

## Supporting Unconstrained Interaction with Application Sharing Systems

Du Li, Rui Li, and Prabhu A. Inbarajan  
Texas A&M University

**Abstract:** Traditional application sharing systems generally resort to turn-taking protocols for concurrency control. This fundamentally constrains human-computer interaction and is not suitable for a range of cooperative work that demand high local response and free flow of thought. On the other hand, supporting unconstrained interaction with application sharing systems ensues significant challenges on consistency maintenance. Traditional concurrency control protocols cannot be directly applied to maintain consistency in application sharing systems. In this paper we analyze this problem and explore novel techniques to support unconstrained, concurrent work in a class of groupware applications that are built atop application sharing technologies.

**Keywords:** Human-Computer Interaction, Application Sharing, Concurrency Control

### 1 INTRODUCTION

Application sharing systems allow end users to interact through familiar single-user applications without modifying source code. According to Grudin [1994], leveraging familiar single-user application for cooperative work often increases the productivity of groups and provides an alternative approach to groupware engineering that is more economic than developing specialized collaborative applications. As a result, a number of application sharing systems have been developed, e.g., XTV [Abdel-Wahab and Feit 1991], Flexible JAMM [Begole et al. 1999], MMConf [Crowley et al. 1990], and Dialogo [Lauwers et al. 1990].

Application sharing systems are typically used for real-time collaboration in which distributed users work on a common task at the same time. The interactive nature of these systems requires the effects of a user's action be seen by himself (lo-

cal response) and other users (remote response) in a timely manner. Local response time is determined by round-trip delay between the user terminal (or display server) and the application process, which is fundamentally limited by the speed of light. When a system runs over a wide-area network such as the Internet, the round-trip delay to the other side of the earth is at least 200ms, while interactive users typically expect a response time at the magnitude of 50-100ms [Bhola et al. 1998]. Therefore, centralized application sharing in general is not able to satisfy the responsiveness requirements of interactive cooperative work. Theoretically, a replicated architecture is more preferable: Each user works on his own copy of the application simultaneously for local response and a concurrency control protocol maintains consistency among the application replicas. Concurrency control thus becomes a critical problem.

Concurrency control protocols found in groupware systems, can be generally characterized as optimistic or pessimistic [Greenberg and Marwood 1994]. Optimistic protocols allow local operations to proceed immediately (without being blocked) for quick response and then seek methods to repair inconsistencies, while pessimistic protocols delay the response to local operations until consistency can be guaranteed. Although quick local response is always desirable in interactive groupware applications, optimistic concurrency control is not applicable in situations where inconsistencies cannot be easily repaired.

Specifically, in application sharing systems, the only *general* way for concurrency control protocols to affect the state of the shared application is to simulate external event sequences [Lauwers et al. 1990]. If an optimistic concurrency control protocol is used, for example, (optimistic) serialization, disastrous consequences may occur when execution has been found out of order but undo

of previous executions is not possible. Therefore, all application sharing systems to our knowledge resort to pessimistic protocols, e.g., floor control [Crowley et al. 1990], to ensure consistency instead of to repair inconsistencies. Floor control essentially implements application-level locking so that collaborators take turns to manipulate the shared application to prevent conflicts from happening [Begole et al. 1999] [Lauwers et al. 1990].

According to Hymes and Olson [1992], however, turn-taking protocols serialize group interaction and is hence counterproductive for intellectual activities such as group brainstorming. Disadvantages of serial interaction include, among others, (1) that there is limited time for each individual to contribute because only one person can input at a time, (2) that being forced to hold on to their contributions possibly prevents individuals from thinking actively, and (3) that groups tend to pursue fewer ideas because ideas are generated serially. Thus, to increase group productivity, tools designed to support intellectual activities should permit users to input simultaneously and freely at any time, in order to facilitate unconstrained and natural information flow.

Therefore, there is a discrepancy between the need for replicated application sharing systems that provide familiar user interfaces for cooperative work and potentially interactive responsiveness, and their incapability of supporting non-blocking, unconstrained, and simultaneous interaction. Unfortunately, this discrepancy has not been addressed in existing application sharing systems, to the best of our knowledge.

