Real Time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface

Saul Greenberg
David Marwood

1994

Cite as:

Greenberg, S. and Marwood, D. (1994) “Real time groupware as a distributed system: Concurrency control and its effect on the interface.” Proceedings of the ACM CSCW Conference on Computer Supported Cooperative Work, October 22-26, North Carolina, ACM Press.

Real Time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface

Saul Greenberg and David Marwood
Department of Computer Science, University of Calgary
Calgary, Alberta, Canada T2N 1N4
Tel: +1 403 220-6015
E-mail: saul@cpsc.ucalgary.ca

ABSTRACT

This paper exposes the concurrency control problem in groupware when it is implemented as a distributed system. Traditional concurrency control methods cannot be applied directly to groupware because system interactions includes people as well as computers. Methods, such as locking, serialization, and their degree of optimism, are shown to have quite different impacts on the interface and how operations are displayed and perceived by group members. The paper considers both human and technical considerations that designers should ponder before choosing a particular concurrency control method. It also reviews our work-in-progress designing and implementing a library of concurrency schemes in GROUPKIT, a groupware toolkit.

KEYWORDS: Real time groupware, computer supported cooperative work, distributed systems, concurrency control algorithms.

INTRODUCTION

Real time distributed groupware allows two or more geographically-separated people to work together at the same time through a computerized environment. These systems typically support a group's ability to manipulate their artifacts (documents) through a shared working space. The groupware controlling the workspace is often distributed (or replicated) at each participant's site, where each site's software is kept synchronized with its counterparts by inter-changing appropriate control messages. If care is not taken, a distributed groupware system can suffer concurrency control problems due to events arriving out of order, leading to inconsistencies in the image, the underlying document, and the group's mental model of what is actually going on.

The point of this paper is to expose the concurrency control problem in groupware, and to illustrate the negative effects that common synchronization schemes could have on the groupware interface and ultimately the groupware user. It does this by surveying traditional concurrency control approaches, and then applying them to sample groupware application tasks. While there are many schemes for managing concurrency in the distributed systems literature, we will show that groupware must be treated differently because it includes not only computers but people as well.

The groupware class we are addressing is those supporting highly interactive real time shared computational workspaces. Examples are group sketchpads [10,30], drawing tools [11,20] and group word processors [2,19]. We expect that participants in these conferences:

- are in real-time communication with each other e.g., through audio and video channels;
- focus and coordinate their attentions on what seems to be a shared visual workspace or document e.g., "what you see is what I see" [26];
- are aware of each other's fine-grained actions in the workspace;
- can interact simultaneously in the workspace;
- use the workspace as either a shared cognitive artifact for exploring ideas (like a whiteboard); or
- use it as a revision tool for discussing and tuning documents.

An Overview of the Problem

It is easy for people to conceptualize a multi-user shared space. We—designers and users—expect a shared space to behave like its physical counterparts. We perceive them as a single space containing distinct objects. We expect to see immediately what all others are doing, and we expect to be physically constrained from doing particular types of actions. Computerized shared spaces are not like that. There are usually multiple copies of the space and the objects they contain; there are time delays when showing the actions of others; and physical constraints are a simulated, rather than a natural, property of the objects.

In particular, concurrency control problems arise when the software, data and interface are distributed over several computers. Time delays when exchanging potentially conflicting actions are especially worrisome. We have been beguiled by the fast local area nets used to test groupware; the high latency of most wide area networks increase the probability of conflicting operations. If concurrency control is not established, people may invoke conflicting actions. As a result, the group may become confused because displays are inconsistent, and the groupware document corrupted due to events being handled out of order.

Concurrency control concerns researchers and designers in both CSCW and distributed systems. On one hand, this is a technical problem that—at least on the surface—appears amenable to approaches forwarded by conventional distributed systems research. On the other, this is a human problem, for the interface design must reflect the way people want and expect to work together. Both human and technical issues must be considered together, for the design of the interface and choice of concurrency control algorithm compromise each other.

This paper is not a conventional distributed systems paper, as it does not propose new algorithms for concurrency control. Rather, it introduces and surveys traditional approaches to concurrency control, outlines their importance to groupware, and indicates unavoidable problems and tradeoffs that must be considered when applying them to groupware interface design. It begins by reviewing basic concurrency control strategies used by conventional distributed systems. This sets the scene, and should make the paper accessible to CSCW researchers who are not computer scientists. The subsequent section explains how the groupware environment differs from conventional distributed systems, and illustrates the profound effects concurrency control methods can have on the interface and consequently on its users. Next, we summarize the human and technical considerations that groupware designers should ponder before choosing a particular method. We then briefly discuss our own work-in-progress designing and implementing concurrency schemes in GROUPKIT, our groupware toolkit. A brief appendix revisits the need for concurrency control within the broader context—and ongoing debate—of centralized and replicated architectures.

TRADITIONAL METHODS FOR MANAGING CONCURRENCY CONFLICTS

Management of conflicts due to concurrency is a well-researched topic in distributed databases and parallel simulation [5,7]. However, the application of concurrency control to the nuances of groupware is often neglected. While groupware researchers point to its importance [6,12,14,15,19,23], application developers typically ignore it outright, or consider concurrency control to be an issue to be remedied by some textbook approach. To set the scene, this section will review what is meant by concurrency control, and will present typical remedies to concurrency conflicts used in the database and simulation field. A later section will show why many of these classical approaches cannot be applied outright to groupware due to their effect on the user interface.

The Synchronization Problem

Interaction between distributed sites can be thought of as the exchange of messages or execution events. An event goes through a number of stages in its existence, usually something like: creation, local execution, transmission, reception, and remote execution (expanded from Ellis and Gibbs [6]). Without concurrency control, events are normally executed locally when they are created, transmitted to the remote site, and executed when they are received. The catch is that this is a multi-way process, and events can be interleaved—executed out of order—at different sites, which could lead to interference and inconsistencies [5]. Figure 1 shows a simple example of two sites losing synchronicity. Site 1 creates, executes and transmits event A at time *t1* then site 2 does the same with event B at *t2*. Event A is then received and executed by site 2 at *t3,* and event B is later received and executed by site 1 at time *t4*. At this point, site 1 has executed event A then B, while site 2 has executed B then A. The different ordering of actions may result in both sites being inconsistent and out of step. For example, problems will occur when these execution actions are not commutative, or when several local events (e.g., A_1, A_2, A_3) are considered part of a local transaction operation that are vulnerable to corruption by other events unless they are treated atomically.

