

Optional Instant Locking in Distributed Collaborative Graphics Editing Systems

David Chen 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

Real-time collaborative editing systems are distributed groupware systems that allow multiple users to edit the same document at the same time from multiple sites. A specific type of collaborative editing system is the object-based collaborative graphics editing system. Traditionally, locking has been used as the major concurrency control techniques in this type of system. This paper examines locking in a supporting role to the concurrency control technique of multi-versioning. Two types of locks are examined: object and region. Two optional and responsive locking schemes, instant locking and instant exclusive locking, are presented. Their advantages and disadvantages are discussed.

1 Introduction

Computer-Supported Cooperative Work (CSCW) or computer-based groupware systems assist groups of people working simultaneously on a common task by providing an interface for a shared environment [2]. Groupware systems range from asynchronous or non-real-time tools such as electronic mail, to highly interactive synchronous systems such as Real-time Collaborative Editing Systems (CESs). CESs are distributed groupware systems which allow multiple users in different sites to edit the same document simultaneously.

A particular type of CES are the collaborative object graphics editing systems (OCESs). In OCESs, graphical objects such as line, rectangle, circle, etc., can be created. Each object is represented by attributes such as type, size, position, color, group, etc.. Editing operations are used to create, modify or delete graphical 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 operations involved in a conflict are called *conflicting*

operations. The application of conflicting operations may result in inconsistency which is a typical problem in distributed systems. For example, two concurrent *move* operations are conflicting if both move the same object to different positions. This may result in inconsistency where the same object appears in different positions at different sites.

Locking is a technique originally developed for concurrency control to maintain consistency in database systems. Locking is also used to maintain consistency in many OCESs including Ensemble [7], GroupDraw [5], GroupGraphics [8], and GroupKit [4]. 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.

However, traditional locking for concurrency control has some disadvantages. As the main/only concurrency control mechanism, locking is compulsory. For situations where conflicts rarely occur, compulsory locking is inefficient. One of the approaches to achieve locking is pessimistic locking. With this approach, when a user wants to edit an object, the system has to obtain an exclusive lock on that object before the user can edit that object. This approach has the disadvantage that the response time is slowed down by the time it takes to obtain the exclusive lock. To obtain an exclusive lock, synchronization between all sites are required. This network synchronization time is variable, and it can be large when the network is congested or the editing sites are far apart, this means slow response time.

To overcome this problem, some systems use optimistic locking. With this approach, the system assumes the exclusive lock will be granted, therefore, editing operations generated are executed locally without delay. This means two or more users may concurrently edit the same object. To ensure consistency, only the operations from one user is kept, concurrent operations from other users are aborted. To abort an executed operation, a roll back method is used which involves undoing the effects of executed operations to be aborted. This solution works well for database systems. However, having an operation undone by the system is unconventional in interactive editing systems. Furthermore, there is the problem of which operation to abort. The system decides which operations are to be aborted. However, this decision is not based on the merits of the operations such as the new position, color or size. This decision is made based on unrelated information such as the arriving order of the operations to a particular site. Randomly select operations to abort does not contribute to collaboration.

Our OCES called GRACE [1, 9] (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 (described in the next subsection) which provides fast response time and maintains the consistency properties. This scheme informs the users of the occurrence of conflict. It facilitates the resolution of conflicts by displaying the effects of all conflicting operations. This allows the users to make informed decisions on the desirable effect (this can be done at anytime that is convenient to the users).

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* [11]. 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.

This paper examines optional locking for GRACE. It is organized as follows: the next section presents the existing results on GRACE; Section 3 examines the types of locks suitable for OCESs, locking operation generation, and inconsistency associated with locking; two locking schemes are devised to maintain locking consistency, they are presented in Sections 4 and 5; Section 6 discusses when locks are stabilized; lastly, the conclusion is presented in Section 7.

2 Existing results on GRACE

The aim of this section is to provide some brief definitions and properties of GRACE to facilitate discussion. For more details about these definitions and properties, please refer to their respective referenced papers.

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 [10] as defined in Definition 1.

Definition 1 Causal ordering and dependency relationships:

- Given two operations O_a and O_b , generated at sites i and j , then O_a is causally before O_b , denoted by $O_a \rightarrow O_b$, iff: (1) $i = j$ and the generation of O_a *happened before* the generation of O_b , or (2) $i \neq j$ and the execution of O_a at site j *happened before* the generation of O_b , or (3) there exists an operation O_x , such that $O_a \rightarrow O_x$ and $O_x \rightarrow O_b$.
- 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 [10].

