J. Parallel Distrib. Comput. 71 (2011) 354–368

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Replicated abstract data types: Building blocks for collaborative applications

Hyun-Gul Roh[a,*], Myeongjae Jeon[b], Jin-Soo Kim[c], Joonwon Lee[c]

[a] Department of Computer Science, KAIST, Daejeon, Republic of Korea
[b] Department of Computer Science, Rice University, Houston, TX, United States
[c] School of Information and Communication Engineering, Sungkyunkwan University (SKKU), Suwon, Republic of Korea

ARTICLE INFO

Article history:
Received 23 July 2009
Received in revised form
11 October 2010
Accepted 4 December 2010
Available online 15 December 2010

Keywords:
Distributed data structures
Optimistic replication
Replicated abstract data types
Optimistic algorithm
Collaboration

ABSTRACT

For distributed applications requiring collaboration, responsive and transparent interactivity is highly desired. Though such interactivity can be achieved with optimistic replication, maintaining replica consistency is difficult. To support efficient implementations of collaborative applications, this paper extends a few representative abstract data types (ADTs), such as arrays, hash tables, and growable arrays (or linked lists), into replicated abstract data types (RADTs). In RADTs, a shared ADT is replicated and modified with optimistic operations. Operation commutativity and precedence transitivity are two principles enabling RADTs to maintain consistency despite different execution orders. Especially, replicated growable arrays (RGAs) support insertion/deletion/update operations. Over previous approaches to the optimistic insertion and deletion, RGAs show significant improvement in performance, scalability, and reliability.

1. Introduction

Optimistic replication is an essential technique for interactive collaborative applications [8,30]. To illustrate replication issues in collaboration, consider the following scenario in an editorial office publishing a daily newspaper.

A number of pressmen are editing a newspaper using computerized collaboration tools. Each of them is browsing and editing pages consisting of news items, such as text, pictures, and tables. When a writer collaborates on editing the same article with others, his local interaction is never blocked, but interactions of the others are shown to him as soon as possible. After all interactions cease, all the copies of the newspaper become consistent.

Human users, the subjects of these applications, prefer high responsiveness and transparent interactivity to strict consistency [8,30,13]. Responsiveness means how quickly the effect of an operation is delivered to users, and interactivity is how freely operations can be performed. Optimistic operations that are executed first at each local site enable to achieve these properties, but consistency should be maintained as sites execute operations in different orders.

Optimistic replication contrasts with pessimistic concurrency control protocols [30], such as serialization [5,14] or locking [3,12]. Even if a global locking protocol allows optimistic operations [13], it not only requires a state rollback mechanism, but also damages interactivity due to the nature of the locking protocol. There has been research on genuine optimistic replication oriented to specific services, such as a replicated databases [36,37], Usenet [21,9,6], and a collaborative textual or graphical editor [8,27,34,33]. However, these service-oriented techniques are inflexible for various complex functions of modern interactive applications; e.g., electronic blackboards, games, CAD tools, and office tools such as Microsoft Office and Google Docs, all of which can be extended for collaboration.

Interactive applications, e.g., CAD tools designing for skyscrapers or spaceships, have demanded managing of indeterminate data; one data may consist of limited elements, another data may need quick access to unlimited elements, and the other data may contain ordered elements frequently inserted and deleted. Sensible developers would make use of various abstract data types (ADTs) to reflect such a demand. When those applications are extended for collaboration, however, developers may abandon to use ADTs for shared data owing to inconsistency. Hence, we suggest *replicated abstract data types* (RADTs), a novel class of ADTs that can be used as building blocks for collaborative applications.

RADTs are multiple copies of a shared ADT replicated over distributed sites. RADTs provide a set of primitive operations corresponding to that of normal ADTs, concealing the details of consistency maintenance. RADTs ensure *eventual consistency* [30], a weak consistency model for achieving responsiveness and interactivity. By imposing no constraint on operation delivery

* Corresponding author.
E-mail addresses: hgroh@calab.kaist.ac.kr, knowhunger@gmail.com (H.-G. Roh).

0743-7315/$ – see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2010.12.006

except causal dependency, we accommodate RADT deployment in general environments. This allows a site to execute operations in any causal order. We model such executions and explore principles to achieve eventual consistency.

This paper suggests two principles that lead to successful designs of non-trivial RADTs. First, *operation commutativity* (OC) requires that every pair of concurrent operations commutes. Though the concept of commutativity was discussed in many distributed systems [39,1,27], it was not fully assimilated. We formally prove that OC guarantees eventual consistency for all possible execution orders; so, we *mandate* RADT operations to satisfy OC. Second, *precedence transitivity* (PT) requires that all precedence rules are transitive. RADTs require precedence rules to reconcile conflicting intentions. PT is a guideline on how to design remote operations so that RADT operations satisfy OC and preserve their intentions. In short, OC is a sufficient condition to ensure eventual consistency, while PT is a principle for exploiting OC.

We present efficient implementations of three RADTs: replicated fixed-size arrays (RFAs), replicated hash tables (RHTs), and replicated growable arrays (RGAs). Although some key ideas for RFAs and RHTs were already present in the literature [36,37,21,9,6], we introduce them again 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 problem of *optimistic insertions and deletions* into a replicated ordered set. As these operations have been highly desired in collaborative applications [8,13], the operational transformation (OT) framework is the classic approach for these operations. Various OT methods have been introduced [7,27,35,34,19,22], and one of them is adopted by a web collaboration tool Google Wave [11]. However, the OT framework has difficulty in verifying correctness, and an evaluation study on recent OT methods reports that their performance and scalability are poor and non-negligible [18].

Thanks to OC and PT, RGAs provide full correctness verification not only for insertions and deletions, but also for *updates* [29]. In addition, RGAs are superior to most of the previous works in complexity, scalability, and reliability. Whereas remote operations of OT methods generally have quadratic time-complexity, remote RGA operations can perform in $O(1)$ time by proposing the s4vector index (SVI) scheme. Our evaluation shows that operations needing about hundreds of ms in OT methods take only dozens of μs in RGAs. Due to the optimal remote operations and the fixed-size s4vectors, RGAs scale. Additionally, RGAs have a chance to enhance reliability by autonomous causality validations of the SVI scheme. RGAs, therefore, can be a better alternative of OT methods.

Section 2 describes three RADTs and their inconsistency problems. Sections 3 and 4 formalize OC and PT, respectively. Concrete algorithms of RADTs are proposed in Section 5. We survey the related work in Section 6, and contrast RGAs with previous work in Section 7. Section 8 presents the performance evaluation, and we conclude this paper in Section 9.

2. Problem definition

2.1. Preliminary: causality preservation among operations

The replication system discussed in this paper is characterized by a set of distributed sites and operations as shown in the time–space diagram of Fig. 1 which describes the propagations and executions of operations. Lamport presented two definitions for causality [15]: happened-before relation (‘$\rightarrow$’) and concurrent relation (‘$\|$’). Given a time–space diagram consisting of n operations, all $\frac{n(n-1)}{2}$ relations are obtained; every pair of distinct operations is in either of the two relations.

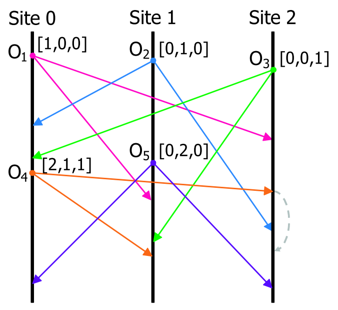

Fig. 1. A time–space diagram in which three sites participate. A vector on the right of each operation is its vector clock.

While no uniquely correct order is defined for concurrent operations, partial orders defined by happened-before relations need to be preserved at every site [27,35] owing to the causality that might exist; e.g., imagine O_4 is to delete the object inserted by O_2 in Fig. 1. Vector clocks can ensure such causality (or causal execution orders) by preserving happened-before relations [5,35]. Following Birman et al.'s CBCAST scheme [5], in our replication system consisting of N sites, site i updates its own N-tuple vector clock $\overrightarrow{v}_i$ according to lines 2, 8, 10, and 16 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 vector clock $\overrightarrow{v}_O$, O is causally ready if $\overrightarrow{v}_O[j] = \overrightarrow{v}_i[j] + 1$ and $\overrightarrow{v}_O[k] \le \overrightarrow{v}_i[k]$ for $0 \le k \le N - 1$ and $k \neq j$. To illustrate, consider site 2 in Fig. 1. After O_1's execution, site 2 has $\overrightarrow{v}_2 = [1, 0, 1]$. When O_4 arrives at site 2 with $\overrightarrow{v}_{O_4} = [2, 1, 1]$, it is causally unready; thus, it is delayed until executing O_2. According to Birman and Cooper [4], CBCAST is 3–5 times faster than ABCAST that supports a 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 one issued locally, whereas a *remote operation* is one received from a remote site.
- At a site, every local operation is immediately executed on the RADT of the site according to its *local algorithm*.
- Every local operation modifying the 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 its *remote algorithm* when it is causally ready.

For the operations modifying RADTs, two kinds of algorithms are given: local and remote. In RADTs, local algorithms are almost the same as those of the normal ADTs, but *remote algorithms might operate differently in order to maintain consistency*. Since an operation is executed first at its local site and later at remote sites, different sites execute operations in different orders. Section 3 will go into detail on operation execution.

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, and thus consistency is not defined for *Reads*. Instead, RADTs guarantee an *eventual consistency model* which is defined only for the operations modifying replica states as follows.

Algorithm 1 The main control loop of RADTs at site *i*

```
MainLoop():
  ∀k: $\vec{v}_i[k]$ := 0;
  i := this site ID;
  initialize queue Q;
  initialize RADT;
  while(not aborted)
     if(O is a local operation but not Read)
       $\vec{v}_i[i] := \vec{v}_i[i] + 1$;
       if(RADT.localAlgorithm(O) = true) broadcast (O, $\vec{v}_i$);
       else $\vec{v}_i[i] := \vec{v}_i[i] - 1$;
     if(O is a Read) RADT.localAlgorithm(O);
     if(an operation O arrives with $\vec{v}_O$ from site j)
       enqueue set (O, $\vec{v}_O$) into Q;
       while(there is a causally ready set in Q)
            (O, $\vec{v}_O$) := dequeue the set from Q;
            ∀k: $\vec{v}_i[k]$ := max($\vec{v}_i[k]$, $\vec{v}_O[k]$);
            RADT.remoteAlgorithm(O);
```

Definition 1 (*Eventual Consistency of RADTs*). Eventual consistency is the condition that all the states of RADTs are identical after sites have executed the same set of modifying operations from the same initial states regardless of any causal execution order of the operations at each site.

