

An Internet Collaborative Environment for Sharing Java Applications

H. Abdel-Wahab and B. Kvande
Department of Computer Science
Old Dominion University
Norfolk, Va 23529
{wahab,kvande}@cs.odu.edu

O. Kim and J.P. Favreau
Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, Md 20899
{okim,favreau}@sncl.nist.gov

Abstract

In the Internet community there is a strong demand for platform-independent collaboration software. Java is developed with the major design goals of being a platform-independent, and Internet-oriented programming language. In this paper we show how a group of Internet users can share single-user Java applications for synchronous collaboration. Our approach is based on replicated tool architecture in which each participant runs a copy of the application and the activity of each user is multicast to all the participants in the conference. We have developed a system called Java Collaborative Environment (JCE), on which the Java's Abstract Window Toolkit (AWT) is extended such that mouse and keyboard events are intercepted and distributed among all copies of the shared Java application. In addition we provide an infrastructure and a simple interface for session management and floor control.

1 Introduction

Most current existing collaborative systems require the participants in a conference to use the same window system. For example, XTV [1, 2] and Suite [4] are based on the X window system and require that the participant's machines run the X server. Other systems such as WTV [3] have tried to replicate the functionality of XTV replacing the X windows with Microsoft Windows. Ideally, each participant in a collaborative conference should be able to use whatever platform he or she prefers. For example, some may use PCs running MS Windows 95. Others may use workstations running different version of UNIX and X windows, yet others may use PowerPC Macintoshes. Before the introduction of Java, this sort of collaboration was enormously difficult to achieve. Java programs are compiled to an architecture neutral byte-code format and thus can run on any system that implements a Java virtual machine and its abstract window system. Java provides a fortuitous opportunity for the

Computer Supported Cooperative Work (CSCW) [5] community to overcome a barrier which hitherto hindered the wide spread use of collaboration technology.

To overcome the platform-dependency problem for application sharing in heterogeneous platforms, NIST (National Institute of Standards and Technology) and ODU (Old Dominion University) are jointly conducting a research project to investigate mechanisms for sharing multimedia applications among participants on not only heterogeneous windowing and operating systems, but on different hardware platforms.

We have developed mechanisms to intercept, distribute and recreate the user events that allow single-user Java applications to be shared among conference participants. These mechanisms can be run transparently on any system implementing Java. The mechanisms incorporate the services of network communications, conference management and floor control management. The network communications services include distribution of the data among conference participants; conference management includes joining and leaving a session; and floor control includes participant's control and interaction with the application during a session. In this paper, we refer to the prototype which has been developed as JCE, an acronym for Java Collaborative Environment.

2 JCE System Architecture

Figure 1 depicts the overall JCE system architecture, and the relationship and communication paths among all processes of the system, for a given conferencing session.

The Java applications denoted as *Java App 1 (and 2)* in Figure 1 are not part of the system. They are single-user applications developed using the collaborative package we developed and discussed later in Section 3. Participants can invoke one or more applications in a given conference. Our model is based on the *replicated architecture* [6] in which an instance of each application runs locally at each

