Collaboration Transparency in Java through Event Broadcasting

James "Bo" Begole Craig A. Struble
Clifford A. Shaffer
Department of Computer Science
Virginia Tech
Blacksburg, VA 24016
{begolej, cstruble, shaffer}@cs.vt.edu

February 20, 1997

Abstract

Widespread use of the Internet for education and research is yielding many opportunities for network-based synchronous collaboration. For example, the Java runtime environment provides a platform independent vehicle for new collaborative applications. While many toolkits are becoming available that support development of collaborative applications, they do not enable collaborative use of existing single-user Java applets, a process called **collaboration transparency**. This paper discusses two approaches to collaboration transparency: display broadcasting and event broadcasting. We then consider the suitability of each within the context of the Java runtime environment. Unfortunately, simple modifications to the standard Java class libraries as they are currently implemented are not sufficient to support collaboration transparency. We describe the problems and suggest solutions.

1 Introduction

The advent of new World Wide Web (WWW) interactive runtime environments provides the opportunity for network-based synchronous collaboration. In particular, the Java programming language and its Application Programming Interface (API) libraries provide a platform-independent environment for creating interactive applications and applets that may be delivered over the World Wide Web[8, 5]. Many educators and researchers have a need for collaborative applications that permit users to work together from remote sites.

Collaborative applications are either **collaboration aware** or **collaboration unaware**. Collaboration aware applications are developed with the aim of allowing multiple users to work cooperatively. Collaboration unaware applications were originally developed for a single user, but may be made collaborative by modifying the application or its runtime environment. After modification, multiple users may share the view and interact with the application. This is called **collaboration transparency**, since the application does not see that more than one user is interacting with it. [15]. When the runtime environment supports collaboration transparency, an application programmer need not write new code to make an application collaborative. Examples of systems that provide this capability are SharedX [7], Sun Microsystems' SharedApp [25], and XTV [2].

A number of toolkits are being developed, based on Java, for implementing collaboration aware applications [6, 19, 26]. By definition, an application programmer must explicitly "build in" collaboration through calls to the toolkit support functions. While providing a significant capability, collaboration toolkits do not address the rapidly growing base of single-user Java applets available to Internet users. Many applications, while designed for single-use, could prove quite useful in a collaborative setting. For example, consider a simple single-user text editor shared among multiple users, with all users simultaneously seeing the current screen-view. In conjunction with a standard chat or real-time audio channel, multiple users could productively work together even though the text editor has no built-in features to support collaboration. A simple chess game interface could instantly become a two-player game if both players saw the same screen at the same time, and both were able to move the pieces. Likewise, many other applications designed for a single-user environment could immediately be of use in a collaborative setting.

In our examples of a text editor and a chess interface, since the application has no built-in support for collaboration, uncooperative participants could make the application unusable. However, our philosophy is that most users can and will negotiate potential conflicts that arise from collaboration transparency in such a way as to take maximum benefit from the available capabilities of the program.

In Section 2 we describe two fundamentally different approaches to implementing collaboration transparency. In Section 3, we discuss the possibility of modifying the Java class libraries to provide collaboration transparency. Unfortunately, the current implementation of the Java runtime environment requires extensive modification to achieve this goal. In Section 4, we propose modifications to the Java runtime environment that would greatly simplify the necessary changes to the class libraries.

2 Collaboration Transparency

2.1 Display versus Event Broadcasting

Most existing collaboration transparency environments follow the display broadcasting model shown in Figure 1. A central **conference agent** receives all user input and serializes the events so that a single instance of an application receives a single stream of events [14, 23, 15, 3]. The conference agent then distributes display updates to the participants' windowing systems. We call this approach **display broadcasting** since only a graphical depiction of the screen state is distributed to the users. Most collaboration transparency systems, both commercial and research, use this model: SharedX, Sun Microsystems' SharedApp, XTV, Microsoft NetMeeting, Intel ProShare, and Farallon Timbuktu. The X Window System is particularly suited to this centralized architecture; due to X's distributed display and input capabilities, the display of a single instance of an application may be distributed to multiple users. Display broadcasting is also feasible on other platforms. The International Telecommunications Union (ITU) is considering a recommendation based on display broadcasting, called T.SHARE, for inclusion under T.120 Data Protocols for Multimedia Conferencing. Microsoft NetMeeting conforms to the proposed T.SHARE.

