Applet-Based Telecollaboration: A Network-Centric Approach

Shervin Shirmohammadi, Jauvane C. de Oliveira, and Nicolas D. Georganas
University of Ottawa, Canada

Real-time collaboration systems, in which participants share multimedia documents and applications, have attracted interest for many years. The JETS system provides a generic multimedia telecollaboration framework that enables sharing of Java applets through the Internet. Experimentation with JETS revealed practical design and implementation issues, as well as the essential requirements of such systems.

The introduction of Java as a platform-independent programming language for the Internet has significantly affected related technologies. Embedding Java applets in HTML documents means applications now go to users in the form of applets instead of users having to find and install them. Users only need a Java-enabled platform, such as a Web browser, to access these applications.

This Web-based approach is sometimes referred to as *network-centric computing*—users gain access to "computing" by simply connecting to a network and clicking on the correct link. They can then run any application provided by the network without having that application preinstalled on their machines, worrying about application level platforms, or considering hardware compatibility. Superficial analysis of such a computing method indicates that this approach has many advantages over desktop-centric computing. Users can transparently launch applications from a computer, a network station, or even a Java-enabled Web television in a just-as-needed, just-in-time fashion. Also, the management and maintenance of overall computing infrastructures will become simpler and cheaper. System administrators will upgrade or manage applications on a few servers as opposed to each and every desktop machine in their system.

Today, many experts consider Java Internet computing's de facto programming language. Java's Web accessibility and platform-independence—its most important properties—set it apart from other type of programs. We assume that most readers are familiar with Java applets and refrain from discussing them further.

Almost parallel to the Java paradigm, Web-based telecollaboration has also received much recent attention. Although computer supported cooperative work (CSCW) systems have existed for a long time,[1] Web-based collaboration tools that permit sharing multimedia applications among participants on the Internet have emerged only recently. Microsoft NetMeeting, Vocaltec ICP, and Netscape Collaborator, for example, are also equipped with audioconferencing capabilities. Some Java-based collaboration tools available, like the NCSA Habanero and the Java Collaborative Environment (JCE), allow sharing of basic multimedia applications. However, few of these tools use applets as their collaborative applications base.

Although applets can usually be shared and converted to applications, a fundamental difference exists between sharing applets and applications: In practice, applets can be shared through a Java-enabled Web browser or a network station. Not only platform-independent, this method also requires no downloading or installation of any specific software or application by the user. If a user navigates to a URL and joins a session using new applets, the browser downloads them from the server. Furthermore, users will always have the latest version of the applet—not the case with applications. Stand-alone applications cannot be shared using a Web-browser. This means that a package must be downloaded and installed on every user's machine taking part in a collaboration session—an approach very similar to non-Java applications using collaboration. In short, Java application sharing emphasizes Java's portability, whereas Java applet sharing takes a more network-centric approach.

There are two main schools of thought for designing telecollaboration sessions, or any multi-user environments for that matter. One approach uses a central server to pass information between clients as well as provide specific services. The other uses a noncentralized system and eliminates the need for a server. In this article, we present a client-server architecture for sharing Java applets. We also discuss collaboration issues that we came across while designing our Java-Enabled Telecollaboration System (JETS) prototype and propose solutions based on our experience with that prototype.

1070-986X/98/$10.00 © 1998 IEEE

Collaboration architecture

Server-based and server-less collaboration have certain trade-offs: server-based has simpler clients, avoids complex distributed algorithms, and easier implementation, while server-less has a potential increase in the maximum number of clients and avoids an extra entity performing server duties. In the case of Java applets, using a centralized approach proves more practical than using a fully distributed architecture.

From a network-centric point of view, remember that client machines have to download the Java applets from a central server. This means that a server already exists, integral to the system. Furthermore, as part of Java's built-in security restrictions, applets only make network connections back to the server from which they came and can only manipulate files on that server. Although new releases of Java have more flexibility, letting the user give more permissions to a Java applet, it is recommended writing applets that can work in any environment. Hence, using a server provides many benefits. In addition, using a server-based approach solves many implementation issues more easily than an approach without a central server.

Implementation issues

Figure 1 shows a simple sketch of a client-server system. The identical applets run on different clients. When a user interacts with the applet, clicking a button, drawing a line on the screen, or rotating a 3D object generates one or more events. Each event produced at the client is sent to the server, which multicasts it to all other clients. The other clients then process the event as if it were generated locally and thus recreate the original client's intended actions. This event broadcasting method has proven a better approach than display broadcasting, where sharing results from graphically displaying the applications with which the originating user interacts.[2]

Two types of servers run on the central machine: a Web server that simply sends HTML documents with embedded applets to requesting clients and an application server to which the applets establish connection to telecollaborate. As a convention, the term *server* in this article refers to the second type unless otherwise stated. Many issues arise that a multiuser collaboration environment such as that in Figure 1 must address, mainly latecomers, participant awareness, management, event collision, and synchronization.

