# LEVERAGING JAVA APPLETS: TOWARD COLLABORATION TRANSPARENCY IN JAVA

JAMES BEGOLE, CRAIG A. STRUBLE, AND CLIFFORD A. SHAFFER • VIRGINIA TECH

THE JAVA ABSTRACT WINDOW TOOLKIT DOES NOT CURRENTLY SUPPORT THE COLLABORATIVE USE OF APPLICATIONS DEVELOPED FOR A SINGLE USER. MODIFICATIONS TO THE AWT WOULD PUT THIS CAPABILITY WITHIN REACH.

Widespread use of the Internet is giving rise to the need for collaborative applications that link users at remote sites. Many toolkits support the development of *collaboration-aware* applications—those developed specifically for cooperative work by multiple users. Another approach is *collaboration transparency*—the collaborative use of applications originally developed for a single user. When the runtime environment supports collaboration transparency, an application programmer need not write new code to make an application collaborative. Thus, collaboration transparency leverages the existing base of single-user applications by extending them to collaborative use.

In this article, we review options for providing collaboration transparency. We also discuss how collaboration transparency can be incorporated into Sun Microsystems' Java, the most widely used vehicle for developing interactive World Wide Web applications.

Our approach is to provide collaboration transparency through *event broadcasting*, in which a conference agent synchronizes independent instances of an application and broadcasts only user input events. This contrasts with the more commonly used method of *display broadcasting*, in which all users receive a graphical depiction of the shared application but not the application itself. However, display broadcasting violates the Java runtime model of client-side processing, while requiring the conference agent to distribute large amounts of graphics data to participants through the Internet. Event broadcasting, on the other hand, conforms to the Java runtime model and requires much less data to be distributed (because only user input events are broadcast).

We believe event broadcasting is a promising solution to providing collaboration transparency in Java. Unfortunately, the native platform implementations of the Java Abstract Window Toolkit do not currently provide support needed for event broadcasting, such as

1089-7801/97/$10.00 ©1997 IEEE

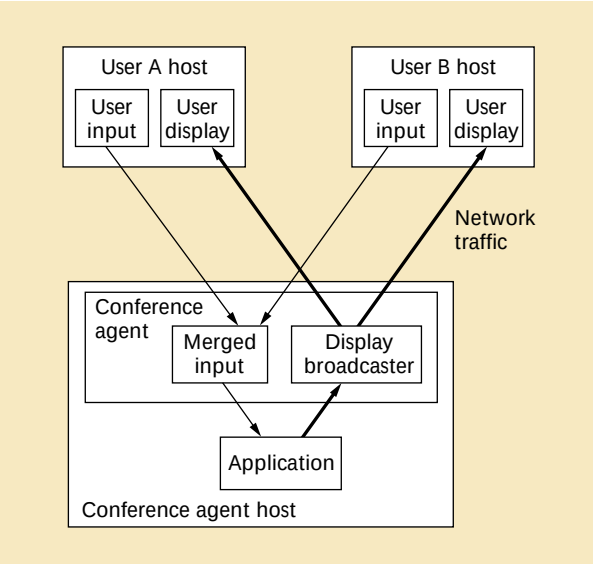


***Figure 1. Centralized application with display broadcasting. The thick arrows represent display data, illustrating how much more graphical data is broadcast relative to event data (regular arrows).***

event tagging and complete event propagation. The required modifications are impractical for any party other than JavaSoft, the implementers of the Java runtime environment. We are working with JavaSoft to include these modifications in future releases of the AWT.

## WHY COLLABORATION TRANSPARENCY?

Several Java-based toolkits are being developed to implement WWW-based collaboration-aware applications, including the National Center for Supercomputing Applications' Habanero,* the University of Erlangen-Nürnberg's Promondia,* and Old Dominion University's Java Collaborator Toolset.* However, although these collaboration toolkits provide a significant capability, they do not address the rapidly growing base of single-user Java applets available to Internet users.

Applications developed for a single user may be used collaboratively by modifying either the application or its runtime environment. After modification, multiple users may share the view and interact with the application. An environment that provides this application-sharing capability is called a collaboration transparency system because the shared single-user application is "unaware" that more than one user is interacting with it.[1]

To illustrate the usefulness of collaboration transparency, consider a simple single-user text editor shared among multiple users. All users simultaneously see the current screen view. If they add a standard chat or real-time audio channel, they can work together productively—even though the text editor has no built-in features to support such collaboration.

