[SEGSt01] *Shervin Shirmohammadi, Abdulmotaleb El Saddik, Nicolas D. Georganas, and Ralf Steinmetz.* **JASMINE: A Java Tool for Multimedia Collaboration on the Internet**; Journal of Multimedia Tools and Applications, January 2001.

JASMINE: A Java Tool for Multimedia Collaboration on the Internet

Shervin Shirmohammadi [1], Abdulmotaleb El Saddik [2], Nicolas D. Georganas [1], and Ralf Steinmetz [2,3]

[1]Multimedia Communications Research Laboratory, School of Information Technology and Engineering, University of Ottawa, Ottawa, Canada

[2]Industrial Process and System Communications, Dept. of Electrical Eng. & Information Technology, Darmstadt University of Technology, Darmstadt, Germany

[3]GMD IPSI, German National Research Center for Information Technology, Darmstadt, Germany

Abstract

Although collaboration tools have existed for a long time [9], Internet-based multimedia collaboration has recently received a lot of attention mainly due to easy accessibility of the Internet by ordinary users. The Java platform and programming language has also introduced yet another level of easy access: platform-independent computing. As a result, it is very attractive to use Java to design multimedia collaboration systems for the Internet. Today there are many systems, which use Java for multimedia collaboration. However most of these systems require the shared Java application to be re-written according to the collaboration system's Application Programming Interface (API) - a task which is sometimes difficult or even impossible. In this paper, we describe a practical approach for *transparent* collaboration with Java. Our approach is transparent in that the Java application can be shared as is with no modifications. The main idea behind our system is that user events occurring through the interactions with the application can be caught, distributed, and reconstructed, hence allowing Java applications to be shared transparently.

Our architecture allows us to make the huge installed base of Java applications collaborative, without any modification to their original code. We also prove the feasibility of our architecture by an implementation.

1. Introduction

Since the advent of the Internet, the computing and communications industry have progressed very rapidly. Today, any user with a desktop computer can access and share multimedia documents with others through the Internet. Furthermore, this accessibility is being extended beyond desktop computers and into *Information Appliances:* consumer devices that bring together computing and communications in one box to ordinary users. Examples of these devices are Web TVs, Net Gaming devices, Internet Screen Phones, and Network Computers. According to a report published by the International Data Corporation, some five million such devices were used in 1998, with over forty million projected to be in use by 2001 [5]. In addition, all these devices will be interconnected through pervasive computing technologies and systems such as JINI [15]. It seems certain that in the near future every person, no matter where located geographically, will be equipped with some sort of network computing capability, either by means of conventional desktop computing or through information appliances. This not only means that geographically-distributed people will be able to easily communicate, but also "collaborate"; i.e., share multimedia documents and applications. Examples are joint editing, whiteboarding, joint browsing, and multi-user presentations, used in a variety of applications such as conferencing, collaborative design, training and telelearning.

A problem with many collaborative applications is their platform dependence leading to the fact that users communicating in heterogeneous environments are restricted in their choice of a cooperative application. For example some users might choose UNIX-workstations, while others might prefer Windows 95/98/NT or Macintosh. But with the introduction of Java it became possible to overcome these problems. Consequently diverse approaches emerged which used Java for developing collaborative systems, producing a variety of toolkits and platforms [12] [1] [2] [8]. However, almost every system described in the literature requires the use of an API, or tries to replace some core Java-components with self-defined collaborative components.

The approach presented in this paper differs from other approaches in the way that we neither propose a new API for developing collaborative systems nor try to replace core components at run time. In fact a great variety of well-designed applets already exist on the World Wide Web which were developed to be run as stand-alone and it would not be acceptable or possible for many developers to re-implement or change these programs to make them work in a collaborative way. In our architecture, we make use of the Java Events Delegation Model [13] to extend the capabilities of Java applications in a way that stand-alone applets can be used in a collaborative way. The delegation event model of JDK1.1 provides a standard mechanism for a source component to generate an event and send it to a set of listeners. Furthermore, the event model also allows to send the event to an adapter, which then works as an event listener for the source and as a source for the listener. Because the handling of events is a crucial task in developing an application, this enhancement makes the development of applets much more flexible and the control of the events much more easier.

The practicality of our architecture is proven by an implementation. We have developed a collaboration system, called JASMINE[1], which facilitates the creation of multimedia collaboration sessions and enables users to share Java applets and applications, which are either pre-loaded or brought into the session live. The system also provides basic utilities for session moderation and floor control. Our approach applies to both applets and applications and hence these terms are sometimes used interchangeably in this document. The rest of the paper is organized as follows. Section 2 discusses the system architecture, while section 3 describes the implementation of JASMINE. Section 4 presents a performance evaluation of our system, followed by discussion of related work in section 5. Finally section 6 concludes the paper and gives an outlook for future work.

2. Architecture

Basically speaking, the core technology behind any collaboration tool is a mechanism to enable a user to send updates to other users about the interactions that are made to a shared application, as illustrated in figure 1. For example, when one user draws a line on a whiteboard, the system informs the whiteboards of other users so that they also draw the same line. The mechanisms to propagate these "updates" vary according to the design or intended use of the system. Some systems send graphical display updates of the portion of the screen that was changed; the receiver simply redraws that portion using the graphics update. Some other systems send the system's graphical events that were generated as a result of a user's interaction, the receivers then process the events as if generated locally; hence reproducing the interaction at every user [6] [7]. Another approach is the use of

[1] JASMINE: Java Application Sharing in Multiuser INteractive Environmnets.

object tokens, whereby an update message is preceded by a token that defines the semantic of that update message. By looking at the token, the receivers can determine what action to perform; for example draw a line, erase an area, etc. [12].