The display broadcast approach may be inappropriate for Web-based collaboration transparency for two reasons. First, it may not fit the model of the runtime environment, while it does fit the model of X. In particular, X has the separation of display from computation as a fundamental feature of its architecture. X also uses the model of a relatively "dumb" local display server supporting a remote client with potentially unlimited compute power. On the other hand, Java for example requires that anyone loading a Java applet within a WWW browser 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. Thus, the local system must be "smarter," and network traffic is minimized.

Second, the display broadcast approach requires distributing large amounts of data from the conference agent to the participants, and so may not be appropriate for widespread use on the Internet. In principle, display broadcast systems such as SharedX could be used today to support collaboration transparency on the Internet. But while the performance of sharing X displays is acceptable on a Local Area Network, it becomes unacceptable on a Wide Area Network. According to Smith, analysis of network traffic for the X-based shared environment "Kansas" found nearly a 10:1 ratio of graphics display to event traffic on the network [22]. We conclude that a dramatic decrease in network traffic will result if only the events are shared.

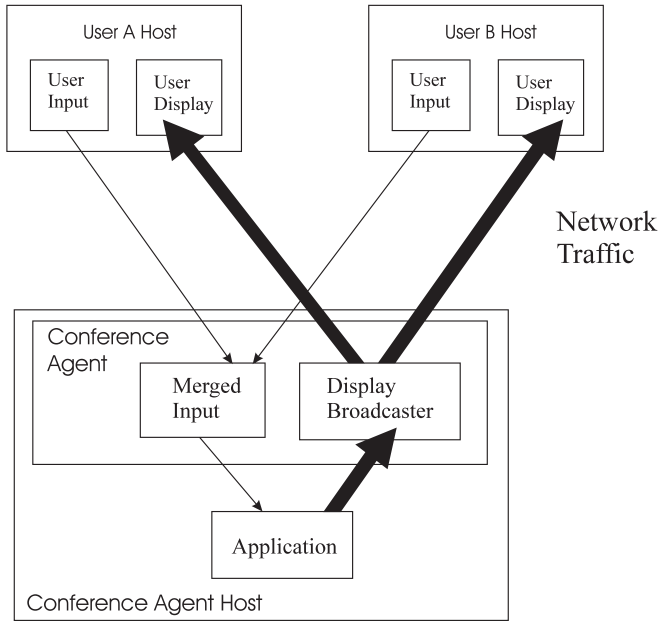

Figure 1: Centralized application with display broadcasting.

Instead of users sharing the display of a single instance of an application, we claim it is better to allow multiple users to share a single *virtual* instance of the application, although in practice each user has a separate copy. This is a replicated architecture for collaboration transparency[16, 17, 20] that we refer to as **event broadcasting** or **event sharing**.

Figure 2 illustrates the event broadcasting approach in which 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 allows every collaborator to invoke events to the system, and those events along with the feedback that the events create are seen by all collaborators. 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. Consistency among the separate instances of the applet is assured because each deterministic applet receives the same input stream.

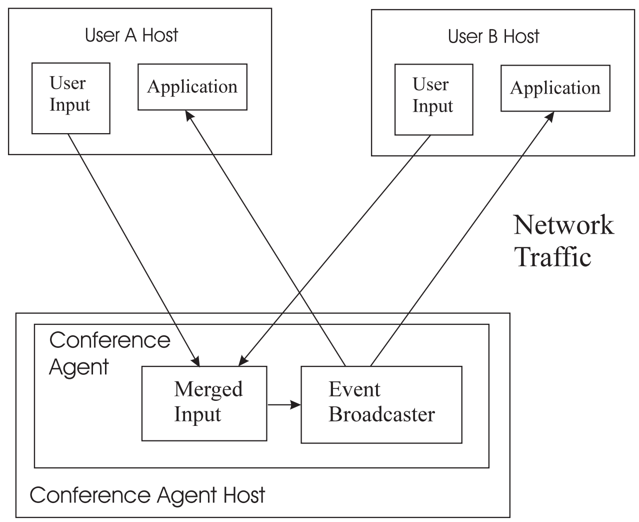

Figure 2: Replicated applications with event broadcasting.