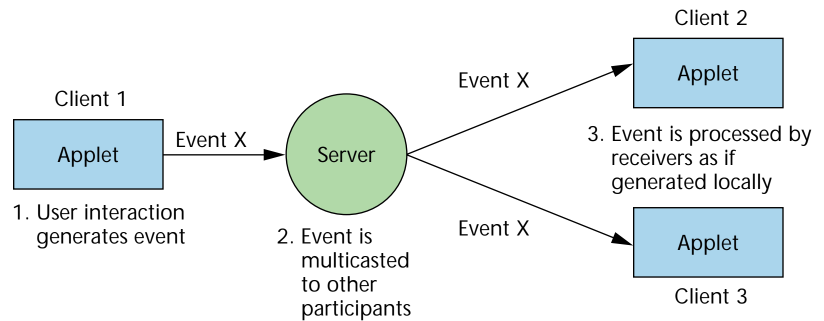


Figure 1. A simple client-server architecture for event multicasting.

Latecomers

A user joining a session already in progress must obtain the current states of the shared applications. Using a central server, the newcomer can receive these states in two ways. One is to play back the sequence of events—event logging.[3] This not only requires huge buffers, but also might be unstable because of an expired entity (a data file being erased at some point). The other approach keeps track of the applications' current object state and sends them to the newcomer.[4] Java's *object serialization* can easily achieve this by transparently sending objects with their current state over the network.[5] However, this requires the server to multicast user events, keep a copy of the application object, and execute the received events to keep the application's state current.

Another interesting issue now arises. What about user events occurring while the state is being initialized to run in the application or be sent to the newcomer? Two approaches can accommodate these events. One locks the application while a latecomer joins so that no events will be accepted by the application until the newcomer has successfully joined. The drawback is that it could take a long time for the newcomer to join, causing interruption in the session. Another approach uses event logging to buffer events from when the application state has been sent to the latecomer until the latecomer is ready to participate in the session. At this point, the buffered events are sent to the application state to keep it fresh. The disadvantage with this approach is the need for a buffer at the server for every latecomer.

Event collision

Event collision is not a new issue. It has existed in database management systems (DBMS) for many years. The problem occurs when the events of two or more users—generated at approximately the same time—affect the same data and create unwanted results. In an on-line presentation, one

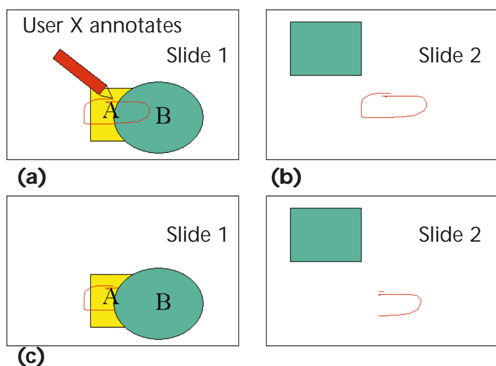


Figure 2. Event collision (a) what user X intended, (b) what actually happened because another user changed the slide, (c) another scenario of what happened: half of the annotation is on slide 1, another half on slide 2.

user might advance the presentation to the next slide while another tries to annotate the previous slide. As a result, the annotation might appear on the new slide or half of it might appear on the previous slide with the other half on the new slide. As demonstrated in Figure 2, only one of these events should have been allowed.

Some type of access control usually addresses event collision. Optimistic techniques, such as validation, usually don't work in situations where many users in the session have a high degree of interactivity.[6] In validation, checking for collisions occurs initially and after executing operations. The assumption in optimistic approaches is that not many user interactions occur at the same time. Although sometimes true, this is often not the case with telecollaboration sessions. Pessimistic techniques, which perform a certain degree of checking before executing events, seem more appropriate for real-time telecollaboration. One of the common pessimistic approaches is locking. When a client wants to interact with an applet, the applet checks with the central server to see if the shared application is currently available. If so, it locks the application to deny other clients availability and performs its operations. After finishing, it releases the lock so others can use the application. This approach works on a first-come, first-serve basis.

Awareness

Awareness is the ability of a given participant in a collaborative session to feel the existence and actions of others. According to the literature, as many as 11 different elements provide this "feeling":[7]

1. Presence: relates to who occupies the workspace
2. Location: addresses where they are working
3. Activity level: measures activity of other participants
4. Actions: what they are doing
5. Intentions: what they will do next
6. Changes: concerns what and where changes have been made
7. Objects: shows what objects they are using
8. Extents: what they can see and how far they can go
9. Abilities: what they can do
10. Influence: where they can make changes
11. Expectations: what the current participant will do next

This issue has attracted interest for many years in teleconferencing systems, virtual reality, and 3D simulations. In teleconferencing systems, video is used to watch the other participants, to be aware of their existence and actions. Awareness in VR and 3D simulation systems usually comes through an avatar that tries to mimic the behaviors and actions of its user. In non-3D environments, such as telecollaboration sessions, other techniques are required.