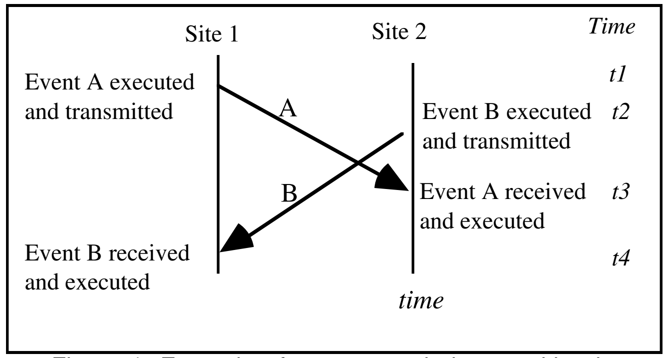

Figure 1: Example of event re-ordering resulting in possible loss of data integrity.

Concurrency Control through Serialization

Concurrency control is the activity of coordinating the potentially interfering actions of processes that operate in parallel. Algorithms work by synchronizing event interleaving so that atomic transactions (which may consist of several events) are executed serially across the entire system, or by repairing effects of out of order events to give the illusion that they are executed serially [5]. They usually create a total ordering of the events (e.g., see Lamport [16]). A scheduler then decides how to execute the events, or how to detect and repair order inconsistencies.

The "optimism" policy of a scheduler determines how events can be received[1]. Non-optimistic techniques ensure that events can only be received in order, thus guaranteeing consistency. Optimistic methods allow events to be received out of order, in which case inconsistencies must be eventually detected and repaired. Since both give the illusion of global *serialization*, we will consider them under that name[2].

Non-optimistic serialization ensures events are executed in the correct order at all sites by not allowing events to be received out of order. In Figure 1, for example, Event B

[1]Terminology for optimism varies in the literature. Another name for an optimisitic approach is "aggressive". Non-optimistic methods are called "conservative" or "pessimistic".

[2]This is another teminology problem, as the literature often talks about optimistic scheduling (in databases) or synchronization (in parallel simulation). We use the word "serialization" to distinguish it from locking, which we discuss later.

must wait for Event A to arrive and be executed before it can proceed. The cost is that the total execution time of a sequence could be quite slow, for the scheduler delays each event execution until its predecessors arrives or until it knows that it is safe to continue.

Optimistic serialization is based on the assumption that conflicting events are rarely received out of order, and that it is more efficient to proceed with execution and then repair problems than it is to guarantee correct ordering at all times. Of course, repair of order problems can be tricky, since one out of order event can put other incoming and otherwise valid events out of step. This requires the entire chain to be repaired.

One approach to repair is to have the system roll back to its state to just before the out-of-order event happened, and then re-execute the events in order. Consider Figure 1 from this perspective. At site 2, event B is executed out of order at t2, which is only noticed upon A's arrival at t3. Site 2 would roll back to its state before t2, and then execute events A and B in the correct order. This can be implemented in several ways. Jefferson's TIMEWARP system [13], for example, actually reverts back to previously saved states—even sending "anti-messages" to undo effects of messages it had sent over the network—and continues from there. Alternatively, a return to an old state can be simulated by applying undo functions [14], and then redoing events in the correct order.

Another approach to repair is to transform, via a set of rules, an arriving out-of order event so that its effect is the same as if it had arrived in order. For example, let us say the events in Figure 1 append items to the end of a list. When Site 2 receives A out of order, it applies a transformation rule that inserts A before B in the list, rather than at the end. In practice, many undo systems perform transformations as well, for it improves efficiency by reducing the number of undos [14,22].

Each of the above serialization policies have problems. We have already mentioned that non-optimistic schemes can be slow. For optimistic schemes, implementing state-saving and undo can be complex and expensive. A graver concern is that not all operations are "undoable" or "transformable". For example, a system may not be able to recall or transform events that trigger real world activities.

Privileged Access Through Locking

Another approach to concurrency management is locking, a method of gaining privileged access to some object (such as shared data) for a length of time. Locking can be used by a scheduler to ensure global serialization by controlling the order that sites obtain and release their locks [5]. Alternatively, it can be used as a way to guarantee local serialization over an atomic sequence, whose global order may otherwise not matter.

Typically, a site will *request* a lock to an object and, if no one else holds its lock, the request is approved and that site gains the lock. In this scheme, there may be only one approved holder of an object's lock, so if another site requests that lock, it will be *denied*. When the lock-holder no longer requires the lock, it is *released* and made available to the community.

The optimism of a lock policy determines whether or not execution pauses before a lock is approved, and whether a tentative lock must be approved before it can be released. Three levels of optimism are summarized in Table 1, and described in detail below.

Level of Optimism	Can manipulate the object while waiting for its lock	Can release the changed object while waiting for its lock
Non-optimistic	No	No
Semi-optimistic	Yes	No
Fully-optimistic	Yes	Yes

Table 1: Levels of optimism for locking.

Non-optimistic locking policies force a site to wait until a lock request is answered before it is allowed to manipulate the object. If requests are blocking, the site makes the request, waits for the response, and then manipulates the object if and only if it received the lock. If they are non-blocking, the site may perform other actions while the request is being served. However, it is not allowed to manipulate the object until the lock is granted.

Optimistic locking is similar in principle to its serialization counterpart. It assumes that sites will be frequently granted its lock requests. After requesting a lock, the requester immediately acquires a *tentative approval,* and it starts manipulating the object before it knows if it really has the lock. If the lock is then approved, it continues as normal. If it is denied, the object must be returned to its original state. Again, the premise is that proceeding without waiting is worth the occasional repair due to incorrect optimistic actions.

Fully- and semi-optimistic locking determine what a site is allowed to do if it has a tentative lock and has finished manipulating the object. With a *fully optimistic* lock policy, the site can put forth a "pending release", and go on to do other things (which may require further lock requests). If the lock is eventually denied, the site must remember how to revert the object to its original state. The difficulty is that if the site had manipulated other objects based on the tentative state of the old one, a full undo or state restoration system may be required to untangle all the cascading ill-effects of the illegal manipulation. These problems are avoided in a *semi-optimistic* scheme. While sites are allowed to manipulate objects with a tentative lock, they are blocked from moving onto other objects until the lock is approved or denied. At most, only a one step undo is required.

