How to Design Optimistic Operations for Peer-to-peer Replication

Hyun-Gul Roh, Jin-Soo Kim, and Joon-Won Lee
Computer Science Division
Korea Advanced Institute of Science and Technology (KAIST)
Guseong-Dong, Yuseong-Gu, Daejeon 305-701, Korea
{hgroh@calab, jinsoo@cs, joon@cs}.kaist.ac.kr

Abstract

As collaboration over the Internet becomes an everyday affair, it is increasingly important to provide high quality of interactivity. Distributed applications can replicate collaborative objects at every site for the purpose of achieving high interactivity. Replication, however, has a fatal weakness that it is difficult to maintain consistency among replicas. This paper introduces *operation commutativity* as a key principle in designing operations in order to manage distributed replicas consistent. In addition, we suggest effective schemes that make operations commutative using the relations of objects and operations. Finally, we apply our approaches to some simple replicated abstract data types, and achieve their consistency without serialization and locking.

Keywords: consistency, peer-to-peer replication, operation commutativity, optimistic operations

1. Introduction

Replication is a common technique to enhance performance and responsiveness of distributed applications by reading local replica and executing local operations immediately[1]. Especially, collaborative applications with high interactivity adopt replication in order to display interactive data efficiently. Whenever each site modifies the objects by user interactions, it transfers operations to remote sites for the purpose of notifying its modifications.

To minimize network delay, we assume that transferring operations among replicas is accomplished in a peer-to-peer way, that is, no sequencer or master intervenes between peers. In addition, for achieving higher quality of interactivity, local operations are executed in an optimistic way without global arbitration such as locking. Each site, however, may execute operations in a different order, which leads to difficulty in maintaining consistency among replicas. This paper suggests a way that makes optimistic executions in different orders eventually identical without any locking or serialization.

As a key principle to make replicas consistent, we introduce *operation commutativity*. Operation commutativity is the condition that every pair of operations is in commutative relation. In this paper, we show why operation commutativity guarantees consistency among replicas. Furthermore, we provide novel schemes which help to achieving commutativity of operations. We first define relations between objects and operations. Then, we present new orders, *contemporary and obsolete orders*, which are useful to make optimistic operations commute based on the relations. Finally, we actually exploit operation commutativity for some representative operations of simple replicated abstract data types.

2. Operation Commutativity

This section theoretically analyzes the aspect that operations are executed in a peer-to-peer replication system. As a key principle to make replicas consistent, we suggest operation commutativity, and show what is operation commutativity and why it guarantees consistency of replicas.

2.1. Operations in a distributed system

In a distributed system, an operation has either causal relation(order) or concurrent relation with any other operation [2]. For example, from a time space diagram in figure 1, we derive relations of every pair of operations as follows: $O_1 \parallel O_2$, $O_1 \parallel O_3$, $O_1 \rightarrow O_4$, $O_1 \parallel O_5$, $O_2 \parallel O_3$, $O_2 \rightarrow O_4$, $O_2 \rightarrow O_5$, $O_3 \rightarrow O_4$, $O_3 \parallel O_5$, $O_4 \parallel O_5$[1]. Between two concurrent operations, no uniquely correct order exists, because they are generated without knowing each other. Causal order (causality), however, must be preserved, because the later generated operation might use the result of previously generated one. In figure 1, while site 0 and 1 preserve all of causal orders, site 2 violates the causal order of $O_2 \rightarrow O_4$. Causality preservation can be achieved using the state vector issued when the operation is generated [3]. Assume that N is the number of sites, and sites are identified by integers $1, \ldots, N$. Each site n maintains an N-tuple state vector SV_n. Initially $SV_n[i] := 0$, for $1 \leq i \leq N$. After site n executes an operation generated at site i, the site times-

[1] $O_a \rightarrow O_b$ denotes a causal relation(order), which means that O_b is generated after observing O_a's execution. $O_a \parallel O_b$ denotes a concurrent relation, which means that both O_a and O_b are generated without knowing the generation of each other.

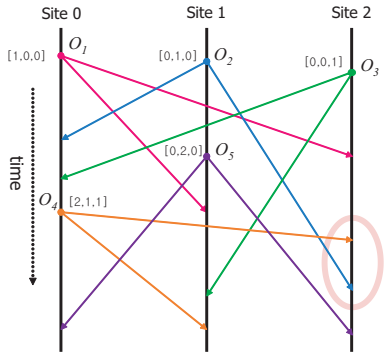

Figure 1: A time-space diagram that three sites participate in and five operations are generated. Each operation is specified with its state vector.

tamps its sequence number as $SV_n[i] := SV_n[i] + 1$. Let O be an operation generated at site k and SV_o be the last timestamped state vector, which is transferred to other sites with O. O is causally ready to be executed at site $l (k \neq l)$ with a state vector SV_l only if the following conditions are satisfied: (1) $SV_o[k] := SV_l[k] + 1$ (2) $SV_o[i] \leq SV_l[i]$, for $1 \leq i \leq N$ and $i \neq k$. To preserve causality, causally unready operations should be selectively delayed.