To achieve awareness, a system can use explicit mechanisms, such as direct communication, or indirect ones, such as simple observation of others' work by noticing the effect of their actions. The use of a telepointer, like in Microsoft's NetMeeting, can improve this awareness. When a user moves the mouse over an application, the other participants screens reflect the movement. This gives all users a clear perspective of elements 1 to 8 defined above.

However, with many users, this method needs some enhancements. Let's say 50 users are all allowed to interact with the application at the same time; naturally, there will be too many pointers on the screen to follow. A simple solution for this problem, the widely used locking mechanism, allows only one user at a time to have access to the application.[8]

Again, a central server proves beneficial for this feature. The server can track information all participants and relay to a newcomer, notify others when a user leaves the session, and manage the locking mechanism.

Management

Often overlooked by existing implementations, a session manager proves a key component of a functional telecollaboration system. Smooth functioning requires a chairperson who controls partic-

ipants' access rights for different applications. This person can dynamically set individual users' access rights such as *no-access*, *view-only*, and *view/interact*. For instance, whenever users abuse their access rights by controlling an application longer than allowed, the chairperson can intervene by changing the user's access rights. Without this management system, a noncooperative participant can spoil a session. Although not many, some collaboration and conferencing systems implement session management. The Telemídia Videoconferencing System (TVS), for example, implements a management system controlling participants' access rights both manually and automatically.[9]

A central server can assist implementation of this issue. The session manager can predetermine users' access rights. These rights could be categorized into generic rights such as *guest*, *member*, and *assistant*, or specific rights such as *John Smith*. The access rights can then be associated with the client applets by specifying them as parameters in the applets' HTML code, or by sending them from the server to the client when the applet is first initialized. Every time a client tries to perform a certain interaction, the applet first checks locally to see if the user has access rights. If not, an "access denied" error message is returned; otherwise, the user action executes. During the session, when a manager changes these access rights, the server sends these changes to the appropriate client applet.

Synchronization

Another important aspect of collaboration is synchronization. Typically, individual users in the same session have different processing powers and different network access in terms of network bandwidth and latency. This creates the possibility that the state of one copy of the application differs from the state of another copy. Lack of synchronization will lead to both temporal and application state inconsistencies. For example, a user with high-bandwidth access might quickly bring up an image from a server and edit it while another user has not even finished downloading the image.

Hence, a synchronization scheme is crucial in a collaboration space. Such a synchronization scheme must support timeliness, causality, and state awareness to insure consistency among participants—not a trivial task.[10] In the above example, you could implement a mechanism so that no one can start editing until all participants have downloaded the image. Obviously, this will not be very efficient in a case where every user has a 1.5-megabit per second (Mbps) network access except for one participant who has a 28.8-kilobit per second (Kbps) network access. Another approach is to have the editing process of the faster stations buffered and played back for a slower station once the image has been downloaded. However, this approach will not give users of slower stations a fair chance to participate in the editing process. A central server could be used in both of the above approaches for different tasks. These tasks could include tracking image downloading, buffering the editing process, and playing it back for the slower stations.

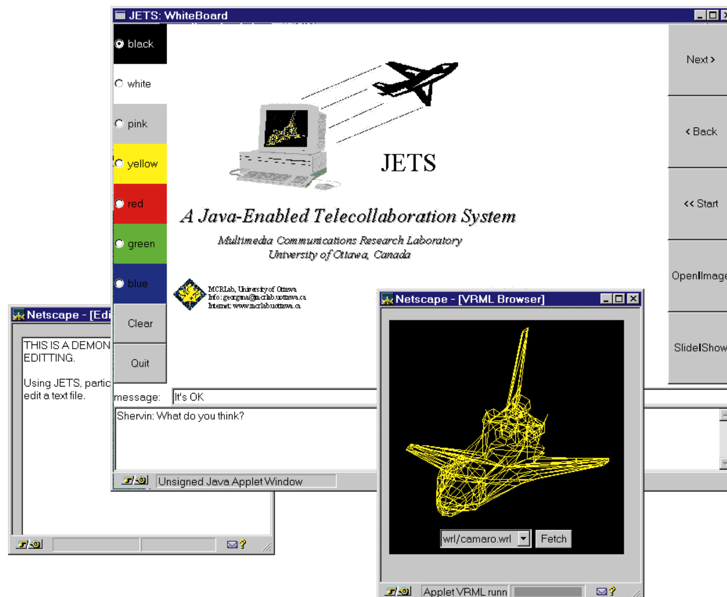
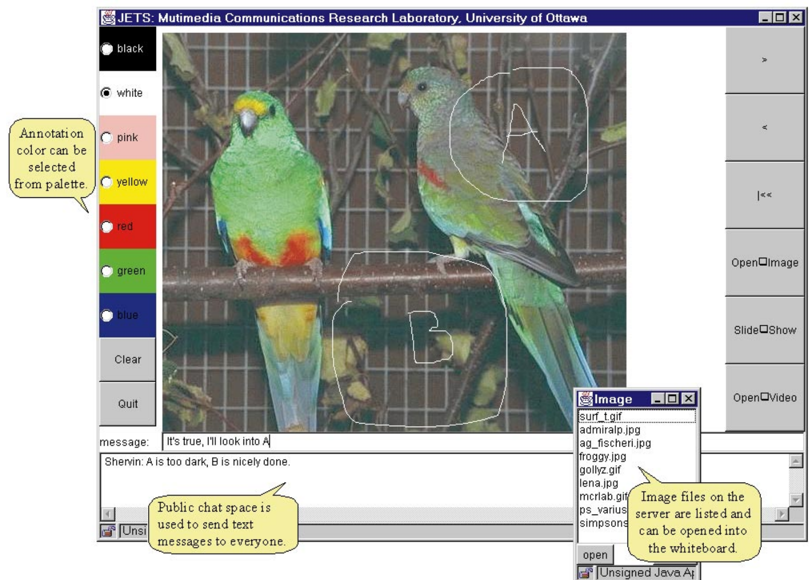


