Optimistic Operations for Replicated Abstract Data Types

Hyun-Gul Roh*
KAIST

Jin-Soo Kim† Joonwon Lee‡
Sungkyunkwan University

Seunghyoul Maeng§
KAIST

CS/TR-2009-318

September 30, 2009

KAIST
Department of Computer Science

* hgroh@camars.kaist.ac.kr
† jinsookim@skku.edu
‡ joonwon@skku.edu
§ maeng@camars.kaist.ac.kr
Optimistic Operations for Replicated Abstract Data Types

Hyun-Gul Roh\textsuperscript{a,1}, Jin-Soo Kim\textsuperscript{b}, Joonwon Lee\textsuperscript{b}, Seungryoul Maeng\textsuperscript{a}

\textsuperscript{a}Department of Computer Science, KAIST, Daejeon 305-701, Republic of Korea
\textsuperscript{b}School of Information and Communication Engineering, Sungkyunkwan University (SKKU), Suwon, Republic of Korea

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

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

1. Introduction

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

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

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

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

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

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

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

Preprint submitted to **

September 30, 2009
RADT operations to satisfy OC. Second, precedence transitivity (PT) is a guideline on how to design remote operations in order to make operations commute and preserve their intentions. RADTs require precedence rules to reconcile the conflicting intentions. PT is a condition that all precedence rules are transitive. PT explains how concurrent relations have to be designed against happened-before relations while OC is a general principle only on concurrent operations.

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

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

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

2. Problem Definition

2.1. Preliminary: Causality preservation among operations

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

While no uniquely correct order is defined for concurrent operations, partial orders defined by happened-before relations need preserving at every site [13, 19]. The reason is owing to causality that might exist between operations; e.g., imagine \( O_4 \) is to delete the object inserted by \( O_2 \) in Fig. 1. Causality has been detected and preserved by vector clock timestamping [4, 19]. Following Birman et al.’s CBCAST scheme [4], in RADTs of N sites, site i updates its own N-tuple vector clock \( \nabla_i \) as lines 2, 8, 11, and 17 in Algorithm 1. To preserve causality, causally unready operations are delayed using a queue (lines 13–15). When an operation \( O \) arrives at site j (\( i \neq j \)) with its timestamp \( \nabla_j \), \( O \) is causally ready if \( \nabla_j[j] = \nabla_i[j] + 1 \) and \( \nabla_j[k] \leq \nabla_i[k] \) for \( 0 \leq k \leq N - 1 \). To illustrate, consider site 2 in Fig. 1. After \( O_1 \)’s execution, site 2 has \( \nabla_2 = [1,0,1] \). When \( O_5 \) arrives at site 2 with \( \nabla_5 = [2,1,1] \), it is causally ready; thus, it is delayed until executing \( O_2 \). According to Birman et al.[24], CBCAST is 3 to 5 times faster than ABCAST that supports the total ordering. Nevertheless, this causality preservation scheme is so strict that it might incur a chain of inessential delays when a site fails to broadcast operations; in Section 7, we discuss relaxing of this scheme.

2.2. System model of RADTs

A replicated abstract data type (RADT) is extended from a normal ADT. The system model of RADTs can be summarized below, and the main control loop is presented in Algorithm 1.
An RADT is a particular data structure with a definite set of operation types (OPTYPE).

RADTs are multiple copies of an RADT, each of which is replicated at one of the distributed sites.

At a site, a local operation is the one issued locally while a remote operation is the one received from a remote site.

At a site, every local operation is immediately executed on the RADT of the site according to the local algorithm of its operation type.

Every local operation modifying its local RADT is broadcast to the other sites in the form of the remote operation.

At a site, every remote operation is immediately executed according to the remote algorithm of the operation type if it is causally ready by its vector clock.

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

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

**Definition 1. Eventual consistency**

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

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

2.3. Definitions of RADTs and inconsistency problems

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

A replicated fixed-size array (RFA) is an array of a fixed number of elements with OPTYPE: [Write, Read]. In Algorithm 2, their local algorithms are presented. An element is an object container of RFAs. A local Write(int i, Object o) replaces the object at the ith element of the array with the new object o. In RFAs, different execution orders lead to inconsistency. For example, if three operations of Fig. 2 are given as RFA[1]: Write(1, o1), RFA[2]: Write(1, o2), and RFA[3]: Write(1, o3) in RFAs, the element of index 1 lastly contains o1 at sites 1 and 2, but o2 at site 0.

**Algorithm 2 The local algorithms for RFA operation types**

1. Write(int i, Object o):
   2. if (RFA[i] exists) //RFA[i]: the ith element
   3. RFA[i].obj := o; //replaces the ith object with o
   4. return true;
   5. else return false;
6. Read(int i):
   7. if (RFA[i] exists) return RFA[i].obj;
   8. else return nil;

Hash tables are extended into replicated hash tables (RHTs), which access shared objects in slots by hashing unique keys with OPTYPE=[Put, Remove, Read], as shown in Algorithm 3. This paper assumes that an RHT resolves key collisions by separate chaining scheme. If a Put performs on an existing slot, it updates the slot with its new object. RHTs have an additional source of inconsistency because Puts and Removes dynamically create and destroy slots. This necessitates the idea of tombstones, which are invisible object container left behind after Remove [10, 11, 12]. Despite the tombstone, if the remote algorithms are the same as the local ones, RHTs might diverge. Consider the scenario of Fig. 2 again, where the operations are given as RFA[1]: Remove(k1), RFA[2]: Put(k1, o2), and RFA[3]: Put(k1, o3). Having executed the two Puts, sites 1 and 2 have different objects for k1. After executing all the three, sites 1 and 2 have the tombstone for k1 while site 0 has o2 for k1.

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

Since the order among nodes matters, RGAs have additional inconsistency problems. Suppose that the operations in Fig. 2 are given as $O_1$: Update(2, $o_1$), $O_2$: Insert(1, 0), and $O_3$: Insert(1, $o_1$) and that they are executed on an initial RGA $[o_0,o_2]$ by the local algorithms. After executing both $O_2$ and $O_3$, two results are available according to the execution order: $[o_0,o_0,o_2]$ at site 1, and $[o_0,o_0,o_2]$ at site 2. Though both results are not wrong, only one must be chosen for consistency.

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

**Definition 2. Intention of an operation**

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

Obviously, the local algorithms accomplish all the intentions of local operations. In RGAs, intentions can be violated at remote sites because Inserts or Deletes change integer indices of some nodes located behind their target nodes. This intention violation problem was first addressed by Sun et al. [19]. In the example, though $O_1$ intends to replace $o_0$ on $[o_0,o_2]$ with $o_1$ at its local site, $O_1$ of site 2 may update $o_0$ on $[o_0,o_0,o_2]$, which is not the intention of $O_1$. The operations of RGAs incur many other puzzling problems [20]. We show that RGAs can solve them with concrete examples in Section 5.