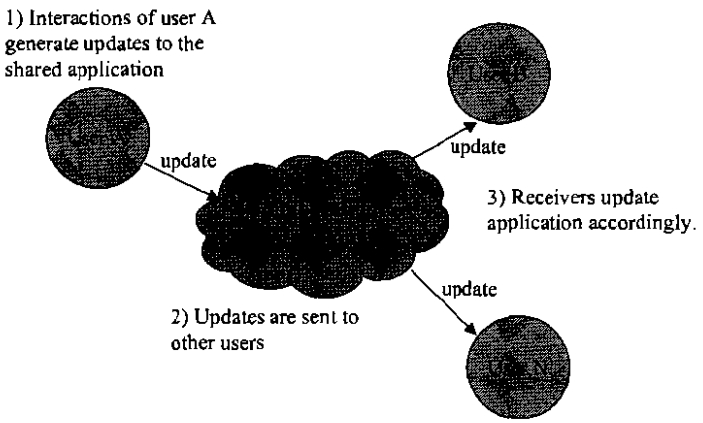


Figure 1. A generic collaboration system

All of these approaches can be implemented using a centralized or fully distributed communication infrastructure. Furthermore, they can be implemented as real-time or near-real-time systems. However, they all have one thing in common: they all must use reliable communication, such as TCP, for their update messages. Although suitable for real-time video/audio data transfer, unreliable communication such as UDP or regular multicasting is not suitable for the transfer of application update messages since these applications, by nature, cannot afford to lose any update data.

To optimize the use of bandwidth and compensate for latency, we have to choose an approach that sends as small amount of information as possible for the updates. Graphics updates are therefore not suitable because of their bulkiness and heavy use of bandwidth. Event updates and object tokens are better candidates. Object tokens are heavily based on the specific application, and must in fact be hard-coded into the shared application - an approach, which is not transparent. This leaves us with Event updates, which are what we

use in our approach.

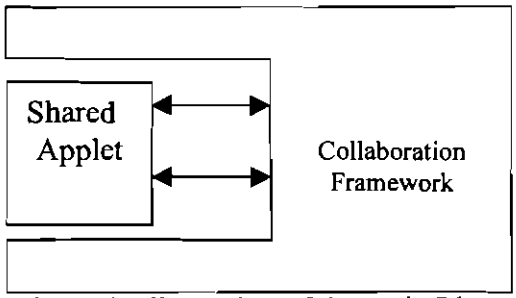


Figure 2. Illustration of the main Idea.

Figure 2 illustrates the overall concept, where our collaboration framework wraps around an applet that is to be shared. The framework listens to all events occurring in the graphical user interface of the applet and transmits these events to all other participants in order to be reconstructed there. The framework captures both AWT-based and Swing-based events. After capturing the event, it is sent to the communication module where the event is sent to all other participants in the session (Figure 3).

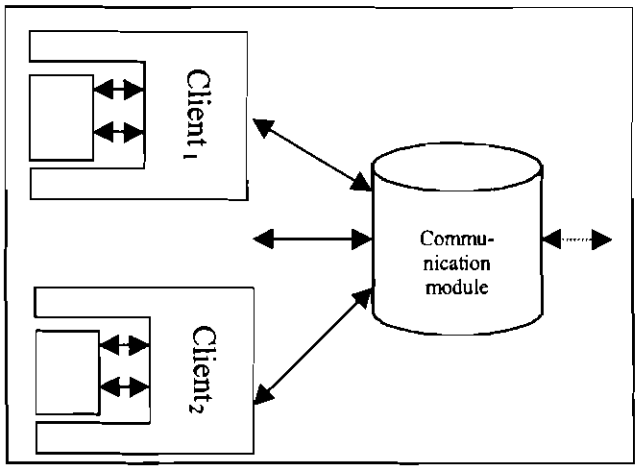


Figure 3. Overall System Architecture of JASMINE.

In the next sections we are going to discuss the architecture in more details, first the client side, and then the communication module.

2.1 Client side

The JASMINE client can be seen as a component adapter. Every event occurring at the graphical user interface of the application is sent to this adapter, which then sends the events to the collaboration server (JASMINE-Server). The client is a Java application, which consists of the following components:

- Collaboration Manager
- Component Adapter
- Listener Adapter
- Event Adapter

These components are discussed next.

2.1.1 Collaboration Manager

The Collaboration Manager is the main component on the client side and provides the user with a graphical interface offering options such as joining the session, starting and sharing applications/applets and chatting with other participants. The collaboration manager is also responsible for dispatching external events coming from the communication module and forwarding them to the component adapter, as well as receiving internal events from the component adapter and sending them to the communication module.

2.1.2 Components Adapter

The Component Adapter maintains a list of the GUI-components of all applications and applets. This list is created with the help of the *java.awt.Container* class, which allows us to get references of all applet components [13]. With the help of the main window of an application, a list of the GUI components in the application can directly be created. Therefore, the main window of an application loaded by the Collaboration Manager is

registered by the Component Adapter. However, Java applets do not use stand-alone windows. They are an extension of the class *java.applet.Applet* and thus of *java.awt.Panel*. Hence, applets can be easily placed into a window, which can then be registered as the main window for the applet. All these registrations are done at the Component Adapter. An example syntax of the registration by the Component Adapter is shown in Figure 4.

```
.....
Class cl = Class.forName(className);
// If it is an applet, instantiate and locate
// it in a Frame
myApplet = (Applet)cl.newInstance();
myApplet.init();
myWindow = new Frame("Titel");
myWindow.add("Center", myApplet);
// Otherwise (if it is an instance of Window) just
// instantiate it
myWindow = (Window)cl.newInstance();
// Register this Frame as main Frame
// by Components Adapter
ComponentsAdapter.addContainer(myWindow);
....
```

Figure 4: Excerpt of the instantiation method

After the registration is done a list of all Swing and/or AWT-components within the loaded application/applet is created. This task is done in the same order on each client, so that a component has the same reference identification at all clients. These references are used to point to specific components, which are the source of the events generated internally and the recipient of the events generated externally. With the help of the references, the recipient of an incoming event is located and the event is reconstructed on each client, as if it occurred locally.

2.1.3 Listener Adapter

The Listener Adapter implements several AWT listeners, which listen to *MouseEvent* and

KeyEvent for all AWT-components except of *java.awt.Scrollbar*, *java.awt.Choice* and *java.awt.List*. For these components the Listener Adapter listens to *AdjustmentEvent*, *ItemEvent* and *ActionEvent*. When an event occurs on the GUI of the application, the Listener Adapter catches it, converts it to an external event, and forwards it to the Collaboration Manager. The Collaboration Manager in turn sends this event to the communication module, which propagates the event to all other participants.