In this model, even if every site executes the same *Read* at the exact same time, sites might read different values. As in the scenario of editing newspapers, however, if consistency of shared objects, displayed to human users, accords with this model, momentary inconsistency is acceptable to human users. In particular, the eventual consistency is appropriate for achieving high responsiveness and transparent interactivity; thus, it has been widely accepted in collaborative applications [7,8,13,30].

2.3. Definitions of RADTs and inconsistency problems

This paper focuses on three kinds of representative ADTs and extends them into RADTs: fixed-size array into RFA, hash table into RHT, and growable array or linked list into RGA. A real example of the growable array is the `Vector` class of JAVA or STL. As shown in Fig. 2, their functionality is prevalently demanded in such applications as in the newspaper editing scenario. Note that multiple RADTs can handle the same object in memory. For example, news items can be managed not only with RHTs as in Fig. 2 but also with RGAs to display a number of news items on a page. To manage the overlapping order of news items, i.e., *Z*-order, consistently over all sites, RGAs can be used even if some items are inserted or deleted. Therefore, just like linked lists or growable arrays are widely used, RGAs will be highly demanded in collaborative applications. However, if remote algorithms are not properly designed, RADTs suffer pathological inconsistency problems. Below, we precisely define each RADT and show its potential inconsistency problems when it executes operations *naïvely*.

A replicated fixed-size array (RFA) is a fixed number of *element*s with OPTYPE = {*Write*, *Read*}. An element is an object container of RFAs, which contains one object. In Algorithm 2, the local algorithm of *Write*(int *i*, Object *o*) replaces the object at the *i*th element with a new one *o*. In RFAs, different execution orders lead to inconsistency. For example, if three operations of Fig. 3 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.

Hash tables are extended into replicated hash tables (RHTs), which access shared objects in *slot*s by hashing unique keys with OPTYPE = {*Put*, *Remove*, *Read*}, as 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

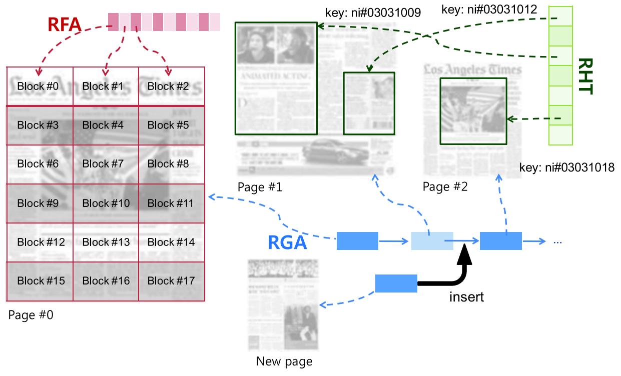

Fig. 2. A usage example of RADTs in the newspaper editing scenario. A newspaper page might be divided into a fixed number of blocks, which can be managed by an RFA. An RHT makes it possible to rapidly access news items with unique keys. An RGA enables pages to be inserted and deleted, respecting the order of existing pages.

Algorithm 2 The local algorithms for RFA operations

```
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;
```

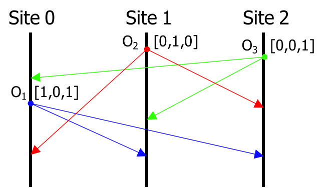

Fig. 3. A simple example of a time–space diagram.

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 containers kept up after *Removes* [21,9,6]. Despite the tombstone, if the remote algorithms are the same as the local ones, RHTs might diverge. Consider Fig. 3 again, assuming 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. Finally, sites 1 and 2 have the tombstone for k_1 while site 0 has o_2 for k_1.

Algorithm 3 The local algorithms for RHT operations

```
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;
```

A replicate growable array (RGA) of primary interest to this paper 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. Since *nodes*, the object containers of RGAs, are ordered and inserted/deleted, an RGA adopts a linked list internally for efficiency. *Update* is also required because *Insert*s cannot update nodes and modifications on nodes should be explicitly 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 OPTYPE with the local pointer operations, which are parameterized with node pointers instead of integers by using *findlink* in Algorithm 4. This paper, however, mainly deals with the operations of integer indices since their semantics have been frequently studied in collaborative applications [7,27,35,34,19,22]. Note that local RGA operations on tombstones fail by *findlist* or *findlink* and are 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. 3 are given as O_1: *Update*$(2, o_x)$, O_2: *Insert*$(1, o_y)$, and O_3: *Insert*$(1, o_z)$ and executed on an initial RGA $[o_1o_2]$ by the local algorithms.[1] After executing both O_2 and O_3, sites 1 and 2 have different results: $[o_1\boldsymbol{o}_z\boldsymbol{o}_yo_2]$ at site 1, and $[o_1\boldsymbol{o}_y\boldsymbol{o}_zo_2]$ at site 2. Here, only one must be chosen for consistency.

Algorithm 4 The local algorithms for RGA operations

```
findlist(int i):
    n := head of the linked list;
    int k := 0;
    while(n != nil)
        if(n.obj != nil)       // skip tombstones;
            if(i = ++k) return n;
        n := n.link;      // next node in the linked list;
    return nil;
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;
    return target_n.obj;
```

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 with parameter(s) on an RADT, its intention is the effect of its local algorithm on the RADT.

In RGAs, intentions can be violated at remote sites because *Insert*s and *Delete*s change integer indices of some nodes located behind their intended nodes. This intention violation problem was first addressed by Sun et al. [35]. In the example, although O_1 intends to replace o_z on $[o_1\boldsymbol{o}_zo_2]$ with o_x at its local site, O_1 of site 2 may update o_y on $[o_1\boldsymbol{o}_yo_zo_2]$, which is not the intention of O_1. RGAs may incur many other puzzling problems regarding intentions [19], but solve them in Sections 4 and 5.

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

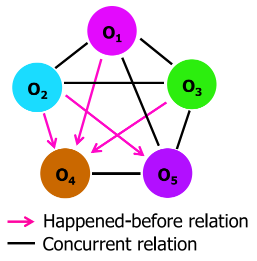

Fig. 4. CEG of the time–space diagram of Fig. 1.

The local RADT algorithms ensure the same responsiveness and interactivity as the normal ADTs. Note that local Algorithms 2–4 are *incomplete* since we present no exact details of the data structures yet. After introducing two principles, the remote algorithms, which mandate consistency maintenance, will be presented with the details of the data structures in Section 5.

3. Operation commutativity

RADTs allow sites to execute operations in different orders. To denote an execution order of two operations, we use '$\mapsto$'; 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 that O_a and O_b change a replica state RS_0 into RS_1 and then into RS_2 in order. We abbreviate this as $RS_0 \overset{O_a \mapsto O_b}{\Rightarrow} RS_2$. Though time–space diagrams, such as Figs. 1 and 3, 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. 4 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.*

Proof. According to the definitions of happened-before and concurrent relations [15], 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. 4, 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, $O_3 \mapsto O_1 \mapsto O_4 \mapsto O_2 \mapsto O_5$ violates causality because O_4 is executed before O_2, whose order is the reverse of edge $O_2 \rightarrow O_4$. A causality-preserved 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 $\mathbf{s} : O_1 \mapsto \cdots \mapsto O_n$ is a *causally executable sequence* (CES), *iff*: all the operations in V participate only once in $\mathbf{s}$, and *no* $O_j \rightarrow O_i$ for $1 \le i < j \le n$.

Unless all the edges in E are directed ones, a CEG has more than one CES. According to the system model, RADTs permit the executions of all possible CESes. Eventual consistency, therefore, is achieved if all the CESes lead to 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. 4. If a pair of adjacent operations on $\mathbf{s}_1$ is concurrent, another CES can be derived by swapping the order in the pair; e.g., if $O_1 \parallel O_2$ is swapped, $\mathbf{s}_1$ is transformed into another CES $\mathbf{s}_2$: $O_2 \mapsto O_1 \mapsto O_3 \mapsto O_4 \mapsto O_5$. 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 all the possible CESes derived from G can be transformed from $\mathbf{s}$ and find the condition that they yield the same result, eventual consistency will be guaranteed. This is the basic concept of operation commutativity (OC), which is developed from the commutative relation.

Definition 5. [Commutative Relation '$\leftrightarrow$']. Given concurrent O_a and O_b, they are in *commutative relation*, denoted as $O_a \leftrightarrow O_b$, *iff*: for a 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.

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 if $O_1 \parallel O_5$ 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 where $O_1 \parallel O_5$ are executed are different and because other operations may or may not intervene between them. Indeed, to make all the possible CESes consistent, the following condition is necessary.

Definition 6 (*Operation Commutativity (OC)*)**.** Given a CEG $G = (V, E)$, *operation commutativity* is established in G, *iff*: $O_a \leftrightarrow O_b$ for $\forall(O_a \parallel O_b) \in E$.

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. 4, $O_1 \leftrightarrow O_2$, $O_4 \leftrightarrow O_5$, $O_3 \leftrightarrow O_5$, 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.

Theorem 1. *If OC holds in a given CEG $G = (V, E)$, all the possible CESes of G executed from the same initial replica state eventually produce the same state.*

This theorem, proved in [29], 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, we show that each pair of concurrent operations actually commutes on all the replica states defined exhaustively (see [29] for detail).

However, OC suggests no guideline to exploit OC itself. In the next section, precedence transitivity gives a practical way to achieve OC for the RADT operations.

4. Precedence transitivity

4.1. Precedence transitivity

In RADTs, operations relate to object containers, i.e., elements of RFAs, slots of RHTs, and nodes of RGAs. This relationship would be clarified by causal object (*cobject*) and effective operation (*eoperation*).

- *cobject*: For a local operation O, its cobject is the object container indicated by the index of O. If O is an *Insert*, it has two cobjects: one is called as *left cobject* which is indicated by the index of O (say i), and the other is *right cobject* which is the one of $i + 1$ when O is generated.
- *eoperation*: For an object container, its eoperation is the operation whose local or remote algorithm succeeds in creating/destroying/updating the container.

Except *Insert*s, a local operation on an existing container regards the container as its cobject (cf. a *Put* on no slot has no cobject) while it becomes an eoperation on its cobject. An *Insert* has two cobjects, but becomes an eoperation only on its new node.

The intention of a remote operation is preserved, (1) if a remote *Insert* places a new node between its two cobjects, (2) if a remote *Put* on no slot becomes the eoperation on a new slot for its key, or (3) if a remote *Write*, *Put*, *Remove*, *Delete*, or *Update* becomes the eoperation on its cobject.

The intentions of different operations might be in conflict, if they are supposed to be eoperations on a common cobject or their cobjects overlap. To decide which operation has higher priority than the other conflicting one in preserving their intentions, precedence rules are needed. Enacting precedence rules is, however, complicated since the rules should not conflict with each other. We, therefore, suggest *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 7 (*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 $O_a \parallel O_b$, O_b has higher priority than O_a in preserving their intentions.