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 as defined 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.

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 OCGEs. 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, then 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

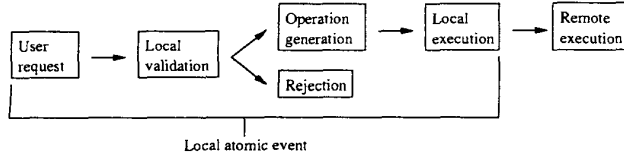


Figure 1: The operation generation process

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 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.

3.3 Inconsistency problems

Let I be an item which is unlocked at all sites and O_1 be a locking operation generated from site k to lock I . At the same time, another operation O_2 is generated independent of O_1 by a different user from site j to lock I . O_1 and O_2 would be first executed locally, then sent to remote sites for execution. However, when these operations arrive at the remote site, I has already been locked by the other user. As the result, these operations would have to be aborted. An inconsistency result will be produced where at site j , I is locked by O_2 , and at site k , I is locked by O_1 . A similar problem would occur if O_2 is an editing operation. O_2 would be applied at site j , but has to be aborted in site k because I has already been locked.

Divergence can be resolved by using serialization. However, serialization would lead to intention violation. This is because to achieve serialization, the effect of one operation needs to be undone after it has already been executed at the local site. For example, if the serial effect is that I should be locked by O_1 instead of O_2 . At site j , I is already locked by O_2 when O_1 arrives, so O_2 will need to be undone to allow O_1 to lock I . Therefore, the effect of O_2 is not preserved.

In the following sections two different locking schemes will be presented, instant locking and instant exclusive locking.

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. So, the basic idea behind this approach is to **allow independent operations targeting the same item to be applied to that item**.

In order to examine the effects of this approach in detail, consider the application of this approach to the example from the previous section. 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 such situation occurs. 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). 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 be 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

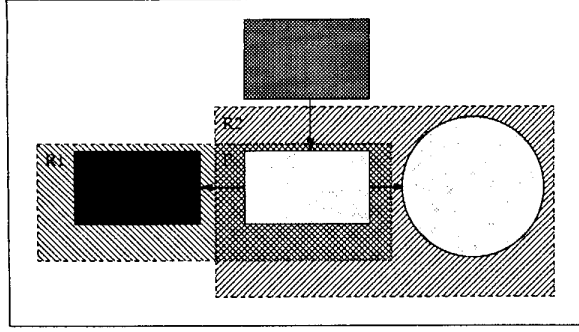


Figure 2: Region locks R_1 and R_2 with overlapping area P .

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 two 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 exclusively as shown in Figure 2.

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 (as shown in Figure 3)? 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:

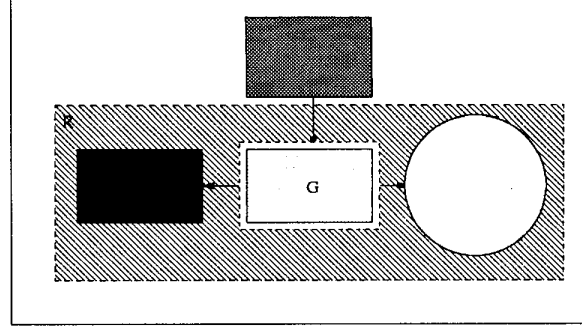


Figure 3: Independent object and region locks where the target object G is inside the region R .

- 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 ?

The first approach is to let the users who owned the lock on G own the locks on all versions of G . For example, let U_1 and U_2 be the users who share the lock on G . U_1 and U_2 generated conflicting operations O_1 and O_2 respectively to edit G . Versions of G will be made, the version G_1 for the application of O_1 and the version G_2 for the application of O_2 . Both U_1 and U_2 will own the lock on both G_1 and G_2 . The end result is that the lock on G_1 and G_2 is still shared.

The second approach is to let the users who caused the creation of the versions to 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.

Which approach is better? The goal of locking is to reduce conflict by (eventually) granting a user exclusive access to an item. With the first approach, the lock on the versions are shared and conflicts may still occur on

these versions. With the second approach the lock on the versions are exclusive and no conflict will occur on these versions (until they are unlocked). Therefore, the second approach is more desirable.

The example for the second approach 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 some lock owners may generate operations which are applied to more than one version and some may not generate any operation at the time. What should happen to their lock ownership?

Let U_3 be the user who also owns the lock on G (in addition to U_1 and U_2). U_3 generated an operation O_3 which is independent and compatible with O_1 and O_2 . O_3 will be applied to both G_1 and G_2 . So which version's lock U_3 should own?

