The Jupiter Audio/Video Architecture: Secure Multimedia in Network Places

Pavel Curtis, Michael Dixon, Ron Frederick, and David A. Nichols
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304 USA
+1 415 812 4000
{pavel,mdixon,frederick,nichols}@parc.xerox.com

ABSTRACT
Jupiter is a multi-user, multimedia virtual world intended to support remote collaboration, including support for shared documents, shared tools, and audio/video communication. In addition, it provides a rich set of end-user customization and programming facilities, designed to support a community of developers with a wide range both of skills and of trust in each other.

This combination of features leads to an interesting tension in the system's design, between flexibility and privacy: how can we make it easy for "power users" to build applications that control the distribution and recording of the real-time audio and video streams in new and useful ways, while simultaneously assuring *all* users that they understand and control where their own A/V streams are going at any moment?

In this paper we describe our solution to this dilemma; it combines the automatic use of encryption, a flexible model for dynamically interconnecting media sources and sinks, and user-interface widgets that can be relied upon to accurately represent and control the workstation's audio/video capabilities.

KEYWORDS
Network places, collaboration, audio, video, multicast, security, encryption.

INTRODUCTION
Jupiter is a system supporting long-term collaboration through the use of *network places* [5]. Each Jupiter user runs a client program on a workstation, preferably equipped with video camera and microphone, and connects to a Jupiter server that maintains a persistent virtual world of network places.

Compared with virtual reality systems, the world itself is simple, represented with only two-dimensional maps and textual descriptions. The emphasis instead is on interaction, with the other users currently connected and with the objects that they jointly construct and leave behind.

The Jupiter world is composed of a set of interconnected places that serve to structure these interactions: to a first approximation, each user sees and interacts with the other users and objects currently located in the same network place as themselves. By moving themselves and the objects from place to place, they can flexibly and naturally control the extent of their participation in the collaboration.

The objects in Jupiter can include simple text documents and drawing surfaces, general tools like web browsers and tape recorders, tools designed for specific types of work like data analysis, and even active agents that move about the Jupiter world and interact with other objects and users. Jupiter provides tools for creating new objects, creating new places, and defining or modifying the behavior of objects and places with a simple but powerful object-oriented programming language.

The Jupiter server consists of a database that maintains the state of the world and the objects in it, an interpreter for the internal programming language, and a core set of objects that provide the basic functionality of network places, communication facilities, and programming tools.

An important design assumption is that users adding new capabilities to Jupiter might not be trusted by other Jupiter users, so we require an unusually fine-grained security model that allows users to maintain control of their own objects and places, but at the same time share and build on each others' programming efforts when desired.

The Jupiter programming environment has made it easy for users to extend the world in many ways; our goal in the current work was to design audio/video facilities with comparable flexibility, so that novel or customized media connection applications could be implemented easily by end users. Furthermore, such extensions should not require reimplementing basic facilities like the management and distribution of network addresses and encryption keys.

These user-written applications should not be able to compromise other users' control or understanding of their own audio/video access. For example, we would like to prevent

- a programmer creating an object that allows them to see or hear what's going on in another room without the occupants of that room knowing it,
- a programmer creating a new communication tool that, while seemingly secure within Jupiter, is actually transmitting the media streams unencrypted and is therefore vulnerable to external tapping,
- user 1 playing back a (possibly edited) recording of user 2, making other people present believe that user 2 is actually speaking, or
- a programmer creating an object that allows unsolicited broadcast to everyone else in the system.

The previous prototype of Jupiter supported much of what we would like, but often in an *ad hoc* or inflexible way. After having used the system for several months, we decided to completely redesign the facilities for audio/video control, coordination, and security; this paper presents our new design.

In what follows, we first lay out the range of applications we had specifically in mind as design targets. Next, we cover two other major influences on the design: the technical infrastructure underlying Jupiter and the security and trust considerations. At the core of the paper is a presentation of the user and programming interfaces in the new design, followed by a brief discussion of how this design enables the implementation of several key applications.

