Optional and Responsive Locking in Distributed Collaborative Object Graphics Editing Systems

David Chen and Chengzheng Sun
School of Computing and Information Technology
Griffith University
Brisbane, QLD 4111, Australia
D.Chen@cit.gu.edu.au, C.Sun@cit.gu.edu.au

Abstract

Object-based collaborative graphics editing systems allow multiple users to edit the same graphics document at the same time from multiple sites. This paper examines the use of locking to prevent the generation of conflicting operations in this type of systems. Two types of locks are examined: object and region. A locking scheme which preserves the intentions of all operations is proposed. Furthermore, the problems of lock ownership caused by concurrent operations are resolved.

Keywords locking, consistency maintenance, collaborative editing, graphics editing, distributed system.

1 Introduction

Real-time collaborative editing systems (CESs) allow multiple users from different sites to edit the same document at the same time. A particular type of CESs is the object-based graphics collaborative editing systems (OCESs). An OCES document contains one or more graphical objects. Each object is represented by a set of attributes such as type, size, position, color, group, etc.. Editing operations can be used to create, modify or destroy objects. In this type of systems *conflicts* may occur when concurrent operations are generated from multiple sites to change the same attribute of the same object. The execution of conflicting operations may cause inconsistency.

Locking is a technique originally developed for database systems [2] can be used to avoid conflicts. The concept of locking can be easily understood by the users, i.e. if someone locks an item, other people cannot access it. Therefore, locking is also used to maintain consistency in many OCESs including Ensemble [6], GroupDraw [4], GroupGraphics [7], and GroupKit [3]. In these systems, before an operation can be generated to edit an object, an exclusive lock on that object *must* be obtained. For example, to move an object, a lock on that object must first be obtained. This will guarantee that only one user, the lock owner, can edit an object at a time and conflict will not occur. Since locking is required before each request to edit an existing object, most systems provide locking implicitly. Once a user generate a request to edit an object, the system will automatically try to obtain the lock on that object.

Our OCES called GRACE [1, 8](GRAphics Collaborative Editing system) works in an environment where the users coordinate their activities, and conflict is possible but would be rare. So compulsory locking would be inefficient. GRACE is designed to provide high responsiveness, so pessimistic locking is not suitable. There are certain consistency properties GRACE needs to maintain. The roll back method used in optimistic locking scheme does not satisfy these properties. Therefore, GRACE uses a multi-versioning scheme which provides fast response time and maintains the consistency properties.

Despite the use of the multi-versioning scheme, locking still has an important role in GRACE. Locking can be used to reduce the amount of conflicting operations. Locks can be placed by the system or the users where conflicts are likely to occur. Once exclusive locks are obtained, future conflicts will be prevented. In summary, without locking, consistency will still be maintained by the multi-versioning scheme. With the use of locking, the amount of possible conflicts can be reduced. Therefore, this locking scheme is *optional* [10]. This is in contrast to other OCES locking schemes where locking is compulsory.

In order to incorporate locking into GRACE, concurrency control issues regarding locking operations need to be addressed. For example, how to solve inconsistency problems caused by concurrent locking operations targeting the same object? The solution to these problems should not slow down the response time and need to satisfy the consistency properties maintained by GRACE. The next section

0-7695-0577-5/00 $10.00 © 2000 IEEE

presents the existing results on GRACE, which includes the consistency properties maintained by GRACE. Section 3 examines the types of locks suitable for OCESs, locking operation generation, and inconsistency associated with locking. The locking scheme is presented in Section 4. Finally the results presented in this paper are summarized in Section 5.

2 Existing results on GRACE

GRACE has a distributed replicated architecture. Users of the system may be located in geographically-separated sites. All sites are connected via the Internet. Each site runs a copy of the GRACE software and has a copy of the shared document being edited. When an operation is generated, it is executed immediately at the local site. Then it is sent directly to all remote sites for execution. Depends on their orders of generation and execution, operations may be dependent or independent of each other. This dependency relationship in Definition 1, is based on the causal ordering relationship "$\rightarrow$" [9] which follows Lamport's event ordering [5].