In this paper, we explore novel techniques to support unconstrained interaction in a class of application sharing systems. In particular, we are interested in the applicability of a specific optimistic concurrency control protocol, operational transformation (OT) [Sun and Ellis 1998] which are originally developed in specialized group editors for unconstrained cooperative editing of shared documents. We examine the applicability of OT in replicated application sharing and analyze problems that arise in this research in Section 2. Then we address these problems in Sections 3 and 4, contributing a novel operational transformation algorithm for concurrency control and some techniques for interference reduction in replicated application sharing. This is followed by a comparison to related previous work in Section 5. We conclude this paper and point out future research directions in Section 6.

## 2 APPLICABILITY OF OPERATIONAL TRANSFORMATION

Operational transformation has been widely used in real-time group editors for concurrency control [Sun and Ellis 1998]. The shared document and the editing process are replicated at each site. Local operations are always executed immediately. Remote operations that are concurrent to locally executed operations are **transformed** before execution. Each user can edit any part of the local document replica at any time as if a single-user editor is used. There are two important auxiliary data structures in addition to the document itself: a history buffer for logging operations that have been executed locally, and an operation queue for temporarily storing local and remote operations that have been received but not yet executed.

Specialized group editors [Sun and Ellis 1998] only need to run one process at each site which implements the graphical user interfaces (GUI), data structures, and the OT algorithm. The OT algorithm takes one operation from the queue at a time, transforms it against operations in the history buffer that are concurrent, executes the transformation result on the document, appends it to the history buffer, and then refreshes the screen with the updated document [Ellis and Gibbs 1989]. The number of concurrent operations is usually small and thus the transformation time is negligible. The time for intra-process communications between components is negligible as well. Therefore local response time is usually good enough and execution of remote operations will unlikely introduce screen flickers to disrupt local editing, if it is not too frequent.

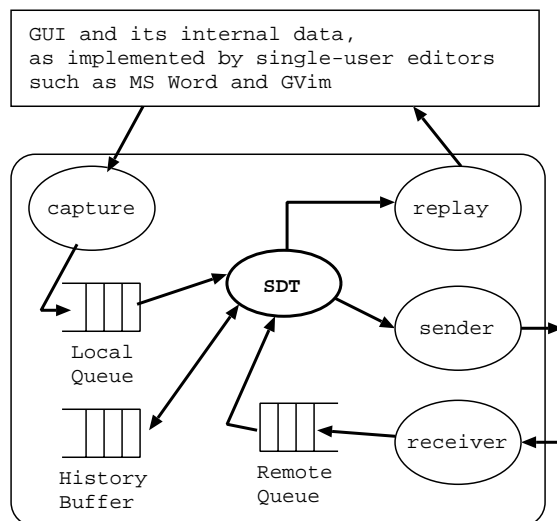
However, as noted in [Li and Li 2002], if we implement a group editor by replicated application sharing, the GUI part has to be a separate process implemented by a single-user editor such as MS Word. The application sharing mechanism runs an agent process at each site to coordinate the cooperating editors. The editor and agent processes each have their own data structures and computation logic. The editor maintains the actual replica of the shared document and the agent encodes a model of the shared document. In addition to synchronizing between replicas of the shared data model at different sites, as in specialized group editors, each agent must also synchronize its own replica of the shared data model and the local editor [Li and Li 2002].

According to previous work on application sharing [Abdel-Wahab and Feit 1991] [Lauwers et al. 1990], the only general way for the agent to learn about the local user’s editing operations is by intercepting his window-level input events. And the only general way for the agent to affect the editor’s internal state is to simulate user operations by replaying window messages to the editor. As in [Li and Li 2002], the agent works externally at each site to capture the user’s inputs to an editor, translate them into operations that are executed on the shared data model, and then propagate these operations to other sites. Upon receipt of remote operations, the agent executes them on the local model replica, translates them into equivalent message sequences, and then replays the messages to the local editor.

The separation between GUI and coordination processes has subtle impacts on system design: **First**, *local editing operations must go through the coordination process for consistency*, because the time between the generation of a local operation and its actual execution is no longer negligible.