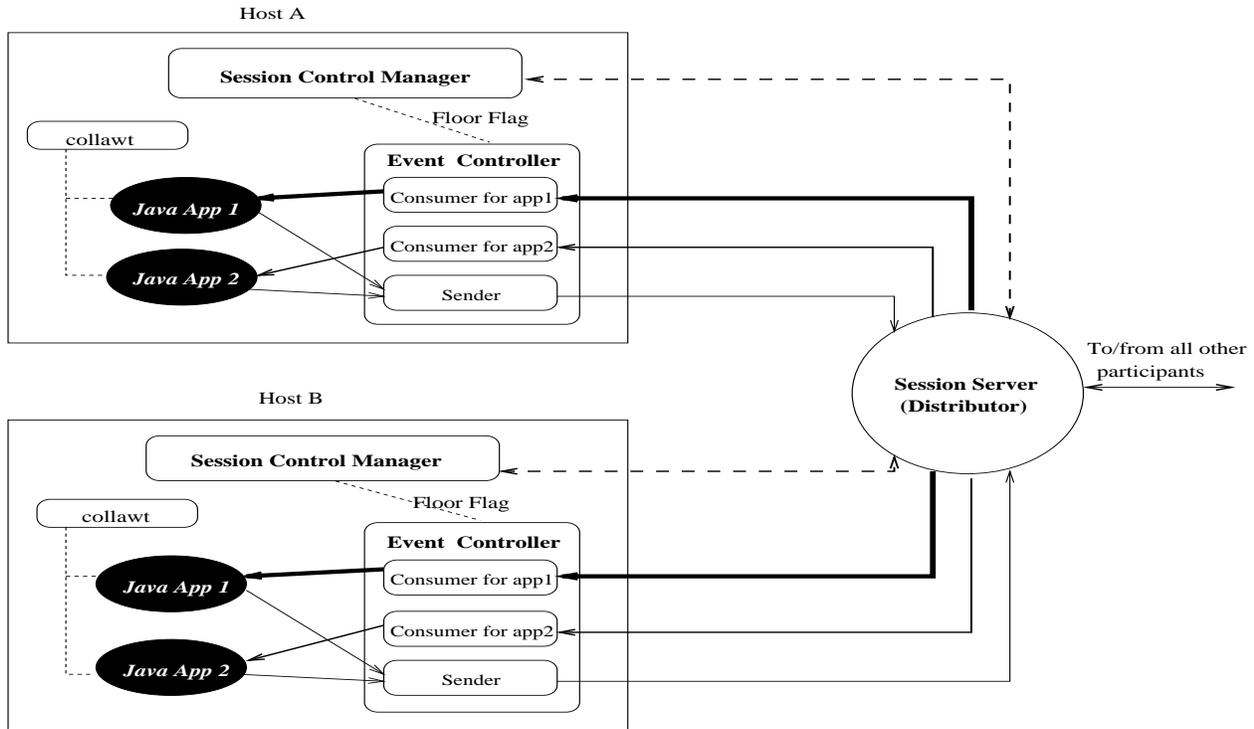


Figure 1. Overall System Architecture

participant's site and the activity of each user is distributed to all the participants in a conference.

As shown in Figure 1 JCE consists of three components: the *Session Control Manager*, the *Event Controller*, and the *Session Server*.

The *Session Control Manager* (SCM) provides the user with a graphical interface offering the following options: to call, join or leave a session; to start applications; and to request or release a floor. Each participant is given an SCM process that exchanges control information with the Server for the duration of the session.

The *Event Controller* is the core of the collaboration mechanisms. It is composed of two processes: the event *Sender* and *Consumer*. When an application is started an *Event Controller* for the application is automatically instantiated by the *Session Control Manager*. When two or more applications are shared, two or more *Consumers* are created as shown in Figure 1. The *Sender* is declared as a static (i.e., class) method of the *Event Controller*, so only one *Sender* method exists for all applications. The *Sender* method first checks the intercepted event to determine whether or not it should be sent to the *Session Server*, since events originating from shared applications are always forwarded. The *Consumer* processes receive events redistributed by the *Session Server* from other participants, and post them to the local instance of the application as if they were originated locally. This process is completely transparent to the application,

i.e., the application is unaware that it is being shared.

The *Session Server* in Figure 1 provides three distinct functions: distribution of all messages to all participants; group management for a given session, including joining or leaving a session; and server floor control management.

3 Collaborative AWT

We have developed a mechanism to intercept all the user interface events and send them to other participants in the conference. All the GUI components defined in the standard *java.awt* package version 1.1 [8] are extended in JCE to implement this mechanism. The extended package is called *collawt*.

For example, here is the class *Button* in *collawt* showing the overridden method *processEvent*, with the interception statement shown in **boldface**.

```
public class Button extends java.awt.Button {
    public void processEvent(AWTEvent evt) {
        if (EventController.sender(evt)) {
            if (evt instanceof(ActionEvent) {
                super.processActionEvent((ActionEvent)evt);
                return;
            }
            super.processEvent(evt);
        }
    }
}
```

```

public static boolean sender
    (java.awt.AWTEvent evt) {
    //Check if event came from network.
    if (evt instanceof CollEventFromNet)
        if ((CollEventFromNet)evt.getFromNet())
            return true;
    //don't send/post if floor isn't yours
    if ((selFloorChoice!=PacketDef.FLOOR_OFF)
        && (!floorFlag))

        return false;
    //create/send packet based on event type.
    if (evt instanceof(ActionEvent))
        sendActionEvent((ActionEvent)evt);
    else if (evt instanceof(MouseEvent))
        sendMouseEvent((MouseEvent)evt);

    ... etc.. handle other events similarly
    return true;
}