---

**Algorithm 3 The local algorithms for RHT operation types**

1. Put(Key k, Object o):  
2. $s := $RHT[hash(k)];  
3. if($s$ = nil) return false;  
4. else $new_s :=$ make a new slot;  
5. $new_s$ = obj;  
6. $RHT[hash(k)] := new_s$;  
7. return true;  
8. Remove(Key k):  
9. $s := $RHT[hash(k)];  
10. if($s$ = nil) return false;  
11. $s$ = obj = nil;  
12. return true;  
13. Read(Key k):  
14. $s := $RHT[hash(k)];  
15. if($s$ = nil or $s$.obj = nil) return nil;  
16. return $s$.obj;

---

**Algorithm 4 The local algorithms for RGA operation types**

1. findlist(int i):  
2. $n :=$ head of the linked list;  
3. int $k := 0$;  
4. while($k$ < $i$)  
5. $n := n$.link;  
6. if($n$ = tail) return nil;  
7. if($n$.obj = nil) return nil;  
8. return $n$;  
9. findlink(node n):  
10. if($n$.obj = nil) return nil;  
11. else return $n$;  
12. Insert(int i, Object o):  
13. if((refer$O_n :=$ findlist(i)) = nil) return false;  
14. new$O_n :=$ make a new node;  
15. return true;  
16. Delete(int i):  
17. if((target$O_n :=$ findlist(i)) = nil) return false;  
18. target$O_n$.obj = nil;  
19. return true;  
20. Update(int i, Object o):  
21. if((target$O_n :=$ findlist(i)) = nil) return false;  
22. return true;  
23. return true;  
24. return true;  
25. return true;  
26. Read(int i):  
27. if((target$O_n :=$ findlist(i)) = nil) return nil;  
28. return target$O_n$.obj;

---

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

### 3. Operation Commutativity

RADTs allow different sites to execute operations in different orders. To denote an execution order between two operations, ‘$\Rightarrow$’ is used; e.g., $O_a \Rightarrow O_b$ 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., $R_{S_0} \Rightarrow R_{S_1} \Rightarrow R_{S_2}$ means $O_a$ and $O_b$ change a replica state $R_{S_0}$ into $R_{S_1}$ and $R_{S_2}$ one after the other and can be shortened by $R_{S_0} \Rightarrow O_{a,b} R_{S_2}$. Though time-space diagrams are intuitive and illustrative, we present a better definition for formal analysis as follows.

**Definition 3. Causally executable graph (CEG)**

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

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

**Lemma 1. A CEG has no cycle with its directed edges and is a complete graph.**
PROOF. According to the definitions of happened-before and concurrent relations [23], they are not defined reflexively and happened-before relations are all transitive; thus, G has no cycle. Unless every pair of two distinct operations is in happened-before relation, it is concurrent; hence, G is complete.

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

**Definition 4. Causally executable sequence (CES)**

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

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

**Definition 5. Causally executable sequence set (CESS)**

*Given a CEG \( G=\langle V, E \rangle \), a set of CESes is a causally executable sequence set of \( G \), denoted as \( S(G) \), iff: \( S(G) \) contains all the possible CESes derived from \( G \).*

According to the system model, RADTs permit the execution of every CES in a CESS. Eventual consistency, therefore, can be achieved if all the CESes produce the same replica state. To observe the relationship among CESes of a CEG, consider a CES \( s_1: O_1\rightarrow O_2\rightarrow O_3\rightarrow O_4\rightarrow O_5 \) in the CEG of Fig. 3. If a pair of adjacent operations on \( s_1 \) is concurrent, another CES can be derived by swapping the order of the pair. If the execution order of \( O_1\parallel O_2 \) is swapped, another CES \( s_2: O_2\rightarrow O_1\rightarrow O_3\rightarrow O_4\rightarrow O_5 \) can be obtained. Only if both \( O_1\parallel O_2 \) and \( O_2\parallel O_1 \) yield the same result from an identical replica state, will \( s_1 \) and \( s_2 \) produce a consistent result. In this regard, given a CES \( s \) from a CEG \( G \), if we show that the other CESes in \( S(G) \) can be derived from \( s \) and find the condition that they yield the same result, eventual consistency can be ensured. This is the basic concept of operation commutativity (OC), which is developed from commutative relation as follows:

**Definition 6. Commutative relation ‘\( \leftrightarrow \)’**

*Given two concurrent operations \( O_a \) and \( O_b \), they are in commutative relation, expressed as \( O_a\leftrightarrow O_b \), iff: for any identical replica state \( RS_0 \), when \( RS_0 \models RS_1 \) and \( RS_0 \models RS_2 \), \( RS_1 \) is equal to \( RS_2 \) (\( RS_1 = RS_2 \)).*

To illustrate the effect of a commutative relation in CESes, consider two CESes in Fig. 1: \( s_1: O_1\rightarrow O_2\rightarrow O_3\rightarrow O_4\rightarrow O_5 \) at site 0 and \( s_2: O_3\rightarrow O_4\rightarrow O_1\rightarrow O_2\rightarrow O_5 \) at site 1. Even though \( O_1\parallel O_5 \) is \( O_1\leftrightarrow O_5 \), we are not sure if this commutative relation helps \( s_1 \) and \( s_2 \) to be consistent because the initial states of \( O_1\parallel O_5 \) are different and because other operations may or may not intervene between them. Indeed, to make all the CESes of a CEG consistent, the following condition is necessary.

**Definition 7. Operation commutativity (OC)**

*Given a CEG \( G=\langle V, E \rangle \), 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 \( s_1 \) and \( s_2 \) again. If OC holds in the CEG of Fig. 3, \( O_1 \leftrightarrow O_2 \), \( O_4 \leftrightarrow O_5 \), and \( O_1 \leftrightarrow O_3 \) are ensured. By applying the properties of those commutative relations in sequence, \( s_1 \) can be transformed into \( s_2 \). For completeness, we present the following theorem and prove it in Appendix A:

**Theorem 1.**

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

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

4. Precedence Transitivity

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

**Definition 8. Causal object for an operation (cobject)**

*Given an operation \( O \) modifying RADTs, an object container is a causal object (cobject) for \( O \), if it has existed in its local RADT at the issued time of \( O \) and affects the execution of the local and remote algorithms of \( O \).*
DEFINITION 9. Effective operation on a container (eoperation)
An operation is an effective operation (eoperation) on an object container, if its local or remote algorithm succeeds in creating/deleting/updating the container.

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

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

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

DEFINITION 11. Precedence relation ‘→’
Given two operations O_a and O_b, O_b takes precedence over O_a, denoted as O_a → O_b, iff: (1) O_a → O_b or (2) for O_a ∥ O_b, the intention of O_b is more preservable than that of O_a.