The examples of potential uses are endless. A simple chess interface could instantly become a two-player game if both players saw the same screen at the same time, and were able to move the pieces. Any drawing application becomes a shared whiteboard. An instructor may lead a class in a demonstration of any application with remote or local students. Research scientists may share dynamic visualizations of data using their precise analysis applications. Technical support operators can remotely control customers' applications while helping them with a problem.

## DISPLAY VS. EVENT BROADCASTING?

As we mentioned earlier, two collaboration transparency models are possible.

### Display broadcasting

Most existing collaboration transparency environments follow the model in Figure 1. A central conference agent receives all user input and serializes the events—that is, sends a single stream of events to the application.[1,2] The conference agent then distributes display updates to the participants' windowing systems. Display broadcasting is so named because the users receive only a graphical depiction of the screen state.

Examples of collaboration transparency systems that use this model are Hewlett-Packard's SharedX,* Sun Microsystems' ShowMe SharedApp,* Microsoft's NetMeeting,* Intel's ProShare,* and Farallon's Timbuktu.* The X Window System is particularly suited to this centralized architecture. X's distributed display and input capabilities make it natural to distribute a description of a single application instance's display to multiple users.

However, despite its popularity, display broadcasting may be inappropriate for Web-based collaboration transparency. First, it does not fit the model of the Java runtime environment. The X architecture separates display from computation and uses the model of a relatively "dumb" local display server supporting a remote client with potentially unlimited computational power. This contrasts sharply to Java, in which anyone loading a Java applet within a Web browser must have a new and separate instance of that applet. The philosophy is to distribute the complete applet, including both computation and display, to local machines as a unit. The local system is "smarter," and there is less network traffic.

Second, in display broadcasting, the conference agent must distribute large amounts of data to the participants, which may not be practical for widespread use on the Internet. In principle, display broadcast systems such as SharedX could be used to support collaboration transparency, but although the performance of sharing X dis-

plays is acceptable on a LAN, it is not on a WAN. According to Randy Smith, analysis of network traffic for Kansas,* an X-based shared environment, found that the ratio of graphics to events was nearly 10:1.[3] From this, we conclude that network traffic will dramatically decrease if only events are shared.

### Event broadcasting

In event broadcasting, multiple users share a single *virtual* instance of the application, although each user actually has a separate copy, as Figure 2 shows.[4] This contrasts to display broadcasting, in which users share a display of a single application instance.

Thus, one component of the conference agent is an event broadcaster whose job is to multiplex an input event from any participating collaborator to every other collaborator. This lets every collaborator invoke events to the system. All collaborators see the invoked events along with the feedback the events create. For example, if one collaborator moves the scrollbar for a text component, all participants see their corresponding copy of the scrollbar move in their copy of the same text component.

This approach is not without potential problems. As we have described it, event broadcasting is a replicated architecture because an application instance is *copied* to multiple users. Such architectures have not been considered suitable vehicles for achieving collaboration transparency. Wladimir Minenko, for example,[5] cites several reasons for not using it. We believe only one of his objections is serious in the context of Java, and we believe even this one is manageable.

- *Sharing must be based on the assumption that application behavior is deterministic.* Applications are, in fact, inherently deterministic. Applications that seem nondeterministic by behaving differently from one session to another are receiving input from sources other than the users' input events. Interacting with the runtime environment for information such as current time, random number seeds, files, and sockets, can lead to seemingly nondeterministic behavior. However, if all input to an application is replicated to each copy of the application, the states of all copies should be consistent.
- *The application environments of all involved hosts must be identical.* Aside from inputs, the Java Virtual Machine is intended to make this so. We need ensure only that each copy of the application receives the same response for the same system call (such as random numbers) and that all applications share input/output resources, such as files and sockets. This implies that the conference agent must control such resources and distribute input and output to all participants.
- *Copies of the shared application must be available on each host.* With Java, all participants can easily access a copy of the application software.