CONCURRENCY CONTROL IN REAL TIME GROUPWARE

Most concurrency control approaches are designed for non-interactive computer systems. They assume that computers can tolerate the delays associated with non-optimistic serialization and locking, or that they can accept the local inconsistencies and requirements of repair demanded by optimistic schemes. For example, TIMEWARP optimistic synchronization is often used for parallel simulation; its goal of optimism is to increase machine efficiency by pursuing a possible simulation path that is aborted only if an inconsistency is detected [13]. In replicated databases, concurrency control is used to guarantee that a data transaction is permanently recorded only when it can be completed correctly [5].

Groupware is quite different, because the distributed system includes not only computers but people as well. As this section will argue, people can be both more and less tolerant of concurrency problems than computers. As well, particular concurrency control schemes and the way it is presented in the interface will have quite different effects on the way people perceive groupware interfaces.

The following sub-sections will illustrate the effects of neglecting concurrency control, of using locking, or of using serialization on groupware and its participants. Two simple application scenarios will be used as examples. The first scenario (Figure 2a), considers a groupware drawing package that allows people to simultaneously create, manipulate, and edit objects such as bitmaps, lines, circles, and so on. The second scenario (Figure 2b), considers a groupware text editor that allows multiple people to enter and manipulate text at the same time. Both examples show two users, each with their own cursor on the display.

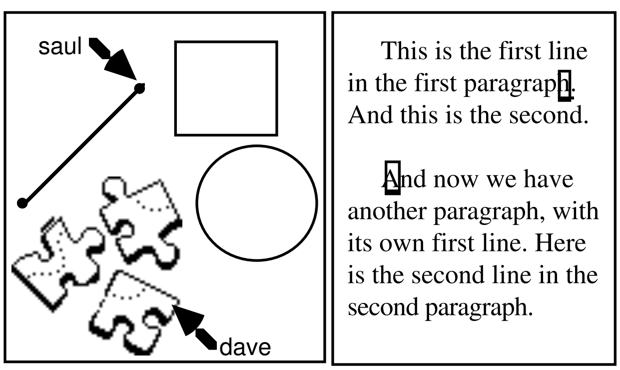

Figure 2: a) A group drawing tool, and b) a group editor.

Groupware Without Computer-Mediated Concurrency Control.

With no concurrency control, sites are free to exchange events, leading to possible inconsistencies because things occur out of order, or to competition for a resource that only one person should acquire. While this may appear unacceptable, it could be a reasonable strategy for some groupware situations. In particular, inconsistencies may not matter, or people will be able to mediate and repair their own actions and conflicts.

Inconsistencies may not matter. Some types of inconsistencies in the shared area and final document may be quite acceptable to people. Let us say participants are creating a bitmapped object with the tool shown in Figure 2a, where each can only set or clear pixels with a fine paint brush (e.g., sketching the jig-saw puzzle pieces). An ordering conflict can occur (as in Figure 1) when (say) one user sets some points at the same moment another clears it; the resulting images will differ by a few pixels at the two sites. This is illustrated in Figure 3, which shows a zoomed view onto 9 pixels of a larger bitmap. Users Saul and Dave start with the same image (top frame). Saul then draws with a diagonal stroke, while Dave erases with a vertical stroke; the different local views of the bitmap are shown in the middle frames. The operations then get transmitted and executed at the other sites, leading to the inconsistent final images in the bottom frame.

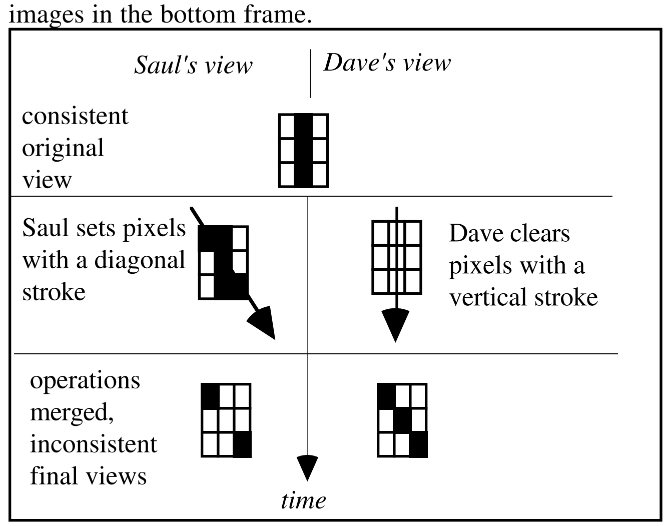

Figure 3: A blown-up view of a drawing, showing a very small 3x3 pixel region. Two users are painting a bitmap, resulting in a 1 pixel inconsistency due to out of order events, and neither users' goals being satisfied.

A difference of a few pixels in a large image will probably not matter to the participants, especially if they use the shared drawing space—as many do—to elaborate ideas, rather than to produce a document [27,11]. Our own GROUPSKETCH program ignores concurrency [10,11], and no user had noticed, let alone complained, about the occasional small-grained image inconsistencies that crept in. However, this is not true for all situations. If the differences become noticeable (e.g., if a wide brush stroke is used), or if the data integrity of the image must be maintained (e.g., if the final document is important), then some form of concurrency control must be used.

People mediate their own actions. People naturally follow social protocols for mediating interactions, such as turntaking in conversations, and the ways shared physical objects are managed. In a shared drawing or editing tool, we see what others are working on, and we usually would not perform an action that would interfere with them. Stefik, Bobrow et al [26] first noticed this behavior in the way people used COLAB, a face to face computer-supported meeting room. Whether we are working with a real or groupware whiteboard, it would be rude to scribble over another's marks as they are drawing them, or engage in a tug of war over a pen. Of course, there are situations where conflict can occur, such as:

- accidental interference due to one person not noticing what the other is doing;

- side effects of one action that have consequences on another (e.g. entering text at one point can repaginate the document, thus affecting the other person's view);
- intentional changing of control (interruptions, power struggles).