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; thus, $O_a \dashrightarrow O_b$. In a similar sense, the precedence relation between concurrent operations is defined. For instance, suppose $O_a \dashrightarrow O_b$ for $O_a \parallel O_b$. If they are two *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 performed so that O_b preserves its intention rather than O_a. If O_a and O_b are two *Insert*s of the same cobjects, O_b should insert its new node closer to the left cobject than O_a because that makes the effect similar to the effect of $O_a \dashrightarrow O_b$ derived from $O_a \rightarrow O_b$. Obviously, intentions of no conflict are preserved at once.

If precedence relations on current operations are arbitrarily enacted, they might conflict with each other. To illustrate, suppose that the operations in Fig. 3 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 arbitrary 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 a *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 [2]. The PRG of the above precedence relations is shown in Fig. 5(a). Assuming that the three operations are executed according to this PRG, the element of index 1 at each site will be as follows.

Site 0: $o_? \overset{O_3}{\Rightarrow} o_3 \overset{O_1}{\Rightarrow} o_1 \overset{O_2}{\Rightarrow} o_x$,

Site 1: $o_? \overset{O_2}{\Rightarrow} o_2 \overset{O_3}{\Rightarrow} o_3 \overset{O_1}{\Rightarrow} o_y$,

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 by $O_3 \dashrightarrow O_1$ and o_3 at sites 1 and 2 by $O_2 \dashrightarrow O_3$, respectively.

When the third operation arrives at each site, its execution must obey the precedence relations with the previous two operations.

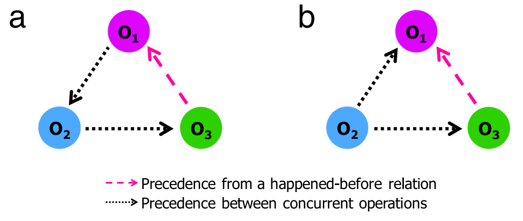
a
b
O1
O2
O3
Precedence from a happened-before relation
Precedence between concurrent operations

Fig. 5. Two PRGs of the time–space diagram of Fig. 3.

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. 5(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. 5(b) is an acyclic tournament. Since all the edges in an acyclic tournament are transitive (see Theorem 3.11 in [2]), 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 8 (*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 [2]), which visits all the vertices of the graph once, precedence relations are totally ordered; e.g., the PRG of Fig. 5(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, each object container has only to store a few hints for its *last eoperation(s)*, which are used to reconcile the intention of a new operation. 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 decide concurrent precedence relations, a derived PRG might have cycles. For example, suppose that an operation issued at a higher site ID takes precedence over an operation issued at a lower site ID. The graph of Fig. 6(a) is the PRG derived from the CEG of Fig. 4. 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 clocks that arrange distributed operations in a particular total order, such as Lamport clocks or vector clocks, can be used. For instance, since RADTs are in need of vector clocks for causality preservation, we can use the condition deriving a total order of vector clocks [35]; then, the PRG of Fig. 6(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 an *s4vector* that is derived from a vector clock. Based on the s4vectors, we define a transitive s4vector order.

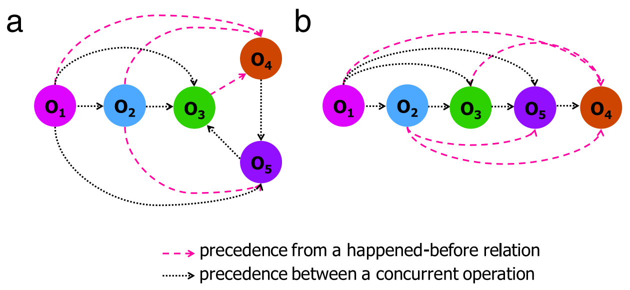
a
b
O1
O2
O3
O4
O5
precedence from a happened-before relation
precedence between a concurrent operation

Fig. 6. Two PRGs of Fig. 4. (a) is the PRG based on static priorities, and (b) is the PRG based on vector clock orders.

In RADTs, precedence relations are mostly determined on the basis of s4vector orders and will be realized in remote algorithms by 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, on which PT is based, are differently implemented with specific operation types, it is difficult to prove that, without loss of generality, PT guarantees eventual consistency. In this paper, we apply PT to the implementations of operation types, and thus make pairs of operation types commute. Hence, in our report [29], 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 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.

4.2. Discussion

In summary, the relationship among the various concepts introduced so far can be represented as follows:

$$\begin{array}{c}\boxed{\text{Responsiveness and interactivity}}\\ \text{can be enabled by}\\ \boxed{\text{Eventual Consistency}}\\ \text{can be guaranteed by}\\ \boxed{\text{Operation Commutativity (OC)}}\\ \text{can be exploited by}\\ \boxed{\text{Precedence Transitivity (PT)}}\end{array}$$

In fact, PT is not the only solution to exploiting OC. Especially, for insertion and deletion, several techniques have been introduced to achieve OC: some approaches derive a total order of objects from partial orders of objects [16,17,19,23], or introduces dense index schemes [26,40]. In Section 6, we compare those approaches with PT in more detail.

On the other hand, for some types of operations, defining precedence is not possible. For example, consider the four binary arithmetic operations, i.e., addition, subtraction, multiplication, and integer division, which are allowed on replicated integer variables. Since some pairs of these operations are not commutative, this data type does not spontaneously ensure OC. Unlike the RADT operations, their intentions are realized depending on the previous value as an operand. Therefore, the precedence relation defined for RADT operations is difficult to apply 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. 3, O_1 is to multiply 3, O_2 to add 2, and O_3 to subtract 5 on the initial shared variable 10. If O_1 is transformed into the addition of 10, i.e., an increased value by the multiplication, the replicated integers 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 $\overrightarrow{v}_O$ be the vector clock of an operation issued at site i. Then, an S4Vector $\overrightarrow{s}_O$ can be derived from $\overrightarrow{v}_O$ as follows: (1) $\overrightarrow{s}_O[\texttt{ssn}]$ is a global session number that increases monotonically, (2) $\overrightarrow{s}_O[\texttt{sid}]$ is the site ID unique to the site, (3) $\overrightarrow{s}_O[\texttt{sum}]$ is $\sum(\overrightarrow{v}_O) := \sum_{\forall i} \overrightarrow{v}_O[i]$, and (4) $\overrightarrow{s}_O[\texttt{seq}]$ is $\overrightarrow{v}_O[i]$ reserved for purging tombstones (see Section 5.6). To illustrate, suppose that $\overrightarrow{v}_O = [1, 2, 3]$ is the vector clock of an operation that is issued at site 0 at session 4. Then, the s4vector of $\overrightarrow{v}_O$ is $\overrightarrow{s}_O = \langle 4, 0, 6, 1\rangle$. As a unit of collaboration, a session begins with initial vector clocks and identical RADT structures at all sites. When a membership changes or a collaboration newly begins with the same RADT structure stored on disk, $\overrightarrow{s}_O[\texttt{ssn}]$ increases. The s4vector of an operation is globally unique because $\sum(\overrightarrow{v}_O)$ is unique to every operation issued at a site. We define an order between two s4vectors as follows.

Definition 9 (*S4vector Order '$\prec$'*)**.** Given two s4vectors $\overrightarrow{s}_a$ and $\overrightarrow{s}_b$, $\overrightarrow{s}_a$ precedes $\overrightarrow{s}_b$, or $\overrightarrow{s}_b$ succeeds $\overrightarrow{s}_a$, denoted as $\overrightarrow{s}_a \prec \overrightarrow{s}_b$, *iff*: (1) $\overrightarrow{s}_a[\texttt{ssn}] < \overrightarrow{s}_b[\texttt{ssn}]$, or (2) $(\overrightarrow{s}_a[\texttt{ssn}] = \overrightarrow{s}_b[\texttt{ssn}]) \wedge (\overrightarrow{s}_a[\texttt{sum}] < \overrightarrow{s}_b[\texttt{sum}])$, or (3) $(\overrightarrow{s}_a[\texttt{ssn}] = \overrightarrow{s}_b[\texttt{ssn}]) \wedge (\overrightarrow{s}_a[\texttt{sum}] = \overrightarrow{s}_b[\texttt{sum}]) \wedge (\overrightarrow{s}_a[\texttt{sid}] < \overrightarrow{s}_b[\texttt{sid}])$.

Lemma 2. *The s4vector orders are transitive.*

Proof. The s4vectors of different sessions are ordered by monotonous $\overrightarrow{s}[\texttt{ssn}]$s. In the same session, the s4vectors of a site are totally ordered because $\overrightarrow{s}[\texttt{sum}]$ grows monotonously. If $\overrightarrow{s}[\texttt{sum}]$s are equal across different sites, they are ordered by unique $\overrightarrow{s}[\texttt{sid}]$s. Since all s4vectors are totally ordered by the three conditions, s4vector orders are transitive. □

In this section below, $\overrightarrow{s}_O$ denotes the s4vector of the current operation derived from $\overrightarrow{v}_O$ in Algorithm 1.

5.2. Replicated fixed-size arrays

To inform a *Write* of the last eoperation, an element encapsulates a single s4vector with an object. Using C/C++ language conventions, an element of RFAs are as follows.

```
struct Element {
   Object*   obj;
   S4Vector  s⃗_p;
};
Element RFA[ARRAY_SIZE];
```

An RFA is a fixed-size array of Elements. Based on this data structure, Algorithm 5 describes the remote algorithm of *Write*, where $\overrightarrow{s}_O$ is the s4vector of the current remote operation, and $\overrightarrow{s}_p$ is the s4vector of the last eoperation on the Element.

Only when $\overrightarrow{s}_O$ succeeds $\overrightarrow{s}_p$ in line 2, does a remote *Write(int i, Object* o)* replace *obj* and $\overrightarrow{s}_p$ of the *i*th Element with a new object *o* and $\overrightarrow{s}_O$. This *Write* becomes the new last eoperation on the

Algorithm 5 The remote algorithm for *Write*

```
1  Write(int i, Object* o)
2     if(RFA[i].s⃗_p ≺ s⃗_O) // s⃗_O: s4vector of this Write;
3        RFA[i].obj := o;
4        RFA[i].s⃗_p := s⃗_O;
5        return true;
6     else return false;
```

Element by replacing RFA[i].$\overrightarrow{s}_p$ with $\overrightarrow{s}_O$. Since the s4vector of a local operation is always up-to-date at its issued time, it succeeds any s4vectors in the local RFA; hence, PT holds in both the local and remote algorithms of *Write*.