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

Precedence relations for a set of operations should be carefully enacted so that they will not conflict with each other. To illustrate, suppose the operations in Fig. 2 are given as O_1: Write(1, o_1), O_2: Write(1, o_2), and O_3: Write(1, o_3). For each pair of operations, assume the following precedence relations: O_1 → O_2, O_3 → O_1 (from O_3 → O_1), and O_2 → O_3. These precedence relations on an element are expressed with a graph called precedence relation graph (PRG). A PRG can be derived from a CEG by keeping the directed edges intact and by choosing a direction for each undirected edge. Such a directed complete graph is called a tournament in graph theory [25].

PRG of the above precedence relations is shown in Fig. 4-(a). Assuming that three operations are executed according to this PRG, the element of index 1 at each site will be as follows.

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_1 is effectively executed. At each site, the second operation is effectively executed if it takes precedence over the first one, otherwise it does nothing. Thus, the elements of index 1 become o_1 at site 0 due to O_3 → O_1 and o_3 at sites 1 and 2 due to O_2 → O_3, respectively.

When the third operation arrives at each site, its execution must obey the precedence relations with the previous two operations. For example, at site 1, the execution of O_1 should obey both O_1 → O_2 and O_1 → O_3. However, O_1 cannot satisfy both; if O_1 does nothing according to O_1 → O_2, it violates O_1 → O_1, but otherwise O_1 → O_2 is disobeyed. We can find the reason from the PRG of Fig. 4-(a). Note that PRG (a) has a cycle, that is, the precedence relations are intransitive such that O_1 → O_2 and O_2 → O_3, but not O_1 → O_3. Hence, obeying two precedence relations among the three inevitably leads to violating the rest in this PRG. On the other hand, another PRG shown in Fig. 4-(b) is an acyclic tournament. Since all the edges in an acyclic tournament are transitive (see theorem 3.11 in [25]), the third operation at each site can be applied while obeying all the precedence relations; thus, o_1, o_2, and o_3 become o_1. In the final analysis, we suggest the following condition as a key principle to realize OC.

DEFINITION 12. Precedence transitivity (PT)
Given a CEG G=(V, E), precedence transitivity holds in G, iff: if O_a → O_b and O_b → O_c for ∀(O_a ≠ O_b ≠ O_c) ∈ V, O_a → O_c.

PT is a condition in which all precedence relations are transitive. Since an acyclic tournament has a unique Hamiltonian path (see theorem 3.12 in [25]), which visits all the vertices of the graph once, precedence relations are totally ordered; e.g., the PRG of Fig. 4-(b) are ordered as O_2 → O_3 → O_1. Note that PT is not a principle that regulates operation executions and operations need never be executed in this order. Instead,
Since precedence relations, in which PT is grounded, are differently implemented according to specific operation types, it is difficult to prove that, without loss of generality, PT guarantees the eventual consistency. In this paper, we apply PT to the implementations of operation types, and thus make pairs of operation types commute. Hence, in Appendix B, we prove OC for every pair of operation types to which PT is applied. As the proofs show, PT is a successful guideline to achieve OC. Although this paper uses PT as a means of achieving OC, PT itself could accomplish the eventual consistency for the implementations of RFAs or RHTs. Furthermore, unlike OC, PT might be able to ensure consistency for the execution sequences that are not CESes. We will discuss this issue further in Section 7.

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

5. RADT Implementations

5.1. The S4Vector

For optimization purpose, we define a quadruple vector type.

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

Let $\mathcal{V}_O$ be the vector clock of an operation issued at site $i$. Then, $\mathcal{V}_O$ of the S4Vector type can be derived from $\mathcal{V}_O$ as follows: (1) $\mathcal{V}_O[ssn]$ is the global session number that increases monotonically, (2) $\mathcal{V}_O[sid]$ is the site ID which is unique to the site, (3) $\mathcal{V}_O[sum]$ is $\sum u_i \mathcal{V}_O[i]$, and (4) $\mathcal{V}_O[seq]$ is $\mathcal{V}_O[i]$, which is reserved for purging tombstones (see Section 5.6). To illustrate, suppose $\mathcal{V}_O = [1, 2, 3]$ is the vector clock of an operation that is issued at site 0 at session 4. Then, the S4vector of $\mathcal{V}_O$ is $\mathcal{V}_O = [4, 0, 6, 1]$. As a unit of collaboration, a session begins with the same initial vector clocks and the same RADTs at all sites. When a copy of RADT is reloaded over all sites and a new editing begins, the session number $\mathcal{V}_O[ssn]$ increases. The S4vector of an operation is globally unique because $\sum u_i \mathcal{V}_O[i]$ is unique to every operation issued at a site. We define an order between two S4vectors as follows.
Appendix in algorithms. Formal proofs of consistency, i.e., OC, is presented.

Operation; thus, PT is applied to date and thus necessarily succeeds the s4vector of any last eop-

rithm for comes the new last eoperation of the Element 
tively, only when \( \vec{s}_O \rightarrow \vec{s}_P \) succeeds \( \vec{s}_O \rightarrow \vec{s}_P \) respectively, as a mark of a tombstone.

Lemma 2. The s4vector orders are transitive.

Proof. The s4vectors issued at a site are totally ordered because \( \vec{s}[\text{sum}] \) increases monotonously. If \( \vec{s}[\text{sum}] \)'s are equal, they must be issued at different sites and are ordered by unique \( \vec{s}[\text{sid}] \)'s. The s4vectors of different sessions are ordered by monotonous \( \vec{T}[\text{ssn}] \)'s. Since all s4vectors are totally ordered by the three conditions, s4vector orders are transitive.

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

5.2. Replicated Fixed-Size Arrays

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

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

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

Algorithm 5 The remote algorithm for Write type

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

In RFAs, a remote Write(int i, Object* o) replaces obj and \( \vec{T}_P \) of the i-th Element with a new object o and \( \vec{T}_O \), respectively, only when \( \vec{s}_O \rightarrow \vec{s}_P \) succeeds \( \vec{s}_O \rightarrow \vec{s}_P \) respectively, as a mark of a tombstone.

5.3. Replicated Hash Tables

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

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

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

Algorithm 6 The remote algorithm for Put type

1. Put(Key k, Object* o)
2. Slot *pre_s := nil;
3. Slot *s := RHT[hash(k)];
4. while(s != nil and s.k != k) // find slot in the chain;
5. pre_s := s;
6. s := s.next;
7. if(s != nil and \( \vec{T}_O < \vec{T}_P \)) return false;
8. else if(s != nil and s is a tombstone) Cemetery.withdraw(s);
9. else if(s is not tombstone)
10. Cemetery.enroll(s);
11. p := make new slot;
12. if(pre_s != nil) pre_s.next := s;
13. s.k := k;
14. s.next := nil;
15. s.obj := o;
16. return true;

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

Algorithm 7 The remote algorithm for Remove type

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

A Remove first finds its target Slot addressed by its key k (lines 2–3). Although a local Remove can be invoked on a non-existent Slot, it is not propagated to remote sites as shown in Algorithm 1 and 3. Therefore, if there is no Slot for a key, a remote Remove throws an exception and does nothing (lines 4–6). In line 7, a Remove is ignored if its s4vector precedes the last operation’s, or otherwise it demotes its target Slot into a tombstone by assigning nil and \( \vec{s}_P \rightarrow \vec{T}_O \rightarrow \vec{T}_P \) of the tombstone, respectively, as a mark of a tombstone (lines 9–10). Local Reads, therefore, regard tombstones as no Slots. Unless a tombstone remains for a key, any concurrent operation
will miss its object, and thus precedence relation with the last Remove is lost. Hence, if an operation arrives to a tombstone, its effectiveness can be decided based also on s4vector orders. Recall the example of RHTs in Section 2.3. Here, O₁ becomes the last eoperation of the tombstone for k₁ while O₂ is ignored at site 0.

