GRIFFITH UNIVERSITY

Operation propagation in real-time group editors

Author

Sun, Chengzheng, Li, D., Zhou, L., Muntz, R.

Published

2000

Journal Title

IEEE Multimedia Magazine

DOI

10.1109/93.895155

Rights statement

© 2000 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

Downloaded from

http://hdl.handle.net/10072/3233

Link to published version

http://ieeexplore.ieee.org/Xplore/dynhome.jsp

Griffith Research Online

https://research-repository.griffith.edu.au

Operation Propagation in Real-Time Group Editors

Du Li, Limin Zhou, and Richard R. Muntz
University of California, Los Angeles

Chengzheng Sun
Griffith University, Australia

Operation propagation refers to the mechanisms and policies used in group editors for the participants to notify each other of their individual editing activities. The timing and granularity of propagation depend on different requirements such as network bandwidth, session scale, computational power of involved sites, and preferences of each user. Here, we draw on some common design principles in two group editors we have implemented.

Synchronous or real-time group editors let a group of users view and edit the same document simultaneously from geographically dispersed sites connected by communication networks such as the Internet. Traditionally, a text editor edits textual documents. A graphics editor or whiteboard draws graphic objects such as lines, boxes, and freehands. Document types edited by modern editors have recently extended to images, hypertext, hypermedia, 3D worlds, and so forth. This category of groupware contains not only useful tools for collaborative work, but also provides excellent vehicles for exploring a range of fundamental and challenging issues facing the designers of real-time groupware systems in general.

Similar to editing documents with single-user editors such as Microsoft Word, users in a collaborative setting expect the group editor to respond quickly. For example, after you insert or delete a character, you'd like to see it appear or disappear on the screen immediately or at least within a certain time constraint. Therefore, the state-of-the-art, real-time group editors typically adopt a replicated architecture in which an editor process runs at each site and the shared document is replicated at each site as well. You edit directly on your own document replicas with your own favorite editor processes. Meanwhile, you're notified of other users' operations. In case conflicting operations occur—for example, one user attempts to delete an object and another wants to modify it—we can use various strategies to resolve conflicts and keep all replicas consistent. While problems such as these have been widely addressed in the literature—such as Grove,[1] GroupDraw,[2] Licra,[3] GroupDesign,[4] Ensemble,[5] Jupitor,[6] Real-Time Distributed Unconstrained Cooperative Editing (Reduce),[7] and Grace[8]—we devote this article to the problem of operation propagation in real-time group editors.

Operation propagation—the mechanisms and policies used in notifying collaborators of each other's activities—is an important issue in group editors. Each operation typically has a finite duration. For example, when you draw a line with a graphics editor, you first click at the beginning point, drag the mouse around, then release it at the end point. This process often takes a few seconds and even longer to finish if, for example, you're not certain about the end point in the beginning. In addition, other users may have performed other drawing operations during this period. As you can see, working in collaborative settings often has different requirements. Among others, the most distinguishing requirement is mutual awareness. In a collaborative environment, users must be cognizant of the progress of other users and, conversely, must provide others the appropriate information on their own activities. Such information provides a more complete picture of what has happened, what is happening, and what might happen next. Users should then be able to decide, within this context, if their individual contributions relate to the group goal and progress.

In this article, we summarize the design principles in the two group editors we developed regarding operation propagation. One is the Collaborative Active whiteBoard (Cab),[9] a distributed whiteboard application developed at the University of California at Los Angeles (UCLA). The other is Reduce,[7] a collaborative text editor developed at Griffith University, Australia. Although these two group editors are different in a number of ways (for example, Cab is a graphics editor and Reduce is a text editor) and they were developed independently, the principles adopted to handle operation propagation are fundamentally similar.