5.3. Replicated hash tables

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

```
struct Slot {
    Object*   obj;
    S4Vector  s⃗_p;
    Key       k;
    Slot*     next;
};
Slot* RHT[HASH_SIZE];
```

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

Algorithm 6 The remote algorithm for *Put*

```
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 s⃗_O ≺ s.s⃗_p) return false;
8     else if(s != nil and s is a tombstone) Cemetery.withdraw(s);
9     else if(s = nil)
10       s := 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.s⃗_p := s⃗_O;
16    return true;
```

A *Put* first examines if the Slot of its key k, mapped by a hash function *hash*, already exists (lines 3–6). If a *Put* precedes the last eoperation on the Slot, i.e., $\overrightarrow{s}_O \prec s.\overrightarrow{s}_p$, it is ignored (line 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 $s.\overrightarrow{s}_p \prec \overrightarrow{s}_O$ or no Slot exists.

A *Remove* first finds its cobject 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 by Algorithms 1 and 3. Consequently, a remote *Remove* on no Slot throws an exception and does nothing (line 4). In line 5, 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 $\overrightarrow{s}_O$ to *obj* and $\overrightarrow{s}_p$ (lines 7–8). Thanks to tombstones, no concurrent operation misses its cobject; so, the precedence relation with the last *Remove* will not be lost. Obviously, local *Reads* regard tombstones as no Slots. If we recall the example of RHTs in Section 2.3, O_1 becomes the last eoperation of the tombstone for k_1 while O_2 is ignored at site 0.

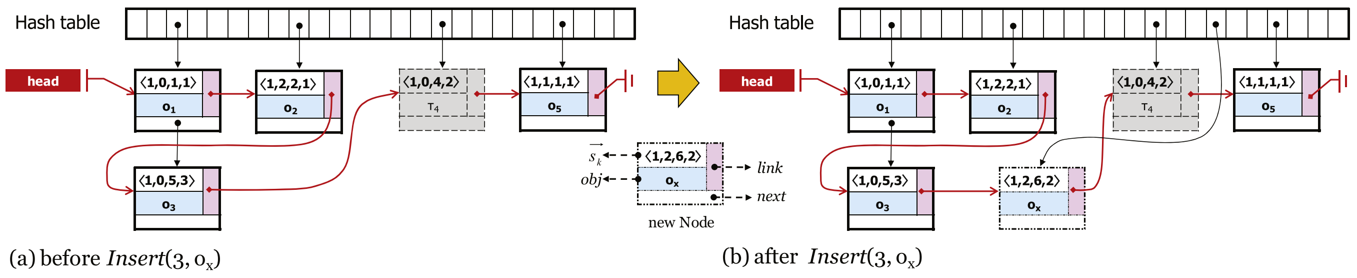

Fig. 7. An example data structure of an RGA. The Node of τ_4 is a tombstone.

Algorithm 7 The remote algorithm for *Remove*

```
Remove(Key k)
    Slot *s := RHT[hash(k)];
    while(s != nil and s.k != k) s := s.next;
    if(s = nil) throw NoSlotException;
    if($\vec{s}_O \prec$ s.$\vec{s}_p$) return false;
    if(s is not tombstone) Cemetery.enrol(s);
    s.obj := nil;
    s.$\vec{s}_p$ := $\vec{s}_O$;
    return true;
```

Removes enrol tombstones in Cemetery, a list of tombstones for purging. In Section 5.6, we discuss their purging condition. 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 not be purged.

5.4. The S4Vector index (SVI) scheme for RGAs

In RGAs, *Inserts* and *Deletes* induce the intention violation problem due to integer indices, as stated in Section 2.3; that is, the nodes indicated by integer indices might be different at remote sites. To make the remote RGA operations correctly find their intended nodes, this paper introduces an *s4vector index (SVI) scheme*. A local operation with an integer index is transformed into a remote one 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 reserved in the hash table by using the s4vector of the *Insert* as a 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 *Inserts*. For precedence of *Deletes* and *Updates*, $\vec{s}_p$ is prepared. Two pointers to Nodes, 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 Nodes like an RHT, and head is a starting point of the linked list.

Fig. 7 shows an example of an RGA data structure which is constructed with a linked list combined with a hash table. The local algorithms also 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. 7(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., the left cobject o_3, from the linked list. Then, it creates a new Node that contains the new object o_x in *obj* and the s4vector $\vec{s}_O = \langle 1, 2, 6, 2\rangle$ in both $\vec{s}_k$ and $\vec{s}_p$. 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. 7(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) generated on the RGA of Fig. 7(a) will be transformed into *Insert*($\langle 1, 0, 5, 3\rangle$, o_x) before it is broadcast. In this way, three 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. Here, $\vec{i}$ is from $\vec{s}_k$ in the cobject of a local *Delete*/*Update* or the left cobject of a local *Insert*. If an *Insert* adds its object at the head, $\vec{i}$ should be *nil*.

5.5. Three remote operations for RGAs

Algorithm 8 shows the remote algorithm for *Insert*. As shown in Fig. 8, a remote *Insert* is executed through four steps. (i) First, a remote *Insert* looks for its left cobject in the hash table with the s4vector index $\vec{i}$ (lines 5–6). The SVI scheme ensures that this left cobject is always the same of the corresponding local *Insert*. For non-*nil* $\vec{i}$, the left cobject always exists in the remote RGAs because tombstones also remain after *Deletes*. To this end, an *Insert* throws an exception, unless finding its cobject (line 7). (ii) Next, 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 8–13).

(iii) A remote *Insert* might not add its new Node on the exact right of the left cobject in order to preserve the intentions of some other concurrent *Inserts* that have already inserted their new Nodes next to the same cobject. If an *Insert* has a succeeding s4vector, it has higher priority in preserving its intention; thus, it places its new Node nearer its left cobject. Accordingly, in line 20, a remote *Insert* scans the Nodes next to its left cobject until a preceding Node whose $\vec{s}_k$ precedes ins.$\vec{s}_k$ is first encountered. As lines 14–18 are needed for inserting a new object at the head, the conditions of line 15 are the converse of line 20; if not inserted at the head, the comparison continues again from line 20. (iv) Finally, the new Node is linked in front of the first encountered preceding Node by lines 21–22.

The following example, known as the dOPT puzzle [34] (see Section 6), illustrates how *Inserts* work.

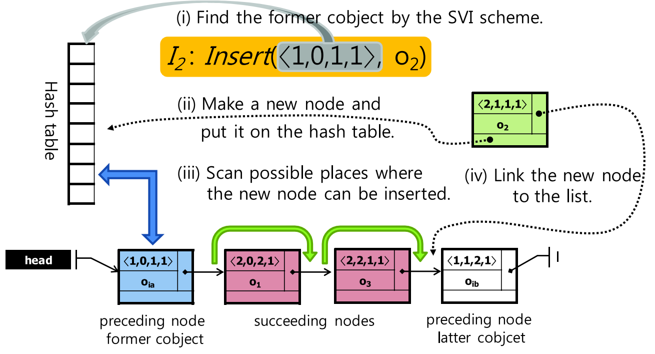
(i) Find the former cobject by the SVI scheme.
I_2: Insert(⟨1,0,1,1⟩, o$_2$)
Hash table
(ii) Make a new node and put it on the hash table.
⟨2,1,1,1⟩ o$_2$
(iii) Scan possible places where the new node can be inserted.
(iv) Link the new node to the list.
head
⟨1,0,1,1⟩ o$_{ia}$
⟨2,0,2,1⟩ o$_1$
⟨2,2,1,1⟩ o$_3$
⟨1,1,2,1⟩ o$_{ib}$
preceding node former cobject
succeeding nodes
preceding node latter cobjcet

Fig. 8. The overview of the execution of I_2 in Example 1.

Algorithm 8 The remote algorithm for *Insert*

```
1  Insert(S4Vector $\vec{i}$, Object* o)
2     Node* ins;
3     Node* ref;
4     if($\vec{i}$ != nil) // (i) Find the left cobject 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) throw NoRefObjException;
8     ins := new Node; // (ii) Make a new Node
9     ins.$\vec{s}_k$ := $\vec{s}_O$;
10    ins.$\vec{s}_p$ := $\vec{s}_O$;
11    ins.obj := o;
12    ins.next := RGA[hash($\vec{s}_O$)]; // place the new node
13    RGA[hash($\vec{s}_O$)] := ins;      // into the hash table;
14    if($\vec{i}$ = nil) // (iii) Scan possible places
15       if(head = nil or head.$\vec{s}_k$ $\prec$ ins.$\vec{s}_k$)
16          if(head != nil) ins.link := head;
17          head := ins;
18          return true;
19       else ref := head;
20    while(ref.link != nil and ins.$\vec{s}_k$ $\prec$ ref.link.$\vec{s}_k$) ref := ref.
         link;
21    ins.link := ref.link; // (iv) Link the new node to the list.
22    ref.link := ins;
23    return true;
```

Example 1 (Fig. 3). dOPT puzzle, on initial RGAs $= [o_{i_a} o_{i_b}]$ with $\vec{i}_a = \langle 1, 0, 1, 1\rangle$ and $\vec{i}_b = \langle 1, 1, 2, 1\rangle$,

$$I_1\text{: Insert}(1 = \vec{i}_a, o_{i_1}) \text{ with } [1, 0, 1] \rightsquigarrow \vec{i}_1 = \langle 2, 0, 2, 1\rangle,$$
$$I_2\text{: Insert}(1 = \vec{i}_a, o_{i_2}) \text{ with } [0, 1, 0] \rightsquigarrow \vec{i}_2 = \langle 2, 1, 1, 1\rangle,$$
$$I_3\text{: Insert}(1 = \vec{i}_a, o_{i_3}) \text{ with } [0, 0, 1] \rightsquigarrow \vec{i}_3 = \langle 2, 2, 1, 1\rangle.$$

We assume that I_1, I_2, and I_3 correspond to O_1, O_2, and O_3 of Fig. 3, respectively. As all their remote forms have the same s4vector index $\vec{i}_a = \langle 1, 0, 1, 1\rangle$, their intentions are to insert new nodes next to o_{i_a}. 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 2. As $\vec{i}_2 \prec \vec{i}_3 \prec \vec{i}_1, I_2 \dashrightarrow I_3 \dashrightarrow I_1$; I_1 has the highest priority, then I_3, and I_2 in order. At each site of Fig. 3, PT of *Insert*s is realized as follows.

At site 0, remote I_3 places o_{i_3} in front of the preceding Node o_{i_b} of the previous session. Then, I_1 is executed as $[o_{i_a} o_{i_1} o_{i_3} o_{i_b}]$ by the local *Insert* algorithm. Finally, remote I_2 is executed as Fig. 8. In line 20, o_{i_1} and o_{i_3} are skipped in turn because they are the succeeding Nodes whose $\vec{s}_k$ succeeds ins.$\vec{s}_k$. Thus, I_2 inserts o_{i_2} past o_{i_1} and o_{i_3} as $[o_{i_a} o_{i_1} o_{i_3} \mathbf{o}_{i_2} o_{i_b}]$.