2.2 Feasibility of Event Broadcasting

Replicated architectures for collaboration transparency has attracted prior research with mixed results. At the time of their work on VConf and Dialogo, Lauwers, et al [16, 15], concluded that while a replicated architecture holds promise, it could not maintain collaboration transparency with the then-present technology. From their experiences with MMConf, an event distribution collaborative application system, Crowley, et al, determined that "it is almost impossible for a replicated architecture to assure correct behavior in an unmodified single-user application [4]." However, they further conclude that relatively minor modifications to applications and toolkits are required to allow applications to run robustly under a replicated architecture. Summarizing earlier work, Minenko [17] deemed event broadcasting to coordinate independent instances of an application as "unusable" for the following reasons.

1. Sharing must be based on the assumption that application behavior is deterministic.
2. Application environments of all involved hosts must be identical.
3. Copies of the shared application must be available on each host.
4. Accommodating late-comers into an existing collaborative session is nearly impossible because the state of shared applications must be updated by playing back the input events of the whole session. This process is unstable and can continue for a long time.

5. The most significant disadvantage is difficulty in maintaining the consistency between the multiple copies of the application. The loss of input to one instance of the application due to network loss, for example, throws it out of synchronization with the others.

However, we feel that these difficulties can be overcome in the following ways.

1. Applications are in fact 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, sockets, etc. 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.
2. The application environment can be made identical by ensuring that each copy of the application receives the same response for the same system call (e.g. random numbers) and that all applications share input/output resources such as files and sockets. This implies that such resources must be controlled by the conference agent, which will distribute input and output to all participants.
3. With the advent of Java, access by all participants to a copy of the application software is no longer a concern.
4. By using an object persistence mechanism, such as Java Object Serialization [13], late-comers can be provided with a copy of a shared application in its current state.
5. Consistent input to all instances will result in consistent application state.

Some types of applications, such as large-scale simulations with significant use of random number generators, are likely to be unsuitable for this model. The best architecture for this situation may be a single instance of the computational part of the simulation combined with a shared front-end interface. In a highly networked world, we predict that separation of computation and interface threads will become the model of choice for many large applications, regardless of collaboration issues.

In practice, many useful applications do not make use of resources whose sharing makes the system inefficient, including most existing Java applets.

2.3 Event Collisions

Before event broadcasting becomes feasible, a number of difficulties must be addressed. The most conspicuous problem is maintaining con-

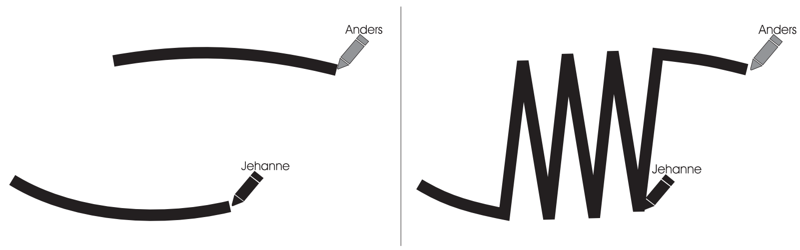

Figure 3: Desired and actual output caused by event collisions

sistency among the copies of a shared application. If events are naively broadcast between collaborators, event streams for events like mouse drags become confused, and conflicts between collaborators arise. For example, suppose two users share a drawing application and both attempt to draw separate free-hand curves by dragging their mouse cursor within the application. The location of the previous mouse drag event for Collaborator 1 is different from that of Collaborator 2. When the next mouse-drag events are broadcast, they conflict because the event for Collaborator 2 uses the context from the event for Collaborator 1. That is, the application will draw a line between both collaborators' mouse positions, instead of between the sequential positions of Collaborator 2's mouse. This is an unexpected result for most users. Figure 3 illustrates the desired output and the actual output occurring from event collisions.

One approach to dealing with the problems of event broadcasting is to use an explicit floor control model[15]. An example of floor control is "pen passing," where only one collaborator can manipulate the application while all other collaborators are watching. Control is explicitly passed from collaborator to collaborator during the use of the application. Since only one collaborator is in control, there is only one event stream that needs to be handled, so event collision and related problems no longer exist.

While explicit floor control simplifies problems with event broadcasting, it burdens the users with having to pass control during application use. Therefore, we prefer implicit floor control models over explicit ones. Implicit floor control does not require user intervention to allow any collaborator to control the application.

Another solution to the problem of event collision is to tag nonatomic events with appropriate context information, including their originating location. With such information available, the local applet can filter mouse drag events and post only one source's inputs at a time to the applet.

