Optimistic Concurrency Control for Context-Specific Consistency in Distributed Real-Time Group Editors

Zhiyi Huang†
†Dept of Computer Science
University of Otago
Dunedin, New Zealand
Email:{hzy}@cs.otago.ac.nz

Chengzheng Sun‡
‡School of Comp. and Infor. Tech.
Griffith University
Brisbane, Qld 4111, Australia
{scz}@cit.gu.edu.au

1 Introduction

Real-time group editors allow a group of users to view and edit the same text/graphic/image/multimedia document at the same time from geographically dispersed sites connected by communication networks. Those real-time group editors are required to have the following characteristics: (1) *real-time*: the response to local user actions is quick (ideally as quick as a single-user editor), and the latency for reflecting remote user actions is low (determined by external communication latency only); (2) *distributed*: cooperating users may reside on different machines connected by different communication networks with nondeterministic latency; (3) *unconstrained*: multiple users are allowed to concurrently and freely edit any part of the document at any time, in order to facilitate free and natural information flow among multiple users.

The requirements for good responsiveness and for supporting unconstrained collaboration have led us to adopt a replicated architecture for the storage of shared documents: the shared documents are replicated at the local storage of each participating site. One of the most significant challenges in designing and implementing real-time group editors with a replicated architecture is consistency maintenance of replicated documents.

In the past literatures [], consistency maintenance is achieved by operational transformation [] integrated with traditional locking schemes. However, since locking schemes require user intervention, it is inconvenient in an brainstorming editing session and inflexible in an highly-cooperative environment.

In this paper, we propose an Optimistic Concurrency Control (OCC) scheme in distributed real-time group editors, which, incoporating with operational transformation, can achieve consistency maintenance conveniently and flexibly.

2 Inconsistency problems

There are two different types of inconsistency problems in distributed real-time group editors. One is *generic* inconsistency problem, which is concerned with whether editing operations are executed in the right order, achieve the original effects, and result in an identical final result, etc. It is further divided into *divergence, causality-violation*, and *intention-violation*. Divergence means that operations may arrive and be executed at different sites in different orders, resulting in divergent final results. Causality violation means that operations may arrive and be executed in an order different from their causal order, due to different transmission delays. Intention violation means that the actual effect of an operation at the time of its execution may be different from the originally intended effect of this operation at the time of its generation, due to concurrent generation of operations. For the limited space of the paper, we only explain divergence here by using an example. For example, suppose that the initial document at all sites contains the text "ABCD" and two users issues an operation concurrently. User A issues O_1 which inserts "1" at position 1, and user B issues O_2 which inserts "3" at position 3. These operations may arrive and be executed at different sites in different order, resulting in divergent final results. Since local operations are executed immediately, the order of operations executed at user A site is O_1 O_2, and the order at user B site is O_2 O_1. Consequently, the final text is "A1B3CD" at user A site and

"A1BC3D" at user B site.

Interestingly, the generic inconsistency problems can not be solved by the traditional concurrency control schemes, but only by operational transformation [].

The other inconsistency problem is *context-specific*, which is concerned with whether the data integrity of the shared document is maintained with respect to the specific application context. For example, consider a shared document with the following text:

"Transformation preserve operation intention."

In this text there is an English grammar error (indicated by the underlined text), i.e. in the text it should be "can preserve", or "preserves" or the like. Assume that two users observed the error and wanted to correct it in two different ways: one user issues an operation to insert "can " at the starting position of "preserve", while another user issues a concurrent operation to insert "s" at the ending position of "preserve". Suppose the editing system has used the operational transformation to ensure generic consistency. Then, after the execution of these two concurrent operations at all sites, the text would be:

Transformation can preserves operation intention."

From generic consistency point of view, this result is correct since all sites have the same document contents and the intended effects of all operations have been achieved. This result is, however, incorrect in the context of English grammar.

Traditional locking schemes have been used to solve the context-specific inconsistency problems []. The users can use locking facilities to enforce mutual exclusion over specific regions (e.g., an English word, a statement, or a paragraph, etc), then either one of the two users could obtain an exclusive lock on the whole statement before modifying it, and the final text would be either:

"Transformation preserves operation intention" or:

"Transformation can preserve operation intention"

which ensures the context-specific consistency (in terms of English grammar).