SOME MOTIVATING APPLICATIONS

The design of the Jupiter audio/video architecture was influenced in large part by the set of sample applications we had in mind. A major criterion in our deliberations was the ease with which new applications could be prototyped by users with little knowledge of the low-level details of audio/video capture, processing, or transmission. In this section we describe many of these applications in order to give the reader a sense of the breadth we considered. Late in the paper, we revisit a number of the applications to explore how well we've addressed our ease-of-use goals.

- *Intra-room communication*
  By far the most common use of audio and video in Jupiter is for communicating with the other users who are in the same network place, or *room*. The model is that when one user speaks, all of the others in the room can hear them; if any of the users have cameras, all of the others there are able to see their image.
- *Private comments to a single user*
  It often happens that a user would like to speak to just one of the others in the current room, not to all present; we refer to this as *whispering.*
- *Non-speech sounds in rooms*
  Other, non-speech audio communications can be helpful in giving network places more of the affordances of physical places. For example, it should be possible for a room to generate a sound, audible to the users in that room, whenever someone enters or leaves. Another example is ambient, background sounds, which might be capable of subtly giving different places different "feels."
- *Hearing sounds from "nearby" rooms*
  It often happens in physical places that the sound of conversation in one room will draw in participants from the nearby spaces; this kind of casual, unplanned interaction is an important mechanism for improving the overall level of communication in a group. If the presentation of network places to users includes enough information to make clear how far conversation is travelling, it would be useful and should be straightforward to allow this kind of inter-room audio leakage.
- *Connecting physical and virtual places*
  The activities in some physical places, such as conference rooms and auditoriums, could be of interest to remote users. It should be easy to set up virtual analogs of these places, allowing users who are not physically present to see, hear, and even actively participate in meetings and presentations.
- *Non-local audio/video tunnels*
  While network places provide a rich metaphor for a wide range of interactions, there are times when users need other communications channels not tied to any virtual geography. For example, some pairs of users who often interact with each other also often need to work in different network places on essentially different immediate projects. Such pairs should be able to easily set up an A/V 'tunnel' between them, allowing them to stay in communication even as they each move about in the virtual space [1].
- *Workgroup media spaces*
  The same basic concept scales up to larger groups as well. For example, the hardware group in PARC's computer science lab comprises about ten people usually working on several distinct projects at any given time. While their individual work keeps the group members spread apart in both physical and virtual space, they would like to be able to stay casually in touch with each other as a group; they, too, should be able to easily set up a group A/V "space" to meet this need [3].
- *Public address system*
  At the limit of this notion, some communities might wish to allow certain distinguished users to make announcements that are seen and heard by everyone.
- *Virtual video telephones*
  The two-person A/V tunnel concept is similar to a video telephone call that is never disconnected. One could also provide virtual videophones in their full generality, with users able to specify on a call-by-call basis the other endpoint of the tunnel and to cause a receiving window to appear on the callee's screen at call initiation.
- *Virtual answering machines*
  Once there were virtual telephones, the virtual equivalent of an answering machine would be useful, providing an asynchronous complement to the other applications discussed.

- *Connecting to the real world phone system*
  Given appropriate hardware, it should be straightforward to hook up any virtual telephone system to the one in the real world [11].
- *Audio/video recording and playback*
  As is suggested by the answering machine example, not all participants in audio/video communications are human users. One obvious non-user application is a “tape recorder.” With the assistance of a simple auxiliary server that can transmit and receive A/V traffic, it should be easy to implement an object that, when located in some network place, can record the open communications in that place. Once such a recording has been made, of course, it should again be straightforward to play it back later to any authorized listeners. Extra facilities might usefully allow users to insert notes and other markings tied to specific locations in the recording, to aid in later indexing or review [8].
- *External audio/video sources*
  Virtual analogs of physical places is just one example of making an external audio/video source available to Jupiter users. Others might include broadcast radio and television signals or physical security cameras.
- *Hourly news recording service*
  In the radio case, we have found it useful to automatically record the news reports broadcast at the top of every hour, making the latest news always easily available to our users.
- *Application-specific audible alerts*
  Another class of non-user media sources includes such applications as mail readers, status monitors, and games; for the most part, these applications are not concerned with audio/video transmissions, but they may, from time to time, wish to alert the user to some change in their state with a simple sound effect.