Issues in operation propagation

To leverage collaboration, we must render local operations in detail for modern collaborative editing systems. For example, when you draw or

1070-986X/00/$10.00 © 2000 IEEE

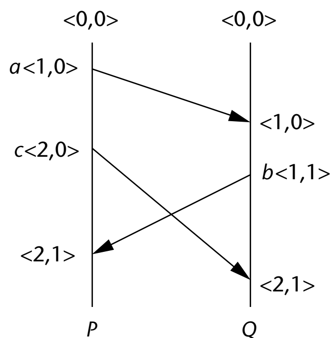


Figure 1. Vector time stamps in group editors.

move an object in a graphics editor, you can continuously see the partial drawing or moving track at the pixel level. Ideally, the whole process should be propagated to your collaborators at the same level of detail so that they can witness the whole process as clearly as you do.

However, this can't be achieved without a price. On one hand, fine-grained sharing penalizes the operation performer (as an information provider) and the communication networks to generate and deliver the notification messages because of the required CPU cycles and available network bandwidth. On the other hand, there's also a price to pay in rendering the received operations on the receiver side. For example, to add or delete a character on a text editor, all the characters following its position must be moved backward or forward. To create a new object (pixel, line, and so on) or modify an existing one on the current page of a graphics editor, the system usually has to refresh the whole screen.

Obviously, if the granule of propagation is too small or the frequency of propagation too high, users may have to work in an extremely dynamic and unstable work context because for example, the whiteboard screen may flicker due to frequent refreshing caused by rendering remote operations. This situation intensifies in editing sessions as the number of active participants grows. Since rendering remote operations takes a considerable fraction of the processor cycles, users can become frustrated because the system isn't as responsive—a locally inserted character may take a few hundred milliseconds (ms) or even longer to appear on screen.

A number of often-conflicting factors determine the timing and granularity of propagation. For example, to propagate a line in a graphics editor, users may configure the system to propagate every pixel as the line is being drawn, which may come at the price of responsiveness and context stability. Another approach might be to propagate only the end points after the drawing operation finishes, which may undermine awareness. Ideally, the following factors should be accounted for in making the design decisions:

- *Bandwidth of the communication networks*. Obviously, frequent and fine-grained notification consumes more bandwidth. Internet, home, and mobile users typically have less available bandwidth. Plus, different users in the same session may have diverse qualities of connectivity.

- *Scale of the editing session*. How many participants are contributing actively? The larger the number of users in a session means a higher frequency of remote operations.

- *Memory size and processor speed*. Group editors usually record the history of operations. As the system propagates more operations session-wide, users need more memory to record the session and CPUs need more processor cycles to process them. Lower end computers such as mobile devices and personal digital assistants (PDAs) typically can't record a very long history nor refresh the screen at a high speed.

- *Users' preferences, work style, and organizational structure*. Designers must keep the users' as well as the application needs in mind. This is part of the so-called collaboration or application semantics. Coarse-grained propagation usually suffices for many applications or is preferred by many users.

Operation causal ordering and vector time stamps

We first introduce the concepts of operation causal ordering and vector time stamps. Distributed systems use vector time stamps[10,11] to order events. For example, Figure 1 shows two editing processes, *P* and *Q*. Note that the vertical lines denote processes and the vertical directions represent time with later times lower than earlier ones. Each process maintains a counter that advances when users perform an operation locally. In this case, the vector time stamp consists of two elements: the first represents the counter for *P* and the second the counter for *Q*. This arrangement associates each operation with a time stamp. Let the initial state vectors of both processes be <0, 0>. When a user performs operation *a* at *P*, the system time-stamps it as <1, 0>. Note the system usually renders local operations immediately.

An atomic sequence (one that shouldn't be interleaved with other operations) of two steps occurs when process *Q* receives operation *a* that *P* propagated. First, the system had to render *a* on the screen of *Q*. For example, to render the operation "add a character *C* at position *k*," you have to move all characters inclusively after *k* one position