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

5.4. The S4Vector Index (SVI) scheme for RGAs

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

```
struct Node
{
    Object* obj;
    S4Vector $\mathcal{S}_I$; // for a hash key and precedence of Inserts
    S4Vector $\mathcal{S}_P$; // for precedence of Deletes and Updates
    Node* next; // for the hash table
    Node* link; // for the linked list
};
Node* Cemetery[HASH_SIZE];
Node head; // the starting point of the linked list
```

A Node of an RGA has five variables. $\mathcal{S}_k$ is the s4vector index as a hash key, and is used for precedence of Inserts. For precedence of Deletes and Updates, $\mathcal{S}_p$ 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. 6 shows examples of an RGA data structure which is constructed with a linked list combined with a hash table. The local algorithms also maintain and operate on such structures. To illustrate, assume that, at session 1, site 2 invokes Insert(3, o₁) with a vector clock $\mathcal{V}_O := [3, 1, 2]$ on the RGA structure of Fig. 6-(a), which can be denoted as $[o₁|0|2|0|5|0]$. As shown in Algorithm 4, the local Insert algorithm first finds its reference Node, i.e., o₁, from the linked list, then creates a new Node. This Node contains the new object o₁, and the s4vector $\mathcal{S}_O = [1, 2, 6, 2]$ is assigned into both $\mathcal{S}_k$ and $\mathcal{S}_p$. Then, this Node is placed in the hash table by hashing $\mathcal{S}_O$ as a key and is connected to the linked list as shown in Fig. 6-(b); thus, $[1,0,5,3]$. We assume line 16 of Algorithm 4 does this.

Once $\mathcal{S}_k$ 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, o₃) will be transformed into Insert([1, 0, 5, 3], o₃) before it is broadcast because $\mathcal{S}_k$ of the third Node is [1, 0, 5, 3] in Fig. 6-(a). In this way, three local RGA operations are broadcast to remote sites in the following forms: Insert(S4Vector $\mathcal{i}$, Object* o), Delete(S4Vector $\mathcal{i}$), and Update(S4Vector $\mathcal{i}$, Object* o), where $o$ is a new object, and where $\mathcal{i}$ is the s4vector index, i.e., $\mathcal{S}_k$ of the reference Node that the local Insert refers to or the target Node that the local Delete or the local Update is executed on. If an Insert adds its object at the head of an RGA, $\mathcal{i}$ will be nil.

5.5. Three remote operations for RGAs

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

```plaintext
Insert(S4Vector t, Object* o)
1. Node* ins;
2. if(t != nil) // find reference node in hash table;
3. ref := RGA[hash(t)]:
4. while(ref != nil and ref.T != t) ref := ref.next;
5. if(ref != nil) throw NoRefObjException;
6. return false;
7. ins := make new Node;
8. ins.T := T;
9. ins.next := RGA[hash(t)]: // place the new node
10. RGA[hash(t)] := ins; // into the hash table;
11. if(t = nil)
12. if(head.link = tail or head.T < T)
13. ins.link := head.link;
14. head.link := ins;
15. return true;
16. else ref := head;
17. while(ref.link = tail and ref.T < T)
18. ins.link := ref.link;
19. ref := ref.link;
20. ref := ins;
21. return false;
```

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

Example 1. [Fig. 2] dOPT puzzle, on initial RGA

I1 := Insert(0 = nil, o1) with [1, 0, 1] <⇒ T1 = [1, 0, 2, 1].
I2 := Insert(0 = nil, o2) with [0, 1, 0] <⇒ T2 = [1, 1, 1, 1].
I3 := Insert(0 = nil, o3) with [0, 0, 1] <⇒ T3 = [1, 2, 1, 1].

We assume the initial RGA are empty (∅), and the operations are propagated as shown in Fig. 2; thus, I1, I2, and I3 correspond to O1, O2, and O3 of Fig. 2, respectively. All their remote forms have nils as the s4vector indices because their intentions are to insert new nodes at the head. In this example, i1, i2, and i3 are the s4vectors derived from the left vector clocks on the assumption of session 1. Due to 1 < 3 < 2, i2 < i3 < i1, I1 is most preservable, then I3, and that of I2 is least. PT is realized in Inserts as follows.

At site 0, when l1 arrives in the form of Insert(nil, o1), o1 is placed at the head by lines 16–20. Then, I1 is executed as [o1, o3] by the local Insert algorithm. Finally, when l1 arrives, head.3 refers to i1 i1 i2 i3 <⇒ T3 = [1, 3], i2 inserts o2 at the tail as [o1, o2, o3].

Algorithm 9 The remote algorithm for Delete type

```plaintext
Delete(S4Vector t)
1. Node* n := RGA[hash(t)]:
2. while(n != nil and n.T != t) n := n.next;
3. if(n = nil) throw NoRefObjException;
4. return false;
5. if(n is not a tombstone)
6. Cemetery.enroll(n);
7. n.obj := nil;
8. n.T := T;
9. return true;
```

The local and remote Delete algorithms also leave a tombstone behind. In Algorithm 9, a remote Delete finds its target Node with t via the hash table (lines 2–3); otherwise, it throws an exception (lines 4–6). Regardless of s4vector orders, Deletes assign nil and T to obj and T (not T), respectively, as a mark of a tombstone, and enrolls it into Cemetery (lines 8–10). Unlike RHTs, an RGA tombstone never revives. In Algorithm 4, findlist and findlink functions exclude tombstones from counting. Being invisible to local operations, a tombstone is never employed as a reference or a remote Node with an integer index. For example, in Fig. 6-(a), local Insert(4, o4) refers to the Node of o5 instead of the tombstone of τ2, and is transformed into remote Insert([1, 1, 1, 1], o5).

