_{k_1}$

(2) $k_1 \neq k_2$: No conflict.

According to (1) and (2), $R_1{\leftrightarrow}R_2$.

Theorem 3. *RHTs guarantee OC.*

Proof. By Lemma 5–7 and Theorem 1, OC holds in RHTs.

Proving OC in RGAs is more difficult than in RFAs and in RHTs owing to the complexity to categorizing RGA states where *Insert*s work. Since the effect of an *Insert* is determined based on $\overrightarrow{s}_k$s of the Nodes next to its reference Node, notations to express RGA states need to be effectively defined. Consider an RGA state given as $[\cdots o_x s_a o_y \cdots]$. Here, o_x and o_y are Nodes while '$\cdots$' is a sequence of Nodes (or an empty sequence) which can have no causality with some given operations. Contrarily, for a sequence that might have causality, we use the prefix 's', e.g., s_a, and represent the ith Node in s_a as $s_a[i]$. In addition, the notations of the set theory such as $\forall$ and $\exists$, will be used to express the properties of Nodes in a sequence. In the following lemmas, we assume the causality between Nodes and given operations is preserved, and all the RGA operations are given in the forms of the remote ones.

Lemma 8. *For* I_1*:* Insert*(*i_1*,* o_1*)* || I_2*:* Insert*(*i_2*,* o_2*),* $I_1{\leftrightarrow}I_2$.

Proof. Let the s4vector order between I_1 and I_2 as $I_1{<}I_2$. An *Insert* compares $\overrightarrow{s}_O$ with $\overrightarrow{s}_k$s of some Nodes next to its reference Node. The range of the compared Nodes begins from the exact next Node of its reference and ends in the first encountered Node preceding $\overrightarrow{s}_O$. When the compared ranges of two *Insert*s do not overlap, the two *Insert*s commute; thus, we regards that as no conflict.

(1) $i_1 = i_2$: Let an initial state of an RGA where I_1 and I_2 will be executed be $RS_1{=}[\cdots o_{i_1}\cdots]$. For $I_1{\mapsto}I_2$ and $I_2{\mapsto}I_1$, RS_1 transitions as follows:

a. $I_1{\mapsto}I_2$: $RS_1\overset{I_1}{\Rightarrow}[\cdots o_{i_1}s_x o_1 o_x\cdots]\overset{I_2}{\Rightarrow}[\cdots o_{i_1}s_{x_1}o_2 s_{x_2}o_1 o_x\cdots]$
I_1 first modifies RS_1 so that s_x and o_x be $o_1{<}\forall o \in s_x$ and $o_x{<}o_1$, respectively.[3] Due to $I_1{<}I_2$, o_2 is always placed in front of o_1, thus s_{x_1} and s_{x_2} become $o_1{<}o_2{<}\forall o \in s_{x_1}$, and $s_{x_2}{=}\varnothing$ or $s_{x_2}[0]{<}o_2$, and $s_{x_1}\cup s_{x_2}{=}s_x$, respectively.

b. $I_2{\mapsto}I_1$: $RS_1\overset{I_2}{\Rightarrow}[\cdots o_{i_1}s_y o_2 o_y\cdots]\overset{I_1}{\Rightarrow}[\cdots o_{i_1}s_y o_2 s_z o_1 o_z\cdots]$
I_2 first modifies RS_1 to be $o_2{<}\forall o \in s_y$ and $o_y{<}o_2$. Due to transitivity, $o_1{<}o_2{<}\forall o \in s_y$; thus, o_1 is inserted after o_2. In addition, if there are some Nodes succeeding I_1 after o_2, i.e., s_z, o_1 is located after them ($o_1{<}\forall o \in s_z$ and $o_z{<}o_1$).

In the final results, the transitivity of s4vector orders ensures $s_{x_1}{=}s_y$, $s_{x_2}{=}s_z$ and $o_x{=}o_z$; thus, $I_1{\leftrightarrow}I_2$ for $i_1 = i_2$.