The locking schemes adopted in group editors can be classified into three categories. The first is *pessimistic locking*, in which a user is not permitted to edit a region until the requested lock has been granted. The second is *optimistic locking*, in which a user is permitted to edit a region while waiting the requested lock. If the locking request is indeed successful, the user is able to continue editing that region without blocking. If the locking request finally fails, the user is not allowed to continue editing that region, and what this user has done during waiting for the lock will be undone. The third is *optional locking* [], in which a user may update any unlocked region without necessarily requesting a lock on it. If a lock has been placed on a region, however, a user can update this locked region only if s/he owns a lock covering the region.

However, locking schemes have the following disadvantages in an cooperative environment. (1) User intervention: no matter what kind of locking schemes is adopted, users are interfered with requesting and releasing locks in an brainstorming editing session. If one user locking a region finishs his/her editing without unlocking it, other users may never edit that region. (2) Inflexibility: once a region is locked, other users have no chance to update it, even when the user doesn't update it at the moment. It is not flexible enough in an cooperative environment.

To avoid those problems caused by locking schemes, in this paper we propose an optimistic concurrency control scheme to solve the context-specific inconsistency problems in distributed real-time group editors.

Optimistic concurrency control was proposed to handle concurrent transactions [**?**]. The idea behind it is simple: just go ahead and do whatever you want to, without paying attention to what anybody else is doing. If there is a problem (conflict), worry about it later.

The key point in optimistic concurrency control is to detect conflicts and resolve them if they occur. What optimistic concurrency control does in our implementation is to keep track of the history of operations and undo/drop some operations if conflicts happen.

For example, to solve the above context-sepcific inconsistency problems, we define that two concurrent operations on the same region (e.g. a statement) conflict with each other. In the above example, suppose user A issues the operation O_1 to insert "can" while user B issues the operation O_2 to insert "s". O_1 and O_2 conflict with each other since they are concurrent and on the same statement

"Transformation preserve operation intention".

According to the optimistic concurrency control, we allow both operations to be immediately executed

locally. When O_1 arrives at user B site and O_2 at user A site, we can discover the conflict at both sites. What we need now is a consistent solution to resolve the conflict at each site. Suppose the solution is to drop O_2 and keep O_1 according to the users' priority. Then O_2 is dropped at user A site and is undone at user B site. Finally both sites have the same text with the correct English grammar (Context-sepcific consistency is ensured).

"Transformation can preserve operation intention".

The big advantages of optimistic concurrency control are that it is user intervention free and allows maximum concurrency because no user ever has to wait for a lock. The disadvantage is that it may undo/drop some operations, which is not a problem in group editors if users are warned properly in advance or informed afterwards.

The challenge of implementing optimistic concurrency control in distributed real-time group editors is how to discover conflicts and how to resolve them consistently in a distributed environment.

3 Optimistic concurrency control in real-time group editors

In this section we will discuss the optimistic concurrency control scheme in distributed real-time group editors in a generic way.

Following Lamport [], we first define a causal ordering relation on operations in terms of their generation and execution sequences as follows.

Definition 1 *Causal Ordering Relation "$\rightarrow$"*
Given two operations O_a and O_b, generated at site i and j, then $O_a \rightarrow O_b$, if and only if (1) i=j, and the generation of O_a *happened before* the generation of O_b; (2) $i \neq j$, and the execution of O_a at site j *happened before* the generation of O_b; (3) there exists an operation O_x, such that $O_a \rightarrow O_x$ and $O_x \rightarrow O_b$.

Definition 2 *Dependent and Independent Operations*
Given any two operations O_a and O_b, (1) O_b is said to be *dependent* on O_a if and only if $O_a \rightarrow O_b$; (2) O_a and O_b are said to be *independent* (or *concurrent*) if and only if neither $O_a \rightarrow O_b$, nor $O_b \rightarrow O_a$, which is expressed as $O_a \| O_b$.