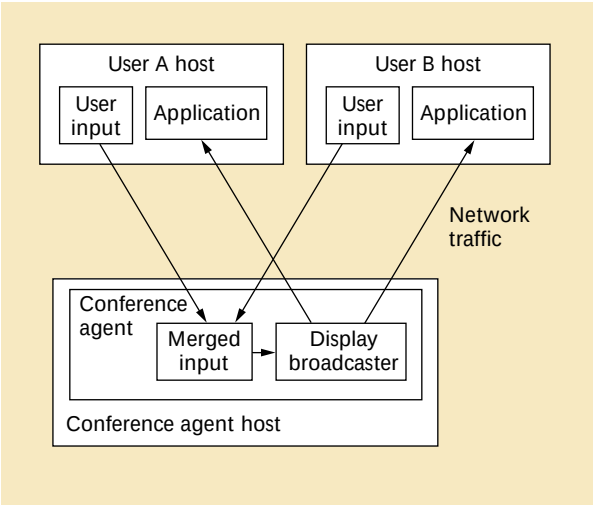


***Figure 2. Replicated applications with event broadcasting.***

- *Accommodating latecomers into an existing collaborative session is nearly impossible.* The reasoning is that the state of shared applications must be updated by playing back the input events of the whole session—a process that is unstable and may continue for a long time. However, by using an object persistence mechanism, such as Java's Object Serialization, latecomers can be provided with a copy of a shared application in its current state.
- *It is difficult to maintain consistency among multiple application copies.* For Java, this is the one serious disadvantage Minenko noted. The loss of input to one instance of the application (for example, because of network loss) may throw it out of synchronization with the others. Again, consistent input to all instances should yield a consistent application state. Ensuring that all copies actually receive the broadcast input stream may prove the greatest challenge to event broadcasting.

Event broadcasting may not be suitable for some types of applications, such as large-scale simulations or ones that use random number generators heavily. The best architecture in this case may be to combine one instance of the computational part of the simulation with a shared front-end interface. In a highly networked world, this separation of computation and interface threads is likely to become the model of choice for many large applications, regardless of collaboration issues.

**URLs from these pages**

**Habanero • www.ncsa.uiuc.edu/SDG/Software/Habanero/**
**Promondia • www4.informatik.uni-erlangen.de/Projects/promondia/**
**JCT • www.cs.odu.edu/~kvande/Projects/Collaborator/**
**SharedX • www.hp.com/xwindow/features/sharedx.html**
**SharedApp • www.sun.com/desktop/products/Multimedia/VCCT.html**
**NetMeeting • www.microsoft.com/netmeeting/**
**ProShare • www.intel.com/proshare/**
**Timbuktu • www.farallon.com/product/tb2/tb2winover.html**
**Kansas • www.sunlabs.com/research/distancelearning/kansas.html**

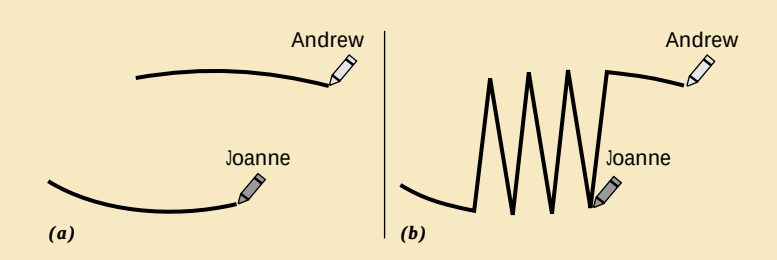


***Figure 3. What happens when two users attempt to draw separate freehand curves simultaneously by dragging their mouse cursors: (a) desired output and (b) actual output caused by a mouse-drag event collision.***

## IMPLEMENTATION ISSUES

To make event broadcasting a feasible way to provide collaboration transparency in Java, we had to overcome two main difficulties.

### Event collision

Collaboration transparency must maintain the intended behavior of the shared application. If events are naively broadcast between collaborators, event streams for nonatomic events such as mouse drags become confused, and can cause conflicts among collaborators.