The point is that, in many groupware applications, concurrency conflicts may be rare because people mediate themselves. When conflicts do occur and slight inconsistencies appear, they may not be problematic in practice. Finally, if people do notice conflicts and problems, they are often quite capable of repairing their negative effects, and consider it part of the natural dialog. The computer's role can thus be seen as one that provides enough feedback and affordances of shared objects to support people's natural abilities. If computer-mediated concurrency control is used, it just becomes a way of avoiding or recovering from rare conflicts [26].

Examples of related work. The COLAB face to face meeting tool used visual feedback to show which screen objects were "busy" [5]. If an object was selected by a participant, it would be graying out, acting as a busy signal to tell others to leave it alone. TIVOLI is a shared distributed drawing system designed for small group meetings [20]. Its designers also noticed that conflicts are rare events and usually result in minor consequences. Still, they worried about concurrency. Because of the simplicity of how they treat shared drawing, they decided to treat drawing strokes as "immutable objects". If an object is subject to concurrent operations, a new object is spawned so that each user is working on their own copy. This is not only a technical solution since both sites are consistent, but also a human solution because people see the copies and can repair the drawing if needed.

Groupware and Locking

Locking can guarantee that people access objects in the shared workspace one at a time. Issues here are the grain size of the lock, the delays at acquiring a lock, and the effects of optimism.

Different grain sizes give a very different feel to groupware. The grain size of a lock determines how little or how much of the object(s) on display are managed by a single lock (grain size has also been referred to as "editable granularity" [19]). From a system's perspective, coarse granularity implies fewer lock requests, but less opportunity for concurrency as locks will be denied more frequently. The opposite is true for fine-grained locks. The choice is a balance between locking overhead and the amount of concurrency desired [5].

From a person's point of view, the coarsest grain is to have a single lock on the entire view or document, which forces people to revert to a serial interaction model with a computer-mediated turntaking protocol [8]. This is a popular strategy for "collaboration-transparent systems" that allow unaltered single-user applications to be shared by several people—it makes sense here because these applications only expect a single input stream [9,17,18]. However, most designers of collaboration-aware groupware recognize that simultaneous access to the view is critical to natural interaction [11,27].

Grain size can be progressively finer. The drawing package in Figure 2a, for example, can have locks on a set of objects (e.g., the three jig-saw pieces), a single object (the line or the circle), and even to a single handle of the object (a line could have two locks corresponding to its endpoints). This could have quite radical effects on the interaction feel. For example, consider a simple line. If the lock is on the handle, then two people could grab different endpoints and cooperatively move and resize it. If locking is at the object level, then this richness of interaction is precluded. Similarly, the text editor of 2b can have locks on characters, lines, paragraphs, sections—the coarser the lock, the more difficult it is for people to work closely together. This effect is worsened if object changes are presented to other users only after the lock is released—the interface changes from an "interactive" model of communication to an inferior "parcel-post" model [28].

Care should also be taken on deciding <u>what</u> to lock. The SASSE text editor, for example, locked on a user's text selection [2]. This worked fine, until users started using the selection mechanism as a form of gesture to highlight text they were discussing. Even though users sometimes wanted to overlap their selections, the locking mechanism made this impossible.

We strongly believe that the unit and granularity of lock should be user-centered. If people would find it natural to grab different endpoints of a line, or change different characters in a word, then the locks should be at a fine enough level to allow it. While this seems obvious, some applications violate this premise simply because they are designed from a systems-centered viewpoint.

Waiting for locks is not desirable. With non-optimistic locks, a user will select an object, and then be blocked from continuing until the lock is granted. If delays are barely noticeable, this may not matter. Noticeable delays, however, will interfere with the flow of interaction. For example, selecting a circle and moving it, or moving a text cursor forward and then typing should both be enacted as continuous operations. A delay makes the interface feel jerky, unresponsive, and unnatural. Another issue is that the interface would have to provide feedback showing that the object is waiting for a lock request to be served. Perhaps the color or outline of the object would change to a waiting state, or the cursor would transform itself (e.g., a question mark over a padlock).

Because of their simplicity and ease of implementation, a non-optimistic locking scheme may be chosen by a application developer when prototyping, or when response to lock request are perceived as instantaneous. When the network or processors suffer a visible delay, the non-optimistic lock may translate into a poor interface for the user.

Building a sensible interface for optimistic locking is hard. Optimistic locking avoids the delays mentioned above. Consider the case where a lock would be granted. Because optimistic locks already assume that a request will be met, the interface should be quite responsive. The problem is that it is not at all clear what to do when locks are denied, for the object that has been optimistically manipulated by the user must be restored to its original state. Let us consider several cases.

1. While a user is manipulating an object, its lock is denied. At this point the system could simply revert the object to its original state. If the user were manipulating a line, it would snap back to its starting location. If they were typing a sentence, the new text would disappear.
2. In a fully-optimistic scheme, a person could manipulate several objects in sequence that all have tentative locks. For example, consider a group drawing sequence. A person first moves a circle, then a line inside of the circle, and finally shifts their attention to a different part of the view. Moments later the lock on the line is denied. Should only the line be snapped back, or should the entire sequence be undone (the latter is sensible if actions are interdependent)? Will the person notice that the line has been moved? When they do, how do they know if it was due to a lock denial, or if another person put it back? The late denial of an optimistic lock can easily cause confusion.
3. In a semi-optimistic scheme, a person is not allowed to proceed to a new object until the lock transaction on the last object is completed. Depending on the time granularity of a person's interaction with the object, this could lead to continuity problems similar to non-optimistic locks (e.g., typing in the group editor would be jerky if locks were requested character by character). However, reversion of an object due to a lock denial would be immediately apparent and easily understood; the problem arising with the optimistic lock would be avoided.
4. What do other people see when a person is manipulating an object that has a tentative lock? Consider the case of two people grabbing a line, each with a tentative lock. Should each see a "second" counterpart copy of the moving line? Would a third person see zero, one, or two line movements? If we decide not to transmit object manipulations until a lock is granted, then this would make the group interaction uneven, and perhaps unsynchronized with what people were saying to each other through the voice channel.

Even though optimistic locking has its problems, it is probably a reasonable strategy for many groupware applications. It is an easy concept for participants to understand. Because people mediate their own concurrency control, conflict should be rare and the gains of optimism high. The cost to bear is higher implementation costs, and potential confusion on the (hopefully infrequent) denials.