(2) $i_1 \neq i_2$: All the possible initial RGA states, where I_1 and I_2 will be executed in different ways, can be classified into six cases of the following **a** – **f**:

a. $RS_2{=}[\cdots o_{i_1}s_x o_{i_2}\cdots]$ for $s_x{\neq}\varnothing$ and $\exists o{\in}s_x{<}I_1$:no conflict.

b. $RS_2{=}[\cdots o_{i_1}s_x o_{i_2}\cdots]$ for $s_x{\neq}\varnothing$ and $I_1{<}\forall o{\in}s_x$: This is the compliment of (2)-**a** for $s_x{\neq}\varnothing$ on RS_2. Placing o_1 somewhere at the back of s_x, the effect of I_1 varies according to the s4vector order between I_1 and o_{i_2} as follows:

- $o_{i_2} \prec I_1$: No conflict.
- $I_1 \prec o_{i_2}$: Because I_1 compares $\overrightarrow{s}_O$ with the next Nodes of o_{i_2}, it is equivalent with (1) after s_x; thus, $RS_2 \overset{I_1\mapsto I_2, I_2\mapsto I_1}{\Longrightarrow} [\cdots o_{i_1}s_x o_{i_2}s_{y_1}o_2 s_{y_2}o_1 o_y\cdots]$ for $o_1 \prec o_2 \prec \forall o \in s_{y_1}$, $s_{y_2}{=}\varnothing$ or $s_{y_2}[0]{\prec}\, o_2$, $o_1 \prec \forall o{\in}s_{y_2}$ and $o_y \prec o_2$.

c. $RS_3{=}[\cdots o_{i_1}o_{i_2}\cdots]$:

- $o_{i_2} \prec I_1$: No conflict.
- $I_1 \prec o_{i_2}$: Equivalent with (1) from o_{i_2}; thus, $RS_3 \overset{I_1\mapsto I_2, I_2\mapsto I_1}{\Longrightarrow} [\cdots o_{i_1}o_{i_2}s_{x_1}o_2 s_{x_2}o_1 o_x\cdots]$ for $o_1 \prec o_2 \prec \forall o{\in}s_{x_1}$, $s_{x_2}{=}\varnothing$ or $s_{x_2}[0]{\prec}\, o_2$, $o_1 \prec \forall o{\in}s_{x_2}$, and $o_x \prec o_1$.

d. $RS_4{=}[\cdots o_{i_2}s_x o_{i_1}\cdots]$ for $s_x{\neq}\varnothing$ and $\exists o{\in}s_x{<}I_2$:no conflict.

e. $RS_4{=}[\cdots o_{i_2}s_x o_{i_1}\cdots]$ for $s_x{\neq}\varnothing$ and $I_2 \prec \forall o \in s_x$:

- $o_{i_1} \prec I_2$: No conflict.
- $I_2 \prec o_{i_1}$: Equivalent with (1) after s_x; thus, $RS_4 \overset{I_1\mapsto I_2, I_2\mapsto I_1}{\Longrightarrow} [\cdots o_{i_2}s_x o_{i_1}s_{y_1}o_2 s_{y_2}o_1 o_y\cdots]$ for $o_1 \prec o_2 \prec \forall o{\in}s_{y_1}$, $s_{y_2}{=}\varnothing$ or $s_{y_2}[0] \prec o_2$, $o_1 \prec \forall o{\in}s_{y_2}$, and $o_y \prec o_1$.

f. $RS_5{=}[\cdots o_{i_2}o_{i_1}\cdots]$

- $o_{i_1} \prec I_2$: No conflict.
- $I_2 \prec o_{i_1}$: Equivalent with (1) after o_{i_1}; thus, $RS_5 \overset{I_1\mapsto I_2, I_2\mapsto I_1}{\Longrightarrow} [\cdots o_{i_2}o_{i_1}s_{x_1}o_2 o_{x_2}o_1 o_x\cdots]$ for $o_1 \prec o_2 \prec \forall o{\in}s_{x_1}$ $s_{x_2}{=}\varnothing$ or $s_{x_2}[0] \prec o_2$, $o_1 \prec \forall o{\in}s_{x_2}$, and $o_x \prec o_1$.

The states from RS_2 to RS_5 cover all the possible initial RGA states where distinct o_{i_1} and o_{i_2} appear, thus $I_1{\leftrightarrow}I_2$ is verified for $i_1 \neq i_2$.

According to (1) and (2), $I_1 \leftrightarrow I_2$.