We had already implemented versions of most of these applications in the previous Jupiter prototype, but in several cases they relied on *ad hoc*, specialized mechanisms; they were thus somewhat fragile and poorly integrated with one another. Our desire to clean up that situation and to enable the rest of these applications was a major reason for our decision to redesign the media coordination system from scratch.

THE JUPITER INFRASTRUCTURE

Several properties of the technical infrastructure underlying the Jupiter system have significantly influenced the design of the new audio/video architecture. In this section we describe enough of that infrastructure to bring out these design requirements.

Communications and networking

Jupiter has a very sharp client/server distinction, with major differences in the responsibilities and capabilities of the two components. Each user runs a client program on their local workstation that communicates across the network with a single, shared server.

The clients each initiate a single TCP connection to the server that stays up for periods ranging from minutes to months, depending on the user; users who disconnect may later reconnect at will. The server stays up, ready to accept new connections and to do its other processing, for arbitrarily long periods of time. The TCP connection is essentially the only means of communication between any client and the server. All communication on these TCP connections will soon be encrypted for privacy, using the proposed standard Secure Sockets Layer protocol, SSL [7].

The clients transmit and receive audio and video data using IP multicast [6] and the proposed standard Real-time Transport Protocol (RTP) [10]. All Jupiter audio and video traffic is encrypted for privacy.

IP multicast is a routing technology that inexpensively sends copies of network packets to every host in the entire Internet that has expressed interest in that packet’s specific destination address; the sender of such a packet can not, in general, control which hosts are allowed to receive it or even discover, after the fact, which ones did so.

> **Requirement:** We should not use the same multicast address for two transmissions to different sets of receivers.

Otherwise, receivers who were in one set and not the other would suffer a useless increase in local network load.

> **Requirement:** We should not use the same encryption key for two transmissions with different sets of intended participants.

This requirement applies even to incremental changes, such as growing or shrinking a set by a single member. When the set grows, we cannot reveal the old key to the new member because they may have recorded old transmissions that they were not meant to receive. When the set shrinks, the key must again change to keep the removed member from deciphering any further transmissions.

Encryption and decryption are not computationally free operations; we would like to minimize the number of cryptographic operations clients must perform. In particular, they should be able to successfully decrypt incoming packets without resorting to trial and error. Fortunately, the two previous requirements together imply that the client can determine which key to use solely from the destination address of the packet.

There are, on a number of client platforms, significant costs associated with each separate network address the client must listen to.

> **Requirement:** We should minimize the number of addresses that clients must handle; in particular, we should usually delay assigning addresses until they are actually needed.

In contrast to the clients, the main server never transmits or receives any multicast data. In this regard, it acts only as a coordinator and key generator, notifying the other parties of the appropriate multicast addresses and keys to use for their own out-of-band communications. Note that cryptographically secure key generation is computationally expensive.

> **Requirement:** We should minimize the number of keys that the server must create; in particular, where possible, we should delay generating keys until they are actually needed.

Jupiter Clients

The Jupiter client programs (of which there are separate versions for UNIX systems, Macintoshes, and PCs) are primarily responsible for managing the local details of the user interface. Guided mostly by commands from the server, the client displays windows on the user's screen, conveys information about user input actions back to the server, and sends and receives audio and video data via IP multicast. The client employs no knowledge about network places, their contents, their behavior, or even other users; rather, it is a more-or-less straightforward engine for manipulating the local machine's I/O devices under the direction of the server and the user. All of the semantics of the various actions it performs is carried in the server's code and the user's mind.