Examples of related work. Coarse-grain locks, the simplest locking strategy, is quite effective in particular situations. Greenberg [8] describes a variety of turn-taking protocols for shared view systems, which is the equivalent of document locking. "Lock" requests are usually handled through special turr-taking interfaces, each with their own policy that controls how a user can request a turn or relinquish control. Many version control systems also use coarse grain locking, where people "check in" and "check out" portions of documents for lengthy periods of time, such as software modules (e.g., Tichy's RCS [29]).

A variety of concurrent text editing systems have locks based on variable-sized text regions. In MACE, for example, a user specifies a pair of locks that bounds the top and bottom of a text region [19]. Since locked regions cannot overlap, the text is effectively partitioned amongst its participants. This scheme also means that the size of a locked region will grow and shrink as its contents are edited. MACE also allows a user to control how local changes within the locked region are displayed to others. In the SASSE text editor, non-optimistic locking of regions are specified by a user's text selection operation [2]. The DISTEDIT toolkit [15,22] also provides region locks on text, which is set explicitly by the user or automatically by the system (it tries to acquire the smallest possible lock on the text region being modified). What makes DISTEDIT interesting is its handling of multi-operation actions that affect many regions in a document (such as a global replace-string operation) [15]. DISTEDIT first executes the action locally without displaying it to the user, decides what locks are required, and then tries to acquire all the locks for the action. If the lock request succeeds, so does the transaction. If it fails, the operation is internally undone; since the display was never updated, the user only sees that the requested operation had failed and is not put into the position of trying to make sense of the undo operations.

There are a variety of other approaches to locking. Our own GROUPDRAW system [11] contains a distributed scheme to handle fine-grained object locking, where locks are requested at the handle level. The SUITE Multi-User Framework employs a flexible access-control model to associate data displayed by the groupware application with a set of collaboration rights [25]. When conflict occurs, conflict resolution rules examine the access rights to determine who gets the object. Finally, specific lock policies can use heuristics to decide whether a lock in use should be taken away and granted to another requester. For example, Grief, Seliger and Weihl [12] describe the "tickle lock" that reassigns the lock if the current holder is inactive. Similarly, "pause detection" automatically releases a lock after a prescribed period of inactivity [8].

Groupware and Serialization

While serialization guarantees data integrity, it also comes at a tradeoff at the user's interface to groupware. As with locking, some of the problems that will occur can depend on the optimism of the scheme. Others are due to the unique properties of serialization.

Serialization can lead to strange interaction behaviors. Let us assume, for a moment, a system that provides instantaneous serialization. What would a typical interaction feel like? Consider the drawing program in Figure 2a, where two people try to move the circle object. The actual interaction is illustrated in Figure 4; the cross hairs show the original center of the circle. In Scene 1, both

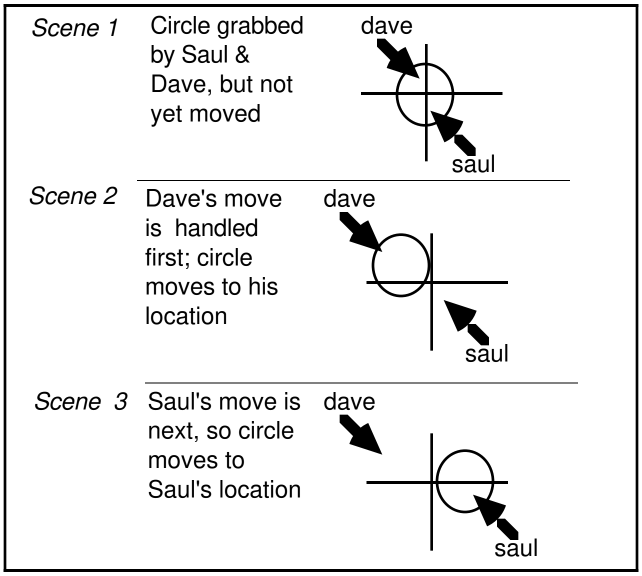

Figure 4: Effects of serialization: the “fight” for control.

Saul and Dave grab the circle, but have not yet moved it. In the next scene, Dave and Saul both move the circle, but Dave’s move is handled first. The circle moves left to Dave’s cursor location. Saul move is handled next (Scene 3), and the circle moves right to his cursor location. The circle continues to bounce between the two; its final resting place will be at the location of the last person to let go of it. While the overall behavior is “correctly” serialized, it leads to an interface tension where users can end up fighting over control of a shared object.

Serialization is acceptable in some situations. In applications such as in bitmap drawing, the granularity of interaction is so small that participants may not even notice the effects of serialization. The advantage here is that document integrity is maintained. Another possibility is that the effect shown in Figure 4 can serve as “feedback” to participants, where participants see that a control conflict is occurring, and may then use their own social protocols to resolve it.

The above scenario assumed instantaneous serialization. Because life is rarely like that, we have to consider the effect of mitigating delays through optimization schemes.

Non-optimistic serialization blocks event handling until there is a guarantee that events are executed in a global order. Users now have to wait for synchronization to occur before they can continue. As with a non-optimistic locking scheme, this interferes with the flow of interaction, and begs the question of how feedback to this “pending” state should be handled.

Optimistic serialization does not require the user to wait for synchronization before events are transmitted. As with locking, this bodes well when events are in order most of the time. The difficulty is how the interface shows repair of the occasional out-of-order sequences through undo or transformation. To illustrate, consider again the sequence in Figure 4 from Saul’s perspective, where events occur in the order shown and starting from Scene 1. Saul’s move is

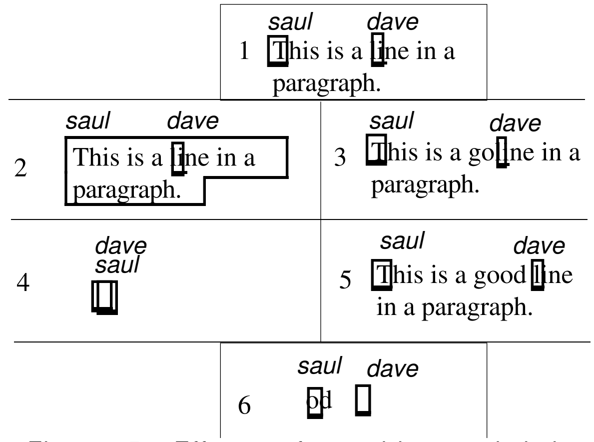

Figure 5: Effects of repairing optimistic serialization in a group editing session. Common views are centered, Saul’s view is on the left, Dave’s on the right.

optimistically considered in-order, so he would then see the illustration in Scene 3. Dave’s event arrives, so the system would “undo” back to Scene 1 (because this is the last common state), then show Scene 2, and then Scene 3. In effect, the circle will appear to jump erratically between several cursor locations. Such action would undoubtedly be confusing and disrupting to the user. While confusion could be minimized by having the undo system only show the final undo state rather than the intermediate steps, there is no guarantee that it would be eliminated.