2.1.4 Event Adapter

The Event Adapter works opposite to the Listener Adapter: it converts incoming external events to AWT events, which can then be processed locally.

2.1.5 Data Flow

Let us summarize the client side's architecture through the following data flow diagram. Figure 5 shows the overall event circulation of the system.

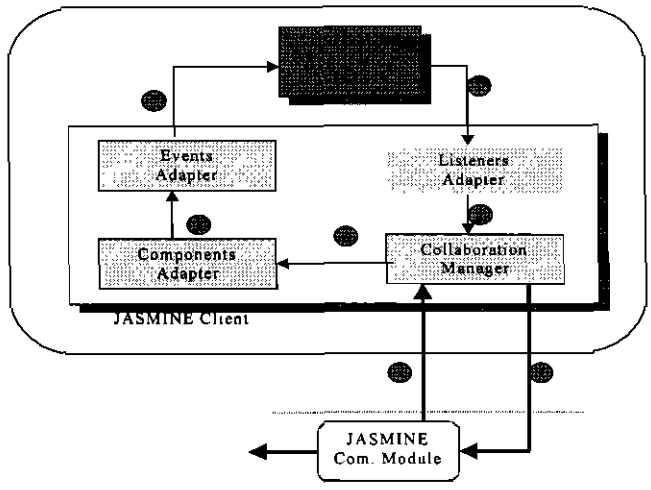


Figure 5: Events circulation

There are two main data paths in the system: the first path is labeled with numbers 1,2 and 3. This path is used to send the internal AWT events to the communication module, and it

works as follows: any Event occurred in a Java-application is caught by the Listener Adapter. The Listener Adapter first tests whether the event is an external or an internal event. It then sends only the internal events, which were not received from other clients, to the Collaboration Manager, which in turn sends the events to the communication module.

Via the second data path shown in figure 5 with numbers 4, 5, 6 and 7, the external AWT events received from the communication module are captured by the Collaboration Manager and the Component Adapter in order to reconstruct the event locally. After receiving the remote event, the Component Adapter extracts the information about its target component and sends this information together with the events to the Event Adapter. The Event Adapter converts the event to normal AWT events and sends them to the application, which then reacts to the event in the same manner as it would to a local user's interaction with the application's GUI.

2.2 Communication Module

The comminations module's main purpose is to receive events from the collaboration manager and propagate them to all participants in the session. It abstracts the network and communication functionalities from the client side so that the client side need not worry about how the events are actually transmitted over the network. This module is separated from the rest of the system because it can be implemented in many different ways based on the communication environment. As mentioned earlier, reliability is a non-negotiable requirement for collaborative applications since loosing even one event can potentially disrupt the collaboration session. This means that the communication module must be implemented by either reliable multicast (RM) or TCP.

Each of these approaches has certain advantages and disadvantages. The obvious

disadvantage of the one-to-one TCP linking approach is scalability: the more users there are, the more network connections are required. What's worse is that the number of connections increases non-linearly with increasing number of users. However, using a client-server approach can substantially decrease the number of network connections. Each client establishes a TCP connection only to the server as opposed to each and every other client. The server then becomes responsible for sending messages between users. The main disadvantage of a server-based approach is the additional delay caused by the server's processing of the incoming events, which sometimes makes the server the bottleneck of the system.

There are many advantages in using RM for the implementation of the communication module. RM technology uses substantially less bandwidth and produces lower delays than a TCP-client-server based approach. But the disadvantages of the RM approach are practical ones, not theoretical ones. All RM technologies available are based on UDP multicast. Multicast support in today's Internet is far from acceptable and in fact today's Internet leaves a lot to be desired when it comes to multicasting. The M-bone tries to compensate this lack of adequate support, but even then a great portion of a multicast session's traffic are "tunneled" through the non-multicast portions of the Internet. In addition, joining and maintaining M-bone connectivity is not a trivial task; one cannot expect ordinary users to easily connect to the M-bone. Furthermore, RM technology is still maturing and standards are still being made. As a result there are many incompatible RM implementations today but mostly for research/experimentation purposes [14].

With JASMINE, our main goal is to create a system that can be accessed by the most number of people and in fact that's why we chose the Java technology. From practical

stand-point, it makes little sense to implement the communication module in such a way that most ordinary Internet users won't be able to use it, or will have to go through a great deal of connection set-up just to use our system. We therefore decided to implement the communication module with a TCP-client-server design. As we will see later in the performance evaluation section, the performance of this design is quite adequate for small to medium-sized group collaboration sessions.

2.2.1 JASMINE Server

JASMINE uses a multithreaded server, where the main server launches a sub-server for each user joining the session. The sub-server is responsible for processing only the update messages or requests coming in from its own client. Once the sub-server receives the update message, it will send it to all other clients in the session (figure 6). This will create a fast system response, at the expense of more resources utilized due to sub-server threads. However, usually only one client at a time can control and interact with an application (due to floor control as we will see), and most threads will simply be waiting and won't consume too much resources.

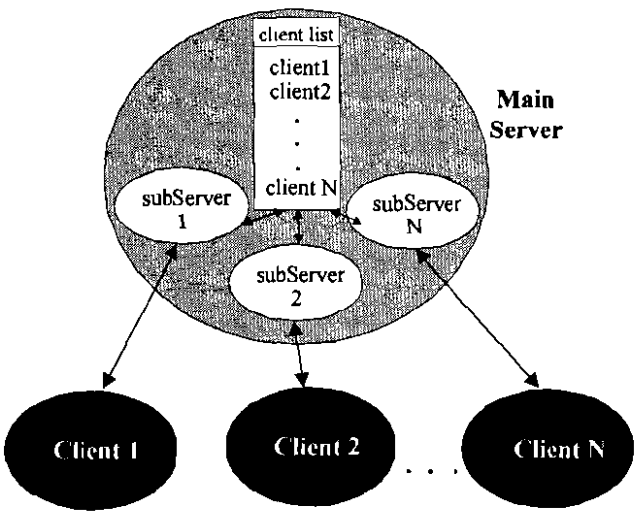


Figure 6. JASMINE's client-server architecture.