Algorithm 10 The remote algorithm for Update type

```plaintext
Update(S4Vector t, Object* o)
1. Node* n := RGA[hash(t)]:
2. while(n != nil and n.T != t) n := n.next;
3. if(n = nil) throw NoRefObjException;
4. return false;
5. if(n is not a tombstone)
6. Cemetery.enroll(n);
7. n.obj := o;
8. n.T := T;
9. return true;
```

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

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

Example 2. [Fig. 1] Initially, RGAs=[o_o] with $\rightarrow_a=[1,0,1,1]$, $U_1(O_1) : Update(I=\bar{i}_a, o_i)$ with $[1,0,0] \rightarrow \bar{i}_1 = [2,0,1,1]$, $U_2(O_2) : Update(I=\bar{i}_a, o_i)$ with $[0,1,0] \rightarrow \bar{i}_2 = [2,1,1,1]$, $D_2(O_1) : Delete(I=\bar{i}_a)$ with $[0,0,1] \rightarrow \bar{i}_3 = [2,2,1,1]$, $I_4(O_3) : Insert(0=\bar{nil}, o_i)$ with $[2,1,1] \rightarrow \bar{i}_4 = [2,0,4,2]$, $I_5(O_5) : Insert(I=\bar{i}_a, o_i)$ with $[0,2,0] \rightarrow \bar{i}_5 = [2,1,2,2]$.

For $U_1 \parallel U_2$ being in conflict, the intention of $U_2$ is more preservable than that of $U_1$ due to $\bar{i}_1 < \bar{i}_2$; thus, as shown in Execution 2, $U_1$ is ignored at site 1 by line 8 of Algorithm 10. When $D_2$ conflicts with $U_1$ and $U_2$, $D_2$ always succeeds in leaving the tombstone of $o_o$, i.e., $\tau_i$, regardless of $\bar{s}_p$ of $o_o$, but the Updates do nothing on the tombstone by line 7 of Algorithm 10. Due to $\tau_i$, $I_3$, which is concurrent with $D_2$, can also find its reference node at sites 0 and 2. In addition, when $I_3$ performs at sites 1 and 2, it places $o_o$ in front of $\tau_i$ since Algorithm 8 regards $\tau_i$ as a normal node whose $\bar{s}_i$ is $\bar{i}_a \times \bar{i}_4$ in lines 16–20. In conclusion, RGAs eventually converge at all the sites as follows.

Execution 2. At each site of Fig. 1,

- site 0: $o_o \Rightarrow [\bar{o}_i] \Rightarrow [\bar{o}_i] \Rightarrow [\tau_i] \Rightarrow [\tau_i] \Rightarrow [o_o, \tau_i, o_i]$
- site 1: $o_o \Rightarrow [\bar{o}_i] \Rightarrow [\bar{o}_i, o_o] \Rightarrow [\bar{o}_i, o_o] \Rightarrow [\tau_i, o_o] \Rightarrow [\tau_i, o_o] \Rightarrow [\tau_i, o_o] \Rightarrow [\tau_i, o_o]$
- site 2: $o_o \Rightarrow [\tau_i] \Rightarrow [\tau_i] \Rightarrow [\tau_i] \Rightarrow [\tau_i] \Rightarrow [\tau_i] \Rightarrow [\tau_i] \Rightarrow [\tau_i]$

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

5.6. Cobject preservation

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

Example 3. [Fig. 7] Initially, RGAs=[o_o] with $\rightarrow_a=[1,0,1,1]$, $I_1 : Insert(0=\bar{nil}, o_i)$ with $[1,0,0] \rightarrow \bar{i}_1 = [2,0,1,1]$, $D_2 : Delete(I=\bar{i}_a)$ with $[0,1,0] \rightarrow \bar{i}_2 = [2,1,1,1]$, $I_3 : Insert(I=\bar{i}_a, o_i)$ with $[0,0,1] \rightarrow \bar{i}_3 = [2,2,1,1]$

Two cobjects of $I_1$ are the head and $o_o$, while $o_o$ and the tail are the cobjects of $I_3$. The former cobject looks indispensable to the execution of an Insert because it prescribes the position where an Insert has to be executed at all sites. However, the necessity of the latter cobject might be overlooked, though it is used in the conditions to terminate the comparisons, i.e., lines 17 or 22 in Algorithm 8. In our RGA implementation, if the latter cobject of an Insert is purged before the remote execution of the Insert, Inserts of different intentions can be in conflict. To illustrate, see Execution 3 where $\tau_i$ is assumed to be purged at the time $T_1$ of Fig. 7 (⇒ means purging).

Execution 3. If purging $\tau_i$ at $T_1$, $I_1 : Insert(0=\bar{nil}, o_i)$ with $[1,0,0] \rightarrow \bar{i}_1 = [2,0,1,1]$, $D_2 : Delete(I=\bar{i}_a)$ with $[0,1,0] \rightarrow \bar{i}_2 = [2,1,1,1]$, $I_3 : Insert(I=\bar{i}_a, o_i)$ with $[0,0,1] \rightarrow \bar{i}_3 = [2,2,1,1]$

In this execution, observe the effect of $I_1$ according to the existence of its latter cobject, i.e., $o_o$ or $\tau_i$. At site 0, the local algorithm of $I_1$ places $o_o$ at the head without considering $\bar{s}_p$. Being indispensable to $I_3$ as the former cobject, $\tau_i$ should be retained. At site 2, $I_3$ has to insert $o_o$ in front of $\bar{s}_p$ by line 17 in Algorithm 8, and then $D_2$ performs. Hence, in the final results at sites 0 and 2, $[o_o, o_o]$ is a correct order between $o_o$ and $\bar{s}_p$. At site 1, however, if $\tau_i$ is purged, $I_3$ compares $\bar{i}_1$ with $\bar{s}_3$ of $o_o$ (⇒ $\bar{i}_3$) despite $I_1$ having a different reference; thus, $[o_o, o_o]$. Consequently, the loss of the latter cobject can
lead to the different effect of $I_1$. Instead, if the latter cobject is purged at $T_2$ of Fig. 7, the effect of $I_1$ is consistent with sites 0 and 1 as shown in EXECUTION 4.

EXECUTION 4. At site 1, if purging $\tau_0$ at $T_2$.

site 1: \[o_1o_2P \Rightarrow \tau_0 \Rightarrow [o_1, \tau_0, o_2]_p \Rightarrow [o_1, o_2].\]

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

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

To preserve the latter cobjects of Inserts, an additional condition is needed. Note that, at site 1 in EXECUTION 3, the loss of $\tau_0$ causes problems only when the $s$-vector of $I_1$ precedes $\mathcal{S}_i$ of $o_i$, i.e., the next slot of $\tau_0$. In other words, if it is ensured that the $s$-vector of a new Insert succeeds $\mathcal{S}_i$ of every slot in RGAs, the next slot of a tombstone can substitute the tombstone as a latter cobject. To this end, an RGA needs to maintain VC$_{last}$ which is a set of vector clocks including as many vector clocks as the number of sites $N$, i.e., VC$_{last} = \{\mathcal{V}_{last_0}, \ldots, \mathcal{V}_{last_N}\}$. $\mathcal{V}_{last}$ is the vector clock of the operation that was lastly issued at site $j$ and successfully executed in this site. Using VC$_{last}$, a tombstone $\tau_i$, denoted by $D_i$, can be safely purged if satisfying both of the following conditions.