Figure 3. Screen shot of JETS running in the Netscape Navigator.

Prototype implementation

To implement the above collaboration techniques, we developed the JETS system. JETS is a groupware toolkit, written fully in Java, that offers an API developers can use to create multimedia applets for collaboration. The API consists of mostly server side classes that provide the necessary functionality and services for collaboration. Except for synchronization, JETS supports all of the issues previously described to some extent. In essence, JETS uses the client-server architecture shown in Figure 1. It uses a series of semaphores on the server to provide locking and also provides support for latecomers through object state updates sent to the new client by the server. In addition to these services, JETS includes some basic Java applets specifically designed for collaboration. Figure 3 shows a typical JETS session running in the Netscape Navigator browser.

Figure 4. The whiteboard and its features.



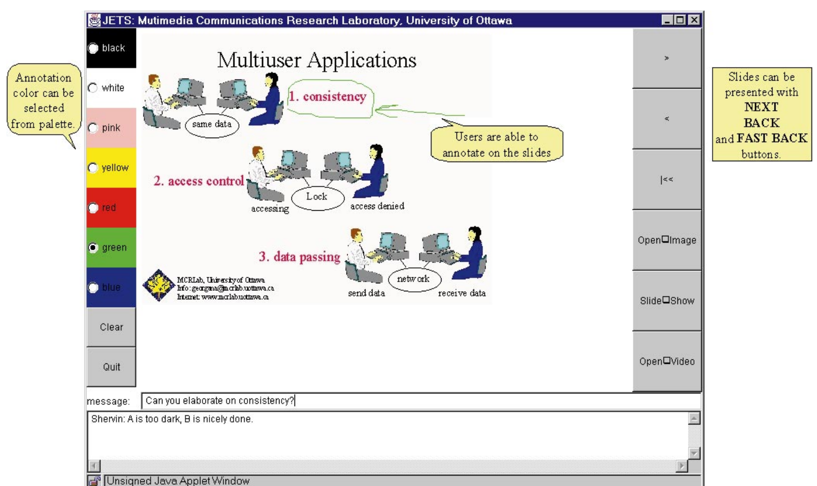


Figure 5. Presentation slides running in the whiteboard.

Whiteboard

Besides serving as a window for shared color drawings, the whiteboard can bring up images in JPEG or GIF format from the Web server and display them. Users can annotate these images and start a discussion. The built-in locking mechanism of JETS forbids simultaneous modification of an object by more than one user. As seen in Figure 4, the whiteboard facilitates user conversations by having a shared chat space participants can use to exchange textual massages— particularly useful where audio access is not available.

Figure 5 shows the whiteboard as an online presentation tool capable of displaying slides brought from the Web server. These can be PowerPoint slides saved in HTML format or simple sequences of images. Participants can annotate these slides in the same way as the images.

Shared video

A very useful feature of JETS is its ability to play International Telecommunications Union Standard (ITU-T) H.263 compliant video in the whiteboard.[11] The video must be stored on the Web server to work correctly. When a user opens and starts playing a video file, the video data streams down to all participants, decoded in real time (processor permitting), and displayed in their whiteboard. Users can also play the video frame by frame as well as annotate a frozen frame (shown in Figure 6). When we first decided to write the H.263 video decoder in Java, we expect-