The server's main job is to propagate the incoming events from a user to all other users. But it also provides other services, which are necessary for maintaining a collaboration session. It provides services for session moderation and management, floor control, and data exchange. Data exchange is of particular importance for multimedia sessions as we will see next.

2.2.2 Advanced Multimedia Applications

As discussed in the literature, a pure transparent collaboration system is not sufficient for multimedia applications [12]. This fact is due to specific services that are required by multimedia applications such as synchronization, quality of service, etc. For example, think of a collaboration session where a video applet is being played. When one user presses the pause button, simply capturing the "pause event' and sending it to all other clients is not sufficient because when other clients receive the pause event and apply it to their video player, at each client the video player will pause on a different frame and clients will not be synchronized. Hence there is a need to send control messages between clients, such as "pause on frame number 57" to maintain consistency among all users. The JASMINE server provides a high-level API that can be used for these type of advanced requirements. However, an application must specifically use the API to take advantage of these functionalities, hence the transparent feature of the system is somewhat diminished.

3. Implementation

Figure 7 illustrates a sample screenshot of a typical JASMINE session. It shows the client's Collaboration Manager and some shared applets and applications running in the session.

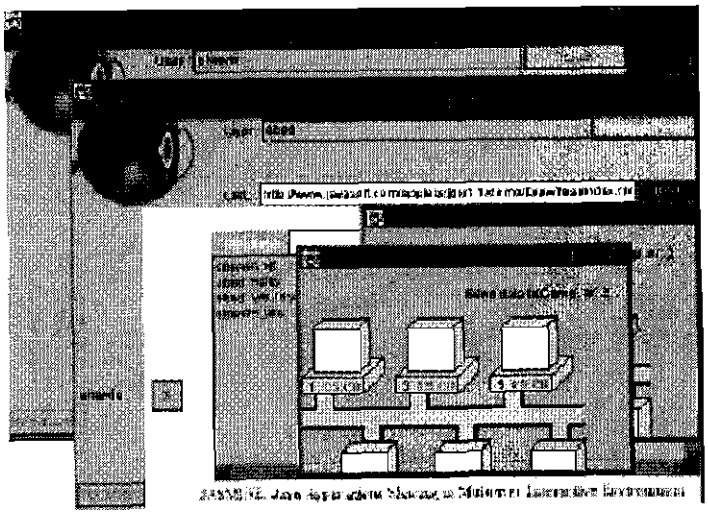


Figure 7. A sample JASMINE session.

In this section we will present our implementation. We start by introducing the Collaboration Browser.

3.1 Collaboration Browser

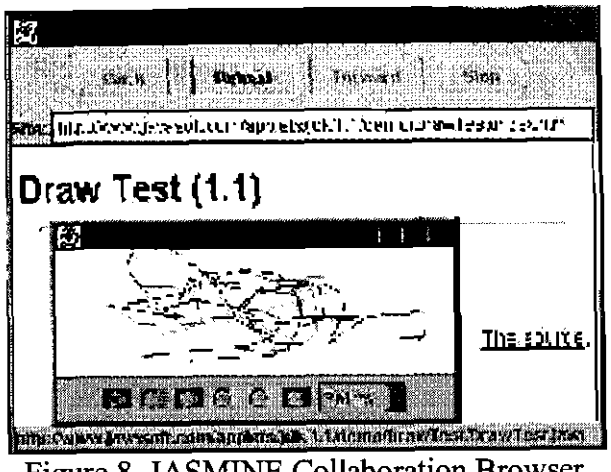


Figure 8. JASMINE Collaboration Browser.

Figure 8 shows the screen shot of the collaboration browser. As mentioned before the shared applets and applications are collaboration-unaware applications developed using the standard Java technology. These applications can be loaded dynamically, after a client joins a session, or can be downloaded on the fly during the session by typing in the URL of the desired applet. In the first case a specific number of preloaded applications/applets that

a user can invoke must be stored in a configuration file before the start of the session and are loaded by the JASMINE-Client at the start-up stage. In the second case, participants can just type the URL of the applet they want to bring into the environment to share. JASMINE then fetches the applet and inserts it into each client's session. There is no limit as to the number of simultaneous applets/applications running in the session.

We have successfully tested our system on a number of applets implemented in JDK 1.1, using normal AWT or Swing components. We did however encounter a limitation for *Frames*, *Dialogs* and *FileDialogs*. These components, when created within the application at run time, cannot be used collaboratively since they are not registered by the Component Adapter explicitly. In other words, collaboration is only possible for the first level windows. To overcome this problem, we developed a collaboration class loader, incorporated in JASMINE as the collaboration browser. The development of such a class loader gives the browser the capability of tracking all components it loads and hence it is able to control *Frames*, *Dialogs* and *FileDialogs* created within the shared applet at run time.

3.2 Configuration file

Information about locally available applications and applets, which can be used in a collaborative way, are read from a configuration file. The configuration file, which is organized as a properties file, contains the names of the applications/applets, which will be presented in the menu and the full names of their main class or URL. The entries have the following syntax:

application.[n].name = [name]

application.[n].class = [class]

where:

n: number of the application in the list.

name: a suitable name for the application to be shown in the menu.

class: full name of the main class.

An example configuration file is illustrated in Figure 9:

```
#Application entries
application.1.name=myTestApplication
application.1.class=kom.develop.apps.MyApp
# Applet entries
applet.1.name=myTestApplet
applet.1.class=kom.develop.applets.TestApplet
# URL entries
url.1.name= TestUrl
url.1.address=http://desiered.server/test.html
```

Figure 9: Excerpt from a configuration file.

Before starting the session, applets and applications that are thought to be useful can be placed in this configuration file. Additional applets and applications can be brought into the session live as needed.

3.3 Floor Control

A collaborative system must address many issues such as synchronization, latecomers, management or moderation, floor control, and awareness [12]. Among these, floor control is perhaps the most primary issue without which a collaborative session won't function properly. In short, floor control ensures that only one person at a time controls the shared application. Without floor control, there will be collisions of events, which leads to unwanted results in the shared application.