At sites 1 and 2, concurrent I_2 and I_3 commute despite different execution order because the scanning of line 20 sorts o_{i_2} and o_{i_3} between the same cobjects o_{i_a} and o_{i_b} as $[o_{i_a} \mathbf{o}_{i_3} \mathbf{o}_{i_2} o_{i_b}]$. Then, the most succeeding I_1 puts o_{i_1} nearest the common left cobject o_{i_a} so that I_1 preserves its intention more preferentially than I_2 and I_3; so, the RGA states eventually converge as follows.

Execution 1. At each site of Fig. 3,

$$\text{Site 0: } [o_{i_a} o_{i_b}] \overset{I_3}{\Rightarrow} [o_{i_a} \mathbf{o}_{i_3} o_{i_b}] \overset{I_1}{\Rightarrow} [o_{i_a} \mathbf{o}_{i_1} o_{i_3} o_{i_b}] \overset{I_2}{\Rightarrow} [o_{i_a} o_{i_1} o_{i_3} \mathbf{o}_{i_2} o_{i_b}],$$
$$\text{Site 1: } [o_{i_a} o_{i_b}] \overset{I_2}{\Rightarrow} [o_{i_a} \mathbf{o}_{i_2} o_{i_b}] \overset{I_3}{\Rightarrow} [o_{i_a} \mathbf{o}_{i_3} o_{i_2} o_{i_b}] \overset{I_1}{\Rightarrow} [o_{i_a} \mathbf{o}_{i_1} o_{i_3} o_{i_2} o_{i_b}],$$
$$\text{Site 2: } [o_{i_a} o_{i_b}] \overset{I_3}{\Rightarrow} [o_{i_a} \mathbf{o}_{i_3} o_{i_b}] \overset{I_2}{\Rightarrow} [o_{i_a} o_{i_3} \mathbf{o}_{i_2} o_{i_b}] \overset{I_1}{\Rightarrow} [o_{i_a} \mathbf{o}_{i_1} o_{i_3} o_{i_2} o_{i_b}].$$

It is worth noting that the 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 precedence relations.

Algorithm 9 The remote algorithm for *Delete*

```
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) throw NoTargetObjException;
5     if(n is not a tombstone)
6        n.obj := nil;
7        n.$\vec{s}_p$ := $\vec{s}_O$;
8        Cemetery.enrol(n);
9     return true;
```

The local and remote *Delete* algorithms leave a tombstone behind. In Algorithm 9, a remote *Delete* finds its cobject with $\vec{i}$ via the hash table (lines 2–3); otherwise, it throws an exception (line 4). Regardless of s4vector order, a *Delete* assigns *nil* and $\vec{s}_O$ into *obj* and $\vec{s}_p$ (but not $\vec{s}_k$) as a mark of a tombstone, and enrols it into Cemetery (lines 6–8). Note that tombstones never revive in RGAs. As *findlist* and *findlink* in Algorithm 4 exclude tombstones from counting, local operations never employ tombstones as cobjects. For example, in Fig. 7(a), local *Insert*$(4, o_y)$ refers to o_5 instead of the tombstone τ_4, and thus is transformed into remote *Insert*$(\langle 1, 1, 1, 1\rangle, o_y)$.

Algorithm 10 The remote algorithm for *Update*

```
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) throw NoTargetObjException;
5     if(n is a tombstone) return false;
6     if($\vec{s}_O$ $\prec$ n.$\vec{s}_p$) return false;
7     n.obj := o;
8     n.$\vec{s}_p$ := $\vec{s}_O$;
9     return true;
```

In Algorithm 10, a remote *Update* operates in the same way as a remote *Delete* until finding its cobject. An *Update* also replaces *obj* and $\vec{s}_p$ (but not $\vec{s}_k$) of its cobject with its owns if $\vec{s}_O$ succeeds $\vec{s}_p$ (lines 7–8). Unlike *Put* of RHTs, an *Update* does nothing on a tombstone as in line 5; thus, always *Update* $\dashrightarrow$ *Delete*. This prevents an *Update* on a tombstone from being translated into the semantic of an *Insert* and makes the 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 $\vec{i}_a = \langle 1, 0, 1, 1\rangle$.
$U_1(O_1)$: Update$(1 = \vec{i}_a, \dot{o}_{i_a})$ with $[1, 0, 0] \rightsquigarrow \vec{i}_1 = \langle 2, 0, 1, 1\rangle$,
$U_2(O_2)$: Update$(1 = \vec{i}_a, \ddot{o}_{i_a})$ with $[0, 1, 0] \rightsquigarrow \vec{i}_2 = \langle 2, 1, 1, 1\rangle$,
$D_3(O_3)$: Delete$(1 = \vec{i}_a)$ with $[0, 0, 1] \rightsquigarrow \vec{i}_3 = \langle 2, 2, 1, 1\rangle$,
$I_4(O_4)$: Insert$(0 = nil, o_{i_4})$ with $[2, 1, 1] \rightsquigarrow \vec{i}_4 = \langle 2, 0, 4, 2\rangle$,
$I_5(O_5)$: Insert$(1 = \vec{i}_a, o_{i_5})$ with $[0, 2, 0] \rightsquigarrow \vec{i}_5 = \langle 2, 1, 2, 2\rangle$.

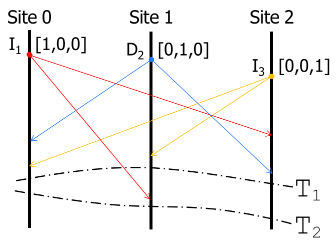

Fig. 9. A time–space diagram of Example 3.

For $U_1 \parallel U_2$ being in conflict, U_2 has higher priority than U_1 owing to $\vec{i}_1 \prec \vec{i}_2$; thus, as shown in Execution 2, U_1 is ignored at site 1 by line 6 of Algorithm 10. When D_3 conflicts with U_1 and U_2, D_3 always succeeds in leaving the Node of o_{i_a} as the tombstone τ_{i_a} regardless of the s4vector order of $\vec{s}_p$, but U_1 and U_2 do nothing on τ_{i_a} by line 5 of Algorithm 10. The tombstone τ_{i_a} also enables I_5 to find the left object after having executed concurrent D_3 at sites 1 and 2. In addition, since τ_{i_a} is regarded as a normal preceding Node in Algorithm 8, I_4 places o_{i_4} in front of τ_{i_a} at sites 1 and 2. Eventually, RGAs converge at all the sites as follows.

Execution 2. At each site of Fig. 1,

Site 0: $[o_{i_a}] \overset{U_1}{\Rightarrow} [\dot{\mathbf{o}}_{i_a}] \overset{U_2}{\Rightarrow} [\ddot{\mathbf{o}}_{i_a}] \overset{D_3}{\Rightarrow} [\tau_{i_a}] \overset{I_4}{\Rightarrow} [\mathbf{o}_{i_4}\tau_{i_a}] \overset{I_5}{\Rightarrow} [o_{i_4}\tau_{i_a}\mathbf{o}_{i_5}]$,

Site 1: $[o_{i_a}] \overset{U_2}{\Rightarrow} [\ddot{\mathbf{o}}_{i_a}] \overset{I_5}{\Rightarrow} [\ddot{o}_{i_a}\mathbf{o}_{i_5}] \overset{U_1}{\Rightarrow} [\ddot{o}_{i_a}o_{i_5}] \overset{D_3}{\Rightarrow} [\tau_{i_a}o_{i_5}] \overset{I_4}{\Rightarrow} [\mathbf{o}_{i_4}\tau_{i_a}o_{i_5}]$,

Site 2: $[o_{i_a}] \overset{D_3}{\Rightarrow} [\boldsymbol{\tau}_{i_a}] \overset{U_1}{\Rightarrow} [\tau_{i_a}] \overset{U_2}{\Rightarrow} [\tau_{i_a}] \overset{I_4}{\Rightarrow} [\mathbf{o}_{i_4}\tau_{i_a}] \overset{I_5}{\Rightarrow} [o_{i_4}\tau_{i_a}\mathbf{o}_{i_5}]$.

To sum up, the SVI scheme enables remote RGA operations to find their intended Nodes 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 are kept up, no remote operations miss their cobjects. Another s4vector of a Node $\vec{s}_p$ is renewed by *Updates* and *Deletes*. The effectiveness of *Updates* is decided by $\vec{s}_p$, but *Deletes* are always successful; i.e., always *Update* $\dashrightarrow$ *Delete*. Nevertheless, OC holds because no operation happening after a *Delete* targets or refers to the tombstone. Separation of $\vec{s}_k$ and $\vec{s}_p$ means that *Insert*s never conflict with any *Updates* or *Deletes*.

5.6. Cobject preservation

Cobjects need to be preserved for consistent intentions of operations. If the cobjects causing the effect of a local operation are not preserved at remote sites, its remote operations may cause different effects. Tombstones enable remote operations to manifest their intentions by retaining cobjects, but need purging. However, the tombstone purging algorithm should be cautiously designed for consistency. In fact, the operational transformation (OT) framework has 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. 9.

Example 3 (Fig. 9). Initially, RGAs = $[o_{i_a}]$ with $\vec{i}_a = \langle 1, 0, 1, 1\rangle$,

I_1: Insert$(0 = nil, o_{i_1})$ with $[1, 0, 0] \rightsquigarrow \vec{i}_1 = \langle 2, 0, 1, 1\rangle$,
D_2: Delete$(1 = \vec{i}_a)$ with $[0, 1, 0] \rightsquigarrow \vec{i}_2 = \langle 2, 1, 1, 1\rangle$,
I_3: Insert$(1 = \vec{i}_a, o_{i_3})$ with $[0, 0, 1] \rightsquigarrow \vec{i}_3 = \langle 2, 2, 1, 1\rangle$.

Two cobjects of I_1 are the head and o_{i_a} while those of I_3 are o_{i_a} and the tail. The left 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 right cobject might be overlooked, though it is required to terminate the comparisons in lines 15 or 20 of Algorithm 8. However, if the right cobject is purged before the remote execution of an *Insert*, *Insert*s of different intentions might be in conflict. For example, see Execution 3 where τ_{i_b} is assumed to be purged at the time T_1 of Fig. 9 ($\overset{P}{\Rightarrow}$ means purging).