To reduce the number of shared locks, U_3 can simply own the lock to one of the versions. Which version U_3 owns does not matter as long as it is the same at all sites. However, operations may arrive in different orders at different sites. Without waiting for all independent operations to arrive, different objects may be selected at different sites. For example, let the subscript attached to the operation also denote the site identifier where the object is generated, i.e. O_1 is from site 1. Let U_3 own the lock of the version made for the operation whose site identifier is the smallest, i.e. G_1 . With only two conflicting operations, the correct object can always be selected. However, if there is another operation O_4 which conflicts with O_1 and O_2 . O_2 and O_4 arrive first at site 3 (where U_3 is), then versions G_2 and G_4 will be created. Now G_2 would be incorrectly selected as the locked object for U_3 because $2 < 4$. When O_1 arrives the system can change U_3 's ownership from G_2 to G_1 . However, this is too late because in the mean time, U_3 could have generated operations to edit G_2 . This is wrong because U_3 is not suppose to have access to G_2 .

Selecting an object for U_3 based on any condition is going to have the same problem. The only solution is let U_3 own all the versions which may be selected. With this approach, the versions which may be selected are only the ones which O_3 is applied to. So if O_3 conflicts with O_4 then U_3 will own the lock of G_1 and G_2 but not G_4 .

Some lock owners may not generate any operation on G while these conflicting and compatible operations are being generated. What should happen to these users' lock ownership on versions of G ? As with the previous case, simply choosing a version would be incorrect. These users did not generate any operation, so select-

ing the version their operations are applied to is out of the question. The only option is to let these users own the lock of all versions of G .

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 .

5 Instant exclusive locking scheme

Although instant locking satisfies the consistency properties, during the unstable period, an item may be modified or locked by other independent operations. This may be undesirable for some situations. This section presents an instant exclusive locking scheme where **once a user locks an item at the local site, the locked item will not be modified or locked by any other user**. Therefore, the user will obtain exclusive access to the item instantly. This type of lock is called *instant exclusive lock* or *IE lock* for short. With this scheme, what the user sees is what s/he has locked.

IE locking scheme can be achieved by multi-versioning. When a user generate a locking operation O_1 to lock item I , an exclusive lock on I is granted instantly. No special action needs to be taken when there is no independent locking operation. However, if there is an operation O_2 which is independent of O_1 and O_2 is a locking operation to lock I , then versions of I are generated, where one version is locked by O_1 and another version is locked by O_2 .

5.1 Multi-versioning of locked items

An existing multi-versioning algorithm [9] is used in GRACE to handle conflicting modification operations. This section will examine how to utilize the existing algorithm to achieve instant exclusive locking.

The multi-versioning algorithm takes in an operation, compares it with a list of executed operations to determine if this operation conflicts with any executed operation. If there is no conflict, then this operation is simply executed. Otherwise, versions of the object targeted by the conflicting operations are made to accommodate the effects of these operations.

The locking operation, like any other modification operations can be feed into the multi-versioning algorithm and compare for conflict. However, the condition for conflict is different for locking operations,

therefore it needs to be redefined. A locking operation conflicts with any independent operation targeting the same item. This is because independent operations are generated by different users. If a user has a lock on an item, then that item cannot be edited/locked by other users. If a conflict is detected, the multi-versioning algorithm will make versions of the target object and assign these objects with unique identifiers.

Definition 5 Conflict IE locking operation

For an IE locking operation O_1 , O_1 conflicts with any operation O_2 iff: (1) $O_1 \parallel O_2$, and (2) O_1 and O_2 target the same item.

Simply defining a conflict condition for locking operations will allow the multi-version algorithm to work with IE object locking operations. However, something extra is needed to determine conflict and make versions when IE region locking operations are involved. A region is regarded as a rectangular area and a list of objects within the region. For any region R , an operation O is targeting R if 1) is targeting any object within R , 2) is creating an object within R .

Once a conflict is detected, what or how to create versions? Simply create object versions for all objects within the region would not work because the object versions would still appear inside the same region. Hence, versions need to be created for both the objects and the editing space. Each region is implemented as a type of object. When creating a version, a new object is created which contains a rectangle that specifies the required region and a list of objects in that region. To differentiate between different region versions, each region version is assigned an object identifier. The same object identification scheme used in multi-versioning can be directly applied to determine the identifier for regions. Versions of graphical objects in the region needs to be created, one set of objects for each region version. This is so that editing an object in a region version will not change the same object in a different region version. These objects' identifiers need not be changed since they belong to different region versions. An object in a region version can be uniquely identified by the using both the object and region identifiers.