2.2. Causally executable graph and sequences

In peer-to-peer replication, although we restrict executions of operations only to preserve causality, many execution sequences are possible at each site. To analyze the aspect of operation executions in peer-to-peer replication systems, a time-space diagram is useful. As a site issues operations and they arrive at remote sites, a time-space diagram is constituted and updated. If we capture quiescence that all generated operations are executed at all sites, we can build a time-space diagram, and it can be effectively represented by borrowing graph notations. Provided operations and their relations are denoted with vertices and edges, respectively, we can define a graph derived from a time-space diagram as follows:

Definition 1 *Causally executable graph(CEG)*
A graph $G = (V, E)$ is a causally executable graph, where $V = \{O_1, \ldots, O_n\}$ is a set of operations, and $E \subseteq V \times V$ is a set of edges between two operations in V with directed edges, e.g., (O_a, O_b), which corresponds to causal relations, e.g. $O_a \rightarrow O_b$, and undirected edges, e.g., $\langle O_c, O_d \rangle$, which corresponds to concurrent relations, e.g. $O_c \parallel O_d$, iff: G has edges for every pair of distinct vertices (complete), and directed edges of G are transitive, e.g., if $(O_a, O_b) \in E$ and $(O_b, O_c) \in E$, then $(O_a, O_c) \in E$.

From the time-space diagram in figure 1, we can obtain a CEG as shown in figure 2. In this graph, if vertices are traveled without going against directed edges, the execution sequence does not violate causality. We therefore can define an execution sequence which preserves causality and in which all operations participate as follows:[2]

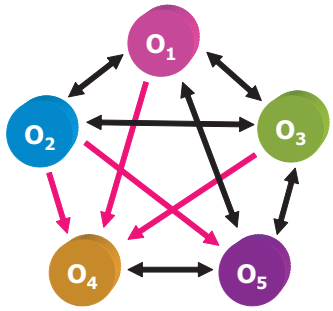

Figure 2: A causally executable graph derived from the time-space diagram in figure 1.

Definition 2 (Causally executable sequence(CES))
Given a CEG, $G = (V, E)$, where $|V| = n$, an execution sequence of operations, s: $O_1 \mapsto \ldots \mapsto O_n$, is a causally executable sequence from G, iff: all operations of V participate only once in s, and for $\exists O_i \in V$ and $\exists O_j \in V$ for $i < j$ in s, the relation of O_i and O_j is either $\langle O_j, O_i \rangle \in E$ or $\langle O_i, O_j \rangle \in E$ or $(O_i, O_j) \in E$.

In peer-to-peer replication, when causality is preserved, all CESs derived from a CEG are the whole possible execution sequences which can be executed at any site.

2.3. Operation commutativity

Assuming that all sites in peer-to-peer replication execute one of CESs, and all CESs executed from the same replica state yield an identical result, we can guarantee consistency of replicas. Then, how can we make all CESs produce the same result? We achieve it by letting all combinations of operations be in commutative relation, when they are concurrent. We define commutative relation as follows:[3]

Definition 3 (Commutative relation)
Given two concurrent operations O_a and O_b, they are in commutative relation, expressed as $O_a \leftrightarrow O_b$, iff: for $RS_0 \overset{O_a \rightarrow O_b}{\Rightarrow} RS_1$ and $RS_0 \overset{O_b \rightarrow O_a}{\Rightarrow} RS_2$, RS_1 is equal to $RS_2 (RS_1 = RS_2)$.

Provided that all operations in concurrent relation are in commutative relation, a CES can be transformed into another CES by changing the execution order of two operations in concurrent relation. Then, we can ensure that the results of the original CES and the changed CES are identical. Repeating this process for the changed CES, all CESs of a CEG are constituted and their results are always identical. We therefore conclude that consistency is guaranteed, if all operations in concurrent relation are in commutative relation. Consequently, this paper define operation commutativity as follows:

[2] If O_a is first executed, and O_b later, their execution sequence is expressed by '$\mapsto$' as $O_a \mapsto O_b$.

[3] If an operation O(or an execution sequence) is executed on a replica state RS_0, thus RS_0 is modified into RS_1, we express this execution as $RS_0 \overset{O}{\Rightarrow} RS_1$.