1. $\tau_i.\mathcal{S}_i[\text{seq}] \leq \min \tau_i.\mathcal{V}[i]$ for $i = \tau_i.\mathcal{S}_i[\text{sid}],$
2. $\tau_i.\text{link}.\mathcal{S}_i[\text{sum}] < \min \tau_i.\text{sum}(\mathcal{S})$ or $\tau_i.\text{link} = \text{tail}.$

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

6. Related Work

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

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

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

**Property 1.** Transformation property 1 ($TP_1$)

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

**Property 2.** Transformation property 2 ($TP_2$)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

As most of optimistic replication systems, RADTs also constrain themselves to preserve the causality defined by Lamport, but this relates to the reliability issue. When a site crashes, some operations issued at this site might be delivered to a part of the other sites. Though such a fault as an operation loss is detected by the causality preservation scheme using vector clocks, it leads to a chain of delays in executing the operations happening after the partially delivered operations at some other sites. In the sense that a fault might result in the failures of
other sites, the reliability could be significantly exacerbated in scalable collaborative applications. In Example 2, under the vector clock scheme, if a site misses \( U_2 \), then \( I_1 \) and \( I_5 \) happening after \( U_2 \) should be delayed. However, \( I_2 \) and \( I_3 \) can perform while \( U_2 \) is being retransmitted because \( U_2 \) has no essential causality with \( I_1 \) and \( I_5 \); in other words, \( I_2 \) and \( I_3 \) can achieve their intentions without \( U_2 \) in RGAs. Therefore, the causality preservation scheme can be relaxed, and this might enhance reliability. However, relaxing causality preservation means to allow sites to execute some operation sequences that preserve only essential causality (say eCES), which belong to a superset of CESS. Since RGA consistency is verified based on OC, it is theoretically not guaranteed that eCESes converge. In fact, though WOOT is the only approach that allows eCESes to be executed, consistency was verified by the model-checker TLC on a specification modeled on the TLA+ specification languages [39]. Since the model checker exhaustively verifies all the states produced by possible operation executions, the states must explode; thus, the verification is made only up to four sites and five characters in WOOT [38]. In any case, for the insertion and deletion types which are executed in eCES order, there is no generalized proof methodology of consistency yet.

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

Especially, in performance, RGAs provide the remote operations of \( O(1) \) with the SVI scheme using s4vectors. This is a significant achievement over previous works and makes RGAs scalable. Furthermore, since the SVI scheme autonomously validates the causality and intention of an RGA operation, reliability would be enhanced. The work presented here has profound implication for future studies of other RADTs such as various tree data types.

A. Proof of Operation Commutativity

To prove Theorem 1, additional definitions are needed.

Definition 14. Happened-before graph (HBG)

Given a CEG \( G=(V, E) \) and \( O_r \in V, \) \( HB(G, O_r)=(V', E') \) is a happened-before graph of \( G \) for \( O_r \), if: \( V' \subseteq V \) consists of operations happened before \( O_r \), i.e., \( V'=\{O_1 \mid (O_1 \rightarrow O_r) \in E \) and \( O_1 \in V\} \), and \( E' \subset E \) consists of all relations composed of operations in \( V' \), i.e., \( E'=\{\{O_1 \rightarrow O_2 \} \mid e_r \in E \) and \( O_1 \in V' \} \cup \{e_s=(O_2 \parallel O_3) \mid e_s \in E \) and \( O_2, O_3 \in V' \} \).

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

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

To illustrate, the HBG for \( O_4 \) of the CEG \( G \) of Fig. 3, \( G_{HB} = HB(G, O_4) = (V_{HB}, E_{HB}) \) is also a CEG; i.e., \( V_{HB} = \{O_1, O_2, O_3\} \) and \( E_{HB} = \{O_1 \parallel O_2, O_1 \parallel O_3, O_2 \parallel O_3\} \). In addition, for \( G_{HB} = HB(G, O_4) \), let \( G_{HB} = (V_{HB}, E_{HB}) \) be defined as follows: \( V_{HB} = V - V_{HB} - \{O_4\} \), \( E_{HB} = \{e_s=(O_2 \parallel O_3) \mid e_s \in E \) and \( O_2, O_3 \in V_{HB}\} \cup \{e_r=(O_1 \rightarrow O_4) \mid e_r \in E \) and \( O_1 \in V_{HB}\}. \)

Note \( G_{HB} \) is also complete and transitive; that is, a CEG. Next, we present the last definition for proving Theorem 1.

Definition 15. Preceding operation set

Given a CES s: \( O_1 \rightarrow \cdots \rightarrow O_{n-1} \rightarrow O_n \), the preceding operation set of \( O_n \), \( P(s, O_n) \), is a set of operations executed before \( O_n \) in \( s \), i.e., \( P(s, O_n)=\{O_1, \cdots, O_{n-1}\} \).

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

Proof. (Proof by contradiction)

Let \( |V| = n \). Assume the contrary there are two CESs \( s_1, s_2 \in S(G) \) that do not converge and are given as \( s_1: a_1 \rightarrow \cdots \rightarrow a_k \rightarrow \cdots \rightarrow a_n \), \( s_2: b_1 \rightarrow \cdots \rightarrow b_k \rightarrow \cdots \rightarrow b_n \) for \( a_i, b_i \in V(1 < k < n) \). If \( RS_0 \Rightarrow RS_1, RS_0 \Rightarrow RS_2 \), then \( RS_1 \neq RS_2 \).

Perform the following procedure on each of \( s_1 \) and \( s_2 \).

Procedure: For \( \exists O \in V, O = a_1 = b_1 (1 \leq x, y \leq n) \), let \( V_{1p}=P(s_1, O), V_{2p}=P(s_2, O), \) and \( G_{HB}=HB(G, O) \), respectively. Then, \( V_{1p} \) and \( V_{2p} \) are shown as below:

8. Conclusions

When developing applications, programmers are used to using various ADTs. Providing the same semantics of ADTs to programmers, RADTs can support efficient implementations of collaborative applications. Operation commutativity and precedence transitivity make it possible to design the complicated optimistic RGA operations without serialization/locking protocols/state rollback scheme/undo-do-redo scheme/OT methods.
The following lemmas and theorems. Object sequences in the RADT state, we specify the state of only the rel-
formed into the same CES because at least one operation is lo-
sequences \( s'_1 \) and \( s'_2 \) are as follows:

\[
\begin{align*}
\mathbf{s}_1 : & \; a_1 \mapsto \ldots \mapsto a_{s-1} \mapsto O \mapsto a_{s+1} \mapsto \ldots \mapsto a_n \\
V_{IP} = & \{ a_1, \ldots, a_{s-1} \} \\
\mathbf{s}_2 : & \; b_1 \mapsto \ldots \mapsto b_{s-1} \mapsto O \mapsto b_{s+1} \mapsto \ldots \mapsto b_n \\
V_{IP} = & \{ b_1, \ldots, b_{s-1} \}
\end{align*}
\]