Definition 1 Dependency relationship:
Given any two operations O_a and O_b. (1) O_b is said to be *dependent* on O_a *iff* $O_a \rightarrow O_b$. (2) O_a and O_b are said to be *independent* (or concurrent) *iff* neither $O_a \rightarrow O_b$, nor $O_b \rightarrow O_a$, which is expressed as $O_a || O_b$.

State vectors are used to determine the dependency relationships between operations. A state vector is a list of logical clocks. Each site maintains a state vector. Whenever an operation is generated at a site, it is time-stamped with the state vector value of that site. By comparing the state vector values between two operations, their dependency relationship can be determined [9].

GRACE maintains three consistency properties [9]: causality preservation, convergence and intention preservation. Causality preservation is maintained by delaying the execution of an operation until all operations causally before it have been executed. Convergence is achieved by serialization. Intention preservation is achieved by object multi-versioning scheme [1, 8].

Definition 2 Conflicting operations
Given two operations O_a and O_b, O_a and O_b are conflicting *iff* $O_a || O_b$ and their effects are to change the same attribute of the same object to different values.

With the multi-versioning scheme, the conflicting relationship between operations is first defined as in Definition 2. The execution of n mutually conflicting operations targeting object G will result in n concurrent versions of G in which each version will accommodate the effect of a conflicting operation. Non-conflicting operations have the compatible relationship. For any pair of compatible operations targeting the same object, there must be an object version which contains the effects of both operations.

As a part of the multi-versioning scheme, an object identification scheme is used to uniquely and consistently identify all objects and to allow concurrent versions made from the same object be determined. An object identifier consists of a set of operation identifiers. When an object is created by a create operation, its identifier set will contain the identifer of the create operation. A concurrent version's identifier contains the identifier of the object it is made from plus the identifier of the operation that created the version.

3 Locking in GRACE

In OCGEs, graphical objects are the obvious and suitable choice for applying locking since editing operations are generated to edit objects. Locking objects can prevent conflicts from occurring on those objects. Therefore, object locks have been chosen as one of the locking operations in GRACE. An object locking operation contains one or more object identifiers which specify the locking targets.

The other type of lock is region lock. A region lock can be used to lock an area of the shared document. Once a region is locked, only the lock owner can modify, create or delete objects within that region. Region locks are useful because users can specify private working areas which no other users can intrude. Conceptually, a region can be regarded as an object which contains a rectangular area (the region) and a list of objects within the region.

The term *lockable item* or simply *item* will be used in the following sections to represent either objects or regions which can be locked.

3.1 To lock or not to lock?

Locking before applying operations is compulsory in other OCESs. However, locking is optional in GRACE. Locks can be generated implicitly or explicitly. Locks are generated implicitly if they are placed automatically by the system. This approach is commonly used when locking is compulsory. However, to apply optional locks implicitly requires an intelligent system which can decide whether locks are required for a certain situation. The discussion of implicit locking generation is outside the scope of this paper. Locks are generated explicitly if they are issued by users (just like other editing operations).

What do the users get by locking before editing? If a user has locked an item, then the system guarantees that user the editing right to that item. If a user U_1 does not lock an item before editing, then it is possible that another user U_2 may lock that item. If U_2 has locked that item then U_1 would

lose editing right to that item. If a user obtains an exclusive lock on an item, then no other user can edit that item, hence no conflict will occur on that item (until that item becomes unlocks). In addition to editing rights, locking can also inform other users which part of the document a user is currently editing.

The terminology of lock ownership will now be introduced. If a user has a lock on an item, then that user owns the lock on that item, or simply that user owns that item. If a user owns an item, than that user has the editing right to that item.