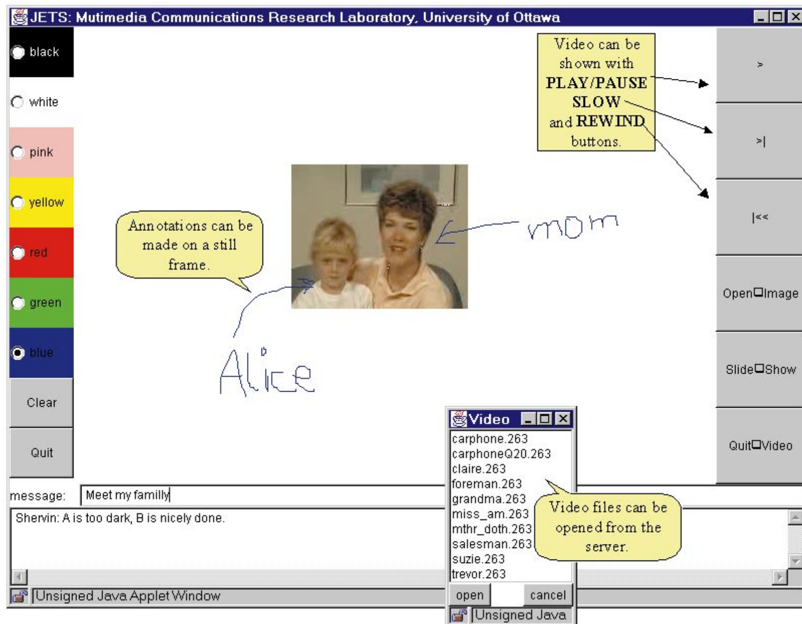


Figure 6. H.263 shared video displayed in the whiteboard.

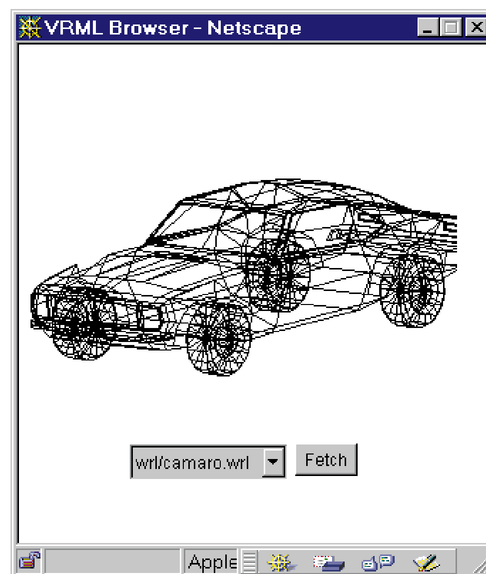


Figure 7. VRML viewer displaying a wireframe model.

ed slow performance because of Java's performance problems. However, we obtained up to 25 frames per second (fps) for a quarter common intermediate format (QCIF) video with the whiteboard running in Netscape Communicator 4.04. (We ran the test on a 200-MHz Pentium Pro with 64 Mbytes of RAM running Windows NT 4.0.)

VRML viewer

Another applet, a simple shared 3D viewer for VRML files, permits real-time collaborative interaction with simple VRML objects. The applet brings VRML 1.0 files from the server and displays them in wireframe mode (Figure 7). A user can then collaboratively interact with the 3D scene—with all the rotations, moving, and zooming reflected on all participants' screens. The JETS VRML server tracks the object state, making it possible for latecomers to be updated with the state of the current session in progress.

Session management

We also implemented a management system to enable session monitoring. Like other JETS applets, the management applets connect to a server program on the central server. However, the management applets are not shared and don't use event multicasting. Each client has a session bar that displays the client's access rights for each application. When a client first tries to join the session, a login menu will ask the client's user name and password for the session (Figure 8a, next page).

Using the session chair applet, the chairperson can extract specific information about participants. This information includes how many users have logged in, who has what type of access to which application, and so on

On the session bar, colored buttons indicates the type of access for each application: red for *no access*, yellow for *view-only* access, and green for *view/interact* access. The client has no access to an application with a red button. This can change on user's request with the chairperson's approval. In

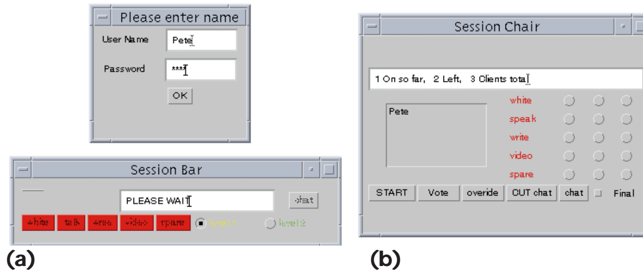


Figure 8. Before the collaboration session. (a) Client "Pete" tries to join the session. (b) The session chair sees that Pete has joined, as well as how many preinvited people remain.

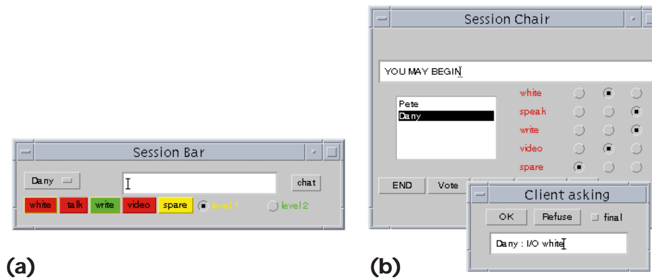


Figure 9. During the conference: (a) Colored buttons indicate a participant's access rights to applications. (b) The chairperson can respond to a participant's access request.