Move \( \forall d_i \in V_{IP} - V_{HB} \) between \( O \) and \( a_{s+1} \) on \( s_1 \) and \( \forall d_j \in V_{2L} - V_{HB} \) between \( O \) and \( b_{s+1} \) on \( s_2 \), respectively, with preserving their relevant orders. Then, the transformed execution sequences \( s'_1 \) and \( s'_2 \) are as follows:

\[
\begin{align*}
\mathbf{s}'_1 : & \; c_1 \mapsto \ldots \mapsto c_n \mapsto O \mapsto d_1 \mapsto \ldots \mapsto d_p \mapsto a_{s+1} \mapsto \ldots \mapsto a_n \\
V_{HB} = & \{ c_1, \ldots, c_n \} \\
\mathbf{s}'_2 : & \; c_1 \mapsto \ldots \mapsto c_n \mapsto O \mapsto e_1 \mapsto \ldots \mapsto e_y \mapsto b_{s+1} \mapsto \ldots \mapsto b_n \\
V_{HB} = & \{ c_1, \ldots, c_n \}
\end{align*}
\]

In \( s'_1 \) and \( s'_2 \), if \( d \in V_{IP} - V_{HB} \) on \( s'_1 \) and \( y \in V_{2L} - V_{HB} \) on \( s'_2 \), which are concurrent with both \( O \) and \( y \in V_{HB} \), can be moved due to OC. Though \( d \) and \( y \) move between \( O \) and \( a_{s+1} \) and between \( O \) and \( b_{s+1} \), respectively, happened-before relations can be preserved; thus, each of \( s'_1 \) and \( s'_2 \) is also a CES.\cdot

Due to \( s_{1L}, s_{2L} \in S(G_{HB}) \) and \( s_{1R}, s_{2R} \in S(G_{HB}) \), the procedure can perform on each of \( s_{1L}, s_{2L} \) of \( G_{HB} \) and \( s_{1R}, s_{2R} \) of \( G_{HB} \) again. If the procedure performs recursively until one or no operation remains in the result CESes, \( s_1 \) and \( s_2 \) are transformed into the same CES because at least one location is located at the same position on the two CESes once the procedure performs. Thus, if \( RS_1 \supseteq RS_1 \) and \( RS_2 \supseteq RS_2 \), then \( RS_1 = RS_2 \), contradicting the assumption.

### B. Proofs for Consistency of RADTs

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

- the \( s\)vector order between two concurrent operations.
- the initial RADT state where the two operations perform.

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

**Lemma 4.** For \( W_1 \): Write\( (i_1, o_{1}) \parallel W_2 \): Write\( (i_2, o_{2}) \), \( W_1 \leftrightarrow W_2 \).

**Proof.** Let the \( s\)vector order between \( W_1 \) and \( W_2 \) be \( W_1 \prec W_2. \footnote{In RFAs and RHTs, if an operation or an object appears at either side of \( \prec \), it is a surrogate of its \( s\)vector, i.e., \( \mathcal{T}_o \) or \( \mathcal{T}_p \) of \( \text{Element} \) or \( \text{Element} \) defined in Section 5.}

(1) \( i_1 = i_2 \): Let \( o_{i_1} \) be the initial RFAn. Since \( s\)vector orders are transitive, a total \( s\)vector order among \( o_{i_1}, W_1 \), and \( W_2 \) is one of the followings:

- \( a_{i_1} \prec W_1 \prec W_2 \); \( o_{i_1} \Rightarrow o_{i_1} \Rightarrow o_{1} \Rightarrow o_{2} \Rightarrow o_{2} \)
- \( a_{i_1} \prec W_1 \prec W_2 \); \( o_{i_1} \Rightarrow o_{i_1} \Rightarrow o_{1} \Rightarrow o_{2} \Rightarrow o_{2} \)
- \( a_{i_1} \prec W_1 \prec W_2 \); \( o_{i_1} \Rightarrow o_{i_1} \Rightarrow o_{1} \Rightarrow o_{2} \Rightarrow o_{2} \)

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

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

**Theorem 2.** RFAs guarantee OC.

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

The proof of OC of RHTs should be shown by \( \text{Put} \leftrightarrow \text{Put} \), \( \text{Put} \leftrightarrow \text{Remove} \), and \( \text{Remove} \leftrightarrow \text{Remove} \).

**Lemma 5.** For \( P_1 \): Put\( (k_1, o_{1}) \parallel P_2 \): Put\( (k_2, o_{2}) \), \( P_1 \leftrightarrow P_2 \).

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

**Lemma 6.** For \( P_1 \): Put\( (k_1, o_{1}) \parallel R_2 \): Remove\( (k_2) \), \( P_1 \leftrightarrow R_2 \).

**Proof.** Note the Slot where \( R_2 \) performs always exists.

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

- \( o_{k_1} \prec P_1 \prec R_2 \); \( o_{k_1} \Rightarrow o_{k_1} \Rightarrow o_{1} \Rightarrow o_{1} \Rightarrow o_{1} \)
- \( o_{k_1} \prec P_1 \prec R_2 \); \( o_{k_1} \Rightarrow o_{k_1} \Rightarrow o_{1} \Rightarrow o_{1} \Rightarrow o_{1} \)
- \( o_{k_1} \prec P_1 \prec R_2 \); \( o_{k_1} \Rightarrow o_{k_1} \Rightarrow o_{1} \Rightarrow o_{1} \Rightarrow o_{1} \)
- \( o_{k_1} \prec R_2 \prec P_1 \); \( o_{1} \Rightarrow o_{1} \Rightarrow o_{1} \Rightarrow o_{1} \Rightarrow o_{1} \)
- \( o_{k_1} \prec R_2 \prec P_1 \); \( o_{1} \Rightarrow o_{1} \Rightarrow o_{1} \Rightarrow o_{1} \Rightarrow o_{1} \)
- \( o_{k_1} \prec R_2 \prec P_1 \); \( o_{1} \Rightarrow o_{1} \Rightarrow o_{1} \Rightarrow o_{1} \Rightarrow o_{1} \)

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

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

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

**Proof.** Let the \( s\)vector order between \( R_1 \) and \( R_2 \) as \( R_1 \prec R_2 \). For \( k_1 \) and \( k_2 \), the followings cover all cases:

(1) \( k_1 = k_2 \): Assume that \( o_{k_1} \) is the initial RHT[\( k_1 \)], and \( \tau_1 \) and \( \tau_2 \) are the tombstones executed by \( R_1 \) and \( R_2 \), respectively.

Then, the two Remove work commutatively as follows:

- \( o_{k_1} \prec R_1 \prec R_2 \); \( o_{k_1} \Rightarrow \tau_1 \Rightarrow \tau_1 \Rightarrow \tau_1 \Rightarrow \tau_2 \Rightarrow \tau_2 \)
b. \( R_1 \prec \alpha_k \prec R_2 \): \( R_1 \prec \alpha_k \prec R_2 \) and \( \alpha_k \Rightarrow \tau_2 \Rightarrow \tau_2 \) and \( R_2 \Rightarrow \tau_2 \Rightarrow \tau_2 \).

c. \( R_1 \prec R_2 \prec \alpha_k \): \( R_1 \prec \alpha_k \Rightarrow \alpha_k \) and \( R_2 \Rightarrow \alpha_k \Rightarrow \alpha_k \).