```

Figure 2. Sender Method

In this class the default event-handler, the `Component.processEvent` method, is replaced in every GUI component that produces user events. To override this method, all the GUI components in `collawt` are subclassed from their corresponding components in `java.awt`. Every widget inherits from its corresponding widget in `java.awt` and overrides the `processEvent` method. The new implementation of this method intercepts the events, and then calls the *sender* method in `EventController` object before it gives it to the original event-handler.

When the Event Controller receives an event, the event is rebuilt and dispatched to the target component. In the `java.awt` package version 1.1 [8], events are represented by a hierarchy of event classes, each of which is defined by the specific event type (e.g., `KeyEvent`, `ActionEvent` and so on) or related group of event subtypes (e.g., `MouseEvent` represents `MouseUp`, `MouseDown`, `MouseDown` and etc). All the event type classes defined in the `java.awt` package are also extended in `collawt`, which are used to flatten and recreate user events that are sent and received, respectively. The dispatching of the received events to the corresponding local components is completely transparent to the application and looks like a normal events originating from the user.

4 Event Handling

When a user interacts with a GUI component in the interface of an application, the events are intercepted by the component's event-handler *processEvent* as explained in the previous section. This method calls the *EventController.sender* method which checks whether the event is from the local

```

public void consumer() {
    while (true) try {
        ReadEvent(packet);
        // locate the target component.
        target = getComponent(packet);
        evttype = getEventType(packet);
        // create event depending on type.
        switch(evttype) {
            case MouseEvent:
                event = new CollMouseEvent(packet);
                break;
            case KeyEvent:
                event = new CollKeyEvent(packet);
                break;

            ..etc.. create other event objects based on types.
        }
        // dispatch the event to the source object.
        ((Component)target).dispatchEvent(event);
    }
}

```

Figure 3. Consumer Method

user and sends the event to the session server for distribution. The event is converted to a string representation before it is sent to the network (see Section 4.2).

The events received from the network are consumed to the corresponding target component by the *EventController.consumer* method, a thread waiting in an infinite loop receiving packets from the network. The received events are reconstructed and dispatched to the correct target component of the application. When an event is posted, the component's event-handler *processEvent* is called as if this were a local event generated by the user. Since this method calls *EventController.sender* method, the received event should not be re-sent on the network. The technique developed to prevent this infinite loop of sending the same event will be described in the Section 4.3. Figures 2 and 3 show snippet of code from *sender* and *consumer* methods.

4.1 Component Identification

In Java, a GUI component has no explicit identity. To be able to identify the components of an application, an identification (CID) number is assigned to each component when opening the application. The CID of the component is sent along with the event when distributed. This CID is then used to locate the target component for the event in the receiving application.

When an application is opened, the hierarchy of components of the windows contained in the application is traversed and a reference to each component is inserted into a vector. Since the components in the application are traversed in the same sequence every time the application is

```

private static void build_component_tree
    (Container root, Vector cVector) {
    // count components contained in 'root'.
    int components = root.getComponentCount();
    for (int i = 0; i < components; i++) {
        //get each component contained in 'root'.
        Component c = root.getComponent(i);
        //if it is container, get its components
        if (c instanceof Container) {
            build_component_tree
                ((Container) c, cVector);
        }
        //add component to vector
        cVector.addElement(c);
    }
}

```

Figure 4. Building Components Index Vector

opened, a component will be placed in the same position of the vector and given the same CID in every replicated instance of the application.

Figure 4 shows the method used to build that vector. In this method, we rely on Java's *awt countComponents* and *getComponent* methods in *Container* class to traverse the tree and construct the corresponding vector.