In Figure 3, for example, Andrew and Joanne are sharing a drawing application, and both attempt to draw separate freehand curves simultaneously by dragging their mouse cursors. However, the location of the previous mouse-drag event for Andrew is different from Joanne's. When the next mouse-drag events are broadcast, they conflict because the event for Joanne uses the context from the event for Andrew. Figure 3a shows what was intended. Figure 3b shows the desired output corrupted by the mouse-drag event collision. In Figure 3b, the application has drawn a line between the alternating mouse positions of both Joanne and Andrew, instead of between the sequential positions of Joanne's mouse, followed by sequential positions of Andrew's mouse. This is an unexpected result for most users.

One approach to dealing with event collision and related problems is to use an explicit floor control model.[4] In the collaborative drawing example, a possible floor control model is *pen passing*: Only one collaborator, say Andrew, can manipulate the application while all other collaborators watch. The collaborators explicitly pass the pen from one to another as each uses the application. Because only one collaborator is in control, only one event stream must be handled, and event collision and related problems no longer exist.

The drawback to explicit floor control is that it requires user intervention to pass control during application use. We generally prefer implicit floor control models, in which the application sharing system automatically passes the pen between collaborators as they use the application.[6] Collaborators might compete more frequently for pen control in this scenario, but we believe that most users can and will negotiate potential conflicts from collaboration transparency to get the most benefit from the program's capabilities.

To implement implicit floor control, nonatomic events must be tagged with appropriate context information, including their originating location. Implementers can then atomicize the entire series of events associated with the drag activity by buffering the drag events, sending them as one atomic action (beginning with the initial mouse down and ending with a mouse up) to remote applications. Two approaches are possible:

- The local application can buffer the series of mouse drags.
- The event broadcaster can buffer (or block) events from other participants until the drag series is complete.

In the second approach, the source of each event must be available to the event broadcaster to perform filtering. This approach may be the more suitable when the event broadcaster is also serializing the collaborators' events. In event serialization, all collaborators receive the same inputs in the same order, thus ensuring consistency among application copies.[7]

Context information provides several other benefits. When a source field is included, the event broadcaster can filter events by user, host, or domain, for example. Such filtering addresses security concerns by allowing events from authorized clients only. Source information is also needed to implement telepointers (representations of remote collaborators' cursors) and other collaborative widgets.

### Collaborator awareness

Another difficulty for transparent collaboration is a collaborator's need to be aware of the actions and locations of all other collaborators. Implementers often use interface techniques such as telepointers and radar views to provide such workspace awareness.[8]

Modern GUI elements are normally expected to deliver feedback upon user input (such as when a button is activated), signaling that the input has been received. It is equally important to provide this graphical feedback to remote collaborators. Without such notice, collaborators may be surprised by sudden changes in the application. As a minimum, users should see GUI element reactions to each collaborator's inputs to the system. For example, if collaborator *A* clicks on and drags the thumb button of a scroll bar, the other collaborators should see *A*'s telepointer move to the scroll bar, and then see the scroll bar itself moving on their displays.

## PROTOTYPE IMPLEMENTATION

With these implementation issues in mind, we set four goals for providing collaboration transparency in Java.

- Any modifications to Java used to support collaboration transparency should come in the form of a class library that replaces the standard Java Core API class library without loss of functionality.
- A collaboration transparency library should not restrict code written by application developers. Application developers should be able to write code without concerning themselves with collaboration details. As a corollary, all Java applets supported within the existing environment should continue to be supported when used collaboratively. (If a choice must be made, some collaboration features should be unsupported within certain applications, rather than limiting which applications can run within the collaborative environment.)
- A collaboration transparency library should keep up with the current releases of the Java runtime system.
- A collaboration transparency library should work with anyone's properly implemented Java runtime system.

With these goals in mind, we conducted two experiments. We first wrote a simple applet, *EventClient*, whose event handler method (`handleEvent()`) would forward any incoming (local) events to the event broadcaster. Figure 4 shows the key code.

The applet has a running thread that receives remote events from the event broadcaster, and then introduces them to the applet using the applet's event posting method (`postEvent()`). In this way, we could see what events were actually sent from the native platform's windowing toolkit to the applet. We could also see whether events posted to the applet were propagated to and acted on by native user interface elements. The test applet included simple user interface elements, such as buttons and text fields.