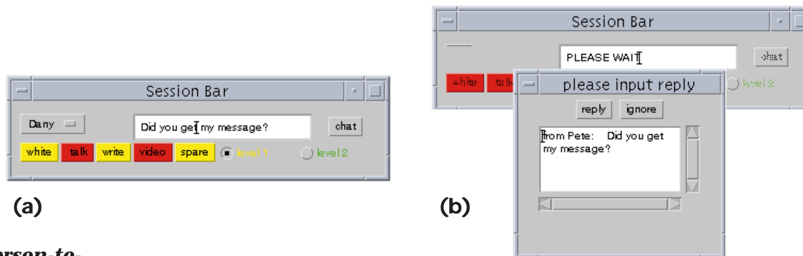


Figure 10. Person-to-person chatting. (a) A user can send messages to specific individuals in the session. (b) The receiver of a message can reply or ignore it.

Figure 9, when user "Dany" requests view access to the white board, the request goes to the chairperson. The chairperson can then grant or refuse the access request.

The chairperson can also change a participant's access rights dynamically during the session without the participant's request. Furthermore, the chairperson has the ability to finalize a participant's access rights so that they won't be able to continually send requests for access.

Each user is aware of the existence of other participants. The session bar comes with a drop-down list that shows the current participants' names—reflecting the "presence" component of awareness. JETS implements items 1, 3, 4, and 6 required for awareness through the participant list and multicasting of user actions. In addition, items 8, 9, and 10 are implemented for the chairperson. Since the management system is not a shared applet system, a user can chat with a specific person in the session. Figure 10 shows a person-to-person chat in progress.

Another feature of the session management system is the ability to vote on a subject. The chairperson can permit participants to vote on a subject indicated on the whiteboard's chat utility. Users vote by pressing yes or no on a vote pop-up menu, with the results reported to the chairperson.

Implementation problems

The main problem we encountered during implementation was event multicasting. We first tried to intercept the generated events in the *handleEvent()* of the applet. If caught, the event can easily be sent to the server as discussed earlier. However, we noticed inconsistencies in the applet's event handling. Although many events were intercepted and received by other clients without any problem, some events such as typing in a text field were not handled properly by the receiving clients. We believed this to be a Java Abstract Window Toolkit (AWT) implementation problem by either the Web browser or the Java Development Kit (JDK) itself. Later, we came across a paper by Begole et al. that confirmed the problem and proposed solutions that must be considered by JavaSoft itself.[2] For JETS, we emulate event multicasting by using a *message multicasting* system architecture where applets, upon user interaction, send predefined messages to other clients through the server.[4] Since these messages are predefined, the receiver applets can tell what events have occurred.

Since the developer must design a predefined messaging scheme for each applet, this creates a problem. Consequently, this shortcoming must be resolved by JavaSoft to allow efficient applet sharing.

Performance analysis

We tested JETS on both local-area (LAN) and wide-area (WAN) networks for both asynchronous transfer mode (ATM) and regular Internet connections to evaluate system delay and response terms. We designed a test applet to measure the effective client-to-client delay (CCD) of the system. CCD is an indication of the average time it takes for a byte of data to travel from one client to another. It also includes the delays caused by all protocol levels, such as the application level, the transport level,

and the network level. We compared the CCD results with quality of service (QoS) parameters for Multimedia Desktop Collaboration (MDC) published by the Multimedia Communications Forum.[12] On LANs and ATM WANs, the JETS system has acceptable quality in terms of user interaction and human perceptions. Typical CCD values fall in the 50 to 60 millisecond range.[4]

Related work

As mentioned earlier, other Java collaboration systems have emerged: the National Center for Supercomputing Applications (NCSA) Habanero, the Java Collaboration Environment (JCE),[13] and Java Applets Made Multiuser (JAMM).[14] Among these, JAMM more closely compares with JETS in terms of its objective—it tries to achieve collaboration using Java applets rather than Java applications.

The main difference between JAMM and JETS is their collaboration enabling architectures. JAMM uses transparent collaboration—events are automatically intercepted and sent to the other copies by the collaboration engine. A developer writes a single-user applet and won't have to know that the applet will be a collaborative one. This requires modification of the core JDK, which is what the developers of JAMM have done. Hence, in order to run JAMM, you must download and install a modified version of the JDK. On the other hand, JETS uses the standard JDK and can run in any Java-enabled Web browser without modifications. It does, however, require the developer to handle the events once intercepted (as mentioned earlier).

Even though a transparent collaboration system makes it easier for the developer to write typical applets, it doesn't always work for the case of multimedia applets. Consider the following scenario for a shared video player. If we assume the video data streams from the server to every client, the difference in access parameters such as network bandwidth and delay will give different users different views of the video stream. If someone presses the pause button during play, simply intercepting the pause event and sending it to all applets is not enough. When other applets receive the pause event, they will not necessarily freeze on the same frame as the originating one. In fact, most likely different users will see a different frame paused, since everyone uses an independent version of the video player on their machines.