To capture the causal relationship among all operations in the system, a timestamping scheme based on a data structure –State Vector (SV)–can be used [].

Definition 3 *State Vector (SV)*
Let N be the number of cooperating sites in the system. Assume that sites are identified by integers 0,...,$N-1$. Each site maintains an SV with N components. Initially, $SV[i]$:=0, for all i {0,...,$N-1$}. After executing an operation generated at site i, $SV[i]$:=$SV[i]$+1. An operation is executed at the local site immediately after its generation and then multicast to remote sites with a timestamp of the current value of the local SV.

Given any two operations O_a and O_b, $O_a \rightarrow O_b$ if and only if their State Vector SV_a and SV_b satisfy the following conditions:
(1) $SV_a[i] \leq SV_b[i]$, for all $i \in \{0,...N-1\}$;
(2) $\exists\ i \in \{0,...N-1\}$, $SV_a[i] < SV_b[i]$.

Definition 4 *Total Ordering Relation "$\Rightarrow$"*
Given two operations O_a and O_b, generated at sites i and j and timestamped by SV_{O_a} and SV_{O_b}, respectively, then $O_a \Rightarrow O_b$, if and only if (1) $sum(SV_{O_a}) < sum(SV_{O_b})$ or (2) $i < j$ when $sum(SV_{O_a}) = sum(SV_{O_b})$, where $sum(SV) = \sum_{i=0}^{N-1} SV[i]$

To facilite *undo* operation in optimistic concurrency control, each site maintains a *history buffer* (HB) for saving executed operations at each site. It is also required in operational transformation.

3.1 Optimistic concurrency control

To discover conflict operations, for each operation, we need to identify its region (a word, a statement, or a paragraph) and its dependency relationship with other operations. To identify a region, we can use the region delimeters, such as space character for word region, period(".") for statement region, and return key for paragraph region. Each site can dynamically maintain a data structure to record the starting position and ending position of every region. It is dynamically changed when an operation is executed. This data structure can help detect if two independent operations update the same region.

Below we present the scheme to identify conflict operations.

Scheme 1: Identification of Conflict Operations (ICO)

1. For a new operation O, first check the history buffer HB if there are any operations which are independent with O.

2. If there is no operations independent with O, report that no conflict is detected.

3. If there are some operations independent with O, check if any of them update the same region as O's.

4. If none of them updates the same region as O's, report that no conflict is detected; otherwise report that conflict is detected.

To resolve conflict between two operations, we need to give each operation a priority. The priority shows how much priviledge an operation can be executed on a region when conflicts happen. We will discuss different schemes to assign priority to operations shortly. Below we present the scheme to resolve conflict operations.

Scheme 2: Resolution of Conflict Operations (RCO)

1. For a new operation O and a set of conflict operations CO, check their priorities.

2. If there is an operation in CO which has priority higher than O's, drop O and return;

3. If all operations in CO have priority lower than O's, undo all operations in CO and return.

3.2 Priority assignment

1. User identifier

We can use the user identifier as the priority for the operations. When a user issues an operation, his/her identifier is attached to the operation as its priority. Since user identifier is unique, the operations from different users have different priorities. The operations with smaller user identifier have higher priority.

The advantage of the scheme is that it is simple. However, since the operations with smaller user identifier always override other conflicting operations from users with larger identifier, it is not fair and flexible.

2. Total order

We can use the total order of the operations as priority. The total order can be decided by the state vector of the operation and the user identifier, as described in the definition of *Total Ordering Relation*. The operations with smaller total order have higher priority.

Besides simpleness, the advantage of the scheme is that it is fair to every user. The problem is, however, none of the users is sure that his/her operations will be faithfully executed if they are concurrent with other users'. Though this problem can be relieved by informing the user if his/her operations are dropped/undone, sometimes we want to ensure one important user's operations on some region are not dropped or undone, just as if that user has locked that region.

3. Ownership and total order

Users may apply for ownership of a region. Once a user has ownership of a region, his/her operations will override any operations conflicting with them. So the owner's operations will have the highest priority. For other non-owner users, if their operations conflict with each other, we resolve the conflict by using the total order as their priority.