3.2 Responsive operation generation

With the introduction of locking, user generated editing and locking requests need to be validated. A user's request is valid if its target item is either unlocked or s/he owns the lock of this item. Once a request has been validated, an operation is generated. Invalid requests are rejected, and the users are informed.

How to determine if an item is locked or not? Conceptually, each item has an attribute which indicates if that item is locked and by who. Each site maintains a list of all items. By finding an item in this list, the locking status of this item at a site can be determined.

Ideally, a user's request should be valid if at all sites the target item is either locked by this user or this item is unlocked. If a user owns the lock of an item at the local site, then this user will own the lock of this item at all sites. In this case, validation for any request by this user on this item can be done by checking the local locking status of this item. However, if an item is unlocked at the local site, it does not mean that this item is unlocked at all other sites. Under this condition, to validate a user request on an item which is unlock at the local site, synchronization is required to determine if this item is also unlock at other remote sites. This would slow down the validation process and thus the response time. In order to achieve fast response time, the synchronization in the validation process needs to be eliminated. Without synchronization, only the local locking status is known. Therefore, the validation condition is reduced to require only the item be unlocked locally as stated in Definition 3.

Definition 3 A valid request
Given an editing or locking request Q generated by user U_Q from site j, to edit/lock item I, Q is valid if either U_Q owns the lock on I or I is not locked in site j.

With this definition of a valid request, the validation process only checks the local document to determine if a request is valid. This means synchronization is not needed in the process starting from when a request is generated, until when an operation is executed locally. Only after that, is

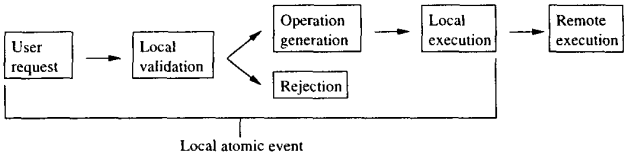

Figure 1. The operation generation process

network communication used to broadcast the operation to all remote sites for execution. It should be noted that these steps from request generation till operation broadcasting are done atomically at the local site. The process of operation generation is as shown in Figure 1.

4 Instant locking scheme

How to apply and preserve the effects of independent operations targeting the same item, where there is at least one locking operation? To satisfy the consistency property, once an operation is generated it has to be applied at all sites. Therefore, the basic idea behind this approach is to **allow *independent operations* targeting the same item be applied to that item.**

In order to examine the effects of this approach in detail, consider the application of this approach to an example with two independent operations O_1 and O_2. O_1 is a locking operation and O_2 can be either an editing or locking operation, so there are two situations at site k after O_1 has been applied to I:

1. If O_2 is an editing operation, then at site k, I is first locked by O_1 before being modified by O_2. This means that a locked item may be modified by a user who does not own the lock on that item.

2. If O_2 is also a locking operation, then at site k, I is first locked by O_1 before it is also locked by O_2. This means after a user has locked an item, it may have to share the ownership of that lock with other users.

So what is the point of locking if after a user has locked an item, that item may be modified or locked by other users? In this example, O_1 and O_2 are independent operations and only because of this will such situation occur. It is impossible to have an operation O_3 where $O_1 \rightarrow O_3$ and O_3 is generated by a different user from O_1 to edit/lock I. Such a request would be invalid.

The users should be informed that after they have locked an item, it is possible that this item may still be modified or locked by other users. At some stage, operations independent of the locking operation will all be applied, then the locked item can only be modified by its owner(s). The users should be informed of this event. This period, starting

from when a locking operation is executed, until all operations independent of that locking operation are executed is called the *unstable period*. During this period, a lock is said to be unstable. After the unstable period, a lock become stabilized.

Definition 4 Unstable period

Let I by any item at site j and O be any locking operation to lock I. The *unstable period* of the lock on I starts when O is executed at I at site j until all operations independent of O have been executed at site j.