Consider an implementation strategy that seems able to offer maximal local responsiveness: GUI operations are executed by the editor immediately before they are recognized and then executed on the data model by the agent. Due to the overheads in inter-process communication, there is a nonnegligible delay between the time when the operation ( $O_L$ ) is executed by the editor on its internal document replica and the time when this operation is intercepted ( $O'_L$ ) and executed by the coordination process on the shared data model. During this period of time, if a remote operation ( $O_R$ ) is executed and replayed, inconsistencies would be resulted: Operation  $O_L$  is performed by the user without knowledge of operation  $O_R$ , while  $O'_L$  is believed in the agent process to have been caused by  $O_R$  due to the definition of causality!

Therefore, to ensure consistency between the editor and the agent, the user inputs must go through the coordination process before they are executed by the the editor. Figure 1 shows a more reasonable design: Local user input events are first intercepted and blocked by the agent. A local event is replayed to the GUI directly if it does not perform significant operations that change the document state. When a significant operation is recognized, however, it is performed on the shared data model and then replayed to the editor after necessary transformations. To achieve



**Figure 1.** Architecture of the application sharing agent. SDT is the concurrency control component.

better local response as possible, local operations are given higher priority than remote operations.

**Second**, *replay of remote operations cause significant screen flickers*, especially when viewport switching is unavoidable.

As shown in Figure 1, a local editing operation causes two process switching to intercept the event and to replay the event. To execute a remote operation by event replaying takes one process switching. When a remote operation happens on the part of the document that is different from the part on which the local user is editing, the agent has to simulate a sequence of three events: switch to the target position, perform the remote operation, and then switch back to the original local position. If these two positions are not within the same viewport and this viewport switching happens frequently, execution of remote operations will generate screen flickers that will possibly disrupt the local editing.

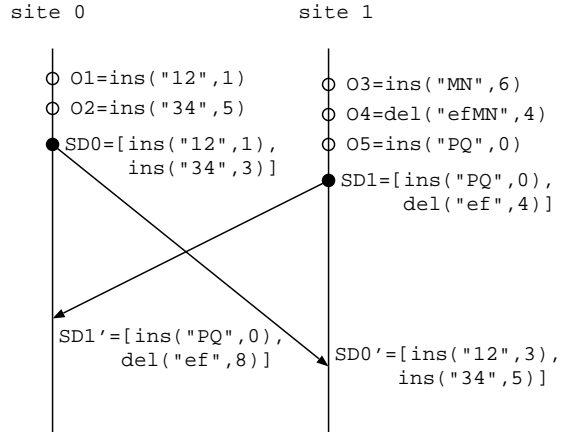
The above analysis motivates the following requirements for implementing OT in replicated application sharing systems: First, we must improve transformation efficiency as possible for local response. Secondly, we must increase the granularity of operation propagation and replay to reduce interferences caused by frequent event replay. Thirdly, we must seek special techniques to handle viewport switching, because it cannot be eliminated by merely reducing replay frequency. We address the first two requirements in Section 3 and the third in Section 4, respectively.

### 3 CONCURRENCY CONTROL

In this section we present a novel operational transformation algorithm called state difference transformation (SDT) [Li and Li 2003] that is more suitable for consistency maintenance in application sharing systems than state-of-the-art OT algorithms [Sun and Ellis 1998]. However, for the scope of this paper, we are not giving technical details but a conceptual description. In particular, we show that SDT is more applicable for the purpose of supporting unconstrained interaction with application sharing systems.

In the seminal dOPT algorithm by Ellis and Gibbs [1989], operations are propagated and transformed at the character level. The state-of-the-art GOTO algorithm by Sun et al. [1998] improves dOPT in several ways, among which it combines temporally *and* spatially consecutive, same-type, characterwise operations into stringwise operations. This reduces the number of transformations to some degree but the resultant stringwise operations are still rather fine-grained. For example, when modifying a document, a user may move the caret back and forth making a sequence of different types of modifications. Because of the differences in operation time and type, *spatially adjacent* operations may not be merged. Instead, GOTO processes operations according to the order in which they were generated.