To avoid this problem, you must specify the frame on which pause was pressed. In other words, additional information must be sent together with the event. As a result, pure transparent collaboration techniques will not suffice in such instances. Alternatively, a secondary system with its own communication channels between the clients could handle this problem. However, this will create redundancy, since a communication infrastructure for event passing already exists. Ideally, this should be enough to pass messages among all copies of the applet. Subsequently, a collaboration system must offer either an optional message passing mechanism or make the underlying communication channels available to the developer. JETS uses the second solution.

One important aspect of JETS is its management system. The JCE system also has a management component—Session Control Manager—that lets users join and leave a session, as well as request and release a lock.[13]

Using Java for multimedia

Besides functioning as a prototype to implement and analyze our design, JETS allowed us to gain practical experience with Java. We were able to observe Java's competency as both a platform and a programming language for implementing collaborative multimedia applets. Before concluding this article, we would like to share our observations.

Most developers know about the advantages and disadvantages of Java as a programming framework. Java's platform independence, full object orientation, portability, and network support balance against its slow performance and security restrictions. We found that from a multimedia system perspective, Java has many useful features.

One of the most useful features is the ability to easily fetch a GIF or JPEG image from a given URL on the Internet and display it within an applet, without worrying about details such as file transfer, image decoding, and display. This is the most important functionality used in our whiteboard. In addition to supporting URLs, the networking APIs also provide some very useful features to easily create a server that listens for an incoming client and to establish a data channel between the server and the client for the transfer of information. All of these tasks can be achieved with relatively few lines of code in Java.

URLs

JAMM (Java Applets Made Multiuser): http://simon.cs.vt.edu/JAMM/

Javasoft (for JMF and PersonalJava): http://www.javasoft.com

JCE (Java Collaborative Environment): http://snad.ncsl.nist.gov/madvtg/Java/Java.html

JETS (Java-Enabled Telecollaboration System): http://www.mcrlab.uottawa.ca/jets

NCSA Habanero: http://www.ncsa.uiuc.edu/SDG/Software/Habanero/

The Object Serialization utility is another useful feature. It allows programs to send and receive, over the network, objects with their current state. The process of marshaling and unmarshaling objects is transparent to the developer. As explained before, this is very useful in supporting latecomers to a session already in progress.

In addition to the problem encountered during event multicasting, another deficiency of Java applets is their lack of built-in support to open and save files on the server. This is important since applet security restrictions allow no local file reading or writing. On the other hand, files on the server can be read using file transfer and written using the server program. However, no built-in mechanism exists in Java that allows a developer to easily display the file list on the server machine, open a file, or save a file on the server. The developer must write code to do these tasks. This is an important issue for multimedia applets because of the frequent need to display images, presentations, video and audio clips, and other multimedia content stored in files.

Another promising Java technology for multimedia is the Java Media Framework (JMF) currently under development. JMF provides a platform-independent infrastructure for synchronization, control, processing, and presentation of both stored and streaming media. The eventual goal of JMF is to provide APIs for Player, Capture, and Conferencing functionalities, even though only Player APIs are currently available. In addition to lying outside the core Java API, JMF's main shortcoming is that current JMF Player implementations have native platform dependencies for most of the low-level processing. Intel's JMF implementation for Windows 95 and Windows NT uses Microsoft's DirectShow, for instance, but Silicon Graphics' JMF implementation uses its own Digital Media libraries. This means a media format might be supported on one platform but have no player on another platform. Since we needed system deployment across all platforms, we did not use JMF in our prototype. However, we are in the process of making our H.263 decoder compatible with JMF so that it can be integrated in any Java environment supporting JMF.

Concluding remarks

Although fully distributed approaches could be used to implement many of the issues discussed, the inherent properties of Java applets make a client-server approach more practical. These properties include a server, which is part of the system to begin with, plus the security restrictions such as communication channels—only allowed between the applet and the server—and file manipulation limitations. However, as the Java technology progresses and legal mechanisms for bypassing applet restrictions become more mature, distributed approaches will become more practical.

One of the main issues left open is synchronization between clients, or interclient synchronization—very important for multimedia collaborative applets. Unlike intraclient synchronization, which ensures synchronization between multiple media on a single client, interclient synchronization addresses synchronizing media between multiple clients. As illustrated earlier in the video decoder example, when one client presses the pause button during the video playback, the video on all other clients must freeze on the exact same frame. Taking the pause-initiating client as the reference, some mechanism must be developed to accommodate the clients who lag behind and those who have already moved ahead in the video presentation. Furthermore, this mechanism must work in a real-time fashion. This argument can be extended to other multimedia applications as well.

If the current network-centric computing trend continues, technologies such as PersonalJava will allow access to Java applets, not only on traditional computing equipment such as PCs, but also on Web-connected consumer devices for home, office, and mobile use. The need to write multimedia applets in Java will then be justified not only by its platform independence and portability, but also by its accessibility and the rapidly growing number of multimedia applets.