4.2 Flattening Events

Before an event is sent it has to be converted to a representation suitable for transportation over the network. This representation has to contain sufficient information for the event to be easily reconstructed and posted to the correct target component. Each collaborative event type class in *collawt* contains a method that flattens an event to a string representation to be sent out. The event is converted to a packet where every field is interpreted as a string separated by commas, which is fairly easy to parse and reconstruct at the receiving application. A layout of a string representation of events is shown below:

```

TopLevelWindowId | CID | EventId | Event-specific-info

```

4.3 Reconstructing and Consuming Events

For an event to be posted, the matching target component in the receiving application must be located. The *ToplevelWindowId* and the *CID* are used to look up the vector containing the application components. Once the target component is located, an appropriate collaborative event object *CollEvent* is created depending upon the event type and then posted to the target component. (See Figure 3). When

a *CollEvent* is posted to the target component, the application interprets it as if it was a normal local event. However, a *CollEvent* received from the network should not be sent back again. To differentiate the *CollEvent* and a normal local events, we have defined the interface *CollEventFromNet*. The interface contains methods that indicate whether or not the event is received from the network. The collaborative event classes in *collawt* implement the interface so that *EventController.sender* method can check whether the event captured is from the network (See Figure 2). The received event is only dispatched to the target component for normal processing and is not resent back to the network.

5 Session Server

The major functions of the *Session Server* shown in Figure 1 are:

1. To act as a *distributor* to multicast events among participants and applications;
2. To provide *session management* such as joining or leaving a session; and
3. To provide *floor control* for regulating access to shared applications.

The distributor maintains a list for each application in use and their current participants, and a queue for floor management. The list is used to distribute events from one application to the other copies of the application. When a user joins a session or opens a shared application, the server establishes a TCP connection and creates a thread to serve the connection. The packet format to exchange information over the TCP connections is:

```

length | packet type | data

```

The values of *packet type* for session management and event distribution are:

NEW_SESSION	Start a new session.
EC_REGISTER	Register a new application.
START_APPLI	Starts an application.
JOIN	Join an existing session.
LEAVE	Leave a session.
EVENT	An event to be distributed.

The event distributor is implemented using two classes: The *TCPServer* and the *TCPConnection*. The *TCPServer* has a full overview of every client connected and registered with the distributor. The class listens for connection requests from clients, and constructs a new *TCPConnection* object when a connection is requested. The *TCPServer* class also registers and removes the clients as users of the application groups, and most importantly it distributes events by calling each *TCPConnection* object instructing them to send the

event to the client. The TCPConnection class sets up the actual connection with the client and does the communications work. It initiates a thread waiting for data from the client and processes the data received by taking action itself or calling the appropriate method in the TCPServer class. Figure 5 shows outline of the Java implementation of the distributor.

6 Floor Management

In a collaborative environment, users should be able to interact with shared applications in a controlled and orderly manner. Most collaborative systems use the concept of *floor*. The floor controls the user's ability to provide input and interact with the shared applications, i.e., the floor holder has the right to "speak". No more than one user at a time can have the floor (unless the nature of the application allows multiple participants to provide input without violating the integrity of the shared workspace).

The floor management schemes currently implemented in JCE are:

1. *Request-and-Get* policy: allows a requesting user to immediately get the floor possibly preempting the current floor holder.
2. *Reuest-and-Wait* policy: allows the requester to get the floor when becomes available. The current floor owner must release it so that the requesting user can get it. Floor requests are queued and granted in the order of they are received by the Session Server.
3. *No-Floor* policy: allows any participant to interact with the shared application.

The floor control GUI interface has buttons to change the floor policy and for requesting and releasing the floor. It addition the interface always display information about the current floor holder.

The floor management is handled by the event distributor. It distributes the floor to the requesting client and knows at all times who is the current floor holder and what policy is being used. The messages used for this purpose are shown below:

REQUEST_FLOOR	User Requested the floor.
RELEASE_FLOOR	User Released the floor.
GRANT_FLOOR	User got the floor.
GIVE_FLOOR	User give up the floor.
FLOOR_OWNER	Who is the current holder.