In the above example, a transformation scheme would avoid extraneous “bounces” of the circle, for it would notice that the undo/redo operation simply restores the drawing to the same state. But consider an optimistic text editing operation, as illustrated in Figure 5. Saul’s view of events is on the left, and Dave’s view on the right. Here both Saul and Dave start in the same paragraph (Time 1, center top). The global order of operations, marked next to each view, is:

Time 2: Saul selects the paragraph.
Time 3: Dave types “go”
Time 4: Saul deletes the paragraph
Time 5: Dave types “od “

Due to delays, neither site realizes that events are out of order until they are all performed. Saul believes that he has deleted the paragraph, and Dave believes he is typed the word “good” before the word “line, and the optimistic serialization provides them the appropriate feedback as shown in their respective views. When the system notices that events are out of order, it either undos or transforms them to be in-order, giving the state shown at time 6. Of course, this is not what either person wanted, leaving them in a state of confusion. In this case, an “undo” trace may marginally help them understand how the screen got that way, while a transformation would not.

The choice of serialization scheme is a trade-off between the benefits it gives and the problems it introduces. The designer needs to consider carefully the particular effects of applying a serialization approach to a groupware application, and decide whether or not it is appropriate to the situation. The designer must also consider how repair effects are presented to the user.

Examples of related work. The GROVE outline editor uses the distributed operational transformation algorithm (dOPT) to transform, via a set of rules, a pair of operations so that its effect is the same at both sites regardless of order [6]. dOPT is not really a serialization algorithm, for the final outcome may be quite different than actually executing operations in order; the goal is to produce a consistent sensible result rather than maintain order. The designer has the additional burden of specifying the transformation matrix, which is non-trivial. "Dependency detection", proposed but not implemented by Stefik et al [26], detects time-stamp conflicts. Rather than repairing the conflict itself, the system would inform the group about the conflicting operations, and they are expected to fix it through manual intervention. The GROUPDESIGN shared drawing tool uses the Optimal Response Time (ORESTE) algorithm to manage serialization [14]. Instead of serializing all events (marked by time-stamps), ORESTE allows events to be executed out of order when its effects are the same as if it were executed in order (i.e., commutative). When this is not possible, ORESTE applies undo/redo to reorder the events. This scheme improves efficiency and unpleasant interface effects by minimizing the number of unnecessary undos and redos. However, it does require the designer to state, using the semantics of the application, what operations are commutative. Finally, the DISTEDIT toolkit [15,22] and the GINA application framework [4] explore the notion of group undo from the different perspectives of concurrency repair and user-initiated undo.

CHOOSING CONCURRENCY METHODS FOR REAL-TIME GROUPWARE

The choice of a concurrency control method can be difficult. A wrong choice can lead to an unusable system. Selecting an overly powerful approach could be overkill for the application, and much development time could be expended for schemes that are unnecessary or used only rarely. Selection of a conflict management scheme can depend heavily on implementation considerations and interface tradeoffs. The discussion below summarizes what we believe are the main considerations.

Human considerations. Unlike most distributed applications, groupware demands that we consider the role of the human when selecting concurrency control schemes. The foremost human consideration is whether or not it agrees with a person's model of events. For example, a person grabbing and dragging a shape or handle in an object-oriented drawing normally considers the shape to be theirs to position and modify. If it were moved by another user, they could both become confused or irritated. Thus a locking scheme could work here, or the system may eschew concurrency control all together and leave its management to its users. On the other hand, users would not expect to gain control of a bitmap picture area, and may in fact want to work very close to each other. In this case, a serialization scheme may suffice.

The latency of interactions over networks and processors has a major impact on how interactions are perceived by users, and ultimately the choice of concurrency strategies. If the system can approve/deny a lock before the person is ready to use the object , then a non-optimistic lock is fine. If it is a bit slower, but can approve/deny a lock before the object is released, then a semi-optimistic lock would suffice. Fully optimistic locks should only be considered when the response times are quite slow, as interaction continuity may suffer. Similarly, if serialization occurs almost immediately, a non-optimistic serialization scheme is fine.

If optimistic serialization and locking schemes are considered necessary, then attention should be paid to the way pending operations, undo/redos, and transformations are presented to the user. "Clever" interface presentations of lock denial or concurrency repair may minimize user confusion in particular situations. Unfortunately, there is no recipe on how this can be achieved in general. The bottom line is that the interface must present the user with an understandable model of what is going on when locks are denied and order problems repaired.

In practice, the designer should walk through a variety of task scenarios, and consider the effects of particular concurrency control methods, latency, and feedback mechanisms on the user's model of interaction.

Technical considerations. Optimistic serialization and locking are considerably more difficult to implement than their non-optimistic counterparts. Optimistic serialization schemes must be able to receive events out of order but execute them in order (implying undo/redo and transformations). Similarly, optimistic locking schemes must be concerned about returning one or more objects to their original states if locks are denied after a sequence of events. Semi-optimistic locking is far easier, for only a one-step undo is required. Both problems bring complex implementation issues, making optimistic schemes significantly more complex than non-optimistic ones.

Resource use must be considered as well, for different schemes may require significant overhead. The computational complexity of some optimistic serialization algorithms can be quite high, particularly those that use complex undo/redo functions. Similarly, the number of network transactions should be a concern. Some schemes may require many message exchanges between sites, increasing overall network and processor latency. In the worst case, a system which would have been fairly responsive without complex concurrency control (and thus not need sophisticated schemes), could have its performance dragged down to make it necessary! Memory requirements can also be excessive on some algorithms. For example, state-saving is memory intensive, particularly if multiple copies must be kept.