JETS is a working example of an applet-based multimedia collaboration system even though still in the primitive stages of development. Despite the performance deficiencies pointed to by Java critics, JETS successfully demonstrates that Java collaboration frameworks are not only possible, but can perform at acceptable levels of quality if designed correctly. **MM**

Acknowledgments

We thank Pierre Desmarais for implementing the session management system. We also acknowledge the financial assistance of the Telelearning Network of Centers of Excellence Canada (TL-NCE), the Ontario Graduate Scholarship Program, and the Brazilian Ministry of Education Agency's CAPES scholarship.

References

1. J. Grudin, "Computer-Supported Cooperative Work: History and Focus," *Computer*, Vol. 27, No. 5, May 1994, pp. 19-26.
2. J. Begole, C. Struble and C. Shaffer, "Leveraging Java Applets: Toward Collaboration Transparency in Java," *IEEE Internet Computing*, Vol. 1, No. 2, Mar.-Apr. 1997, pp. 57-64.
3. O. Kim et al., "Issues in Platform-Independent Support for Multimedia Desktop Conferencing and Application Sharing," *Proc. Seventh IFIP Conf. on High Performance Networking (HPN'97)*, Chapman & Hall, London, 1997, pp. 115-139.
4. S. Shirmohammadi and N.D. Georganas, "JETS: A Java-Enabled Telecollaboration System," *Proc. IEEE ICMCS*, IEEE Computer Society Press, Los Alamitos, Calif., 1997, pp. 541-547.
5. T.B. Downing, *RMI: Developing Distributed Java Applications With Remote Method Invocation and Object Serialization*, IDG Books Worldwide, Foster City, Calif., 1998.
6. R. Elmasri and S.B. Navathe, *Fundamentals of Database Systems*, 2nd ed., Benjamin/Cummings Publishing Company, Redwood City, Calif., 1994, pp. 555-572.
7. C. Gutwin and S. Greenberg, "Workspace Awareness for Groupware," *Proc. ACM Computer-Human Interface (CHI '96)*, ACM Press, New York, 1996, pp. 208-209.
8. H.P. Dommel and J.J. Aceves, "Floor Control for Multimedia Conferencing and Collaboration," *ACM Multimedia Systems*, Vol. 5, No. 1, 1997, pp. 23-38.
9. J.C. Oliveira, *TVS—A Videoconferencing System*, Master's dissertation (in Portuguese), Computer Science Dept., Pontifical Catholic University of Rio de Janeiro, Brazil, Aug. 1996.
10. W. Robbins and N.D. Georganas, "Shared Media Space Coordination: Mixed Mode Synchrony in Collaborative Multimedia Environments," *Proc. IEEE ICMCS*, IEEE Computer Society Press, Los Alamitos, Calif., 1997, pp. 466-473.
11. K. Rijkse, "H.263: Video Coding for Low-Bit-Rate Communication," *IEEE Comm.*, Vol. 34, No. 12, Dec. 1996, pp. 42-45.
12. Multimedia Communication Forum, "Multimedia Communication Quality of Service," MMCF document MMCF/95-010, Approved Rev 1.0, Sept. 24, 1995.
13. 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)*, IEEE Computer Society Press, Los Alamitos, Calif., 1997, pp. 112-117.
14. J. Begole et al., "Transparent Sharing of Java Applets: A Replicated Approach," *Proc. 97 Symp. User Interface Software and Technology (UIST'97)*, ACM Press, New York, 1997, pp. 55-64.

Shervin Shirmohammadi is currently pursuing a PhD at the University of Ottawa under an NSERC (National Sciences and Engineering Research Council of Canada) scholarship. He received his BASc and MASc degrees in electrical engineering at the University of Ottawa. His main research interests are telecommunication software, multimedia communications, and collaborative virtual environments. See http://www.mcrlab.uottawa.ca/~shervin.

Jauvane C. de Oliveira is currently pursuing his PhD at the University of Ottawa. He received his BS degree in computer science from the Federal University of Ceara, Brazil, and his MS degree in computer science from the Pontifical Catholic University of Rio de Janeiro, Brazil. He is under CAPES scholarship from the Brazilian Ministry of Education. His main research interests include teleconferencing systems, telecollaboration, real-time applications, and VR. See http://www.mcrlab.uottawa.ca/~jauvane.

Nicolas D. Georganas is a professor at the School of Information Technology and Engineering, University of Ottawa. He received his Dipl.Ing. in electrical engineering from the National Technical University of Athens, Greece, and his PhD in electrical engineering Summa cum Laude from the University of Ottawa. His research interests are multimedia communications and collaborative virtual environments. He is a Fellow of the IEEE, the Engineering Institute of Canada, the Canadian Academy of Engineering, and the Royal Society of Canada. See http://www.mcrlab.uottawa.ca/~georgana.

Readers can contact Shirmohammadi at the Multimedia Communications Research Laboratory, School of Information Technology and Engineering, University of Ottawa, 161 Louis Pasteur Priv., Ottawa Ontario K1N 6N5, Canada, e-mail shervin@mcrlab.uottawa.ca.