By comparison, SDT propagates and transforms state differences of the shared document among cooperating sites. In accordance to the analysis of the preceding section, it improves local response time and reduces interference in the following two ways: **First**, spatially adjacent operations that are performed at overlapping or adjacent positions are combined, no matter whether they are of the same type or executed at consecutive times or not. State differences are calculated on the fly as each local operation is executed, and are propagated automatically at predefined intervals or manually by the user at desirable synchronization points. This increases the granularity of operations to be propagated and transformed. **Second**, state difference lists are ordered by the positions of the difference units. Then transformation of one list against another only needs to be performed between operations that are spatially dependent. This further eliminates transformations that are otherwise redundant in existing OT algorithms. As a result, SDT can significantly improve the efficiency of transformation.



**Figure 2.** Two sites edit the same document at the same time and propagate state differences to each other. Both sites start from initial document state “abcdefg” and converge at “PQa12bc34dg”.

#### 3.1 A simple example

In the following we give an example to illustrate the main ideas of SDT. Figure 2 shows a scenario of two sites collaborating to edit the same document at the same time. As a convention of distributed systems, vertical lines denote processes and the vertical directions represent time with later times lower than earlier ones. To reduce interferences, these sites do not propagate operations until a certain number of operations have been executed locally. Then state differences are calculated and propagated. Suppose the initial document state at both sites are “abcdefg”.

Note here by following the conventions of OT algorithms [Sun and Ellis 1998], a (text) document is represented by a linear string and we only consider two interesting operations:  $INS(S, P)$  for inserting a string  $S$  at position  $P$ , and  $DEL(S, P)$  for deleting a string  $S$  from position  $P$ . A state difference consists of a sequence of difference units that are ordered by their position. The position parameter of a state difference unit is based on the *original* document state.

First consider at site 0, two operations are executed in a row:  $O_1 = INS(“12”, 1)$  and  $O_2 = INS(“34”, 5)$ . The document state becomes “a12bc34defg”. After adjusting the position parameter of  $O_2$  to base it against the original document state, site 0 generates the state difference  $SD_0 = [INS(“12”, 1), INS(“34”, 3)]$  and then

propagates it to site 1.

At the same time, site 1 has executed the following sequence of operations:  $O_3 = INS("MN",6)$ ,  $O_4 = DEL("efMN",4)$ , and  $O_5 = INS("PQ",0)$ . The document state becomes "PQabcdg". Realizing that  $O_4$  effectively deletes the two characters "MN" that were inserted by  $O_3$ , site 1 replaces these two operations with one that deletes only characters "ef". After position reordering, site 1 generates the state difference  $SD_1 = [INS("PQ",0), DEL("ef",4)]$ , which is then propagated to site 0.

When  $SD_0$  is received at site 1, it is transformed against  $SD_1$  in the history buffer which has been executed locally and concurrent to  $SD_0$ . As a result, we get  $SD'_0 = [INS("12",3), INS("34",5)]$ . After  $SD'_0$  is executed, the document state of site 1 becomes "PQa12bc34dg".

When  $SD_1$  is received at site 0, we transform  $SD_1$  against  $SD_0$  for the same reason so that  $SD'_1 = [INS("PQ",0), DEL("ef",8)]$ . The document state of site 0 becomes "PQa12bc34dg" after the execution of  $SD'_1$ . At this point, all operations generated have been executed at all sites and their document states are consistent, although they were executed in different orders.

### 3.2 Implications on group interaction

SDT is a general algorithm that can be used in any group editing systems. Certainly it can also be used in a group editor that is built on application sharing technology, as has been motivated in this paper. In fact, SDT can demonstrate the following advantages in supporting unconstrained interaction with application sharing systems.

First, *SDT allows for unconstrained interaction and fast local response.*

For the purposes of concurrency control and consistency maintenance, there is no need for turn-taking as in traditional application sharing. Since most editing operations are characterwise, there is no contention for objects at the character level. Hence locking is generally not necessary. There is also no need to enforce a total order on operation execution, as in serialization approaches. Cooperating users are able to edit any part of the shared document at any time. After transformation, operations can be executed at any site in any order, given causality is preserved [Sun and Ellis 1998]. Therefore editing is not constrained and local response can be comparable to single-user editors when not shared.