Execution 3. If purging τ_{i_b} at T_1,

Site 0: $[o_{i_a}] \overset{I_1}{\Rightarrow} [\mathbf{o}_{i_1}o_{i_a}] \overset{D_2}{\Rightarrow} [o_{i_1}\tau_{i_a}] \overset{I_3}{\Rightarrow} [o_{i_1}\tau_{i_a}\mathbf{o}_{i_3}] \overset{P}{\Rightarrow} [o_{i_1}o_{i_3}]$,

Site 1: $[o_{i_a}] \overset{D_2}{\Rightarrow} [\tau_{i_a}] \overset{I_3}{\Rightarrow} [\tau_{i_a}\mathbf{o}_{i_3}] \overset{P}{\Rightarrow} [o_{i_3}] \overset{I_1}{\Rightarrow} [o_{i_3}\mathbf{o}_{i_1}]$,

Site 2: $[o_{i_a}] \overset{I_3}{\Rightarrow} [o_{i_a}\mathbf{o}_{i_3}] \overset{I_1}{\Rightarrow} [\mathbf{o}_{i_1}o_{i_a}o_{i_3}] \overset{D_2}{\Rightarrow} [o_{i_1}\tau_{i_a}o_{i_3}] \overset{P}{\Rightarrow} [o_{i_1}o_{i_3}]$.

Observe the effect of I_1 concerning the existence of its right cobject, i.e., o_{i_a} or τ_{i_a}. At site 0, I_1 places o_{i_1} at the head. Being indispensable to I_3 as the left cobject, τ_{i_a} is retained. At site 2, I_1 has to insert o_{i_1} in front of the preceding o_{i_a}, and then D_2 performs. Hence, sites 0 and 2 have the correct final result $[o_{i_1}o_{i_3}]$. At site 1, however, if τ_{i_a} is purged, $\vec{i}_1$ of I_1 is compared with $\vec{s}_k$ of o_{i_3} $(= \vec{i}_3$ of $I_3)$ despite I_3 having different cobjects; thus, $[o_{i_3}o_{i_1}]$. Consequently, the loss of the right cobject can lead to the different effects of I_1. Instead, if the right cobject is purged at T_2 of Fig. 9, the effects of I_1 are consistent as follows.

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 for consistent operation intentions. However, in RGAs, tombstones, impeding search for Nodes in the linked list, need purging as soon as possible. We, therefore, introduce a safe tombstone purging condition using s4vectors.

Let D_i be a *Delete* issued at site i and τ_i be the tombstone caused by D_i. Recall that D_i assigns its s4vector into $\tau_i.\vec{s}_p$ and that RGAs guarantee two properties for a tombstone: (1) a tombstone never becomes the cobject of 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. By retaining τ_i as far as any concurrent operations with D_i can arrive, we can prevent those concurrent operations from missing cobjects. Golding already introduced a safe condition for this [10]. The existing condition enables RHTs and RGAs to preserve the cobjects of their operations, except the right cobjects of *Insert*s.

To preserve the right cobjects of *Insert*s, an additional condition is needed. Note that, at site 1 in Example 3, the loss of τ_{i_a} causes problems since the next Node of τ_{i_a} succeeds the s4vector of I_1. In other words, if it is ensured that a new arrival *Insert* succeeds $\vec{s}_k$ of every Node, the tombstone can be substituted by its next Node as a right cobject. To this end, an RGA needs to maintain a set of vector clocks including as many vectors as the number of sites N, i.e., $VC_{\mathrm{last}} = \{\vec{v}_{\mathrm{last}_0}, \ldots, \vec{v}_{\mathrm{last}_{N-1}}\}$; here, $\vec{v}_{\mathrm{last}_j} \in VC_{\mathrm{last}}$ is the vector clock of the last operation issued at site j and successfully executed at the site of VC_{last}. Using VC_{last}, a tombstone τ_i of D_i can be safely purged if satisfying both of the following conditions.

(1) $\tau_i.\vec{s}_p[\mathtt{seq}] \le \min_{\forall \vec{v} \in VC_{\mathrm{last}}} \vec{v}[i]$ for $i = \tau_i.\vec{s}_p[\mathtt{sid}]$,
(2) $\tau_i.\mathrm{link}.\vec{s}_k[\mathtt{sum}] < \min_{\forall \vec{v} \in VC_{\mathrm{last}}} \sum(\vec{v})$ or $\tau_i.\mathrm{link} = \mathrm{tail}$.

Condition (1) is similar to Golding's [10], 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 arrival operation succeeds that of the Node next to the tombstone that will be purged in the linked list.

Consequently, we prepare Cemetery as a set of FIFO queues, each of which reserves tombstones of different $\vec{s}_p[\mathtt{sid}]$. A *Delete* enrols a tombstone at the end of the queue; thus, enrolling a tombstone takes a constant time. A purge operation first inspects a foremost tombstone in each queue of `Cemetery` to know whether there are any tombstones that can be purged. If they exist, the tombstones are purged in practice with the time complexity of $O(N)$ because the previous `Node` of a tombstone should be found from the singular linked list of an RGA.

Note that, if a session changes, all tombstones in RGAs can be purged. However, 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, thereby continuing to purge.

6. Related work

The concept of commutativity was first introduced in distributed database systems [1,39]. It was, however, applied to concurrency control over centralized resources, but not to consistency maintenance among replicas. In other words, to grant more concurrency to some transactions in a locking protocol, transaction schedulers allow only innately commutative operations, e.g., *Write*s on different objects, to be executed concurrently while noncommutative operations still have to be locked. For maintaining consistency among replicas, the works of [14,21] considered commutativity, but allow only innately commutative operations to be executed in different order.

The behavior of RFAs is similar to that of *Thomas's writing rule*, introduced in multiple copy databases [36,37]. The rule prescribes that a *Write*-like operation can take effect only when it is newer than the previous one [15]. To represent the newness, Lamport clocks are adopted. In fact, Lamport clocks can be used in RADTs on behalf of `sum` of s4vectors. We, however, use the s4vector derived from a vector clock to ensure not only the causality preservation but also the cobject preservation. The idea of tombstones was also introduced in replicated directory services [21,9,6], which are similar to RHTs.

With respect to RGAs, the operational transformation (OT) framework is one of a few relevant approaches that allows *optimistic insertions and deletions* 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 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 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 [7], is another expression of the commutative relation of Definition 5 in terms of the OT framework. As it is not sufficient, a counterexample, called the dOPT puzzle (see Example 1), was found. Ressel et al. proposed TP_2, which means that consecutive TFs along different paths must result in a unique transformed operation [27]. It was proven that TP_1 and TP_2 are the sufficient conditions that ensure eventual consistency of some OT integration algorithms such as adOPTed [20,32], but it is worth comparing them with OC and PT. Though we are not sure that TP_2 is equivalent to OC, TP_2 is clearly a property only on *sequential* concurrent operations. If happened-before operations intervene among concurrent operations, OC and PT explain how operations are designed, but TP_2 nothing.

Various OT methods consisting of different TFs or integration algorithms have been introduced: e.g., adOPTed [27], GOT [35], GOTO [34], SOCT2 [32], SOCT4 [38], SDT [16] and TTF [22]. However, having presented no guidelines on preserving TP_2, counterexamples have been found in most of OT algorithms, such as adOPTed, GOTO, SOCT2, and SDT. Though GOT or SOCT4 avoided TP_2 by fixing up the transformation path through an undo-do-redo scheme or a global sequencer, responsiveness is significantly degraded. In addition, the intention preservation, addressed in [35], also has gone through the lack of effective guidelines. We believe PT could be an effective clue to preserving both TP_2 and the intention preservation. For example, when OT methods break ties among insertions, or write the TF for an *Update*-like operation, PT will explain how to compromise their conflicting intentions.

Another reason why OT methods have failed not only in consistency but also in intention preservation is the loss of cobjects (see Section 5.6) if illustrated in terms of our RADT framework. Similar to Execution 3, once a character is removed, TFs are difficult to consider it. Accordingly, Li et al. have suggested a series of algorithms, such as SDT [16], ABT [17], and LBT [19]. The authors said these algorithms are free from TP_2 by relying on the effects relation, which is an order between a pair of every two characters. However, deriving effects relations incurs additional overhead 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 invites tombstones to the OT framework [22]. TTF introduces 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 incomparable at different sites. Hence, some optimizations, such as caret operations or D-TTF, are provided. Since TTF must be combined with an existing OT integration algorithm, such as adOPTed or SOCT2, it inherits the characteristics of the OT framework.

Recently, several prototypes adopting unique indices were introduced for the optimistic insertion and deletion. Oster et al. proposed the WOOT framework that is free from vector clocks for scalability [24]. Instead, the causality of an operation is checked by the existence of cobjects at remote sites; also in RGAs, this can be in use through the SVI scheme (see Section 7). In WOOT, a character has a unique index of the pair $\langle$site ID, logical clock$\rangle$ and includes the indices of two cobjects of the insertion; also, an insertion is parameterized with the two indices. For consistency, an insertion derives a total order among existing characters by considering both sides of characters, but this makes the insertion of WOOT an order of magnitude slower than that of RGAs. WOOT also keeps up tombstones, but its purging algorithm is not yet presented.

Meanwhile, independently of our early work [28], Shapiro et al. proposed a commutative replicated data type (CRDT) called *treedoc* that adopts an ingenious index scheme [31,26]. 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. Conflicting insertions require special indices like the WOOT indices to place their new characters into the same node. Besides, if a deletion performs on a node that is not a leaf, its tombstone should be preserved not to change indices of its child nodes. Thus, as a treedoc ages, it becomes unbalanced and may 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 vice versa. However, *flatten*, requiring a distributed commitment protocol, is costly and not scalable.

For scalability purpose, Weiss et al. suggested *logoot*, a sparse n-ary tree, which provides new indices continuously like treedoc [40]. However, unlike in treedoc, a node of logoot encapsulates a unique index that is an unbounded sequence of the pair ⟨pos, site ID⟩, where *pos* is the position in a logoot tree. Explicit indices allow logoot trees to be sparse; that is, no tombstone is needed for a deletion. Owing to the absence of tombstones, logoot could incur less overhead than treedoc for numerous operations abounding in a large-scale replication. To enhance scalability, the authors also suggest the causal barrier [25] on behalf of vector clocks for causality preservation. Although causal barriers can reduce the transmission overhead, sites manage their local clock in the same manner as with vector clocks for membership changes. In fact, causality preservation relates to reliability, which is discussed in Section 7.

Above all, none of the above three approaches could derive an underlying principle, such as PT. Therefore, including the effects relations of Li et al. [19], the approaches are bounded only to the consistency of the insertion and deletion. In other words, they present no solution for the *Update*-like operation. 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, the update has *not* been discussed in the OT framework. Instead, independently of the OT framework, Sun et al. proposed a multi-version approach for the update in collaborative graphical editors, where the order of graphical objects does not matter [33]. In this approach, when the operations updating some attributes, such as position or color, are in conflict, multiple versions of the object are shown to users in order not to lose any intentions. Although the behavior of RADTs must be deterministic as building blocks, RADTs can mimic the multi-version approach; 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

As the building blocks, the time complexity of RADTs is decisive for the performance and quality of 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, is the theoretical worst-case of *Put* and *Remove* $O(N)$ for the number of objects N due to the separate chaining.