While a lock is unstable, the number of owner of that lock can increase due to the application of independent locking operations. It is also possible for conflicts to occur on an object while its lock is unstable (the locking effect for this situation is discussed in Section 4.2). After this lock has stabilized, the number of lock owner cannot increase. Only the lock owners can edit or unlock this locked item. The number of owner decreases when an unlock operation is applied to this item. If the number of owner of this item is one, then the lock is exclusive and no conflict will occur on this item.

With this locking scheme, when a locking operation is generated the user who generated this operation will gain ownership to the target item instantly. Therefore this locking scheme is called *instant locking scheme* and the lock placed by this locking scheme is called *instant lock*. The next three subsections address some specific issues associated with this locking scheme.

4.1 Instant lock sharing

This section discusses the details of lock sharing. What should be the lock ownership for independent locking operations whose target item is the same or overlaps? For independent object locking operations targeting the same object, the users who generated those operations will share the ownership on that object. For independent region locks with overlapping regions, the ownership for overlapping regions will be shared and non-overlapping regions remain exclusive. For example, two target regions R_1 and R_2 with overlapping area of P. Only the ownership of P will be share and the ownership for rest of R_1 and R_2 remains exclusive.

Lock ownership for these two situations are obvious. However, what should be the lock ownership if there are independent object and region locking operations where the object G is inside the region R? Let U_R be the set of owners who generated the region locking operations and U_G be the set of owners who generated object locking operations. The lock ownership on G and R is as follows:

- All users in U_R and U_G should own G because G is inside R (or partially inside).
- Only the users in U_R should own R because users in U_G did not request for the region lock.

Since G is within a region lock, its behaviour is different from other locked objects. The effect on G is as follows:

- All users in U_R can edit G and can move G within R.
- All users in U_G can edit G but cannot move G within R, since U_G do not have access to R.
- All users in U_R and U_G can move G outside of R.
- After G has been moved outside of R (i.e. at the completion of drag and drop) then U_R lose their lock ownership on G. This is because the ownership of U_R on G is solely due to G being in R. So if G is outside of R, it is not within the scope of the region lock.

4.2 Instant locking and concurrent versions

For any object G with a lock that is shared or in the unstable period, conflicts may occur on G. Conflict will result in concurrent versions. What should be the lock ownership for these versions of G?

Our approach is to let the users who caused the creation of the versions own different versions according to which object their operation is applied to. For example, users U_1 and U_2 issued conflicting operations O_1 and O_2 on G. O_1 is applied to G_1 and O_2 is applied to G_2. Then U_1 will own the lock on G_1 and U_2 will own the lock on G_2. So the locks on both versions are exclusive.

The example works because each lock owner generated a conflicting operation which is applied to a different version. As the result, the lock ownership can be determined by which versions their operations are applied to. However, it is possible that a lock owner may generate an operation which is applied to more than one version. In this case, that user should own all locks to the versions which his/her operation is applied to. It is also possible that a lock owner may not generate any operation while conflicts occurred on the locked object. The only solution which will produce consistent lock ownership at all sites (without extra synchronization) is to let this user own the locks to all versions of that object.

In summary, let S be a set of independent operations all targeting the locked object G. Assume there is at least a pair of conflicting operations in S. For any user U_G who owns the lock on G, after executing all operations in S:

- If U_G generated an operation $O \in S$, then for any version G' of G which O is applied to, U_G will own the lock on G' .
- If U_G did not generate any operation in S, then U_G owns the lock for all versions of G.

4.3 When does a lock stabilize?

How to determine when a lock stabilizes? How would a site know when all operations independent of the locking operation have been executed at that site? This problem is similar to the garbage collection problem described by Sun et al for REDUCE [9]. The solution is based on the assumption that the underlying network is reliable and order-preserving between any pair of sites (e.g. TCP). Therefore, if a sequence of operations is sent from the same site, then these operations will arrive at its destination in the sending order.