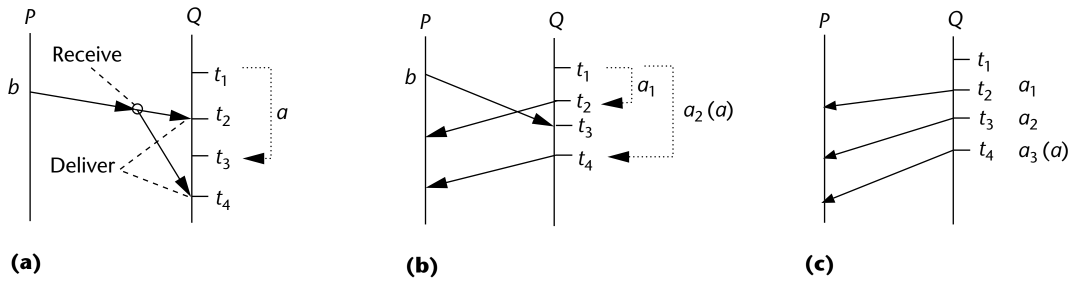


Figure 2. Propagation and delivery of partial results in Cab.

forward and then put *C* at *k*. To render the operation "draw a line from point *x* to *y*," you may have to refresh the screen to reflect this new operation. Second, the system used the time stamp of *a* to adjust the local vector of Q. For example in Figure 1, the state vector of process Q on delivery of *a* becomes <1, 0>. Note the first element of the state vector of *Q* is equal to the first element of the time stamp of *a* since the latter is larger. This larger element indicates *a*'s knowledge of more operations that have been delivered in *P*. The propagation and delivery of operations *b* and *c* are similar.

We say that an operation *u* causes another operation *v*, that is, $u \rightarrow v$, if one of the following occurs:

1. operation *u* is performed before *v* by the same user,
2. operation *v* is performed by a user after the delivery of *u*, another user performed, or
3. there exists an operation *z* such that $u \rightarrow z$ and $z \rightarrow v$.

If neither $u \rightarrow v$ nor $v \rightarrow u$, then we say *u* and *v* are concurrent, denoted by $u \parallel v$. For example in Figure 1, we can conclude that $a \rightarrow b$ and $b \parallel c$. A total order $\prec$ among all operations can be defined, which is consistent with their causal order. For example, in Figure 1 operation *a* totally precedes $b(a \prec b)$ since $a \rightarrow b$. The total order between concurrent operations *b* and *c* is usually decided by some predefined rule, for example, comparing the Internet protocol (IP) addresses of processes *P* and *Q*. We can use the total order between operations to determine the document state. In Figure 1, since $a \prec b$, the system will always render *a* before *b* in a graphics editor. So *b* may cover *a* completely or partly if they happen to overlap in space. In cases where two operations *u* and *v* don't overlap (directly or transitively), we say that they're *commutative*. Exchanging the total order of commutative operations doesn't affect document state. Note that we use *object* and *operation* interchangeably in this article.

Operation propagation in Cab

We developed Cab[9] to study the impact of event duration on the design of group graphics editors. Novel features in Cab include the ability to propagate partial results of each drawing operation as an evolving sequence of versions of the same object. Cab also lets users control the propagation and delivery of such partial results.

Figure 2a shows two drawing processes, *P* and Q. Q is drawing object *a* when *P* delivers object *b*. Since *a* begins at time t_1 without the knowledge of *b*, we can state $a \parallel b$. Depending on the strategies used to deliver *b*, however, a different causal relationship between *a* and *b* may result that can overturn this intuitive conclusion. The issue is when to deliver *b*, because the time stamp of *b* will adjust the logical time of process Q.

If we defer the delivery of *b* to time t_4, immediately after *a* is finished, then we can maintain the conclusion that $a \parallel b$. However, since the duration of *a* isn't predictable, many remote operations may queue up before it's finished. It may not be desirable if all those objects pop up abruptly and simultaneously on the screen.

As an alternative strategy, we might deliver *b* as soon as it's received, as shown in Figure 2b. Intuitively, what has already been performed on object *a* at the time of delivery of *b* has been done without the knowledge of *b*. However, we don't know if the original intention of *a* is affected after the delivery of *b*. For example, user Q might decide to abort *a* or draw it differently than what was originally intended. To reflect that the first part of *a* is concurrent to *b* and the rest of *a* might be causally related to *b*, we split *a* into two versions: a_1 lasts from time t_1 to t_2 and a_2 from t_1 to t_4. Note a_2 is actually the final version of object *a*. We can deliver *b* at time t_3 so that a_1 is propagated right before the delivery of *b*. The system then propagates version a_2 when object *a* is finished. We can conclude then that $a_1 \parallel b$ and $b \prec a_2$.