The local RGA operations with integer indices work in $O(N)$ time since *findlist* in Algorithm 4 searches intended Nodes via the linked list from the head. As mentioned in Section 2.3, RGAs also support the local pointer operations taking constant time by using the *findlink* function. Meanwhile, the remote RGA operations can perform in $O(1)$ time 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 Nodes have been inserted by the concurrent *Insert*s on the same reference Node.

We compare RGAs with WOOT and the recent OT methods, such as ABT, SDT, and TTF, in time-complexity. About the complexity of ABT and SDT, we consult [18]. For the complexity of TTF, we assume that D-TTF is combined with the adOPTed integration algorithms [22,27]. According to [18], the performance of the remote operations of both ABT and SDT fluctuates depending on the size and characteristics of the history buffer. WOOT presents different time complexity for insertion and deletion. Also, the complexity differs according to the policy to find a character; we assume that a local operation finds a character in

Table 1
Time complexity of local and remote operations in a few algorithms.

Algorithms	Local operations	Remote operations
RGAs	$O(N)$ or[a] $O(1)$	$O(1)$
ABT	$O(\|H\|)$	$O(\|H\|^2)$
SDT	$O(1)$	$O(\|H\|^2)$ or[c] $O(\|H\|^3)$
D-TTF w/adOPTed	$O(N)$ or[b] $O(1)$	$O(\|H\|^2+N)$
WOOT	[d] $O(N^2)$ and[e] $O(1)$	[d] $O(N^3)$ and[e] $O(N)$

N: the number of objects or characters, $|H|$: the number of operations in history buffer.
[a] Local pointer operations.
[b] The caret operations.
[c] Worst-case complexity.
[d] WOOT insertion operation.
[e] WOOT deletion operation.

$O(1)$ time but a remote operation in $O(N)$ [23]. Table 1 shows that RGAs overwhelm the others, especially in the performance of the remote operations. More significantly, the remote RGA operations can perform without fluctuation, and thus guarantee stable responsiveness in the collaboration.

We examine scalability in two aspects: membership size and the number of objects. As membership size or the number of objects scale up, performance may degrade. 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 that each of total $s = 16$ sites generates evenly $N = 6250$ operations. Then, though all sites execute equally $s \times N = 100,000$ operations, each site will execute 6250 local and 93,750 remote operations. Consequently, the performance of remote operations are critical to scalability. RGAs have the optimal remote operations that are irrelevant to the number of objects. RGAs, therefore, are scalable, which will be proven in the next section. Meanwhile, OT methods are unscalable because their remote operations are inefficient and because history buffers are likely to grow for larger membership.

In the meantime, scalability can be affected by vector clocks, adopted by most of optimistic replications requiring causality detection and preservation, because the clock size must be proportional to membership size. In the OT framework, a vector clock per operation should be stored in the history buffer for the causality detection. Hence, 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 in order to store a document or effects relations. However, RGAs reserve only two s4vectors per object because PT enables consistency to be achieved without causality detection; thus, the space complexity is $O(N)$. Therefore, the s4vector enhances the scalability of RGAs with respect to the space overhead.

In fact, the overhead incurred by tombstones cannot be ignored in collaborative applications; thus, tombstone purging algorithms have an impact on overhead. The space complexity of treedoc and WOOT is $O(N)$ including tombstones, while unbounded indices of logoot might incur theoretically higher space complexity despite the absence of tombstones [40]. Compared with treedoc or WOOT, RGAs may suffer more overhead owing to entries of tombstones. However, unlike treedoc or WOOT, of which indices are structurally related to tombstones, an RGA can purge tombstones regardless of indices as it continuously receives operations from the other sites. Section 8 presents a simple experiment regarding tombstones.

As most of optimistic replication systems, RADTs also constrain themselves to preserve the causality defined by Lamport [15], but this relates to the reliability issue. When a site broadcasts an operation, some of the other sites may lose it. Though such a fault as an operation loss can be detected by the causality preservation scheme, it leads to a chain of delays in executing the operations happening after the lost operation. In the sense that a fault might result in a failure, preserving causality could significantly exacerbate reliability in scalable collaborative applications. In Example 2 and Fig. 1, if site 2 misses U_2, then I_4 and I_5 happening after U_2 should be delayed. However, I_4 and I_5 can be executed without delays while U_2 is being retransmitted because U_2 has no *essential causality* with I_4 and I_5. In other words, reliability can be enhanced by relaxing causality.

Relaxing causality permits sites to execute some additional operation sequences that are not CESes, but that preserve only essential causality (say eCES). Then, OC does not guarantee consistency for eCESes. In fact, WOOT, the only approach allowing eCESes, verifies consistency by the model-checker TLC on a specification modeled on the TLA+ specification languages [41]. This checker exhaustively verifies all the states produced by all possible executions of operations, leading to the explosion of states; thus, the verification is made only up to four sites and five characters in WOOT [23]. In any case, for eCESes consisting of insertions and deletions, there is no generalized proof methodology of consistency yet.

In RFAs and RHTs, it is simple to show that eventual consistency is guaranteed even for eCESes, though lines 4 in Algorithm 7 needs to be modified not to throw an exception. PT ensures that the last eoperation on a container is always identical, if the same set of operations is executed. Necessarily, to achieve consistency for eCESes in RGAs, the algorithms need to be modified. We leave the causality relaxation as future work, but believe OC and PT can be clues to make eCESes converge and to prove consistency.

Compared with the OT framework, RGAs have much room for improving reliability. In the OT framework, all happened-before operations of a remote operation are indispensable to *satisfying the precondition*, addressed by Sun et al. [35], because integer indices are dependent on all the previously happened operations by nature. Meanwhile, by means of the SVI scheme, the remote RGA operations can validate their cobjects autonomously, i.e., independently of most of the other operations, thereby checking the essential causality without vector clocks. Hence, like WOOT, RGAs have a chance to be free from vector clocks, which could improve the reliability.

8. Performance evaluation

We perform some experiments on RGAs to verify if the RGA operations actually work as the analysis of Section 7 and to compare with some previous approaches. To our knowledge, however, no previous approaches have presented any *performance* evaluation yet, except SDT and ABT [18]. Comparably to the experiments in [18], RGAs are implemented in C++ language and compiled by GNU g++ v4.2.4 on Linux kernel 2.6.24. We automatically generate intensive workloads modeling real-time collaborations with respect to the following four parameters:

- s: the number of sites.
- N: the number of operations that a site generates. In the experiments, every site generates evenly N operations. Hence, every site executes N local operations and $N \times (s-1)$ remote operations on its RGA.
- avd: average delay. As shown in Fig. 10, at every turn, a site either generates a local operation or receives a remote operation. Operations generated at a site are broadcast to arbitrary forward turns of the other sites by keeping the order at their local site. Thus, delays are measured in ‘turns’, and avd is the average number of turns that total $s \times N$ operations take to be delivered. We indirectly control avd by stipulating the maximum delay of an operation.

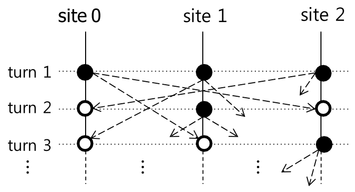

Fig. 10. An example of a workload generation where s=3.

- mo: minimum number of objects. Since all experiments are devised to begin with empty RGAs, mo controls the number of objects in RGAs during the evaluation of a workload. If an RGA at a site has objects fewer than mo (excluding tombstones), it generates only *Insert*s. Otherwise, the site randomly generates one of *Insert*, *Delete*, and *Update* with equal proportions.

For three groups of operations, i.e., (LI) local operations with integer indices, (LP) local operations with pointer indices, and (R) remote operations with s4vector indices, their generated indices are uniformly distributed over their current RGAs. Unrelated to our proposed algorithms, the times for communication or buffering are excluded in the measurement. Currently, a purge operation is invoked whenever every remote operation is executed (see Section 5.6). We run the workloads on Intel® Pentium-4 2.8 GHz CPU with 1 GB RAM.

Fig. 11 shows the average execution time of each operation with respect to the number of objects. By restricting mo, we control the average number of objects as in the line chart of Fig. 11. As predicted in the time-complexity of Table 1, only the execution times of operations (LI) are proportional to the number of objects, whereas those of operations (LP) and (R) are irrelevant. The execution time of the purge operation is also affected by the object number, but less susceptible than those of operations (LI) because tombstones are not always purged.

Compared with the OT operations of SDT and ABT [18], implemented in C++, run on Intel® Pentium-4 3.4 GHz CPU, and evaluated for the workloads generated from only two sites, the RGA operations overwhelm the OT operations in performance. As shown in Table 1, the results of [18] prove that the history size decides the performance of the OT operations. For example, if a site has executed more than 3000 local and 1000 remote operations, it takes more than 600 ms (10^{-3} s) and 100 ms to execute a remote SDT operation and two ABT operations (one local and one remote operations), respectively [18]. Operations (LP) and (R), however, can be executed in 0.4–1.3 μs (10^{-6} s) in our environment, and operations (LI) are also faster enough unless an RGA contains excessively abundant objects.

Fig. 12 shows the effect of delays. In RGAs, delays decide the lifetime of tombstones. As stated in Section 5.6, tombstones can be purged if a site continues to receive operations from all the other sites. In the line chart of Fig. 12, though each of roughly 33,000 *Delete*s makes one tombstone, smaller avd decreases tombstones; irrespective of avd, the average number of objects excluding tombstones is around 800. As a result, longer delays degrade the performance of operations (LI), but not of operations (LP) and (R). Also, avd hardly affects purge operations since the numbers of purged tombstones are similar. Actually, in our experiments, one tombstone is purged for every three purge operations on average.

In treedoc [26] and logoot [40], overhead was evaluated; tombstones and fixed-size indices incur overhead in treedoc, and unbounded indices do in logoot. Though the workloads are obtained from Wikipedia or latex revisions, they cannot model the real-time collaborations where multiple sites concurrently participate. Though overhead depends on workloads in all approaches, purging tombstones in RGAs is less costly than in treedoc, and unlike logoot indices the size of a `Node` is fixed; as the sizes of an `S4Vector` and a `Node` are 12 bytes and 36 bytes,

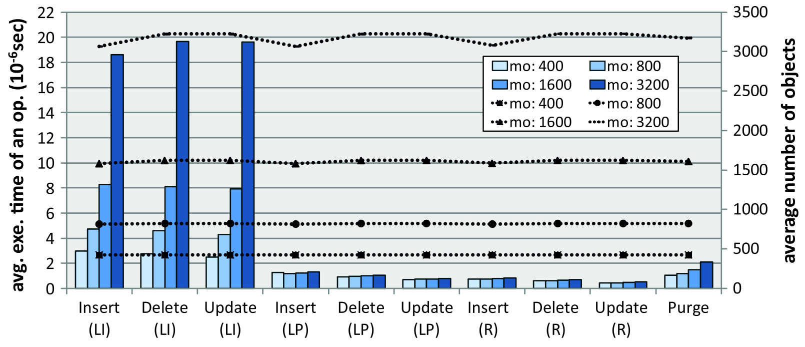

Fig. 11. (Object effect) [$s = 16$ sites, $N = 6250$ ops, avd $= 25.7$ turns] With respect to mo, the average execution time of each operation (the column chart with the left y-axis) and the average number of objects including tombstones (the line chart with the right y-axis).