In the second prototype, we modified the `postEvent()` method in the AWT class library so that the system prints a message when an event is posted. Figure 5 shows the code for this. All local events were sent to the event broadcaster, and then processed locally. The event broadcaster then sent events to all participants other than the originator. Again, the applet has a running thread to receive and introduce remote events into the applet. In

```
public class EventClient extends Applet implements Runnable {
   EventInputStream in;     // Input and Output Streams
   EventOutputStream out;
   Socket socket = null;

   public void init()
   { // Create input and output streams from socket }

   public void start()
   { // start thread to read events }

   public boolean handleEvent(Event event) {
      if (!(event instanceof RemoteEvent)) {
         try {    // Broadcast Event
            out.writeEvent(event);
         } catch (IOException ecp)
         { return false; }
      }
      return super.handleEvent(event);
   }

   public void run() {
      while (true) {
         try {    // Read Remote Event
            Event e = in.readEvent();
            ((Component)e.target).postEvent(e);
         } catch(IOException ecp)
          { continue; }
}}}
```

***Figure 4. EventClient tests event propagation to and from the native environment and the Java applet levels.***

```
// in java.awt.Component
public boolean postEvent(Event e) {
   ComponentPeer peer = this.peer;

   // Find out which events are getting posted
   System.out.println("Target: " + this);
   System.out.println("Event: " + e);

   // Send the event
   if (!(event instanceof RemoteEvent)) {
      try {
         out.writeEvent(event);
      } catch (IOException ecp)
      { return false; }
   }

   // convert the event to a remote event so that
   // it is not rebroadcast
   e = new RemoteEvent(e);

   // rest is as before

   if (handleEvent(e))
      return true;

   // . . . . . .
}
```

***Figure 5. Component.postEvent(Event e) is modified to intercept events and broadcast them to participants.***

experiments with this approach, we discovered that some events were not sent to the applet. This indicated that the event objects we instantiated were not acted on by the AWT objects to which they were posted.

We conducted these experiments using JavaSoft's Java Development Kit version 1.0.2 on a Sun SparcStation 5 and a 120-MHz Pentium-based PC running Windows 95. Although version 1.1, recently released, drastically changes the event model at the Java API level, the problems we discovered using version 1.0.2 remain in the new version. Hence the limitations we describe next will remain for the foreseeable future.

## Java AWT limitations

When we first considered modifying the Java class libraries to make all Java applications collaborative, we assumed that the standard buttons, text fields, and other interface components would pass all events to the event-handling system. We also expected the GUI elements to respond to remotely generated events as if they were generated by a local user. Most events can be passed and processed between two instances of an application, but not all events are handled properly. This is particularly true when dealing with interface components implemented through a platform's native interface library.

The Java AWT binds GUI components, such as buttons, to *peers*. Peers are platform-specific implementations of these elements. In this way, the underlying system environment is responsible for the display and responsiveness of interface elements, such as buttons, text fields, and scroll bars. This provides the advantage of giving users the "look and feel" of GUI components on their native platform. For example, Macintosh buttons look and act like Macintosh buttons. The AWT relies on the local system to implement the code for displaying these GUI elements and reacting to user actions. Thus, the AWT is much smaller than it would be if it had implemented all the interface elements.

Unfortunately, the AWT removes some necessary information and control required to provide collaborators with information about the actions of group members. This limitation takes several forms.

***Event inconsistency.*** The AWT is inconsistent in that some GUI elements send all information about all events that occur within them, while others send only some events or only limited information.

For example, mouse events occurring within a button are not sent to the applet. Only a high-level action event is sent when a user clicks and releases a button. Mouse events are never received by the application; instead the native implementation of the button component handles them directly. Furthermore, important information about the local mouse position is lost once the cursor moves within a button. When an action event is generated, the location of the mouse within the button is always reported as (0, 0), no matter where the button was actually pressed.

***No remote events.*** A more important problem is that some or all generated events posted to GUI elements are ignored—for example, keystrokes generated by remote events and sent to a TextField component. This occurs because the AWT cannot instantiate an event with a valid `pData` field. `pData` is a private field in the `Event` class that contains a platform-dependent representation of an event and is used to reintroduce events into the platform-dependent toolkit.