To generalize, consider that Q receives *n*

remote operations in period $[t_1, t_4]$, then object a is propagated in $n + 1$ versions, that is, $a_1, a_2, \ldots, a_{n+1}$, and $a_1 \rightarrow a_2 \ldots \rightarrow a_{n+1}$. If n is sufficiently big, other users should be able to see almost the whole process. In this case, the delivery of remote operations triggers the drawing process' propagation.

A complementary strategy is that each drawing process propagates the intermediate versions of its drawing operation voluntarily. As illustrated in Figure 2c, process Q samples drawing operation a into three versions, a_1, a_2, a_3, and propagates them to process P. The sampling can be done periodically, say every 100 ms. If the sampling period is small enough and the network latency is relatively constant, other users would be able to track the drawing process. As discussed earlier, we aim to seek a balance between mutual awareness and context stability. As sampling occurs more frequently, the system will generate more notification messages and refresh users' screens more frequently. In Cab, the users can regulate the sampling rate individually or as a group.

Note that only operations that generate a track can possibly generate partial results such as drawing a new object and dragging an existing object to modify its size or position. For example, user Q draws line a in Figure 2c. It begins at point (x_1, y_1) at time t_1. As it moves on, at time t_2 the partial version a_1 of line a is from (x_1, y_1) to (x_2, y_2). At t_3 the partial version a_2 is from (x_1, y_1) to (x_3, y_3), and so on. Drawing a line can easily lead to thousands of partial versions if it's sampled at a high rate. In contrast, we call operations that don't generate partial results *atomic* operations (such as delete and recolor).

We used the following rules to determine the propagation and delivery of operations in a Cab session:

- *Rule 1*: The system propagates an operation right after it's finished.
- *Rule 2*: Delivery of remote operations triggers the propagation of partial results of the local drawing operation in process (if there is one).
- *Rule 3*: The system samples each drawing operation every t ms and propagates the partial results.
- *Rule 4*: The timing and granularity of the propagation of local operations and delivery of remote operations are user and/or system adaptable.

Cab maintains a linear operation log at each site. The system inserts each delivered operation into the log according to a globally unique total order. Local operations are delivered immediately. A remote operation is delivered if causality is preserved. That is, the system will deliver a remote operation, say a, only if it has delivered all operations that might have caused a. Otherwise, the system must defer delivering a. However, if two operations a and b are concurrent, the system delivers a first if it arrives before b, even if $b \prec a$. When the system delivers a local or remote operation, it inserts the operation into the operation log according to the total order. If some operation b has been delivered and it's time to deliver $a(a \prec b)$, then the screen must be refreshed according to the updated operation log. To reduce memory costs, the system must handle the partial results of an operation differently from a finished operation. The system maintains only one entry in the operation log for each operation. Each subsequent version replaces the older one, both from the screen and the log. Obviously, delivery of operations has computational costs at each site. Each partial version takes a message to propagate and causes the receiver's screen to refresh once. If a line is propagated at the pixel level, thousands of messages will be multicast, which may refresh the screen thousands of times. The performance will degrade linearly if the number of active users increases.

Therefore, Cab lets users control the timing and granule of operation propagation and delivery. At the lowest level of control, Cab has two switches on its user interface. One decides whether to propagate partial results to other users before a nonatomic operation is finished. The other decides whether to deliver partial operations performed by other users. Users can choose their propagation and delivery preferences by toggling these two switches. For each pair of users, however, the actual propagation and delivery policy is limited to one of the four combinations.