The entire series of events associated with the drag activity can be **atomicized** by buffering the drag events so that they are sent as one atomic action (beginning with the initial mouse down and ending with a mouse up) to remote applications. Either the local application can buffer the series of mouse drags or the event broadcaster can buffer (or block) events from other participants until the drag series is complete. If the latter approach is taken, the event broadcaster would benefit from having the source of each event available to it to perform the filtering. The latter approach may be preferred to the former when the event broadcaster is also serializing the collaborators' events. Serialization of events means providing a total ordering of events so that all collaborators receive the same inputs in the same order, thus insuring consistency among the application copies each is executing [11].

In addition, source information provides the following further benefits. With a source field included, events could be filtered by source (e.g., user, host and/or domain). This would address security concerns by allowing events from authorized clients only. Including the source tag also eases the implementation of telepointers (representations of remote collaborators' cursors) and collaborative widgets.

2.4 Awareness of Collaborators' Activity

As part of any collaborative software, a collaborator needs to be aware of the actions and locations of other collaborators within an application. Interface techniques such as telepointers, radar views, and other multi-user widgets are often used to provide such workspace awareness to collaborators [12, 10].

Modern GUI elements are normally expected to deliver feedback upon user input (e.g., button depressions), 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, therefore, users should see GUI element reactions to each collaborator's inputs to the system. For example, if group member A grabs and drags the thumb button of a scroll bar, the other members should see A's cursor move to the scrollbar, followed by the scroll bar moving on their displays as well.

3 Java and Collaboration Transparency

We have set forth four goals for achieving collaboration transparency in the context of Java applets running within the standard Java runtime environment, perhaps in conjunction with a standard WWW browser.

1. Any modifications made to the Java libraries to support collaboration transparency should be packaged into a class library that replaces the standard Java Core API class library without loss of functionality.
2. A collaboration transparency library should not place restrictions on code written by application developers. An application developer should be able to write code without concerning themselves with collaboration details. As a corollary, all Java applets that are 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.)
3. A collaboration transparency library should keep up with the current releases of the Java runtime system.
4. A collaboration transparency library should work with anyone's properly implemented Java runtime system.

3.1 A Test of Collaboration Transparency in Java

To investigate the feasibility of event broadcasting, we implemented two prototypes in Java. First, a simple applet was written in which the applet's event handler method (`handleEvent()`) would forward any incoming (local) events to the event broadcaster (See Figure 4). The applet also had a running thread that received remote events from the event broadcaster, and then introduced 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 Java Virtual Machine to the applet. We could also see whether events posted to the applet were propagated to and acted upon by native user interface elements. The test applet included simple user interface elements, such as buttons and text fields.

The second approach was to modify the `postEvent()` method in the Abstract Window Toolkit(AWT) class library so that a message is printed when an event is posted (See Figure 5). All local events were sent to the event broadcaster, then processed locally. The event broadcaster sent events to all participants other than the originator. Again, the applet had a running thread to receive and introduce remote events into the applet. With this approach, we discovered that some events are not sent to the applet.