CONCURRENCY CONTROL IN GROUPKIT: WORK IN PROGRESS

We are now incorporating concurrency control into GROUPKIT, a toolkit that makes it easy to build real time, distributed, and fully replicated applications [24]. Unlike other groupware researchers who have considered concurrency control in specific domains (e.g., text outliners [6], drawing tools [3,11,20,26]; text editors [2,15,19]), we want GROUPKIT to support a wide variety of applications. Our philosophy is that no generic method for concurrency control will suffice, and that the toolkit should provide many methods and allow developers to pick and choose the most appropriate one.

This is work in progress, and we have implemented only a few of the previously mentioned schemes as interchangeable layers. A *lock layer* allows a developer to attach either non-optimistic or semi-optimistic locks to objects, as well as a callback procedure that is executed whenever the lock state is changed. The system then tracks all requests to the lock, changes the state of the lock as appropriate, and triggers the callback so that the application can handle it. It is up to the developer to take the appropriate action i.e., to show feedback of state, and to undo operations if necessary. If non-optimistic locks are used, the states are 'approved', 'released', or 'denied'; requests for a lock are blocked until a definite answer can be provided. Semi-optimistic locks add a 'tentative' state; lock requests immediately return a tentative lock, and the interaction can continue. The final lock approval or denial triggers a state change which executes the call back, and the appropriate action can be taken by the application.

The actual locking algorithm can take several forms, and is implemented in a *network layer*. This layer provides the lock layer with communication among the conference applications. The lock layer asks the network layer if there are any other lock holders. If there are, the network layer may indicate that the lock is not available, or may force the lock holder to release the lock. The application developer need not know the details of the network layer to use the locking package. The advantage is that different underlying locking algorithms can be used.

1. The *polling network layer* asks all the other sites if they hold or want to hold the lock. When all sites have responded, it can return the information to the lock requester. The performance here is dictated by the slowest response from all polled sites, but is very easy to implement.
2. In the *token network layer*, each lock has an associated token that is always held by one site. Only the token holder can hold the lock, so a site wanting the lock must first request the token from the holder. The token holder either passes the token to the requesting site, or informs the site that the token cannot be passed. (This is similar to the "roving lock" idea proposed by Stefik et al [26].) This method works well when there is a high probability that a site will repeatedly request the same lock. Since it already holds the token, no network traffic is generated and response is fast. The cost is that the scheme is more complex to implement.

There are, of course, limitations to our approach. We designed the layers so that locking and network schemes can be globally substituted. For example, a single line of code determines which locking mechanism should be used. On reflection, this was a mistake. We now realize that different objects and operations in a single application may require different concurrency control methods. We will be redesigning our layers to allow methods to be assigned on a per object basis. The second limitation is our handling of undo, for we leave that up to the developer to manage. While single-step undo is easy, multiple undos and redos are hard [22] and only the most serious developer would implement undo from scratch. While we know that we must provide a framework for undo within the toolkit, we are stymied on how to do this, since undo/redo is often tightly intertwined with the application semantics which is not in our control.

SUMMARY

This paper described a variety of concurrency control issues that crop up in distributed real time groupware. Its premise is that groupware is a fundamentally different application domain from traditional distributed systems, because the transaction process includes people as well as computers. Different concurrency control methods, such as locking, serialization, and the degree of optimism, have quite different impacts on the interface and how transactions are shown to and perceived by group members. The presence of people also means that we can consider human-mediated concurrency control, especially when it is part of the natural conversation process. We conclude that there is no superior method, but that each has its merits and problems. The final choice rests upon the application domain, system performance considerations, and—most importantly—the nuances of distributed group interaction.

The paper also reviewed how we are approaching concurrency control in GROUPKIT, our groupware toolkit. Following from the previous discussion, GROUPKIT does not tout a single method or algorithm. Rather, it gives groupware developers the power to choose a concurrency scheme that fits the nuances of their application.

APPENDIX: CONCURRENCY CONTROL IN REPLICATED & CENTRALIZED ARCHITECTURES

Groupware researchers have long argued the merits of centralized vs. replicated architectures [1,9,17,18,21]. On the surface, the simplest way of implementing concurrency control is through a *centralized* architecture. The centralized approach uses a single application program, residing on one central machine, to control all input and output to the distributed application. Server processes residing at each site are responsible only for passing requests to the central program, and for displaying any output sent to it from the central program. The advantage of a centralized scheme is that synchronization is easy, as state information is consistent since it is all located in one place. Events will never be received out of order (they are usually handled first-come, first-served). Locking is also easy, as only one copy of the object exists. *Replicated* architectures, on the other hand, execute a copy of the program at every site. Thus each replication must use specific concurrency control algorithms to coordinate their actions, and must worry about handling undo if optimistic schemes are used.

Because of its simplicity at handling concurrency, centralized architectures for groupware has had many advocates [1,9,21,30], and one may wonder why a replicated approach would ever be considered. The answer concerns the issue of latency. A centralized scheme implies sequential processing, and is inherently non-optimistic. A request is received and handled by the central application before the next one can be dealt with. If the system latency is low, this is not a problem. But if it is high, the entire system will become sluggish. A replicated scheme, on the other hand, implies parallel processing which maximizes the use of optimistic schemes. Events can occur in parallel at each replication, with the optimistic method mediating any problems. While overkill for low latency, it can address the interface issues in systems that have noticeable delays.

There is no real answer to whether a centralized or replicated scheme works best for groupware. Rather, it is a set of tradeoffs that revolve around the way they handle latency, ease of program installation and connection, programming complexity, synchronization requirements, processor speed, the number of participants expected, communication capacity and cost, and so on.

Software Availability: GROUPKIT is available via anonymous ftp from ftp.cpsc.ucalgary.ca, in the directory pub/projects/grouplab/software.

Acknowledgments. This research was supported by the National Sciences and Engineering Research Council of Canada through its strategic and operating grant program.