Lemma 9. *For* I_1*:* Insert*(*i_1*,* o_1*)* || D_2*:* Delete*(*i_2*),* $I_1{\leftrightarrow}D_2$.

Proof. For $D_2{\mapsto}I_1$, D_2 modifies only $\overrightarrow{s}_p$ of its target Node, and the Node is preserved as a tombstone until I_1 is executed. Since I_1 is affected by only $\overrightarrow{s}_k$s of Nodes, $I_1{\leftrightarrow}D_2$.

Lemma 10. *For* I_1*:* Insert*(*i_1*,* o_1*)* || U_2*:* Update*(*i_2*,* o_2*),* $I_1{\leftrightarrow}U_2$.

Proof. Due to the same reason of Lemma 9, $I_1{\leftrightarrow}U_2$.

Lemma 11. *For* D_1*:* Delete*(*i_1*)* || D_2*:* Delete*(*i_2*),* $D_1{\leftrightarrow}D_2$.

Proof. Trivial.

[3]In RGAs, if any Node appears at either side of '$\prec$', it is a surrogate of $\overrightarrow{s}_k$ of the Node.

Lemma 12. *For D_1:* Delete*(i_1)* || *U_2:* Update*(i_2, o_2), $D_1 \leftrightarrow U_2$.*

Proof. Though i_1=i_2, due to $U_2 \dashrightarrow D_1$, $D_1 \leftrightarrow U_2$.

Lemma 13. *For U_1:* Update*(i_1, o_1)* || *U_2:* Update*(i_2, o_2), $U_1 \leftrightarrow U_2$.*

Proof. Since U_1 and U_2 for $i_1 = i_2$ are executed according to the s4vector orders, $U_1 \leftrightarrow U_2$.

Theorem 4. *RGAs guarantee OC.*

Proof. By Lemma 8–13 and Theorem 1, OC holds in RGAs.

References

[1] C. A. Ellis, S. J. Gibbs, G. Rein, Groupware: some issues and experiences, Communications of the ACM 34 (1).

[2] Y. Saito, M. Shapiro, Optimistic replication, ACM Computing Surveys 37 (1).

[3] S. Greenberg, D. Marwood, Real time groupware as a distributed system: concurrency control and its effect on the interface, in: Proceedings of ACM conference on Computer Supported Cooperative Work (CSCW), 1994.

[4] K. P. Birman, A. Schiper, P. Stephonson, Lightweight causal and atomic group multicast, ACM Transactions on Computer Systems 9 (3).

[5] P. A. Jensen, N. R. Soparkar, A. G. Mathur, Characterizing multicast orderings using concurrency control theory, in: Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS), 1997.

[6] P. A. Bernstein, N. Goodman, An algorithm for concurrency control and recovery in replicated distributed databases, ACM Transactions on Database Systems 9 (4).

[7] J. Gray, P. Helland, P. O'Neil, D. Shasha, The dangers of replication and a solution, in: Proceedings of ACM international conference on Management of Data (SIGMOD), 1996.

[8] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, C. H. Hauser, Managing update conflicts in Bayou, a weakly connected replicated storage system, in: Proceedings of ACM Symposium on Operating Systems Principles (SOSP), 1995.

[9] R. H. Thomas, A majority consensus approach to concurrency control for multiple copy databases, ACM Transactions on Database Systems 4 (2).

[10] S. Mishra, L. L. Peterson, R. D. Schlichting, Implementing fault-tolerant replicated objects using psync, in: Proceedings of Symposium on Reliable Distributed Systems, 1989.

[11] M. J. Fischer, A. Michael, Sacrificing serializability to attain availability of data in an unreliable network, in: Proceedings of ACM Symposium on Principle of Database Systems (PODS), 1982.

[12] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, D. Terry, Epidemic algorithms for replicated database maintenance, in: Proceedings of ACM Symposium on Principles of Distributed Computing (PODC), 1987.

[13] M. Ressel, D. Nitsche-Ruhland, R. Gunzenhäuser, An integrating, transformation-oriented approach to concurrency control and undo in group editors, in: Proceedings of ACM conference on Computer Supported Cooperative Work (CSCW), 1996.

[14] C. Sun, C. S. Ellis, Operational transformation in real-time group editors: issues, algorithms, and achievements, in: Proceedings of ACM conference on Computer Supported Cooperative Work (CSCW), 1998.