We conducted this investigation using JavaSoft's Java Development Kit (JDK) version 1.0.2 on a Sun SparcStation 5 and a 120 MHz Pentium based Windows95 machine. As of this writing, JavaSoft will soon release their JDK 1.1, which drastically changes the event model at

```
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 re-broadcasted
    e = new RemoteEvent(e);

    // rest is as before

    if (handleEvent(e))
        return true;

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

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

the Java API level. However, the problems we discovered using JDK 1.0.2 still exist in JDK 1.1. Thus the limitations described in the next section will remain for the foreseeable future.

3.2 Limitations of the Java AWT

The 1.0.2 and 1.1 implementation of the Java Abstract Window Toolkit (AWT) do not support collaboration transparency, and cannot be easily modified to provide this capability. When we first considered modifying the Java class libraries to make Java applications collaborative, we assumed that the standard buttons, text fields, and other interface components will pass all events to the event-handling system. We also expected that GUI elements would respond to remotely generated events just as if a user at the console had generated the event.

Most events can be passed and processed between two instances of an application, but not all of the events are handled properly. This is particularly true when dealing with interface components implemented through a given platform's native interface library.

The Java AWT binds GUI components, such as buttons, to platform specific implementations of these elements, called **peers**. In this way, the underlying system environment is responsible for the 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 (e.g., Macintosh buttons look and act like Macintosh buttons). Additionally, the size of the AWT is much smaller than it would be if the AWT implemented all interface elements, since it relies on the local system to implement the code for displaying these GUI elements and reacting to user actions.

Unfortunately, the AWT removes some necessary information and control required to provide collaborators with information about the actions of group members. For example, developers are unable to draw over or within some GUI elements, such as buttons. As described below, this leads to problems when implementing telepointers. Lack of information about events causes problems for supporting non-atomic actions such as mouse drags.

No Drawing on Native GUI Elements The AWT provides an inconsistent model in that it permits the developer to draw within some of the AWT components, but not all. System dependent GUI elements, such as `Button`, `ScrollBar`, `Label`, `TextField`, `TextArea`, etc. are displayed by the native operating system. As a result, the AWT is unable to change their on-screen appearance. However, GUI elements derived from the `Canvas` class are not drawn by the underlying operating system and the developer is free to draw on them as she pleases.

This inconsistency prevents the implementation of some collaborative GUI elements, such as telepointers. Once a telepointer moves from a canvas into a button, for example, the telepointer disappears, since it cannot be drawn over the button. This may give a local user who is watching a remote user's telepointer the mistaken impression that the remote collaborator is no longer participating.

Peers Do Not Send All Events The AWT is also inconsistent in that some GUI elements send all of the information about all of the events that occur within them while others send only some events and/or only limited information.

For example, a button press sends only an `action` event to the Java application. The `mouseDown` and `mouseUp` events are never received by the application; they are handled directly by the native implementation of the button component. Furthermore, important information about the local mouse position is lost once the cursor moves within a system dependent GUI element. Upon generation of an `action` event, the location of the mouse within the button is always reported as (0, 0), no matter where the button was actually pressed.

Remote Events Not Accepted by Peers A more important problem is that some events posted to GUI elements are ignored. For example, key strokes generated by remote events and sent to a `TextField` component are ignored. This is due to the inability to instantiate an event with a valid `pData` field. The `pData` field 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 JVM receives an event with an invalid `pData` field, it is simply thrown away, and the event is never received by the peer. Thus, the peer cannot provide feedback for those events that are received remotely, since a valid `pData` field cannot be created. More details are provided by Newmarch [18].

No Graphical Response to Posted Events As we envision event broadcasting, when a broadcast event is received, it is *posted* to the corresponding GUI element in each local version of the applet. If the element was written to handle some type of event, it will execute the method associated with that event. However, with the current implementation of the AWT, the element does not react to the posted event graphically. The reason is that when the event is posted to the GUI element, it is not passed on to the platform dependent GUI peer of that element. 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 by which a developer may force an element to display input responses, such as button clicks. This limitation means that we cannot provide awareness and feedback of collaborators' actions as discussed in Section 2.4.