In JASMINE, floor control is achieved by means of locking. Each application has a corresponding *semaphore* on the server. When a user wants to interact with the shared

application, the system first locks the application by locking a semaphore. At this point, any other users trying to interact with the application will be denied access. When the first user is finished, the system releases the semaphore and others can take control of the application.

For a specific shared application, most developers prefer an "intuitive" implementation of the floor control capability; i.e., as soon as the user tries to interact with the application, the client automatically asks for floor control and allows or disallows its user to interact. After the user is finished, the client releases the lock automatically. Figure 10 shows sample Java code that demonstrates how the floor control is used in an intuitive way. This approach is in contrast to the "direct" approach, where a client must specifically ask for control, for example by pressing a "control-request" button.

```
public void mouseDragged(MouseEvent e) {
  //user is dragging the mouse, so ask for control
  if (getControl()==true) {
    // do whatever must be done for a mouse drag
    releaseControl();
  }
  else displayMessage("Aceess Denied!");
}
```

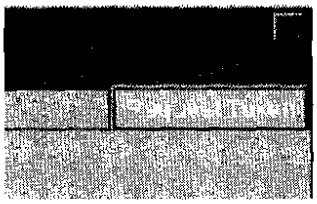
Figure 10. Intuitive floor control.

Just how intuitive the approach in figure 10 really is depends on the system response. If there is a small delay between the time the user tries to interact and the time when something actually happens on the screen, the application is intuitive. If however that delay is large, the application becomes "unnatural". So the Floor Control Delay (FCD) is an interesting parameter that we must also evaluate.

3.4 Moderation

Although floor control addresses the issue of event collisions, it works on a first-come-

first-serve basis. This in turn leads to the possibility of a participant to abuse or disrupt the session by feeding unwanted events into the session. There is therefore a need to have a moderator in order for a session to be more productive, for example, a teacher moderating a distance learning session. The moderator is usually the person who calls for a collaborative session and starts the server. In JASMINE we have two types of sessions: moderated, and non-moderated. The server can be started by specifying a login name and password for the moderator. Once the session starts, the moderator can login at any time and take control of the session. When the session is moderated, no one can send any events to the server. A participant wishing to do so must ask for permission from the moderator as shown in figure 11a. The moderator will subsequently receive a message indicating the participant's request to interact (figure 11b) which the moderator can allow or refuse. Upon moderator's acceptance of the user's request, the user will receive a green light, which indicates that he or she can now send events to the session (figure 11c). The moderator can also dynamically "cut off" a user's permission to interact if needed (figure 11d). In JASMINE, we allow only one user at a time to have permission to send events, although this number can be increased based on the application.



a

b

c d

Figure 11. Moderation capabilities in JASMINE.

4. Performance Evaluation

JASMINE can be considered a real-time tool in the sense that its updating response time, in a network environment capable of supporting real-time applications, is within the acceptable parameters of human quality of service for desktop collaboration, as we shall see. But as with any TCP based multiuser system, there is an upper-bound to the number of simultaneous users before those parameters are violated. This "maximum users" limit depends on the resources utilized by the system, such as processing power, graphics power, memory, network bandwidth and network delay, as well as the design of the communication part of the system.

Depending on the quality desired, the application level end-to-end delay between two users should be less than 1000 milliseconds, with 200 milliseconds recommended for tightly-synchronized tasks [11]. However, these numbers are valid only if the shared application is used in conjunction with some type of media that provide a sense of presence such as video and audio. The reason is that if audio or video or both are present, users have a sense of "awareness" of each other, which in turn requires the shared application to respond within a time that maintains that awareness. For example, imagine

three engineers who are collaboratively designing a bridge in a live session. One of them highlights a section of the bridge and says: "I think this part should be redesigned". If they are using real-time audio conferencing (end-to-end audio delay of 100 msec), then the delay of the shared application must comply with the above numbers in order for the other two engineers to receive the audio message and the event update in such a way as to maintain the real-time quality of the session. This is usually the case in controlled IP environments such as local networks or corporate IP networks.

In the case of typical Internet connections, where audio and video delays are not controllable, or in the absence of audio or video, restrict delay parameters make little sense because the users have no time-wise perception of one another. In such instances, when a user receives an update message, the user has no way of knowing when an actual action occurred. So, even a delay of 5 seconds or more might be acceptable depending on the nature of the application under such circumstances.

Our performance evaluations are done for a controllable environment, where real-time characteristics are required and can be supported.

4.1 Parameters of Interest

The most common parameter that measures the quality of a collaborative application is the Client-to-Client Delay (CCD). CCD tries to measure the average time it takes for an update message to reach other users. It includes all layers between the two clients, including application, transport, networking, and physical layer delays. However, at the application level, it only measures the time it takes for a sender to send or a receiver to receive the update at the application layer. It does not include the delay caused by what the application does with the update because that is application-dependent. For example,

when a line is drawn in a shared whiteboard, CCD measures the average delay from the time the sender application assembles and sends the update message until the time the receiver receives the message and extracts the data from it, just before it makes a graphics call to actually draw the line. Hence, for an overall delay, one must also add the average on-the-screen drawing time, referred to as Rendering Delay, which really depends on the capabilities of the graphics, memory and processing power of various client machines and therefore not constant for all clients. As another example, if one user opens an image in a whiteboard, what we measure is how long it takes for the "open-image" message to reach all clients. We don't measure how long it takes for the receivers to actually download the image from a given URL and show it on their screen, because we can't control those delays and they are not related to the collaboration system shown in figure 1.

As mentioned earlier, the server processing time per packet increases with increasing number of simultaneous users. This is due to one-to-one TCP connection-oriented nature of the system; the server needs to send the update info to each client one by one. This Server Processing Delay (SPD) adds to the overall end-to-end delay of the system and must be taken into account when calculating maximum number of users supported by the system.

As mentioned in section 3.3, Another interesting parameter is the Floor Control Delay (FCD). This is the average time for a user to take control or be denied taking control of an application and measures how intuitive a system is. A system with a smaller FCD is more "natural" and behaves more naturally than a system with a larger FCD.