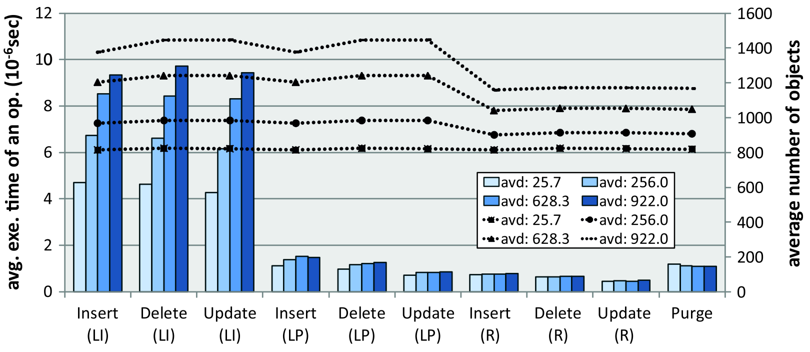

Fig. 12. (Delay effect) [$s = 16$ sites, $N = 6250$ ops, mo $= 800$ objs] With respect to avd, the average execution time of each operation (the column chart with the left y-axis) and the average number of objects including tombstones (the line chart with the right y-axis).

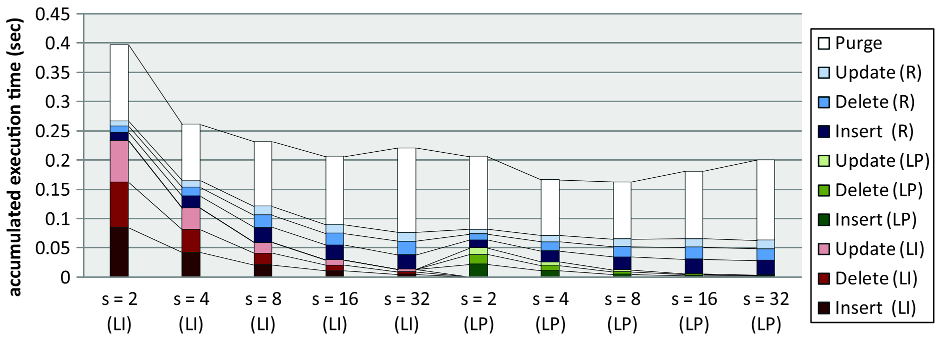

Fig. 13. (Site effect) [$s \times N = 100,000$, mo $= 800$ objs, avd $= 16.5$–27.3 turns] With respect to s, accumulated execution time of total 100,000 operations.

respectively, the overhead of 33,000 tombstones is about 1.1 MB without purging tombstones. In addition, an update, emulated with consecutive deletion and insertion in the two approaches, also produces a tombstone or makes indices long; meanwhile, *Update*s incur no additional overhead in RGAs.

To verify the scalability issue, addressed in Section 7, we have a site execute total 100,000 operations which are equally generated by all s sites; hence, a site executes 100,000/s local operations and $100{,}000 \times (1 - 1/s)$ remote and purge operations. With respect to s, the accumulated execution times are presented in Fig. 13. Except the times for purge operations, the accumulated execution times tend to decrease as s is getting larger. However, the times for purge operations, invoked more frequently, offset the gains by replacing a slow local operation with a fast remote one. Notwithstanding, the result proves that RGAs are scalable with respect to the number of sites owing to the excellent performance of the remote operations.

9. 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. We have demonstrated this outstanding performance of RGAs with intensive workloads. Furthermore, since the SVI scheme autonomously validates the causality and intention of an RGA operation, reliability would be enhanced. The work presented here, therefore, has profound implication for future studies of other RADTs such as various tree data types.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 2010-0000829). We would like to give warm thanks to all the anonymous reviewers and special thanks to Dr. Marc Shapiro at INRIA.

References

[1] B.R. Badrinath, K. Ramamritham, Semantics-based concurrency control: beyond commutativity, ACM Transactions on Database Systems 17 (1) (1992) 163–199.
[2] V. Balakrishnan, Graph Theory, McGraw-Hill, New York, 1997.
[3] P.A. Bernstein, N. Goodman, An algorithm for concurrency control and recovery in replicated distributed databases, ACM Transactions on Database Systems 9 (4) (1984) 596–615.
[4] K. Birman, R. Cooper, The ISIS project: real experience with a fault tolerant programming system, SIGOPS Operating Systems Review 25 (2) (1991) 103–107.
[5] K.P. Birman, A. Schiper, P. Stephonson, Lightweight causal and atomic group multicast, ACM Transactions on Computer Systems 9 (3) (1991) 272–314.
[6] 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, pp. 1–12.
[7] C.A. Ellis, S.J. Gibbs, Concurrency control in groupware systems, in: Proceedings of ACM International Conference on Management of Data, SIGMOD, 1989, pp. 399–407.
[8] C.A. Ellis, S.J. Gibbs, G. Rein, Groupware: some issues and experiences, Communications of the ACM 34 (1) (1991) 39–58.
[9] 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.
[10] R.A. Golding, Weak-consistency group communication and membership, Ph.D. Thesis, University of California, Santa Cruz, 1992.
[11] Google Inc., Google wave protocols, 2009. http://www.waveprotocol.org/.
[12] 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, pp. 173–182.
[13] 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, pp. 207–217.
[14] 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, pp. 586–593.
[15] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Communications of the ACM 21 (7) (1978) 558–565.
[16] D. Li, R. Li, Preserving operation effects relation in group editors, in: Proceedings of ACM Conference on Computer Supported Cooperative Work, CSCW, 2004, pp. 457–466.
[17] R. Li, D. Li, Commutativity-based concurrency control in groupware, in: International Conference on Collaborative Computing: Networking, Applications and Worksharing, CollaborateCom, 2005, p. 10.
[18] D. Li, R. Li, A performance study of group editing algorithms, in: Proceedings of International Conference on Parallel and Distributed Systems, ICPADS, IEEE Computer Society, 2006, pp. 300–307.
[19] R. Li, D. Li, A new operational transformation framework for real-time group editors, IEEE Transactions on Parallel and Distributed Systems 18 (3) (2007) 307–319.
[20] B. Lushman, G.V. Cormack, Proof of correctness of Ressel's adOPTed algorithm, Information Processing Letters 86 (6) (2003) 303–310.
[21] S. Mishra, L.L. Peterson, R.D. Schlichting, Implementing fault-tolerant replicated objects using Psync, in: Proceedings of Symposium on Reliable Distributed Systems, 1989, pp. 42–52.
[22] G. Oster, P. Molli, P. Urso, A. Imine, Tombstone transformation functions for ensuring consistency in collaborative editing systems, in: International Conference on Collaborative Computing: Networking, Applications and Worksharing, CollaborateCom, 2006, pp. 1–10.
[23] G. Oster, P. Urso, P. Molli, A. Imine, Real time group editors without operational transformation, Rapport de recherche RR-5580, INRIA, May 2005.
[24] 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, pp. 259–268.
[25] R. Prakash, M. Raynal, M. Singhal, An adaptive causal ordering algorithm suited to mobile computing environments, Journal of Parallel and Distributed Computing 41 (2) (1997) 190–204.
[26] 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.
[27] 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, pp. 288–297.
[28] 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.
[29] H.-G. Roh, J.-S. Kim, J. Lee, S. Maeng, Optimistic operations for replicated abstract data types, Technical Report CS-TR-2009-318, KAIST, 2009.
[30] Y. Saito, M. Shapiro, Optimistic replication, ACM Computing Surveys 37 (1) (2005) 42–81.
[31] M. Shapiro, N. Preguiça, Designing a commutative replicated data type, Rapport de recherche RR-6320, INRIA, October 2007.
[32] M. Suleiman, M. Cart, J. Ferrié, Concurrent operations in a distributed and mobile collaborative environment, in: Proceedings of International Conference on Data Engineering, ICDE, IEEE Computer Society, 1998, pp. 36–45.
[33] C. Sun, D. Chen, Consistency maintenance in real-time collaborative graphics editing systems, ACM Transactions on Computer–Human Interaction 9 (1) (2002) 1–41.
[34] 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, pp. 59–68.
[35] 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) (1998) 63–108.
[36] 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, pp. 172–182.
[37] R.H. Thomas, A majority consensus approach to concurrency control for multiple copy databases, ACM Transactions on Database Systems 4 (2) (1979) 180–209.
[38] 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, pp. 171–180.
[39] W.E. Weihl, Commutativity-based concurrency control for abstract data types, IEEE Transactions on Computers 37 (12) (1988) 1488–1505.
[40] 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.
[41] 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.

Hyun-Gul Roh received his B.S. degree in computer science from Yonsei University, Korea, in 2002, and is due to receive his Ph.D. degree in computer science from KAIST (Korea Advanced Institute of Science and Technology), in 2011. Currently, he is working as a research intern at INRIA from September, 2010. His research interests include distributed and replication systems, especially, collaboration and version vectors.

Myeongjae Jeon is currently a Ph.D. student in computer science at Rice University. He received his M.S. degree in computer science from Korea Advanced Institute of Science and Technology (KAIST) in 2009 and his B.E. degree in computer engineering from Kwangwoon University in 2005. His research interests include machine virtualization, distributed systems, and storage systems.

Jin-Soo Kim received his B.S., M.S., and Ph.D. degrees in Computer Engineering from Seoul National University, Korea, in 1991, 1993, and 1999, respectively. He was with the IBM T.J. Watson Research Center as an academic visitor from 1998 to 1999. He was the faculty in computer science department at KAIST from 2002 to 2008. Currently, he is the faculty of SungKyunKwan University. His research interests include operating systems, distributed file systems, and grid computing.

Joonwon Lee received his B.S. degree from Seoul National University in 1983 and his Ph.D. degree from the Georgia Institute of Technology in 1991. From 1991 to 1992, he was with IBM T.J. Watson Research Center. After working for IBM, he was the faculty of KAIST from 1992 to 2008. Currently, he is the faculty of SungKyunKwan University. His research interests include operating systems, virtual machines, and parallel processing.