[15] C. Sun, D. Chen, Consistency maintenance in real-time collaborative graphics editing systems, ACM Transactions on Computer-Human Interactaction 9 (1) (2002) 1–41.

[16] W. E. Weihl, Commutativity-based concurrency control for abstract data types, IEEE Trans. on Computers 37 (12).

[17] B. R. Badrinath, K. Ramamritham, Semantics-based concurrency control: Beyond commutativity, ACM Transactions on Database Systems 17 (1).

[18] C. A. Ellis, S. J. Gibbs, Concurrency control in groupware systems, in: Proceedings of ACM international conference on Management of Data (SIGMOD), 1989.

[19] C. Sun, X. Jia, Y. Zhang, Y. Yang, D. Chen, Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems, ACM Transactions on Computer-Human Interaction 5 (1).

[20] R. Li, D. Li, A new operational transformation framework for real-time group editors, IEEE Transaction on Parallel and Distributed Systems 18 (3).

[21] G. Oster, P. Molli, P. Urso, A. Imine, Tombstone transformation functions for ensuring consistency in collaborative editing systems, International Conference on Collaborative Computing: Networking, Applications and Worksharing (2006) 1–10.

[22] D. Li, R. Li, A performance study of group editing algorithms, in: Proceedings of International Conference on Parallel and Distributed Systems (ICPADS), 2006.

[23] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Comm. of ACM 21 (7).

[24] K. Birman, R. Cooper, The isis project: real experience with a fault tolerant programming system, SIGOPS Operating Systtems Review 25 (2).

[25] V. Balakrishnan, Graph Theory, McGraw-Hill, New York, 1997.

[26] R. A. Golding, Weak-consistency group communication and membership, Ph.D. thesis, University of California, Santa Cruz (1992).
URL `citeseer.ist.psu.edu/golding92weakconsistency.html`

[27] B. Lushman, G. V. Cormack, Proof of correctness of ressel's adopted algorithm, Info. Process. Letters 86 (6).

[28] M. Suleiman, M. Cart, J. Ferrié, Concurrent operations in a distributed and mobile collaborative environment, in: Proceedings of the Fourteenth International Conference on Data Engineering (ICDE), IEEE Computer Society, 1998, pp. 36–45.

[29] N. Vidot, M. Cart, J. Ferrié, M. Suleiman, Copies convergence in a distributed real-time collaborative environment, in: Proceedings of ACM conference on Computer Supported Cooperative Work (CSCW), 2000.

[30] D. Li, R. Li, Preserving operation effects relation in group editors, in: Proceedings of ACM conference on Computer Supported Cooperative Work (CSCW), 2004.

[31] R. Li, D. Li, Commutativity-based concurrency control in groupware, International Conference on Collaborative Computing: Networking, Applications and Worksharing.

[32] G. Oster, P. Urso, P. Molli, A. Imine, Data consistency for p2p collaborative editing, in: Proceedings of ACM conference on Computer Supported Cooperative Work (CSCW), 2006.

[33] H.-G. Roh, J. Kim, J. Lee, How to design optimistic operations for peer-to-peer replication, in: Joint Conference on Information Sciences (JCIS), 2006.

[34] M. Shapiro, N. Preguiça, Designing a commutative replicated data type, Tech. rep., inria, `http://hal.inria.fr/inria-00177693/` (Oct. 2007).

[35] N. Preguiça, J. M. Marqués, M. Shapiro, M. Letia, A commutative replicated data type for cooperative editing, in: Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS), 2009.

[36] S. Weiss, P. Urso, P. Molli, Logoot : a scalable optimistic replication algorithm for collaborative editing on P2P networks, in: Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS), IEEE Computer Society, 2009.

[37] R. Prakash, M. Raynal, M. Singhal, An adaptive causal ordering algorithm suited to mobile computing environments, Journal of Parallel Distributed Computing 41 (2) (1997) 190–204.

[38] G. Oster, P. Urso, P. Molli, A. Imine, Real time group editors without operational transformation, Rapport de recherche RR-5580, INRIA (May 2005).

[39] Y. Yu, P. Manolios, L. Lamport, Model checking tla+ specifications, in: CHARME '99: Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods, Springer-Verlag, 1999, pp. 54–66.