4.2 Testing and Results

We tested CCD, SPD, and FCD of JASMINE over both local area network (LAN) and

telephone modem access. During the testing, all machines were running their usual background processes related to the network and the operating system. The testing configuration is shown in figure 12.

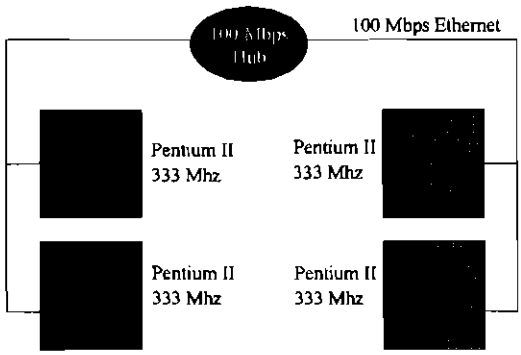


Figure 12. Network configuration for LAN tests.

All machines were running JDK 1.2 on Windows NT 4.0 Workstation. In addition, two 133 Mhz Pentium machine running Windows 95 were used to dial-in into the LAN with 28.8 kbps modems over phone lines. The result of the tests are shown next.

4.2.1 CCD Test

For the CCD test, we had a "sender" applet send an event to a "receiver" applet. Upon receiving the event, the receiver applet extracts all necessary data from the packet, reassembles the event, and sends the event back to the sender. The sender does the same thing and resends the event, and so on. This is repeated for a given duration, which was 10 minutes in our tests. The result of this test was an average CCD of 150 msec on the LAN, and 370 msec between the clients behind 28.8 kbps modem. It is worth mentioning that the transmission delay of the very first event took 750 msec and 2.5 sec on the LAN and modem, respectively. We believe this to be attributed to the Just-in-Time compiler (JIT) utility of JDK 1.2 which compiles the interpreted bytecode of a given method into native code, the first time that method is called, causing a one-time-only larger than usual delay.

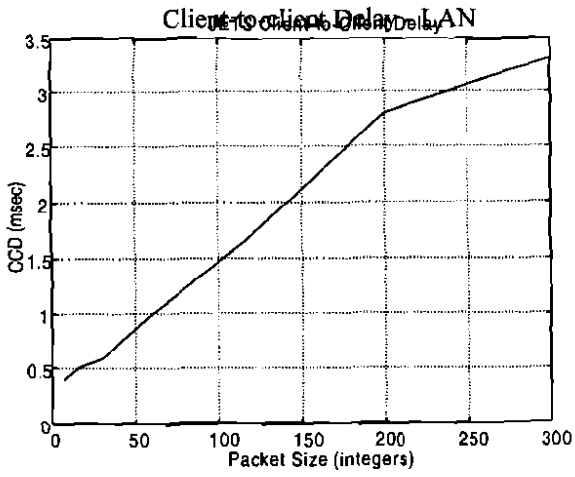


Figure 13. JASMINE CCD results (packet-based).

As argued in section 2.2.2, the system must also be able to send data between clients in addition to the event updates. It is interesting to know the delay of sending such data. We therefore repeated the CCD test for data exchange, this time for data different packet sizes. The result is shown is figure 13. The packet size is measured in number of integers sent per packet. Even though it is very unlikely that a synchronization or control message of size 500 integers is sent in one packet, we did extend our test to that limit to see the effect of very large update messages. Figure 14 shows the same test performed over 28.8 Kbps modem access instead of 100 Mbps Ethernet.

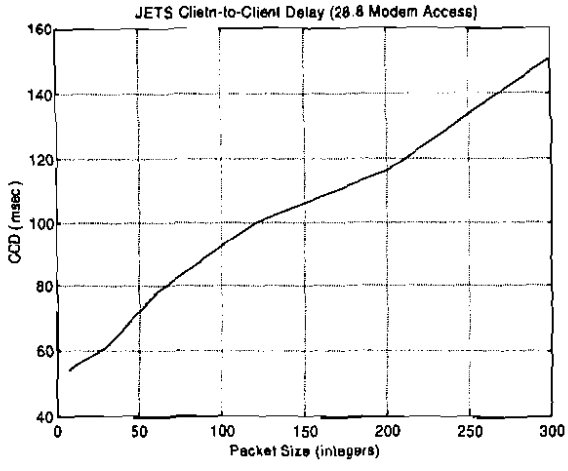


Figure 14. JASMINE CCD results over modem line (packet-based).

4.2.2 SPD Test

For the SPD test, we had the sender applet flood the server with event updates. Then we had the receivers (up to 45) calculate the average delay between receiving adjacent packets from the server. As expected, this delay increases with increasing number of users as seen in figure 15. Figure 16 shows the same test performed for data updates. Note that due to floor control and moderation, no more that one client at a time can send events to the server, a scenario, which is typical of collaborative applications.

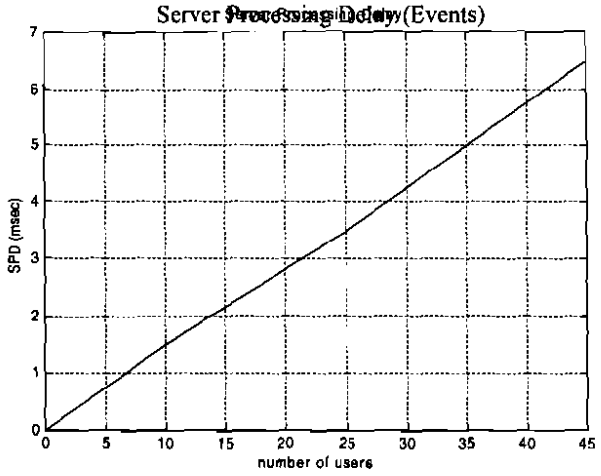


Figure 15. JASMINE Server Processing Delay.