REFERENCES

1. Ahuja, S.R., Ensor, J.R. and Lucco, S.E. (1990) "A comparison of applications sharing mechanisms in real-time desktop conferencing systems." In *Proceedings of the ACM COIS Conference on Office Information Systems*, pp. 238-248, Boston, April 25-27.
2. Baecker, R.M., Nastos, D., Posner, I.R. and Mawby, K.L. (1993) "The user-centred iterative design of collaborative writing software." In *Proceedings of the ACM INTERCHI Conference on Human Factors in Computing Systems*, pp. 399-405, Amsterdam, April 24-29.
3. Beaudouin-Lafon, M. and Karsenty, A. (1992) "Transparency and Awareness in a Real-Time Groupware System." In *Proceedings of the ACM UIST Symposium on User Interface Software and Technology*, pp. 171-180.
4. Berlage, T. (1992) "The GINA Interaction Recorder." In *Proceedings of the IFIP TC2/WG2.7 Working Conference on Engineering for Human Computer Interaction,* Finland, Aug 10-14.
5. Bernstein, P., Goodman, N. and Hadzilacos, V. (1987) *Concurrency control and recovery in database systems*, Addison-Wesley.
6. Ellis, C.A. and Gibbs, S.J. (1989) "Concurrency control in groupware systems." In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pp. 399-407, Seattle, Washington, USA.
7. Fujimoto, R.M. (1990) "Parallel discrete event simulation." *Communications of the ACM*, **33**(10), pp. 31-53, October.
8. Greenberg, S. (1991) "Personalizable groupware: Accommodating individual roles and group differences." In *Proceedings of the ECSCW European Conference of Computer Supported Cooperative Work*, pp. 17-32, Amsterdam, Sept 24-27, Kluwer Academic Press.
9. Greenberg, S. (1990) "Sharing views and interactions with single-user applications." In *Proceedings of the ACM COIS Conference on Office Information Systems*, pp. 227-237, Boston, April 25-27.
10. Greenberg, S. and Bohnet, R. (1991) "GroupSketch: A multi-user sketchpad for geographically-distributed small groups." In *Proceedings of Graphics Interface '91*, pp. 207-215, Calgary, Alberta, June 5-7.
11. Greenberg, S., Roseman, M., Webster, D. and Bohnet, R. (1992) "Human and technical factors of distributed group drawing tools." *Interacting with Computers*, **4**(1), pp. 364-392, December. Butterworth-Heinemann.
12. Grief, I., Seliger, R. and Weihl, W. (1986) "Atomic data abstractions in a distributed collaborative editing system." In *Proceedings of the 13th Annual Symposium on Principles of Programming Languages*, pp. 160-172.
13. Jefferson, D.R. (1985) "Virtual time." *ACM Transactions on Programming Languages and Systems*, **7**(3), pp. 404-425, July.
14. Karsenty, A. and Beaudouin-Lafon, M. (1993) "An algorithm for distributed groupware applications." In *Proceedings of the 13th International Conference on Distributed Computing Systems ICDCS'93*, Pittsburgh, May 25-28.
15. Knister, M. and Prakash, A. (1993) "Issues in the design of a toolkit for supporting multiple group editors." *Computing Systems (The Journal of the Usenix Association)*, **6**(2), pp. 135-166, Spring.
16. Lamport, L. (1978) "Time, clocks and the ordering of events in a distributed system." *Communications of the ACM*, **21**(7), pp. 558-565, July.
17. Lauwers, J.C. and Lantz, K.A. (1990) "Collaboration awareness in support of collaboration transparency" In *Proceedings of the ACM SIGCHI Conference on Human factors in Computing*, Seattle Washington, April 1-5.
18. Lauwers, J.C., Joseph, T.A., Lantz, K.A. and Romanow, A.L. (1990) "Replicated architectures for shared window systems: A critique." In *Proceedings of the ACM COIS Conference on Office Information Systems*, pp. 249-260, Boston, April 25-27.
19. Newman-Wolfe, R. E. and Pelimuhandiram, H. K. (1991) "MACE: A Fine Grained Concurrent Editor." In

Proceedings of the ACM COCS Conference on Organizational Computing Systems, pp. 240-254.

20. Moran, T., McCall, K., van Melle, B., Pedersen, E. and Halasz, F. (in press) "Design principles for sharing in Tivoli, a whiteboard meeting-support tool." In *Designing Groupware for Real Time Drawing*, S. Greenberg, S. Hayne & R. Rada ed. McGraw Hill.
21. Patterson, J.F., Hill, R.D., Rohall, S.L. and Meeks, W.S. (1990) "Rendezvous: An architecture for synchronous multi-user applications." In *Proceedings of the ACM CSCW Conference on Computer Supported Cooperative Work*, Los Angeles, California, October 7-10.
22. Prakash, A. and Knister, M.J. (1992) "Undoing Actions in Collaborative Work." In *Proceedings of the ACM CSCW Conference on Computer-Supported Cooperative Work*, Toronto, Nov 1-4, pp. 273-280.
23. Rodden, T. and Blair, G. (1991) "CSCW and distributed systems: The problem of control." In *Proceedings of the ECSCW European Conference on Computer Supported Cooperative Work*, pp. 49-64, Amsterdam, Klewar Press.
24. Roseman, M. and Greenberg, S. (1992) "GROUPKIT: A groupware toolkit for building real-time conferencing applications." In *Proceedings of the ACM CSCW Conference on Computer Supported Cooperative Work,* Toronto, Nov 1-4, pp 43-50.
25. Shen, H. and Dewan, P. (1992) "Access Control for collaborative environments." In *Proceedings of the ACM CSCW Conference on Computer Supported Cooperative Work*, pp. 51-58, Toronto, Ontario, Nov 1-4.
26. Stefik, M., Bobrow, D.G., Foster, G., Lanning, S. and Tatar, D. (1987) "WYSIWIS revised: Early experiences with multiuser interfaces." *ACM Transactions on Office Information Systems*, **5**(2), pp. 147-167, April.
27. Tang, J.C. (1991) "Findings from observational studies of collaborative work." *Int J Man Machine Studies*, **34**(2), pp. 143-160, February.
28. Tatar, D.G., Foster, G. and Bobrow, D.G. (1991) "Design for conversation: Lessons from Cognoter." *Int J Man Machine Studies*, **34**(2), pp. 185-210, February.
29. Tichy, F. W. (1982) "RCS: A revision control system." In *Proceedings of the ECICS 82 European Conference*, Stresa, Italy, September.
30. Wilson, B. (in press) "WSCRAWL 2.0: A shared whiteboard based on X-Windows." In *Designing Groupware for Real Time Drawing*, S. Greenberg, S. Hayne and R. Rada ed. McGraw Hill.