At a higher level of control, you can use Cab together with the Collaborative Objects Coordination Architecture (COCA)[12] to achieve more flexibility. Actually, we designed and implemented Cab in the spirit of COCA. That is, we separated the mechanisms and policies. The policies can be easily specified in a powerful rule-based coordination language and enforced by the COCA runtime system, a copy of which runs at each site. For example, users can specify the following advanced policies in COCA without modifying Cab. They can control to whom and in what level of detail the partial results will be propagated. Considering fur-

ther that collaborating sites may vary in computational power and network connectivity, we can even divide the users into multiple groups. The system will propagate the more detailed partial results to the more capable groups. Users can also control the timing and granule of delivering remote operations. For example, users can choose to deliver remote operations immediately or in a deferred mode in which remote operations are accumulated and delivered at certain points—when the current local operation is finished, when the user isn't currently active, or periodically, say, every 5 seconds and longer if the application doesn't require fine-grain sharing. The delivery of partial results can be controlled in a similar manner. Since intermediate versions of an object aren't always relevant, users can choose to deliver them or not, when to deliver them, whose partial results they want to see in what level of detail, and so forth.

Operation propagation in Reduce

Reduce[7] first helped us identify and formalize the problem of user intention preservation in group editors. Most relevant to this article is that Reduce extended the original operational transformation algorithm[1] with stringwise operations, which can considerably improve the performance of real-time group text editors.

Rendering graphic objects in a total order that's the same at all sites generally suffices for a group graphics editor. A graphics editor is usually two-dimensional in that only the spatial relationships (overlapping and commutative) between objects count. However, state-of-the-art group text editors are usually linear in that they treat documents as textual strings. Therefore, delivering an operation in graphics editors is relatively simpler. It suffices to refresh the screen in Cab, for example, when the system delivers an operation. There's usually no need to transform the operation in group graphics editors while in real-time group text editors operational transformation is important to ensure responsiveness and correctness.

To achieve high responsiveness in a group text editor, the system delivers local operations immediately. Remote operations are delivered provided that causality is preserved. Concurrent operations may be delivered in any order, but they must be transformed against each other to preserve their intentions and to ensure convergence.[7] The essence of transformation is to adjust the parameters of operation a according to the effect of concurrent operation b that was executed before a so that the transformed a can have the right parameters to perform in the new document state. To support operational transformation, Reduce keeps an operation log (called the history buffer) to maintain all executed operations.

Local operations are normally character-based. They're performed at the local site immediately for achieving high responsiveness. How are local operations propagated? At one extreme, the system propagates every character-based operation once it's performed locally. At another extreme, users work independently and merge their results at some time point. Neither can strike a satisfactory balance between context stability and mutual awareness. Note also that each operation takes one entry in the operation log at each site. Propagation and deliver operations at the character level require more message passing between different sites, which adds more computational costs because of more frequent transformation. Therefore as a trade-off, Reduce lets users choose somewhere in the middle of these two extremes. In other words, the system allocates a buffer to each site to accumulate local operations. Then, it combines as many character-based operations into stringwise operations as is allowed by the factors discussed earlier in this article.

We used the following rules to determine when to convert a sequence of accumulated character-based operations into a single string-based operation and propagate it to remote sites:

- *Rule 1*: If a newly generated character-based operation and the accumulated operations can't be combined and expressed as a single string-based operation, then the accumulated operations will be converted into a single string-based operation. For example, suppose that after the execution of a sequence of N character-based local operations of the same type, such as Ins['1,' K], Ins['2,' $K+1$], …, Ins['N,' $K+N-1$], the $(N+1)$-th character-based local operation comes in with a different operation type (such as Delete). Or the character-based local operation comes with the same operation type but also with a position parameter that isn't in sequence such as Ins['?,' M], where $M \neq K+N$. If this happens, then the accumulated N character-based local operations will be grouped into a single string-based operation O_1 = Ins["12 … N," K] and multicast to remote sites, while the $(N+1)$-th character-based operation and forthcoming suitable character-based local operations will be grouped into another string-based operation such as O_2 = Ins["? …?," M], because a sin-

gle string-based operation can't express the combined effect of both O_1 and O_2.