The effect of OCC with this scheme of priority assignment is similar to locking schemes in that only one user has the absolute right to update a region. It is different from locking schemes, however, in that other users can also update the same region as the owner as long as there is no conflict. Even if the owner forgets to dismiss his/her ownership, other users can still update that region freely. So it is more flexible than locking schemes.

The disadvantage of this scheme is that it needs user intervention to request/dismiss ownership. But compared with locking schemes, it still has the advantage of flexibility.

4. Entrance order

We can also use the entrance order as priority. When a user moves his/her cursor to a new region, an entrance order is automatically assigned to the user by a manager according to his/her order of entrance to that region. Then all operations from the user on that region are given the user's entrance order as their priority. The user who stays in a region longest has the highest priority in the region.

In this scheme, the complication is the dynamic assignment of entrance order. A user may enter and leave a region frequently. When a user enters a region, a unique sequence number should be assigned as his/her entrance order. Each time the sequence number is assigned it is increased by 1. When a user leaves a region, the one who enters the region next to him/her naturally has the highest priority in the region. Since the entrance order is always increased, it may run out of unique numbers after some time and therefore garbage collection has to be used on the resource of unique numbers. A manager is used to handle the assignment of entrance order and the garbage collection.

This scheme has the advantages of user intervention free and fairness to every user.

4 Integrating generic consistency control with OCC

Generic consistency is maintained by scheduling the execution order of operations and operational transformation. A *Undo/Transform-Do/Transform-Redo* scheme has been proposed to maintain the generic consistency []. Before a new operation is executed, the scheme makes sure it is causally ready to avoid causality violation. To achieve convergence, some operations executed before are undone and then redone after the new operation so that the total ordering relation of operations is maintained.

To preserve the generic intention, a *Generic Operation Transformation (GOT)* control algorithm has been proposed in []. It uses *inclusion* and *exclusion* transformations to transform a new operation before its execution. *Inclusion* transforms an operation O_a against another independent operation O_b in such a way that the impact of O_b is effectively included into O_a. Consider the same example in Section *. O_1 inserts "1" at position 1, and O_2 inserts "3" at position 3 on the shared text "ABCD". After O_2 is transformed against O_1 by inclusion, O_2 becomes O_2', which inserts "3" at position 4. *Exclusion* transforms an operation O_a against another independent operation O_b in such a way that the impact of O_b is effectively excluded from O_a. For example, After O_2' is transformed against O_1 by exclusion, O_2' becomes O_2.

Formal description of *Undo/Transform-Do/Transform-Redo* scheme, GOT control algorithm, and inclusion and exclusion transformations can be seen in [].

To handle both generic inconsistency and context-specific inconsistency, we have to integrate OCC with the *Undo/Transform-Do/Transform-Redo* scheme. This integration results in the following *Undo/Transform-OCC-Do/Transform-Redo* scheme. Like the *Undo/Transform-Do/Transform-Redo* scheme, the scheme first undoes the operations whose total order is after that of the new operation, and then transforms the new operation by using the GOT control algorithm. After the transformation of the new operation, we use the *Identification of Conflict Operations (ICO)* scheme to check if it conflicts with any operations in the history buffer HB. We use ICO after the transformation of the new operation because the transformation can guarantee the correct detection of shared regions in ICO. If ICO reports no conflict, we execute the new operation, and transform and redo the undone operations just like the *Undo/Transform-Do/Transform-Redo* scheme. But if ICO reports conflicts, we need to use the *Resolution of Conflict Operations* scheme to resolve the conflicts. If the resolution is to drop the new operation, then we simply redo the undone operations. However, if the resolution is to undo other operations in HB, we need to undo conflict operations and transform other affected operations according to the operational transformation.