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

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

**Theorem 3.** RHTs guarantee OC.

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

Proving OC in RGAs is more difficult than in RFAs and in RHTs owing to the complexity to categorizing RGA states where Inserts work. Since the effect of an Insert is determined based on \( \mathfrak{P}_8 \) of the \( \alpha_k \) next to its reference node, notations to express RGA states need to be effectively defined. Consider an RGA state given as \( [\cdots \alpha_s \alpha_s \cdots] \). Here, \( \alpha_s \) and \( \alpha_i \) are \( \alpha_k \) while \( \cdots \) is a sequence of \( \alpha_k \) (or an empty sequence) which can have no causality with some given operations. Contrarily, for a sequence that might have causality, we use the prefix ‘\( \cdot \)’, e.g., \( s_o \), and represent the \( i \)th \( \alpha_k \) in \( s_o \) as \( s_o[i] \). In addition, the notations of the set theory such as \( \forall \) and \( \exists \), will be used to express the properties of \( \alpha_k \) in a sequence. In the following lemmas, we assume the causality between \( \alpha_k \) and given operations is preserved, and all the RGA operations are given in the forms of the remote ones.

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

**Proof.** Let the \( s \) vector order between \( I_1 \) and \( I_2 \) as \( I_1 \prec I_2 \). An Insert compares \( \mathfrak{P}_9 \) with \( \mathfrak{P}_8 \) of some \( \alpha_k \) next to its reference node. The range of the compared \( \alpha_k \) begins from the exact next \( \alpha_k \) of its reference and ends in the first encountered \( \alpha_k \) preceding \( \mathfrak{P}_9 \). When the compared ranges of two Inserts do not overlap, the two Inserts commute; thus, we regard that as no conflict.

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

a. \( I_1 \rightarrow I_2 \): \( RS_1 \Rightarrow [\cdots o_i s_o o_i o_s \cdots] \Rightarrow [\cdots o_i s_o o_s o_i o_s \cdots] \)

\( I_1 \) first modifies \( RS_1 \) so that \( s_o \) and \( o_i \) become \( o_i \prec o_i \) and \( o_i \prec o_i \), respectively. Due to \( I_1 \rightarrow I_2 \), \( a_2 \) is always placed in front of \( o_i \), thus \( s_2 \) and \( s_3 \) become \( o_2 \prec o_2 \) and \( o_2 \prec o_2 \), and \( s_3 \Rightarrow o_2 \) or \( s_3 \Rightarrow o_2 \), and \( s_2 \Rightarrow o_2 \) and \( s_2 \Rightarrow o_2 \), respectively.

b. \( I_2 \rightarrow I_1 \): \( RS_1 \Rightarrow [\cdots o_i s_o o_s o_i o_s \cdots] \Rightarrow [\cdots o_i s_o o_s o_s o_i o_s \cdots] \)

\( I_2 \) first modifies \( RS_1 \) to become \( o_2 \prec o_2 \) and \( o_2 \prec o_2 \). Due to transitivity, \( o_1 \prec o_2 \prec o_2 \) and \( o_1 \prec o_2 \). In addition, if there are some \( \alpha_k \) succeeding \( I_1 \) after \( o_2 \), i.e., \( s_o \), \( o_i \) is inserted after \( o_2 \).

In the final results, the transitivity of \( s \) vector orders ensures \( s_1 = s_2 \), \( s_2 = s_2 \), and \( o_2 = o_2 \); thus, \( I_1 \leftrightarrow I_2 \) for \( i_1 = i_2 \).

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

a. \( RS_2 = [\cdots o_i s_o o_i \cdots] \) for \( s_o \not\in \) and \( \forall \in s_o \prec I_1 \); no conflict.

b. \( RS_2 = [\cdots o_i s_o o_i \cdots] \) for \( s_o \not\in \) and \( I_1 \prec s_o \); This is the compliment of (2)-a for \( s_o \not\in \) on \( RS_2 \). Placing \( o_i \) somewhere at the back of \( s_o \), the effect of \( I_1 \) varies according to the \( s \) vector order between \( I_1 \) and \( o_i \) as follows:

- \( o_i \prec I_1 \): No conflict.

- \( I_1 \prec o_i \): Because \( I_1 \) compares \( \mathfrak{P}_o \) with the next \( \alpha_k \) of \( o_i \), it is equivalent with (1) after \( s_o \); thus,

\[ RS_2 \Rightarrow [\cdots o_i s_o o_i o_s o_o o_o \cdots] \] for \( o_i \prec o_2 \prec \forall \in s_o \), \( s_o \not\in \) or \( s_o[0] \not\in \), \( o_1 \prec \forall \in s_o \), and \( o_1 \prec o_2 \).

c. \( RS_2 = [\cdots o_i o_i o_i \cdots] \):

- \( o_i \prec I_1 \): No conflict.

- \( I_1 \prec o_i \): Equivalent with (1) from \( o_i \); thus,

\[ RS_3 \Rightarrow [\cdots o_i o_i o_s o_s o_o o_o o_o \cdots] \] for \( o_i \prec o_2 \prec \forall \in s_o \), \( s_o \not\in \) or \( s_o[0] \not\in \), \( o_1 \prec \forall \in s_o \), and \( o_1 \prec o_2 \).

d. \( RS_4 = [\cdots o_i s_o o_i \cdots] \) for \( s_o \not\in \) and \( \forall \in s_o \prec I_2 \); no conflict.

e. \( RS_4 = [\cdots o_i s_o o_i \cdots] \) for \( s_o \not\in \) and \( I_2 \prec \forall \in s_o \).

- \( o_i \prec I_2 \): No conflict.

- \( I_2 \prec o_i \): Equivalent with (1) after \( o_i \); thus,

\[ RS_5 \Rightarrow [\cdots o_i o_s o_o o_o o_o o_o \cdots] \] for \( o_i \prec o_2 \prec \forall \in s_o \), \( s_o \not\in \) or \( s_o[0] \not\in \), \( o_1 \prec \forall \in s_o \), and \( o_1 \prec o_2 \).

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

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

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

**Proof.** For \( D_2 \rightarrow I_1 \), \( D_2 \) modifies only \( \mathfrak{P}_p \) of its target node, and the node is preserved as a tombstone until \( I_1 \) is executed. Since \( I_1 \) is affected by only \( \mathfrak{P}_p \) of \( \alpha_k \), \( I_1 \leftrightarrow D_2 \).

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

**Proof.** Due to the same reason of Lemma 9, \( I_1 \leftrightarrow U_2 \).

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

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

Proof. Though $i_1 \equiv i_2$, due to $U_2 \rightarrow D_1$, $D_1 \leftrightarrow U_2$.

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

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

Theorem 4. RGAs guarantee OC.

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

References

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