Second, *SDT is more efficient than existing operational transformation algorithms.*

SDT improves the efficiency of operational transformation by increasing the transformation granularity and eliminating redundant transformations. On one hand, it is able to merge operations that have been executed at different times but whose positions are adjacent or overlapping. A state difference thus obtained accumulates the maximal net effects of a sequence of operations performed over a period of time. An extreme case is that, for example, a user repeatedly inserts a character say 'x' and then deletes it for multiple times. The state difference will be an empty list! By comparison, other transformation approaches are not able to merge operations this far. The increase of granularity in SDT not only reduces the number of transformations, but also saves network bandwidth and storage space in the history buffer. Because the synchronization points (to propagate local operations and to replay remote operations) are user-controllable, SDT does not necessarily result in loss of mutual awareness between collaborators.

On the other hand, a state difference in SDT is a list of string operations that are ordered by position. When one list is transformed against another, it is only necessary to transform one operation with those whose positions precedes that of itself. The process is intuitively similar to performing a two-way merge sort. Thus the time complexity is  $O(m+n)$ , as compared to  $O(m*n)$  in existing transformation approaches, given  $m$ ,  $n$  are the lengths of the two lists. Therefore a lot of redundant transformations can be saved.

Third, *SDT reduces synchronization time by replaying remote operations by position order.*

String operations in a state difference are ordered by position. Thus when a remote state difference is replayed to the local editor, multiple operations can be executed in a row without scrolling screen if their positions are within the same viewport. Due to the general locality of editing activities, this feature of SDT can potentially reduce the needs for frequent viewport switching and thus reduce the duration of event replay and interference. By comparison, existing OT approaches, when used directly for application sharing, remote operations must be replayed strictly following the *time* order in which they were generated at the original site. Frequent screen scrolling may be necessary if their execution and position orders are not consistent.

## 4 REDUCING INTERFERENCE

A viewport is the range of document content that is visible within the current editor window of a user. One advantage of replicated application sharing is that collaborators can work on different part of the shared document independently [Begole et al. 1999]. However, when local and remote users have different viewports, replay of a remote operation needs to switch to the target viewport by scrolling the local user’s screen and then switch back after the operation is completed. Certainly this will aggravate screen flickers and interferences.

However, if we can somehow replay events in a batch in the background and refresh the screen only after all changes have been made, we can ensure consistency and minimal interferences at the same time. In this section, we explore three special techniques to achieve this goal: mask window, display locking, and direct document access. These techniques are platform dependent but they can be used together with SDT to achieve better interaction effects. We have successfully prototyped these approaches on popular editors including GVim, MS Word and WordPad.

### 4.1 Mask window

Popular window systems including MS Windows and X Windows provide mask window mechanisms that can be used to hide updates to underlying windows from the user. A mask window usually does not have borders as do normal windows. The idea is to create a mask window on top of the editor window, whose sizes are exactly the same so that the mask window covers the text area of the editor that needs to be updated.

The mask window can be created from the beginning of group interaction. It is set to be transparent during normal local operation so that the underlying editor window updates are directly visible to the user. When replaying remote events, it is set to be non-transparent and simply displays an image of the editor text area to create an illusion to the local user. Meanwhile, the events are replayed to the original editor window that is hidden underneath. The mask window is set back as transparent when the event replay is completed and editor has been restored to its original viewport and editing state. Since the user is not able to witness the event replay process, interferences are often negligible.

During the time of event replay, the local user’s

editing operations (if any at all) are captured by the mask window and queued in the application sharing agent. As a possible disadvantage, the user may not be able to see the effects of his actions “immediately” before the replay finishes. However, as noted in [Bhola et al. 1998], human users can tolerate a response time of 50 – 100ms between an action is performed and the effect is visible on the screen. Therefore there is a space for further improvement: First, by using SDT we only need to replay the maximal net effects of an otherwise much longer sequence of remote operations. The fact that these operations are ordered by their positions can significantly reduce the need for screen scrolling. Second, the user may not perform editing operations or will not consider the delay as a problem at all, if he cognitively anticipated that synchronization may delay local response. This can be achieved by initiating the event replay process on demand, e.g., when the local user feels ready to synchronize with remote users and presses the button.