Definition 4 (Operation commutativity)
Given a set of operation types ***O****, operation commutativity is established in* ***O****, iff:* $O_a \leftrightarrow O_b$ *for* $\forall O_a, \forall O_b \in \boldsymbol{O}$*, when* $O_a \parallel O_b$.

Due to space limitation, we omit the formal proof that operation commutativity guarantees replica consistency.

Based on the definition of operation commutativity, we suggest commutativity-based consistency maintaining method. If an operation set is given for a replication system, this method pursues consistency by making all combinations of concurrent operations commutative. For instance, suppose that we have only *Write* and *Append* operations. We need to make any pair of operations, such as *Write-Write*, *Write-Append*, and *Append-Append*, commutative, when they are concurrent.

3. Simple replicated abstract data types

In this section, three simple abstract data types(ADTs) are presented to show how to design optimistic operations in order to achieve operation commutativity.

First, we consider fixed-size replicated arrays which support only *Write* operation. *Write(int i, Object o)* specifies an integer index i to point to the position of the array and an object o to be updated. To show that this replicated ADT is consistent, we should design concurrent *Write* operations are commutative with each other. To illustrate, consider an example where each operation in figure 3 is as follows:

e.g. 1 O_1*:*Write(0, obj1)*,* O_2*:*Write(0, obj2)*,* O_3*:*Write(0, obj3)

If these operations are executed on a fixed-size empty array, what should be the last array state of index 0?

Second, we consider an ADT maintained by only *Append* operation like log data. An *Append* inserts a new object *o* to the last position in the form of *Append(Object o)*. If three operations shown in e.g. 2 are executed on the empty state as in figure 3, how can we make the order of three appended objects consistent at all sites?

e.g. 2 O_1*:*Append(obj1)*,* O_2*:*Append(obj2)*,* O_3*:*Append(obj3)

Finally, we take into account another ADT supporting *Write* and *Append* operations together. This ADT permits to modify appended objects by *Write* operation. In this case, we should show that every possible combination of two operations, i.e., *Write-Write*, *Write-Append*, and *Append-Append*, are commutative. If O_1, O_2, and O_3 in figure 3 are as follows and they are executed on an empty state, how can we make them commutative?

e.g. 3 O_1*:*Append(obj1)*,* O_2*:*Append(obj2)*,* O_3*:*Write(0, obj3)

In the above examples, if all operations are performed without any conflict resolving rules, each site eventually has an inconsistent replica. In the next section, through presenting two design schemes, we answer the aforementioned three questions.

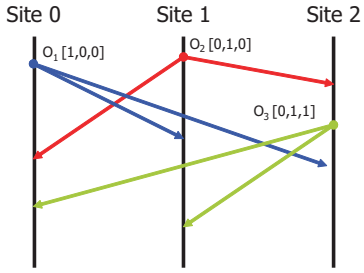

Figure 3: An example time-space diagram. Operations are $O_1 \parallel O_2$, $O_1 \parallel O_3$, and $O_2 \rightarrow O_3$

4. Designing Commutative Operations

4.1. Relations between objects and operations

When an operation is executed, it must have knowledge of its previous concurrent operations in order to make the operation commutative with them. As it might be solved by recording the history of operations, operation transformation algorithms use this approach. They, however, may require to compare an incoming operation with many previous operations unnecessarily, and it is complicated to find out the correct result[4].

Instead, we give conflict resolving clues to operations in different way. Operations introduced in this paper deal with only one object, and conflicts occur only when their target objects are the same or adjacent.[4] It is reasonable to consider relations between objects and operations. If we leave the state vectors of operations on its target object after executing them, another operation can figure out the relation between the operation and its target object when it refers to the object. For example, after executing O_2 at site 1 in e.g. 1, O_1 is concurrent with the object of index 0. On the other hand, O_3 is causal with the object where O_2 is executed at site 2.

4.2. Contemporary and obsolete orders

If an operation is in causal relation with its target object(s), it must be executed according to its intention. In e.g. 1, consider the execution sequence of site 2, $O_2 \mapsto O_3 \mapsto O_1$. O_3 is normally executed, because it is causal with O_2. When O_1 is executed on the result of $O_2 \mapsto O_3$, O_1 must arbitrate its execution to be commutative with not only O_3 but also with O_2. However, because the object of index 0 has the state vector of O_3, O_1 may not know the effect of O_2. We therefore design operations so that operations can infer relations of all previously executed operations. For this, two new orders, contemporary and obsolete orders, are introduced as follows:

Definition 5 (Contemporary and obsolete orders)
Given two operations O_a *and* O_b*, generated at site* i *and* j*, with state vectors* SV_{O_a} *and* SV_{O_b}*, respectively,* $O_a \mapsto$

[4] A target object of an operation is defined as the objet which the operation modifies. While the target object of *Write* is the object pointed by an integer index, *Append*'s is the object inserted at the tail.