5.2 Conflict locking effect

With IE locking scheme there is also an unstable period. During the unstable period the locked item will not be modified or locked by other users, however, concurrent versions of that item may be created. This section will examine specific cases of where concurrent versions are made and what should be their effects.

If all conflicting operations are IE object locking operations then object versions will be created. One

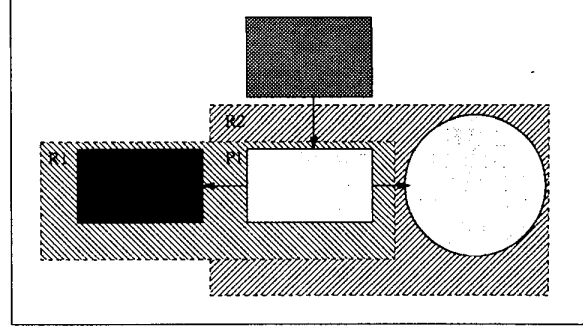


Figure 4: Independent and overlapping IE region locks result in region P being duplicated. P_1 is on top of P_2 .

version is locked for each user who generated the IE locking operation. If the conflicting operations are IE region locking operations, then only the overlapping regions should have different versions. These region versions will overlap each other and are displayed like overlapping objects. Only the region on top is visible to the users. The users are free to choose which region version at the local site should be visible. For example, there are two independent operations to lock regions R_1 and R_2 . If these two regions have overlapping area of P , then only P will have multiple versions. This is as shown in Figure 4, where region P_1 in R_1 is on top of region P_2 in R_2 .

It is also possible that there are concurrent region and object locking operations O_1 and O_2 respectively. The target object G of O_2 is inside the region R to be locked by O_1 . Should versions be created for the object or the region? Consider the case where only the object versions are created. Versions G_1 and G_2 are made for O_1 and O_2 respectively. G_1 will be in R , locked along with the surrounding objects. G_2 would be locked by O_2 . G_2 is located in R , but it cannot appear in R because R is IE locked and should not contain G_2 . Therefore, only creating object versions would not work. Region versions need to be created. In this example, region versions R_1 and R_2 will be made. R_1 is locked by O_1 . R_2 is not locked, except for the object G in R_2 is locked by O_2 .

If the the conflict involves both locking and editing operations, then only the versions the locking operations are applied to should be locked. For example, if O_1 is an IE locking operation to lock region R , and O_2 is an editing operation to move object G located inside of R . The result after the execution of O_1 and O_2 is that versions of R , R_1 and R_2 , are created for the application of O_1 and O_2 respectively. O_1 is applied to R_1 so R_1 will be locked. O_2 is applied to R_2 , then G in R_2 will be changed by O_2 , but R_2 remains unlocked.

At some stage, users may decide to compare the item

versions to make a decision on which item to keep. Supporting systems such as a voting system can be used to help users make such a decision. Once the decision is made, then unwanted items can be deleted or user(s) can issue an undo operation to undo the effect of conflict operations.

6 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 in [10] for the REDUCE system. The solution is based on the assumption that the underlying network is reliable and order-preserving between any pair of sites (e.g. TCP). This means 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 send a message to all remote sites telling them that j has executed O . Any 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 operations from j independent of O must have already arrived.

Definition 6 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.

7 Conclusion

Two optional locking schemes for collaborative graphics editing systems are presented in this paper. The instant locking scheme provides users with locks to objects instantly (although the lock may be shared). The instant exclusive locking scheme provides users with instant and exclusive locks to objects.

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] C. Ellis, S. Gibbs, and G. Rein. Groupware: some issues and experiences. *Communications of ACM* 34, 1:39–58, Jan. 1991.
- [3] J. Fjermestad and S. R. Hiltz. An assessment of group support systems experiment research: Methodology and results. *Journal of Management Information Systems*, 15(3):7–149, 1999.
- [4] 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.
- [5] 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.
- [6] J. F. N. Jr., R. O. Briggs, D. D. Mittleman, and P. B. Balthazard. Lessons from a dozen years of group support systems research: A discussion of lab and field findings. *Journal of Management Information Systems*, 13(3):163–207, 1996–1997.
- [7] 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.
- [8] 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.
- [9] 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, Taipei, Taiwan, Apr. 2000.
- [10] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality-preservation, and intention-preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction*, 5(1):63–108, Mar. 1998.
- [11] C. Sun and R. Sasic. 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.