The server/client windowing protocol is relatively high-level and low-bandwidth, especially in comparison to X windows and similar systems [9]. Windows are described to the client as a tree of high-level widgets, drawn from a fixed set of about 20 defined in the protocol. The client implements these widgets in terms of native facilities on the platform, informing the server only of high-level user actions, such as pushing a button, releasing a slider, etc. Some widgets, as described later, directly support various aspects of audio/video communication.

Note that it may be computationally expensive for clients to capture, compress, and transmit audio and video data. Particularly in the case of video, which provides an essentially constant stream of new data, we would like to be able to avoid any of this processing during times when there were no active receivers.

> **Requirement:** Even if security considerations would permit others to receive transmissions, we should avoid preparing and sending them unless those transmissions would actually be used by someone.

The Jupiter Server

The main server is substantially more complex and general than the client. An instance of the LambdaMOO server software used in a number of MUDs (multi-user dungeons) on the Internet [4], it accepts and manages network connections, maintains an object-oriented database, and, often in response to user commands and client protocol messages, executes code stored in the database.

The database contains a variety of objects, including their methods and instance variables. Some of these database objects model so-called *world objects*: virtually 'tangible', user-visible objects like places, the things in those places, and even the individual users. The last of these, the objects modelling users, have particular importance in the A/V architecture; we call them *user objects* and discuss their role later in the paper.

Other objects in the database are essentially invisible to users, being programming libraries or other parts of the programming infrastructure.

All code for object methods is written in MOO, a simple interpreted language. Essentially all of the Jupiter-specific code in the server is in the database, written in MOO.

All aspects of the MOO database are mutable during execution; objects may be created or destroyed, methods and instance variables added, modified, or removed. Further, all users are allowed to perform any of these actions at any time, allowing them to add to and customize the system to a substantial degree.

We conceptually distinguish certain objects that implement very general, basic facilities, such as the server side of the window system's widget toolkit, from more specialized objects that implement specific tools or other uses of those facilities. When maintained by the system administrators, we call the first kind *core objects;* we refer to the other objects as *applications*. In general, applications are not maintained by the system administrators; any user can create them.

As a security measure, each MOO object, method, and instance variable is *owned* by some specified user and has associated permission settings that control the access allowed to all other users. Users and the code they own may not, in general, modify or destroy objects, methods, or instance variables owned by others and, in some cases as specified by the owner, may not even inspect them.

This multiplicity of, in general, mutually untrusting code authors leads to situations where a method owned by one user is called by a method owned by someone else. If the author of the callee is malicious or even just incompetent, this call might signal an error or never return, causing problems for the caller. Often, though, the caller does not need or expect any returned value; the call is just a one-way *notification* of some condition or event. We have added a facility to our database supporting asynchronous, one-way method calls; these provide a guaranteed safe way for callers to make such notifications.

THE SECURITY/TRUST MODEL

In our thinking about network places, we have not limited ourselves to scenarios where all of the users necessarily trust one another fully. We do expect that the vast majority of users will behave well in the vast majority of situations, but we wish to maintain our generality by not treating this expectation as an assumption. This section discusses the design requirements that this desire for generality imposes.

In our analysis of the security issues in Jupiter, we have identified five classes of *principals*, entities with distinguishable agency or intent:

- users,
- their Jupiter client software,

- objects maintained by users,
- objects maintained by the system administrators, and
- external eavesdroppers.

For the purpose of driving the design of the audio/video architecture, the key security issue revolves around trust. We need to know which principals must trust which others to keep what guarantees. The general applicability of our design is improved whenever we can reduce the level of trust required without also reducing the usefulness of the system.

Users must trust their own clients, like other software they run on their local machines, more-or-less absolutely: not to damage their local computational resources, not to deceive them about what local resources are being accessed in what way, etc. The client is the user's only representative to the rest of the system, the sole conduit for the user's expressions of intent. We therefore equate the client with the user, merging them into a single effective principal for the remainder of the analysis.