However, if we adopt the above idea we will have derived inconsistency problems. For example, we have operations O_1, O_2, and O_3 in Figure **??**. Their dependency relationship is $O_2 \| O_1$, $O_2 \| O_3$, and $O_1 \rightarrow O_3$. Suppose their total ordering relationship is $O_1 \Rightarrow O_2 \Rightarrow O_3$, their priority has the relation $P_{O_1} < P_{O_2} < P_{O_3}$, and a region is one word. Assume the initial text is "ABCD", O_1 inserts "1" after "A" (denoted as $Insert("1", 1)$), O_2 inserts "2" after "B" (denoted as $Insert("2", 2)$), and O_3 inserts "3" after "C" (denoted as $Insert("3", 4)$). At site 0, O_1 is transformed and executed first and then text becomes "A1BCD"; when O_2 comes, conflict is discovered and O_1 is undone; then O_2 is transformed and executed, and the text becomes "AB2CD"; When O_3 comes, conflict is again discovered and O_2 is undone; then O_2 is transformed and executed, and the text becomes "ABCD3". At site 1, the operations comes in an order $O_2O_1O_3$. After conflict resolution, we only get O_3 executed and the final text becomes "ABCD3". At site 2, however, the the operations comes in an order $O_1O_3O_2$. After conflict resolution, we can get O_1 and O_3 executed and the final text becomes "AB2C3D". Obviously, the above results are inconsistent. We have divergence problem again. Moreover, we have intention violation in both site 0 and 1, in which "3" is inserted into a position different from the initial intention (insertion after "C").

The problem is caused by inconsistent handling order of conflict operations. At site 0 and 1, O_2 overrides O_1, and then O_3 overrides O_2. Because O_1 is dropped or undone, O_3 inserts "3" at the wrong position and therefore intention is violated. Because O_2 is overriden by O_3, O_1, which is overriden by O_2, should be resumed and executed if it does not conflict with O_3. At site 3, O_1 and O_3 are first executed, then comes O_2 which is dropped after the conflict resolution. Actually this site happened to preserve the intention of O_3 because O_1 is not overriden by O_2, and have the correct final result. The solution for the problem is we need to maintain a Conflict Operation

List (COL) for each operation, which contains all the conflict operations overriden by the operation. When an operation is overriden by a new operation, every operations in its COL should be tested and redone if it does not conflict with the new operation.

Algorithm (Undo/Transform-OCC-Do/Transform-Redo Scheme) Given a new causally ready operation O_{new}, and HB = [EO_1,...,EO_m,...,EO_n], the following steps are executed:

(1) Undo operations in HB from right to left until an operation EO_m is found such that $EO_m \leq O_{new}$.

Assume the undone operations UDL = [EO_{m+1}, ..., EO_n] and HB_1 = [EO_1, ..., EO_m].

(2) Transform O_{new} into EO_{new} by applying the GOT control algorithm with HB_1 as the history buffer.

(3) Use ICO scheme to check if EO_{new} conflicts with any operations in HB_1 and UDL. If ICO reports any conflict, continue; otherwise go to step 8.

(4) Use RCO scheme to check if EO_{new} should be dropped to resolve the conflicts. If EO_{new} should be dropped, and the conflict operation is EO_c, append EO_{new} into EO_c's COL, mark EO_{new} as *nullified* in HB, and go to step 9; otherwise continue;

(5) Check HB_1 from right to left for any operation EO_i conflicting with EO_{new}. (a) Undo EO_i, mark it as *nullified* in HB, and append it into EO_{new}'s COL.

(b) For every operation after EO_i in HB (including EO_{new} and the *nullified* operations), exclude the effect of EO_i by *exclusion* transformation.

(c) For every operation EO_{i_c} in EO_i's COL, remove its *nullified* mark and redo it, and for every operation after EO_{i_c} in HB, include the effect of EO_{i_c} by *inclusion* transformation.

(6) Repeat step 5 until there is no conflict operation in HB_1.

(7) For each operation EO_k in UDL, (a) Include the effect of EO_{new} by *inclusion* transformation. (b) check if it conflicts with EO_{new}.

(c) If EO_c conflicts with EO_{new}, mark it as *nullified*, and for each operation EO_j after EO_c in UDL, exclude EO_c's effect by *exclusion* transformation.

(8) Execute EO_{new}.

(9) Redo all operations without *nullified* mark in UDL sequentially from left to right.