- *Rule 2*: If a newly generated character-based operation and the accumulated operations can't share the same state vector time stamp, then the accumulated operations will be converted into a single string-based operation. For example, suppose that after the execution of a sequence of N character-based local operations of the same type such as Ins['1,' K], Ins['2,' $K + 1$], …, Ins['N,' $K + N - 1$], remote operation *RO* comes in and is executed. Then, even if the $(N + 1)$-th character-based local operation has the same type and its position parameter is in sequence, such as Ins['?,' $K + N$], the first N character-based local operations should be grouped into a single string-based operation O_1 = Ins["12 … N," K]. The system would multicast to remote sites while the $(N + 1)$-th character-based local operation and forthcoming suitable character-based local operations should be grouped into another string-based operation O_2 = Ins["? … ?," $K + N$], because *RO* is independent of O_1, but *RO* causally precedes O_2. This implies that O_1 and O_2 must be time stamped by different state vectors.

- *Rule 3*: If the accumulated operations have reached the system- or user-selected space and/or time limits, then the system will convert the accumulated operations into a single string-based operation. For example, this would happen if the buffer for accumulating character-based local operations is about to overflow, or if a special timer expires after the user interface has been idle for certain period of time.

- *Rule 4*: If the user has initiated a synchronization operation from the interface to flush whatever accumulated in the local buffer to all remote sites, then the system will convert the accumulated operations into a single string-based operation. Moreover, the detection of some special keyboard or mouse events (carriage return, backspace, and so on) by the user interface may also trigger flushing accumulated operations to remote sites. In addition, when the system propagates an accumulated string-based operation to remote sites, it saves the operation in the local history buffer.

Conclusions

Collaborative text editors and graphics editors do have differences. For example, in a text editor, every character input will become part of the document. However, in a graphics editor, partial results of an object are only intermediate versions that will eventually be replaced by the final version of that object. To achieve an integrative perspective, we can consider each character-based operation in a text editor as a partial result of a coarser grain string operation.

Despite various differences, we adopted the following policies in Cab and Reduce with regard to operation propagation:

- A number of accumulated operations are propagated if a new operation can't combine them. This corresponds to Rule 1 in the sections on Cab and Reduce, respectively. It's not necessary to combine a finished operation with a separate new operation in Cab.

- The delivery of a remote operation triggers the propagation of the partial results of the current local operation. Rule 2 of both sections actually expressed the same thing.

- The system propagates accumulated operations together when the buffer reaches the predefined space or time limit. Rule 3 of the Cab section propagates the partial results periodically. So it's comparable to Rule 3 of the Reduce section and this rule.

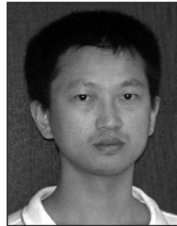
- Systems can explicitly propagate local operations on user commands. In Rule 4 of the Reduce section we said that accumulated operations can be flushed by a synchronization operation. Rule 4 of the Cab section suggests that operation propagation should be user and/or system adaptable.

Sun and Ellis[13] emphasized that group editors should build a sufficient amount of generic supporting mechanisms into the system while leaving the high-level collaboration policy decisions to the system users. In our case of operation propagation in real-time group editors, we'd like the operation performer to choose the timing and granule of propagation and the operation receiver to choose the timing and granule of delivery.

In this article, our focus is on operation propagation of DO operations. One related issue is the operation propagation of corresponding UNDO operations. We also plan to conduct a usability study of various propagation strategies from the end users' perspective. **MM**