When the native implementation of the Java AWT receives an event with an invalid `pData` field, it simply discards it, and the peer never receives the event. Thus, the peer cannot provide feedback for events received remotely, because a valid `pData` field cannot be created. Jan Newmarch describes this limitation in more detail.[9]

***Selected graphical response.*** In event broadcasting (as we consider it here), when a broadcast event is received, it is *posted* to the corresponding GUI element in each local version of the applet. If the developer wrote the element to handle some type of event, it will execute the method associated with that event. However, with the Java Development Kit 1.0.2 and 1.1 implementations of the AWT, the element does not react to the posted event graphically. This is because an event is not passed on to the platform-dependent GUI peer when the event is posted to the Java GUI object. That is, the actual screen element is not notified of the event and so the system does not display the graphical reaction of that element to such an event.

Additionally, the AWT provides no mechanism that lets a developer force an element to display input responses, such as button clicks. This limitation means that implementers cannot provide collaborator awareness to the extent we described earlier.

***Limited cursors.*** The AWT now provides only a small set of predefined cursors, and there is no ability to define new cursor shapes or colors. This limitation affects the ability to create reasonable cursor shapes for telepointers.

Typical collaborative environments give each collaborator's telepointer a color unique to that user. To support What You See Is What I See[10] or the more relaxed What You See Is What I Think You See[11] goals for collaborative software, a user's cursor should be the same shape and color locally as well as remotely.

***Drawing inconsistencies.*** The AWT model is inconsistent because it lets the developer draw within some of its components, but not all. The native operating system displays the system-dependent GUI elements, such as Button, ScrollBar, Label, TextField, and TextArea. Consequently, in the Windows 95 version of JDK 1.0.2

the AWT cannot change the on-screen appearance of these elements. The operating system does not draw GUI elements derived from the Canvas class, however, so developers are free to draw on them as they please. Fortunately, JavaSoft appears to have addressed this problem, since one can draw over GUI elements in the Windows 95 and Solaris versions of JDK 1.1.

This inconsistency made it difficult to implement some collaborative GUI elements, such as telepointers. For example, once a telepointer moves from a canvas into a button, it disappears, since it cannot be drawn over the button. A local collaborator watching the remote user's telepointer may mistakenly think that the remote collaborator is no longer participating.

### Proposed solutions

We had hoped to modify only a relatively small portion of the Java AWT to implement collaboration transparency via event broadcasting. However, we found that the Java AWT as it exists now would require that we rewrite all platform-dependent component peers in Java. We did not want to do this for the same reasons that peers were introduced originally:

- It would further slow the already relatively slow performance of Java programs.
- It would significantly increase the size of the class library.
- GUI components would lose their platform-specific look and feel.
- Any party other than JavaSoft attempting this would be caught in a never-ending version chase of the Java AWT to meet the collaboration transparency library's goals described earlier (keeping up with current Java runtime system releases and working with anyone's properly implemented Java runtime system).

Instead of rewriting all the platform-dependent component peers in Java to implement event broadcasting, we propose the following modifications to the standard Java AWT.

***Propagate events.*** To exploit Java's use of native GUI elements and implement collaboration transparency, all input events must be available at the Java environment level, regardless of where they occur and regardless of whether they involve a component implemented in Java or a component implemented in the native graphics toolkit. For example, mouse events that occur in a button should be delivered to the application.

Additionally, when an event is instantiated by an application and posted to an AWT component, it should fall through to the native platform's windowing toolkit so that the system will react appropriately by displaying graphical feedback to the input event. To achieve that in the current AWT, the `Event.pData` field must be assigned a valid native platform event structure when an `Event` object is constructed.

***Tag events.*** The `Event` class should include information about the source of the event, in particular, which instance of an application generated it. This will allow a transparently collaborative environment to buffer and serialize user inputs to avoid event collision, as we described earlier. Although it is simple to add such information by extending the `Event` class in version 1.0.2, the new event class hierarchy in version 1.1 requires this information to be included in the standard `AWTEvent` class.

***Allow cursor modifications.*** The Java API would benefit from improved support for modifying the displayed mouse cursor. This would allow the cursor on the collaborator's screen to match the color and shape of any telepointer on other screens. The AWT should provide some means to set a cursor bitmap and mask. Such a mechanism could exist within the Cursor class of version 1.1.