The basic approach is that whenever a site j executes a locking operation O, j needs to sent a message to all remote sites telling them that j has executed O. If site k receives this message from j then k knows that any operation independent of O from j must have already arrived and been executed at k. If there are N sites, and k has received $N-1$ messages (excluding itself), then operations independent of O from all sites must have already arrived and been executed at k.

The actual implementation can be done by simply checking dependency of the operations. Dependency can be determined by comparing the state vectors. After j has executed O, it sends a message M containing the state vector of j to all sites. By comparing the state vector of O with M it can be determined that $O \rightarrow M$. So all operation from j independent of O must have already arrived.

Definition 5 Lock stabilization
For any item I locked by operation O, the lock on I at site j becomes stable *iff* j has received an operation dependent on O from all participating sites.

5 Summary

Inconsistency caused by conflicting operations is a typical distributed system problem. This paper has presented a locking scheme used to reduce the amount of conflicts that may occur. This locking scheme is used to support the consistency maintenance scheme of multi-versioning. Unlike other systems where locking is compulsory, this locking scheme is optional. Locks can be generated by the system or by the users like other editing operations. Fast response time for lock generation is ensure by the distributed replicated architecture and optimistic operation execution. Two different types of locking items are proposed: object and region. Object locking is widely used in other OCES, but region locking is unique in GRACE.

GRACE is the first OCES to adopt optional locking. However, optional locking was first proposed for REDUCE [10], which is a collaborative text editing system. These two systems take similar approaches to optional locking, but also solve different context specific problems related to text and graphics editing. The problem both systems need to solve is when independent operations target the same item. REDUCE uses shared locking to solve this problem, which is part of the instant locking scheme in GRACE. The difference between shared locking presented for REDUCE and instant locking presented in this paper is that this paper examines the effects of shared locks in more depth, and address the occurrence of the unstable period and discusses the possible effects duration of this period.

A prototype GRACE has been built as a Java application. Currently, GRACE only supports some basic objects and operations. We plan to extend the functionality of GRACE so it can be used for collaborative CAD or to draw connect diagrams such as ER-diagram, flow charts etc. By implementing and using the system, more research issues will be identified and investigated.

References

[1] D. Chen and C. Sun. A distributed algorithm for graphic objects replication in real-time group editors. In *Proc. of the International ACM SIGGROUP Conference on Supporting Group Work*, pages 121–130, Phoenix, USA, Nov. 1999.

[2] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. The benjamin/cummings publishing company, Inc., 1989.

[3] S. Greenberg and D. Marwood. Real time groupware as a distributed system: concurrency control and its effect on the interface. In *Proc. ACM Conference on Computer Supported Cooperative Work*, pages 207–217, Nov. 1994.

[4] S. Greenberg, M. Roseman, and D. Webster. Issues and experiences designing and implementing two group drawing tools. In *Proc. of the 25th Annual Hawaii International Conference on the System Sciences*, pages 139–250, Jan. 1992.

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

[6] R. E. Newman-Wolfe, M. L. Webb, and M. Montes. Implicit locking in the Ensemble concurrent object-oriented graphics editor. In *Proc. ACM Conference on Computer Supported Cooperative Work*, pages 265–272, Nov. 1992.

[7] M. O. Pendergast. GroupGraphics: prototype to product. In S. Greenberg, S. Hayne, and R. Rada, editors, *Groupware for Real-time Drawing: A Designer's guide*, pages 209–227. McGraw-Hill, 1995.

[8] C. Sun and D.Chen. A multi-version approach to conflict resolution in distributed groupware systems. In *Proceedings of the 20th IEEE International Conference on Distributed Computing Systems*, pages 316–325, Taiwan, Apr. 2000.

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

[10] C. Sun and R. Sosic. Optional locking integrated with operational transformation in distributed real-time group editors. In *Proceedings of ACM 18th Symposium on Principles of Distributed Computing*, pages 43–52, Atlanta, USA, May 1999.