O_b is a contemporary order, and $O_b \mapsto O_a$ is an obsolete order, iff: (1) $sum(SV_{O_a}) < sum(SV_{O_b})$, or (2) $i < j$ when $sum(SV_{O_a}) = sum(SV_{O_b})$, where $sum(SV) = \sum_{i=0}^{N-1} SV[i]$.

The conditions determining the orders are the same ones used in the total ordering [3]. From the conditions of detecting causality described in section 2.1, note that causal order is always a contemporary order. In addition, contemporary order designates a globally unique order for a concurrent relation. Then, the following design makes concurrent *Write* operations commutative:

Design 1 (*Write* operation)
If a Write *operation is a contemporary order with its target object, it replaces the object with its one, and if it is an obsolete order with its target object, it is ignored.*

Returning to the site 2 of e.g. 1, because O_1 is an obsolete order with its target object after the execution of O_3, it is ignored. Although O_2 is a contemporary order with $obj1$ of O_1, the state vector of $obj3$ tells that it need not compare with the object of O_1, because causal operation of O_2 is executed. In other words, since *Write*s are executed as long as the state vector sums of their target objects increase, commutativity of concurrent *Write* operations is guaranteed.

Using contemporary and obsolete orders, commutative *Append* operations can be designed as follows:

Design 2 (*Append* operation)
Compare the objects from the last object until it meets the object of contemporary order, and insert the object after the object of the first contemporary order.

In e.g. 2, O_2 of site 2 can be executed commutatively not only with O_3 but also with O_1 in similar effects of the total ordering.

Then, supposing *Write* and *Append* are permitted together like e.g. 3, is operation commutativity established? If we let each object reserve two state vectors dedicated to each operation, *Write-Write* and *Append-Append* commutativity is established. However, as in the case of e.g. 3, *Write-Append* is not commutative. This is caused by the integer index system. Since concurrent *Append* operations can change the object index targeted by a *Write* operation, the *Write* misses its actual target object. To make *Write-Append* commutative, we fix *Write* operation so that it find the causal object with itself after its target object indexed by the *Write* if the object is concurrent relation with itself. This guarantees commutativity of *Write-Append*, because *Append*s of only concurrent relation with the *Write* can push the object after the back of the position where the *Write* are pointing.

5. Related work

Consistency of peer-to-peer replication has been mainly studied in groupware or computer supported cooperative work(CSCW) fields. Operation transformation(OT) is one of main approaches to maintain consistency [5, 6, 7]. OT transforms indexes of insertion and deletion operations in receiving sites using histories of operations so as to preserve intentions of operations. Although they have tried to apply the concept of commutativity, they have gone through many trial and error, because commutativity was not completely formalized[4].

6. Conclusions and future work

As interactions over the Internet increase, providing higher quality of interactivity becomes important. Although optimistic peer-to-peer replication is the most effective architecture for interactivity, applications have avoided adopting it due to difficulty in maintaining consistency among replicas. In this paper, we have presented a novel way to design optimistic operations that make peer-to-peer replication consistent. As a key principle to achieve consistency, we formalized operation commutativity and suggested commutativity-based consistency maintaining method. In addition, we have exploited operation commutativity for some operation set of simple replicated ADTs.

As future work, we will consider more complicated replicated ADTs employing richer operations such as insertion and deletion.

Acknowledgements

This research was supported by the MIC(Ministry of Information and Communication), Korea, under the ITRC (Inofrmation Technology Research Center) support program supervised by the IITA(Institute of Information Technology Assessment) (IITA-2005-C1090-0502-0031)

References

[1] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42 – 81, Mar. 2005.

[2] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558 – 565, Jul. 1978.

[3] M. Raynal and M. Singhal. Logical time: capturing causality in distributed systems. *IEEE Computer*, 29(2):49 – 56, Feb. 1996.

[4] C. Sun and C. (Skip) Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *ACM CSCW'98 Proceedings*, pages 59–68, Dec. 1998.

[5] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Acheieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction*, 5(1):63 – 108, Mar. 1998.

[6] R. Li and D. Li. Commutativity-based concurrency control in groupware. In *CollaborateCom'05*, pages 19–21, Dec. 2005.

[7] D. Li and R. Li. Ensuring content and intention consistency in real-time group editors. In *In IEEE ICDCS'04 Proceedings*, pages 748 – 755, Mar. 2004.