The implementation of the floor management is divided between the *Distributor* and the *Session Control Manger* (See Figure 1). Floor enforcement is implemented by a *floor-flag* in the *Event Controller* object. This flag reflects

```
public class TCPServer extends Thread {
    protected ServerSocket serverSocket;
    public TCPServer(int port) {
        serverSocket = new ServerSocket(port);
        start();
    }
    public void run() {
        while(true) {
            Socket sd = serverSocket.accept();
            TCPConnection connection =
                new TCPConnection(this, sd);
        }
    }
    public static void main(String argv[]) {
        TCPServer tcpServer;
        tcpServer = new TCPServer(argv[0]);
    }
}

public class TCPConnection extends Thread {
    ... Initalize local variables ...

    public TCPConnection
        (TCPServer server, Socket sd) {
        this.server = server;
        this.socket = sd;
        in = sd.getInputStream();
        out = sd.getOutputStream();
        start();
    }

    public void run() {
        int length = -1;
        while (!done) try {
            length = in.readInt();
            if (length != -1){
                packet = new Packet(readBytes(length));
                processPacket(packet);
            }
            else done = true;
        }
    }
}
}
```

Figure 5. Distributor Implementation

the status of the floor. Whenever an event from a GUI component is to be sent, this flag is checked. If it is enabled, the event is distributed to others participants.

7 Conclusions and Future Work

The Java Collaborative Environment (JCE) presented in this paper allows application sharing among diverse systems such as UNIX workstation-based and PC Windows-based systems. Through JCE any collaborative single-user Java applications can be shared and by using the JCE simple user interface, JCE participants may launch new shared applications and circulate the floor among themselves to control and manipulate these shared applications.

Our next goal is to use JCE from java-enabled Internet browsers such as the Netscape Navigator and the Microsoft Internet Explorer. Due to some of the limitation imposed by Internet browsers for security reasons, the participants may not be able to perform certain basic functions such as saving files on secondary storage. We would like to maximize the functions that the participants can do through the Web and perform the remaining functions using a parallel stand-alone JCE interface.

Among the issues to be addressed in our future works are: replication management, accommodating late comers, scalability through the use of reliable multicasting, and the integration of audio and video in JCE.

As we have explained in this paper, JCE is based on replicated-tool architecture and among the problems associated with this approach is *replication management*. Most applications need to create or use objects during execution, for instance, the environment variables, the initialization dot files, and the files storing multimedia data. These objects must be replicated and available at each site for the correct operation of the JCE system. There are three types of objects to be replicated and managed: environment, operational and final objects. Before the invocation of each copy of the shared application at each site, we must ensure that all sites have identical operating environments.

To increase the system performance as the number of participants increases we should use reliable multicasting for data transport instead of the current server-based (star-topology) TCP connections. If one participant needs to send a message to all other participants, he or she sends it to the server which in turn distributes it to all the participants, one at a time, using the TCP connections. This may be acceptable if the number of participants is small (e.g., 4 or 5), but as the number of participants increases, the system performance degrades and the session quality is reduced, as measured by several parameters, such as view synchronization. The use of *reliable multicasting* (e.g., RMP provided by Berkeley/West Virginia [7]) should greatly improve both the performance and the quality of session.

Beside shared applications, audio followed by video in this order are important to support full and effective collaboration among participants. Almost all PCs and Workstations now have audio devices (microphone and speakers), though they are often not compatible with each other and may use different audio formats. Thus, our task here is to ensure that all participants can talk and hear each other without being concerned about compatibility between their respective devices.

References

- [1] 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, pp. 159-167, April 1991.
- [2] H. Abdel-Wahab and K. Jeffay, "Issues, Problems and Solutions in Sharing X Clients on Multiple Displays", *Journal of Internetworking Research & Experience*, pp. 1-15, Vol. 5, No. 1, March 1994.
- [3] D. Adams, "WTV: An MS Windows based Collaborative System", Master's Project Report, Department of Computer Science, Old Dominion University, Dec. 1995.
- [4] P. Dewan and R. Chouldhary, "A high-level and flexible framework for implementing multiuser interfaces", *ACM Transaction on Information Systems*, Vol. 10, No. 4, 345-380, (October 1993).
- [5] J. Grudin, "Computer-Supported Cooperative Work: History and Focus", *IEEE Computer*, Vol. 27, No. 5, 19-26, (May 1994).
- [6] R. Steinmetz and K. Nahrstedt, *Multimedia: Computing, Communications & Applications* Prentice-Hall, 1995.
- [7] B. Whetten, T. Montgomery, and S. Kaplan, "A High Performance Totally Ordered Multicast Protocol", *Theory and Practice in Distributed Systems*, Springer Verlag LCNS 938, 1994.
- [8] *Abstract Windowing Toolkit (AWT) package, Java Developers Kit (JDK) Version 1.1 API*, Sun Microsystems Inc. Mountain View, CA 94043