Users need not trust the server and its maintainers completely. Of course, the system maintainers can inspect, modify, or destroy anything "owned" by users in the database, so users must certainly trust them to the degree that they value what is stored there. We assume therefore that users only give information to the server that they trust it to maintain for them.

The system maintainers do not, *a priori*, have any control over the local resources of the user, including in particular the A/V capture and display devices on the user's machine.

> **Requirement:** The design should keep as much local device control as possible in the hands of the user; the server should not control, for example, whether or not a user's audio or video signals are transmitted to the network.

User-maintained applications embody, in effect, the intent of the user who created them; through the appearance and behavior of an application, authors make both tacit and explicit statements about what guarantees they offer to users. For example, consider the two-person A/V tunnel application described earlier. In addition to other, less explicit guarantees, it is represented as providing private communications between its two users.

Before a user can believe these statements, he or she must first trust the application's author to honestly present and correctly implement their intent. How easily such trust will be granted is a function of the reputations of the author and application, and of what users have at stake if that trust turns out to be misplaced.

The MOO object system and programming language make a fundamental security guarantee, that code written by one user cannot modify, destroy, or in some cases even inspect data owned by any other user.

> **Requirement:** Users of an application should only be putting at risk data that they explicitly make available.

Returning to the A/V tunnel example, a user should risk only whatever audio and video signals he or she explicitly enables the application to access; the application should not, in particular, somehow gain access to the user's other data. Given this, a user might reasonably be willing to use even a completely unknown (and therefore completely untrustworthy) tunnel application for communications that are not at all sensitive. If the application works at all according to its stated guarantees, the user will succeed in communicating with the intended party. At worst, no such communication will take place and the harmless A/V signals may have been distributed arbitrarily widely.

The code and objects maintained by the Jupiter system administrators are shared resources, providing services to all users, both individually and as a group. Clearly, the applicability and robustness of the system is greatly improved if these *core objects* are not required to trust the users.

> **Requirement:** Core objects should be protected against misbehavior by any user, client, or application. In addition, misbehavior by any one user should be prevented, to the extent possible, from reducing the usefulness of the system to others.

Finally, we consider the trust situation for user-maintained applications. Since the very essence of an application, the code and data in its objects, is totally under the control of the server (and thus of the system administrators), applications must fundamentally trust the server and core objects absolutely. Users and other applications, however, cannot be so relied upon; robust applications must protect themselves against misbehavior by these principals.

> **Requirement:** The same basic MOO security guarantees described for users should apply as well to applications; they should only be risking release of information that they explicitly reveal.

Of course, any information that a user presents to an application must in general also be treated as untrustworthy; this is rarely, however, a threat to the integrity of applications.

The final principal to consider is external eavesdroppers. By and large, the use of encryption on all Jupiter network traffic is sufficient to thwart such unauthorized listening. We do not further consider eavesdroppers as a threat.

AN OVERVIEW OF THE ARCHITECTURE

The earlier sections of the paper lay out in some detail the context for our redesign of the Jupiter audio/video architecture. In this section, we describe the new design in two parts. First, we describe four new user interface components that together provide users with the levels of information and control indicated by the earlier requirements analysis. We then give a high-level introduction to the key components of the programming interface provided to applications. The next section presents the details of the programming interface.

The User Interface

All transmission of a client's audio and video signals, and all audio and video output to a client's local devices, is tied

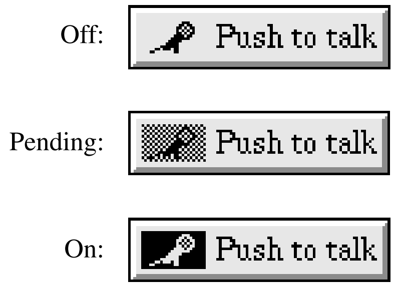

Figure 1: The three states of a Microphone widget.

to widgets that Jupiter applications may include in the windows they open on the user's display. In this way, users can always easily control whether or not their audio or video is being sent anywhere and to choose exactly which applications they are entrusting it to. It also makes it easy for them to identify which applications are currently delivering images or sounds, and to shut them off when they don't wish to see or hear them. There are four new GUI widgets involved: *cameras*, *microphones*, *video panes*, and *speakers*.