Cursor Limitations Finally, a minor but related problem is the limited support for cursor shapes and colors. The Java AWT 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 the "What You See Is What I See" [24] or the more relaxed "What You See Is What I Think You See" [21] goals for collaborative software, a user's cursor should be the same shape and color locally as well as remotely.

3.3 Related Work

Despite the difficulties reported in this section, event distribution can be used successfully with the Java AWT. The Java Collaborator Toolset (JCT) [1] is a class library to create collaboration aware applications that use application replication, event distribution and explicit floor control. JCT extends the standard Java AWT classes such that an interface object transmits events to an event distributor when that user holds the floor. However, Abdel-Wahab, Kvande and Nanjangud also conclude that there is a need for all events to be delivered to the application, particularly `mouseDown` and `mouseUp` events in a button.

4 Solutions to Java AWT Limitations

We originally hoped that only a relatively small portion of the Java API libraries would need to be modified to implement collaboration transparency. However, the Java API as it exists now requires rewriting all platform-dependent component peers in Java to support collaboration transparency by means of event broadcasting. This is undesirable for several reasons (some of which lead to the original decision to use peers to implement the Java API).

- It would further slow the already relatively slow interpreted performance of Java programs, even when using Just In Time Code Generators (JIT) to optimize execution speed [27, 8].

- It would significantly increase the size of the class library since those parts of the GUI elements that are handled by native platform code would be re-implemented in Java.
- GUI components would lose their platform-specific look and feel. This can be argued as being good, since now all of the components will look the same. It can also be argued as being bad, because now components with an unfamiliar look and feel are introduced into the applications. Also, the collaboration is not quite as transparent because a collaborative version of an application looks different from the non-collaborative version.
- Any party other than JavaSoft attempting this would be caught in a never-ending version chase of the Java AWT in order to meet the third and fourth goals specified in the beginning of Section 3

The following are our recommended improvements to the standard Java API. If these are applied, we believe collaboration transparency via event broadcasting becomes practical.

Event Propagation In order to take advantage of Java's use of native GUI elements and implement collaboration transparency, all input events need to be made available at the Java environment level, regardless of where they occur, whether in a component implemented in Java or one that is implemented in the native graphics toolkit. When an event is posted to an application, it should fall through to the system level so that the system will react appropriately by displaying graphical feedback to an input event.

Event Tagging The `Event` class should include information about the source of the event, in particular, which instance of an application generated the event. This will allow a transparently collaborative environment to buffer and serialize user inputs, as described in Section 2.3.

Drawing on all GUI Elements In order to implement certain user awareness features, such as telepointers, developers need to be able to draw over the GUI elements provided by the AWT. For example, it would be useful if the Button class implemented a `paint()` method that we could override in order to draw telepointers on a button. The current implementation of the Button class has no such method.

Mouse Cursor Finally, the API would benefit from improved support for modifying the displayed mouse cursor. This would allow for the cursor on the collaborator's screen to match the color and shape of any telepointer on other screens.

5 Summary

We have presented the benefits of providing collaboration transparency to existing Java Applets available on the World Wide Web. Within the context of the Java runtime environment, the standard approach to collaboration transparency through a single conference agent collecting all events and broadcasting screen updates to the various collaborators requires high bandwidth and does not match the architecture of the Java runtime environment.

As an alternative, we considered the approach of event broadcasting, where independent instances of an application are synchronized through a single conference agent. This approach has the potential of greatly reducing the bandwidth required to share applications. A number of limitations imposed by the current Java class library that make collaboration transparency via event broadcasting difficult or impossible were described. Solutions to these problems can be provided through feasible modifications to the Java runtime environment.

We are currently working with JavaSoft, implementors of the Java runtime environment, to support the solutions proposed here. Preliminary implementation results may be found at the following URL: http://simon.cs.vt.edu/JAMM

6 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 his review of this paper. We also thank Amy Fowler of the JavaSoft AWT group for her comments.

7 About the Authors