## CONCLUDING REMARKS

Even with the current status of the AWT, some level of collaboration transparency is possible. Our preliminary implementation results are available at http://simon.cs.vt.edu/JAMM. JavaSoft is responding to many of the issues we identified.

Collaboration transparency through event broadcasting, as presented here, leaves open some research areas that bear exploration before it may be applied to all applications. In addition to distributing user inputs, we must distribute other inputs such as files, network connections, random number seeds, and current time, to maintain consistency among the copies of the shared application. Additionally, we believe it is important to support latecomers to a shared application session. This enhancement can be implemented by employing *application migration*, copying the current state of the application and migrating that state to the latecomer's host.

We believe these issues are worth resolving. Collaboration transparency will open up new application areas in education, scientific research, and business as these domains continue to stress teamwork and remote computing. The ability to share applications supports the growing shift from personal computing to *inter*personal computing. ■

## ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under grant REC-9554206, and by Sun Microsystems. We thank Randy Smith for his suggestions and review of an earlier draft of this article. We also thank Amy Fowler of the JavaSoft AWT group for her comments.

## REFERENCES

1. J. Lauwers and K. Lantz, "Collaboration Awareness in Support of Collaboration Transparency: Requirements for the Next Generation of Shared-Window Systems," *Proc. Conf. Human Factors in Computing Systems*, ACM Press, New York, 1990, pp. 303-311.

2 O. Jones, "Multidisplay Software in X: A Survey of Architectures," *The X Resource*, Spring 1993, pp. 97-113.
3. R.B. Smith, "Kansas: A Large, Flat, Multiuser Virtual World for Interactive Simulations," *Virginia Tech Computer Science Colloquium Series*, Apr. 1996.
4. J. Lauwers et al., "Replicated Architectures for Shared Window Systems: A Critique," *Proc. Conf. Office Information Systems*, ACM Press, New York, 1990, pp. 249-260.
5. W. Minenko, "The Application Sharing Technology," *The X Advisor*, June 1995; http://www.unx.com/DD/advisor/.
6. T. Crowley et al., "MMConf: An Infrastructure for Building Shared Multimedia Applications," *Proc. Computer-Supported Cooperative Work*, ACM Press, New York, 1990, pp. 329-342.
7. S. Greenberg and D. Marwood, "Real Time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface," *Proc. Conf. Computer-Supported Cooperative Work*, ACM Press, New York, 1994, pp. 207-217.
8. C. Gutwin, S. Greenberg, and M. Roseman, "A Usability Study of Awareness Widgets in a Shared Workspace Groupware System," *Proc. Conf. Computer-Supported Cooperative Work*, ACM Press, New York, 1996, pp. 258-67.
9. J. Newmarch, "The Java AWT: New Event Model," http://pandonia.canberra.edu.au:80/java/xadvisor/eventmodel/eventmodel.html.
10. M. Stefik et al., "WYSIWIS revised: Early Experiences With Multiuser Interfaces," *Proc. Conf. Computer-Supported Collaborative Work*, ACM Press, New York, 1986, pp. 276-290.
11. R.B. Smith, "What You See Is What I Think You See," *SIGCUE Outlook*, Vol. 21, No. 3, 1992, pp. 18-23.

**JAMES "BO" BEGOLE** is a PhD candidate in computer science at Virginia Tech. His research interests include collaborative computing and interactive visualization. Begole received a BS in mathematics from Virginia Commonwealth University and an MS in computer science from Virginia Tech. He is a member of the ACM.

**CRAIG A. STRUBLE** is a PhD candidate in computer science at Virginia Tech. His interests include symbolic computation, programming languages, and collaborative software. Struble received a BS in mathematics and a BS and an MS in computer science—all from Virginia Tech. He is a member of the ACM.

**CLIFFORD A. SHAFFER** is an associate professor of computer science at Virginia Tech. His research interests are in computer-aided education, computer-supported cooperative work, spatial data structures, and data visualization. Shaffer received a PhD in computer science from the University of Maryland at College Park. He is a member of the IEEE Computer Society, ACM, Sigma Xi, and American Association for Advancement of Science.

Readers may contact Shaffer at Dept. of Computer Science, Virginia Tech, Blacksburg, VA 24016; shaffer@cs.vt.edu.