### 4.2 Display locking

Many platforms such as MS Windows provide display locking mechanisms so that the display of an application can be locked when the computation is going on. This provides another way to hold the display until event replay is completed so that the process is not exposed to the local user. Since we do not assume access to the source code, the mechanism must be able to work externally.

MS Windows provides software development kits (Win32 SDK) and application programming interfaces (Win32 APIs) so that applications have more control over their behavior. Win32 SDK has APIs to lock/unlock the window display. It may sound simple to issue a lock display call to the application and unlock back after the event replay is completed. The difficulty arises because of the security management feature in the OS. Most operating systems consider it a security threat that the default behavior of a process is modified by another process. In a nutshell, no application would be able to use API calls to lock the window display of another application. These calls have to come from within the application. But the application has already been written, is there a way to modify its behavior?

The answer to this question is both yes and no. We will not be able to modify the functionality of the application at the source level but would

be able to modify it at a higher level. We can inject a piece of code or a function into the address space of any application through system calls. This function would be responsible for invoking the lock window display and unlock window display selectively. But again the problem is to make the application call this function. To achieve this we would need an understanding of the operating system architecture. All windows applications are message driven and the messages to the applications are handled by a function, which is commonly known as the window procedure. In other words, the window procedure is the gateway or the interface between the operating system and the application. This procedure drives the behavior of the application. Win32 SDK also allows the application to choose its windows procedure as long as the window procedure resides in the address space of the application and conforms to a specific interface. By injecting a custom window procedure in the address space of the application replace the default window procedure, we can have our custom window procedure call the window locking and unlocking functions.

The custom window procedure should not in any way affect the original functionality of the editor which has much processing built inside the default window procedure. So before replacing the window procedure we store the address of the default window procedure. Any normal message will be piped through the custom window procedure to the default window procedure. Only the custom messages of locking and unlocking will be handled by the custom window procedure. As a result, the screen display is locked while the events are being replayed. After that we send another message to the application to unlock the display and refresh the editor screen to synchronize the document state with the screen state.

### 4.3 Direct document access

The third approach is to directly make changes to the target application document instead of using event replay for synchronization. Since we do not use the event replay mechanism, it does not really matter whether or not the remote and local users are on different viewports of the same document. The changes will be made directly to the document model of the target editor and we just have to refresh the target editor to reflect the updated state of its document model.

Most Windows applications provide access to

their document model through a COM/OLE interface. These interfaces are published and provide a standard way of accessing the document model without tweaking much into the source code. Inside these interfaces are methods to access the document structure at any granularity including paragraph, sentence, and word. We can programmatically manipulate this object model and can make changes to the document structure. The updates would be reflected automatically on the editor screen. This relieves us from the burden of event replay and screen scrolling. The approach thus minimizes interferences to a greater extent than the above two techniques.

The downside of this approach is that the COM/OLE interface is supported only by Microsoft editors and is still not standardized, much less implemented in other non-Microsoft applications. A typical example is GVim that only has an OLE interface, through which we can send commands. But it does not directly provide an interface through which we can programmatically access its document model. Fortunately, GVim is an open-source editor. To our rescue, through its application level interface, we can see that the editing area is a text area. According to window internals, all UI widgets (e.g., buttons and text areas) are treated as windows themselves. Text area is a special window whose object representation can be obtained by casting the window handle of this text area to a CTextArea object. This way we can also obtain indirect access to the document model of GVim.

### 4.4 A qualitative comparison

By comparison, mask window is the most general technique that can be implemented in many platforms. Display locking is more effective since it saves the costs of creating and maintaining a separate mask window, but it may only be achievable in few platforms like MS Windows. Direct document access is the most effective of all – in fact, it can make the experience of sharing single-user editors comparable to using specialized group editors – but it tends to be more platform and application dependent. These special hacks, although without modifying source code of the shared editors, undermine the generality of application sharing mechanism itself. Hence they should only be explored in application domains where unconstrained interaction is critical and the loss of generality can be paid off.

## 5 RELATED PREVIOUS WORK

Traditional optimistic concurrency control methods, e.g., (optimistic) locking and serialization, are also able to achieve high local response as operational transformation. However, they may frequently cause the loss of interaction results due to the failure to acquire a lock or the rollback of operations that have been executed out of order. So they are not suitable for many interactive groupware applications [Ellis and Gibbs 1989] [Greenberg and Marwood 1994].