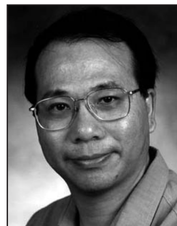
References

1. C.A. Ellis and S.J. Gibbs, “Concurrency Control in Groupware Systems,” *Proc. ACM Sigmod 89*, ACM Press, New York, 1989, pp. 399-407.
2. S. Greenberg, M. Roseman, and D. Webster, “Issues and Experiences Designing and Implementing Two Group Drawing Tools,” *Proc. 25th Ann. Hawaii Int’l Conf. on the System Sciences*, IEEE Press, Piscataway, N.J., Jan. 1992, pp. 139-250.
3. R. Kanawati, “LICRA: A Replicated-Data Management Algorithm for Distributed Synchronous Groupware Application,” *Parallel Computing*, Vol. 22, No. 13, 1997, pp. 1733-1746.
4. A. Karsenty, C. Tronche, and M. Beaudouin-Lafon, “GroupDesign: Shared Editing in a Heterogeneous Environment,” *Usenix J. Computing Systems*, Vol. 6, No. 2, 1993, pp. 67-195.
5. R.E. Newman-Wolfe, M.L. Webb, and M. Montes, “Implicit Locking in the Ensemble Concurrent Object-Oriented Graphics Editor,” *Proc. ACM Conf. Computer-Supported Cooperative Work*, ACM Press, New York, Nov. 1992, pp. 265-272.
6. D. Nichols et al., “High-Latency, Low-Bandwidth Windowing in the Jupiter Collaboration System,” *Proc. ACM Symp. User Interface Software and Technologies*, ACM Press, New York, Nov. 1995, pp. 111-120.
7. C. Sun et al., “Achieving Convergence, Causality-Preservation, and Intention-Preservation in Real-Time Cooperative Editing Systems,” *ACM Trans. Computer-Human Interaction*, Vol. 5, No. 1, Mar. 1998, pp. 63-108.
8. D. Chen and C. Sun, “A Distributed Algorithm for Graphic Objects Replication in Real-Time Group Editors,” *Proc. ACM Conf. Supporting Group Work*, ACM Press, New York, Nov. 1999, pp. 121-130.
9. D. Li, L. Zhou, and R.R. Muntz, “The Gods Must Be Crazy: A Matter of Time in Collaborative Systems,” *Proc. ACM Group 99 Workshop on Consistency Maintenance and Group Undo in Real-Time Group Editors* (*The First Int’l Workshop on Collaborative Editing Systems*), ACM Press, New York, Nov. 1999, pp. 1-4.
10. C. Fidge, “Logical Time in Distributed Computing Systems,” *Computer*, Vol. 24, No. 8, Aug. 1991, pp. 28-33.
11. F. Mattern, “Virtual Time and Global States in Distributed Systems,” *Proc. Int’l Workshop on Parallel and Distributed Algorithms*, North-Holland, Amsterdam, 1989, pp. 215-226.
12. D. Li and R.R. Muntz, “COCA: Collaborative Objects Coordination Architecture,” *Proc. ACM CSCW 98*, ACM Press, New York, Nov. 1998, pp. 179-188.
13. C. Sun and C. Ellis,” Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements,” *Proc. ACM CSCW 98*, ACM Press, New York, Dec. 1998, pp. 59-68.



Du Li is an assistant professor in the Computer Science Department, Texas A&M University. He received his PhD in computer science from the University of California at Los Angeles in June 2000. His research interests include collaborative systems, Internet and distributed computing, middleware, logic programming, programming languages, and database systems.

Readers may contact Li at the Department of Computer Science, Texas A&M University, College Station, TX 77843-3112, e-mail lidu@cs.tamu.edu.



Chengzheng Sun is currently a professor at the School of Computing and Information Technology in Griffith University, Australia. He obtained PhDs in computer science and engineering from University of Amsterdam, Netherlands, and from National University of Defense Technology, China, respectively. His major areas of expertise and research interests include Internet computing technologies and applications, groupware and CSCW, distributed operating systems and networks, mobile computing, and parallel implementation of object-oriented and logic programming languages.

Limin Zhou received an MS in computer networking from Peking University, China, in 1995. Now he is a PhD student at the University of California at Los Angeles. His research interests include CSCW, data mining, and networking.

Richard R. Muntz is a professor and chair of the Computer Science Department, School of Engineering and Applied Science, at the University of California at Los Angeles. His current research interests are sensor-rich environments, multimedia storage servers and database systems, distributed and parallel database systems, spatial and scientific database systems, data mining, and computer performance evaluation.

He received a BS in electrical engineering from Pratt Institute in 1963, an MS in electrical engineering from New York University in 1966, and a PhD in electrical engineering from Princeton University in 1969. He is a fellow of the ACM and of the IEEE.