James “Bo” Begole is a PhD candidate in the Department of Computer Science at Virginia Tech. His current research focus includes collaborative computing and interactive visualization. Mr. Begole received his MS in Computer Science from Virginia Tech in 1994, and a BS in Mathematics from Virginia Commonwealth University in 1992. Prior to his academic life, he spent eight years as an Arabic linguist in the US Army.

Craig A. Struble is a PhD student in computer science at Virginia Tech. His interests include symbolic computation, programming languages, and collaborative software. Mr. Struble received a BS in mathematics and computer science in 1993, and an MS in computer science in 1996 both from Virginia Tech.

Clifford A. Shaffer received his PhD in Computer Science from the University of Maryland at College Park in 1986. Since 1987, he has been a member of the Department of Computer Science at Virginia Tech where he is presently an Associate Professor. Dr. Shaffer's current research interests are in the areas of computer aided education, CSCW, spatial data structures, and data visualization.

References

[1] H. Abdel-Wahab, B. Kvande, S. Nanjangud, O. Kim, J.P. Favreau, "Using Java for Multimedia Collaborative Applications," PROMS'96, Madrid, Spain URL: http://www.cs.odu.edu/~kvande/Projects/Collaborator/

[2] H. Abdel-Wahab and M. Feit, "XTV: A Framework for Sharing X Window Clients in Remote Synchronous Collaboration," *Proceedings, IEEE TriComm '91: Communications for Distributed Applications & Systems*, Chapel Hill, North Carolina, April 1991, pp 159-167.

[3] R. Bentley, T. Rodden, P. Sawyer, and I. Sommerville, "Architectural Support for Cooperative Multiuser Interfaces," *IEEE Computer*, 7(4), May 1994, pp. 37-46.

[4] T. Crowley, P. Miazzo, E. Baker, H. Forsdick, and R. Tomlinson, "MMConf: An Infrastructure for Building Shared Multimedia Applications," *CSCW 90: Proceedings on Computer-Supported Cooperative Work,* Los Angeles, Oct. 7-10, 1990. ACM Press, New York, 1990, pp. 329-342.

[5] D. Flanagan, *Java in a Nutshell,* O'Reilly & Associates, Inc., Sebastopol, CA, 1996.

[6] U. Gall and F.J. Hauck, "Promondia: A Java-Based Framework for Real-Time Group Communication in the Web," to be published in 1997.

[7] D. Garfinkel, B.C. Welti and T.W. Yip, "HP SharedX: A Tool for Real-Time Collaboration," *HP Journal*, April 1994, pp. 23-26.

[8] J. Gosling and H. McGilton, "The Java Language Environment: A White Paper," Sun Microsystems, Inc. Mountain View, CA, May 1995.

[9] S. Greenberg, "Sharing Views and Interactions with Single-User Applications," *Proceedings of the ACM/IEEE Conference on Office Information Systems*, ACM Press, New York, 1990, pp. 227-237.

[10] S. Greenberg, C. Gutwin and M. Roseman, "Semantic Telepointers for Groupware," Research report, Department of Computer Science, University of Calgary, Calgary, Alberta, Canada, June 1996. URL: http://www.cpsc.ucalgary.ca/redirect/grouplab/papers/SemanticCursors/TechReport.html

[11] S. Greenberg and D. Marwood, "Real Time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface," *Proceedings of the ACM Conference on Computer Supported Cooperative Work 1994*, ACM Press, 1994, pp. 207-217,

[12] C. Gutwin, S. Greenberg and M. Roseman, "Workspace Awareness in Real-Time Distributed Groupware: Framework, Widgets, and Evaluation," Department of Computer Science, University of Calgary, Calgary, Canada, January 1996. URL: http://www.cpsc.ucalgary.ca/projects/grouplab/papers/WorkspaceJan96/Workspace_jan96.html

[13] "Java Object Serialization Specification," Revision 0.95, Sun Microsystems, Inc., September 3, 1996. URL: http://chatsubo.javasoft.com/current/serial/index.html

[14] O. Jones, "Multidisplay Software in X: A Survey of Architectures," *The X Resource* (6), Spring 1993, pp. 97-113.

[15] J.C. Lauwers and K.A. Lantz, "Collaboration Awareness in Support of Collaboration Transparency: Requirements for the Next Generation of Shared-Window Systems," *Proceedings of CHI 90, Conference on Human Factors in Computing Systems*, ACM Press, New York, 1990, pp. 303-311.

[16] J.C. Lauwers, T.A. Joseph, K.A. Lantz, L. Romanow, "Replicated Architectures for Shared Window Systems: A Critique," *Proceedings of the Conference on Office Information Systems*, ACM Press, March 1990, pp. 249-260.

[17] W. Minenko, "The Application Sharing Technology," *The X Advisor*, 1(1), June 1995. URL: http://www.unx.com/DD/advisor/

[18] J. Newmarch, "The Java AWT: New Event Model," *The X Advisor*, 1(6), May 1996.

[19] NCSA, "NCSA Habanero," June 1996. URL: http://www.ncsa.uiuc.edu/SDG/Software/Habanero/

[20] W. Reinhard, J. Schweitzer, G. Vvlksen and M. Weber, "CSCW Tools: Concepts and Architectures," *IEEE Computer,* 7(4), May 1994.

[21] R. Smith, "What You See Is What I Think You See," *Spring '92 ACM Conference on Computer Supported Collaborative Learning,* 21(3), ACM Press, New York, 1992.

[22] R. Smith, "Kansas: A Large, Flat, Mutli-user Virtual World for Interactive Simulations," Presentation to the Virginia Tech Computer Science Colloquium Series, 24 April 1996.

[23] S. Sarin and I. Greif, "Computer-based Real-Time Conferencing Systems," *IEEE Computer,* 18(10), 1985, pp. 33-45.

[24] M. Stefik, D.G. Bobrow, S.Lanning and D.Tatar, "WYSIWIS revised: Early experiences with multiuser interfaces," *Proceedings of the Conference on Computer Supported Collaborative Work*, Austin Texas, December 1986, pp. 276-290.

[25] SunSolutions, *The Complete Guide to ShowMe 2.0.1*, Sun Technology Enterprises, Inc., Mountain View, CA, June 1994.

[26] TESSERACT Information Solutions Inc., "RealTime Collaborative Groupware for JAAS - Intro," May 1996. URL: http://www.tisinc.com/realtime/

[27] F. Yellin, "The Java Native Code API," Sun Microsystems, Inc. URL: http://java.sun.com/doc/jit_interface.html