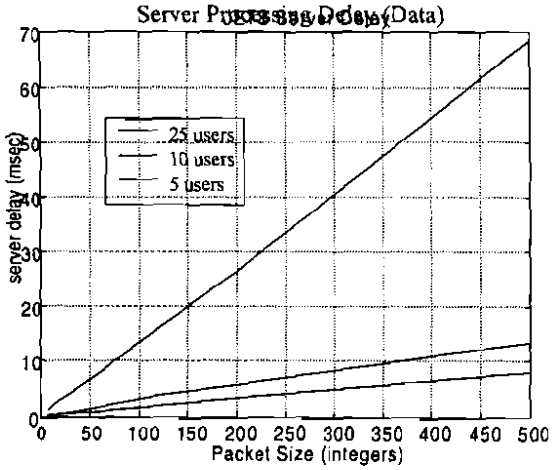


Figure 16. JASMINE Server Processing Delay (data).

We can see that the delay increases linearly. This is due to the fact that the server spends equal amount of processing time per packet per client; therefore it increases linearly with increasing number of clients.

4.2.3 FCD Test

For the Floor Control Delay, we had a client constantly ask for control, and release it upon receipt, for a given amount of time. The average FCD turned out to be less than 5 msec, which affirms the intuitiveness of the floor-control mechanism of the system.

4.3 Subjective evaluation

We also tested a few applets, including a typical whiteboard application, as seen in figure 7, with up to 5 users sitting next to each other and able to see one another's screens. The applications responded in a natural manner in terms of the feel and interaction/perception of the whiteboard. The visual updating delay between the screens of the workstations was very small yet detectable by the naked eye.

4.4 Analysis

As mentioned before, the recommended overall end-to-end delay is less than 1000 msec, with less than 200 msec required for closely-coupled collaboration. This delay includes the CCD, the SPD, and the on-screen rendering/display delay corresponding to the application's GUI. As argued previously, the rendering delay (RD) is not constant and it depends on the hardware/OS/platform used.

From the CCD and SPD tests, we can approximate the overall delay as:

delay = CCD + SPD + RD;

from figure 15: SPD ≈ 0.142* N, where N is the number of users;

hence:

$$delay \approx CCD + 0.142^{*} N + RD$$

which roughly represents the delay experienced from the time an event is generated due to a client's interaction until that interaction is shown on the screen of all other clients. Figures 17 shows achievable number of users based on the expected overall delay, for different rendering delays (RD).

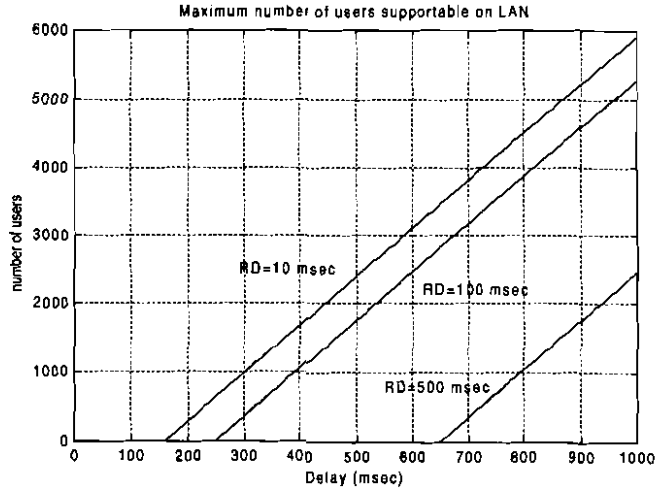


Figure 17. Number of users supported by the system.

Figures 18 shows the same thing with focus on tightly-synchronized tasks (delay<200 msec).

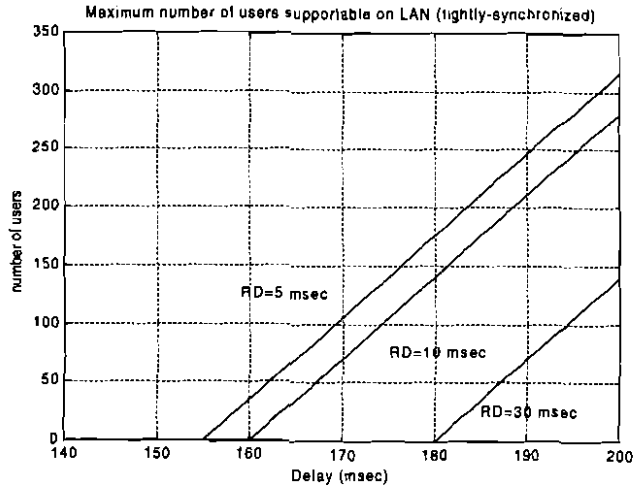


Figure 18. Number of users supported by the system (delay < 200 msec).

Finally, figure 19 illustrates number of users supportable with 28.8 Kbps modem access.

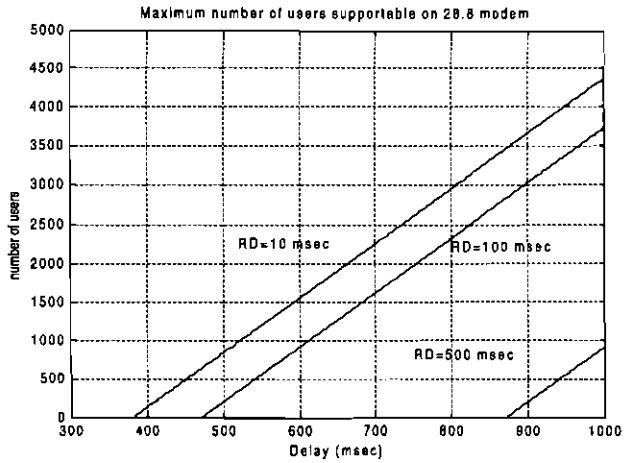


Figure 19. Number of users supported by the system (modem access).

By looking at the above graphs, we can conclude that the system can support "many" users. Even though the plots suggest that theoretically thousands of users can be supported, the fact is that the actual number of users supportable is less. The reason is that the linear behavior of the system diminishes as the number of users increase: the performance of the machine running the server decreases substantially as we approach the limit of maximum allowable socket connections on the machine, also the underlying physical network becomes slower with increasing number of users. So the hardware/OS of the server machine and the network either cannot support so many simultaneous users, or their performance decreases significantly. Nevertheless, this shows that the underlying communication module of JASMINE can support small-size and medium-sized collaboration sessions of hundreds of users, resource permitting.