Flexible JAMM [Begole et al. 1999] is able to share a class of Java applets by dynamically replacing its single-user components (e.g. text boxes) with multi-user versions. Operational transformation can be implemented in multi-user text components for concurrency control. However, these components themselves are essentially no different from specialized group editors. Thus it did not address the same problems.

The presented work reflects our recent progress on the Intelligent Collaboration Transparency (ICT) project. Our previous work [Li and Li 2002] within the same project focused on the heterogeneity and interoperation issues that arise in sharing familiar editors. It implemented a recent OT algorithm but did not address the problems as analyzed in Section 2.

## 6 CONCLUSIONS

The presented work is innovative and revolutionary. With traditional application sharing technologies, users are able to manipulate the shared application only by following a turn-taking protocol and often in a what-you-see-is-what-I-see manner. Supporting unconstrained interaction in our work significantly extends the flexibility of application sharing. This paper elaborates the idea in developing group editors by transparent sharing of familiar single-user editors without modifying source code. Allowing for familiar user interfaces and unconstrained interaction for cooperative work is potentially able to improve the productivity of groups and the acceptance of groupware [Grudin 1994] [Hymes and Olson 1992]. We plan to explore this idea further in other application domains in future work. We also plan to do usability and quantitative studies on the techniques presented in this paper.

## REFERENCES

- ABDEL-WAHAB, H. AND FEIT, M. 1991. XTV: A framework for sharing x window clients in remote synchronous collaboration. In *Proceedings of IEEE Tricomm '91* (Chapel Hill, NC, April 1991), pp. 159-167.
- BEGOLE, J. B., ROSSON, M. B., AND SHAFER, C. A. 1999. Flexible collaboration transparency: supporting worker independence in replicated application-sharing systems. *ACM Transactions on Computer-Human Interaction* 6, 2 (June), 95-132.
- BHOLA, S., BANAVAR, G., AND AHAMAD, M. 1998. Responsiveness and consistency tradeoffs in interactive groupware. In *Proceedings of ACM CSCW'98 Conference* (Seattle, Nov. 1998), pp. 79-88.
- CROWLEY, T., MILAZZO, P., BAKER, E., FORSDICK, H., AND TOMLINSON, R. 1990. MMConf: An infrastructure for building shared multimedia applications. In *Proceedings of ACM CSCW'90 Conference on Computer-Supported Cooperative Work* (Los Angeles, California, 1990), pp. 329-342.
- ELLIS, C. A. AND GIBBS, S. J. 1989. Concurrency control in groupware systems. In *ACM SIGMOD'89 proceedings* (Portland Oregon, 1989), pp. 399-407.
- GREENBERG, S. AND MARWOOD, D. 1994. Real-time groupware as a distributed system: Concurrency control and its effect on the interface. In *Proceedings of the ACM CSCW'94 Conference on Computer-Supported Cooperative Work* (Chapel Hill, NC, Oct. 1994), pp. 207-217.
- GRUDIN, J. 1994. Eight challenges for groupware developers. *Communications of the ACM* 37, 1, 92-105.
- HYMES, C. M. AND OLSON, G. M. 1992. Unblocking brainstorming through the use of simple group editor. In *Proceedings of the ACM CSCW'92 Conference on Computer-Supported Cooperative Work* (Nov. 1992), pp. 99-106.
- 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 ACM OIS'90 Conference on Organization Information Systems* (1990), pp. 249-260.
- LI, D. AND LI, R. 2002. Transparent sharing and interoperation of heterogeneous single-user applications. In *Proceedings of the ACM CSCW'02 Conference on Computer-Supported Cooperative Work* (Nov. 2002), pp. 246-255.
- LI, R. AND LI, D. 2003. A state difference based transformation approach to concurrency control in real-time group editors. Technical Report CSDL 2003-02-02 (Feb.), Center for the Studies of Digital Libraries and Department of Computer Science, Texas A&M University (20 pages).
- SUN, C. AND ELLIS, C. 1998. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proceedings of ACM CSCW'98 Conference* (Seattle, Washington, Dec. 1998), pp. 59-68.