5. Related Work

There are many Java-based collaboration systems, none of which offer a management or moderation feature similar to ours. Kuhmünch [10] at the University of Mannheim developed a Java Remote Control Tool, which allows the control and synchronization of

distributed Java applications and applets. The drawback of this approach is that it is necessary to have access to the original source code of the application or applets in order to make it collaborative. That means every applet must initiate a Remote-Control-Client object, which is usually done in the constructor of the applet. Also the event handling within the applet must be modified in order to receive and/ or send events from / to remote applets. The Java Shared Data Toolkit (JSDT) from JavaSoft is also an API-based framework [15]. Habanero [1] is an approach that supports the development of collaborative environments. Habanero is in its terms a framework that helps developers to create shared applications, either by developing a new one from scratch or by altering an existing single-user application which has to be modified to integrate the new collaborative functionality. Instead of using applets, which can be embedded in almost every browser, the Habanero system uses so-called "Happlets" which need a proprietary browser to be downloaded and installed on the client site. Java Collaborative Environment (JCE) has been developed at the National Institute of Standards and Technology (NIST) coming up with an extended version of the Java-AWT [2] called Collaborative AWT (C-AWT). In this approach AWT-components must be replaced by the corresponding C-AWT components [3].

All these approaches propose the use of an API, which has the cost of modifying the source-code of an application, re-implementing it or to design and implement a new application from scratch in order to make it collaborative.

Java Applets Made Multiuser (JAMM) [8] is a system, which is similar to our approach in terms of its objective: the transparent collaboration of single-user applications. The difference between JAMM and JASMINE is the way collaboration is achieved. In JAMM

[6], the set of applications that can be shared is constrained to those that are developed using Swing user interface components as part of Java Foundation Classes, which are part of the standard JDK since version 1.2. JAMM's set of applications is furthermore restricted to those which implement the Java serializable interface.

Conclusion

We presented the architecture and implementation of our transparent collaboration framework for Java applets and applications. We developed this architecture in order for users to be able to collaborate via collaborative-unaware applications and applets without modifying the source code. Our architecture enables us to use almost all single-user applets and applications in a collaborative way. We have successfully tested our system on a number of applets. We also observed that using the TCP-client-server approach of our communication module can support relatively large number of users. However, when reliable multicasting becomes more practical in the future, it would be more logical to replace the current communication module with one which is RM based.

There are two outstanding issues remaining. These issues are not directly related to JASMINE but are research areas of the transparent collaboration paradigm. The first issue is that of latecomer-support. When a user starts a session later than other participants, there is a need to bring this user up-to-date as opposed to start from scratch. This can be achieved either by sending the entire object state of the shared application to the newcomer using object serialization, or by sending all the events occurred up to now to the new user so that it follows the same sequence of events that other participants have gone through [12]. We're currently using JASMINE to experiment with these methods.

Another issue was brought up in 2.2.2: multimedia inter-client synchronization and control. Transparent collaboration cannot address this issue alone and we believe that using an API is necessary to achieve such functionality for multimedia applications.

Today, computing environments where Java applications and applets are running over IP have become very popular and widespread. Our architecture helps people to collaborate in such environments easier.

Acknowledgments

The authors acknowledge the financial assistance of the Volkswagen Stifftung, Germany, as well as the Telelearning Network of Centers of Excellence Canada (TL-NCE) and the Natural Sciences and Engineering Research Council Canada (NSERC).

References

[1] A. Chabert et al, , "Java Object Sharing in Habanero", Communications of the ACM, Volume 41, No. 6, June 1998, pp. 69-76.

[2] H. Abdel-Wahab et al "An Internet Collaborative environment for Sharing Java Applications" IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems (FTDCS'97), October 29 - 31, 1997, pp. 112-117.

[3] H. Abdel-Wahab et al, *"Using Java for Multimedia Collaborative Applications"* Proc. PROMS'96, Madrid, Spain, 1996.

[4] *Handheld IP Connectivity for 1998*, IEEE Internet Computing, Vol. 2, No. 1, January/February 1998, pp. 12-14.

[5] International Data Corporation, "IDC's Forecast of the Worldwide Information

Appliance Marketplace 1996-2001", IDC Bulletin #w15080, December 1997, (screen phone revisions 5/7/98).

[6] Abdulmotaleb El Saddik, Oguzhan Karaduman, Stephan Fischer, and Ralf Steinmetz. "Collaborative Working with Stand-Alone Applets". In Proc. of the 12th International Symposium on Intelligent Multimedia and Distance Education (ISIMADE'99), August 1999.

[7] J. Begole et al, "Leveraging Java Applets: Toward Collaboration Transparency in Java", IEEE Internet Computing, March-April 1997, pp. 57-64.

[8] J. Begole et al, "Transparent Sharing of Java Applets: A Replicated Approach". Proc. Symposium on User Interface Software and Technology, ACM Press, NY, 1997, pp. 55-64.

[9] J. Grudin, "Computer-Supported Cooperative Work: History and Focus", IEEE Computer, Vol. 27, No. 5, May 1994, pp. 19-26.

[10] Kuhmünch et al, "Java Teachware - The Java Remote Control Tool and its Applications", Proc. ED-MEDIA'98, 1998.

[11] Multimedia Communication Forum Inc., "Multimedia Communication Quality of Service", MMCF document MMCF/95-010, Approved Rev 1.0, September 24, 1995.

[12] S. Shirmohammadi et al, "Applet-Based Telecollaboration: A Network-Centric Approach", IEEE Multimedia, Vol. 5, No. 2, April-June 1998, pp. 64-73.

[13] Stephan Fischer and Abdulmotaleb El Saddik, Open Java: Von den Grundlagen zu den Anwendungen. Springer-Verlag, ISBN: 3540654461 (1999).

[14] K. Obraczka, "Multicast Transport Protcols: A Survey and Taxonomy", IEEE Communications, Vol. 36, No. 1, 1998, pp. 94-102.

URL:

[15] Javasoft (for Java, JINI, RMI, and JSDT technologies) http://www.javasoft.com