Camera and Microphone widgets allow the user to monitor and control the transmission of their local audio and video, respectively. These button-like widgets each contain an icon (provided by the user's client program) representing the local audio or video device they control. As with normal buttons, the application can provide its own textual label or other content in addition to the icon. When one of these widgets is turned "on", the icon's foreground and background colors are reversed, as shown in Figure 1, to signal that audio/video is being sent.

Camera widgets act as relatively simple toggle buttons. Each time the user clicks on them, they switch from on to off and back. Microphone widgets support the same sort of toggling, but they can also be used as *push-to-talk* buttons. In that case, pressing the button enables sending *only* to the destination associated with that widget, temporarily interrupting transmissions associated with any other "on" Microphone widgets for that device. When the mouse is released, the interrupted transmissions are resumed. Because of this press-to-talk functionality, users must hold down a shift key while clicking on Microphones in order to "lock them down" like Camera widgets.

In the process of turning a Camera or Microphone widget on or off, there may be some latency introduced to notify the server and acquire the appropriate multicast address and encryption key information. This is reflected in the user interface by switching background of the widget's icon to a stippled color first, to show that the request is in progress; later, it switches either to the reversed icon, if turning the widget on, or to the normal one, if turning it off. The stippled background signals the user that their request is being processed, but that the audio or video transmission hasn't actually changed state yet. This latency is typically under a second, which is satisfactory for most cases but not for push-to-talk. As described later, our design allows push-to-talk buttons to avoid communications latency after the first time they are used.

Because users' own client programs provide the icon, they can be sure that all genuine Camera and Microphone widgets will be recognizable; if there are no such widgets in any of the Jupiter windows on the screen, or if all of them are in the "off" state, then the user knows that their audio and/or video signal is not being transmitted. Further, the client guarantees that all Camera and Microphone widgets begin in the "off" state when the window containing them is first displayed, and only the user can turn them on. This means that an explicit user action is always required before their audio or video is transmitted. When such a button is displayed by an application, the user can decide whether they trust that application enough to allow it to send audio or video on their behalf.

VideoPane widgets are used to display incoming video streams. Each VideoPane is associated with a single video source at any given time. It resizes the incoming video stream to fit in the space available, if necessary, and can be set to display the video in either grayscale or color. It also takes care of any necessary dithering on displays of limited depth.

A Speaker widget acts as a border around any other collection of widgets. The border highlights whenever the client plays audio from one of the sources associated with the widget. The client will not play any audio stream through the local speaker unless there is a Speaker widget associated with the source of that stream in some open window. This ensures that users can always identify the source of any audible sounds and also guarantees that they have control over whether the sounds continue to be played: they can simply close any offending windows.

The Media Coordination System

Not all Jupiter audio and video transmissions originate from users, and not all are received only by them. The new audio/video coordination system thus generalizes away from the specific widgets described above, to the more abstract notions of *sources* and *sinks*. For active Camera or Microphone widgets, the user's database object acts as a source; in the case of Speakers and VideoPanes, the user object acts as a sink. Other database objects may be sources and sinks as well. An example from the applications mentioned earlier is the tape recorder object. It could be both a source and a sink, depending on whether it was being used for playback or recording.

A source can have several *devices* associated with it; each one is capable of generating a stream of audio or video data. For a user, these devices correspond to the physical audio and video inputs on their workstation. A virtual tape recorder object might have one device for each stream that it is currently playing back.

Sources transmit to sinks through *channels*, an abstraction in the database representing the pipes through which audio and video streams flow. Channels have a *membership*, listing the sources and sinks that are allowed to access them; the membership list can be either static or dynamic. Channels with dynamic membership have an associated *channel*

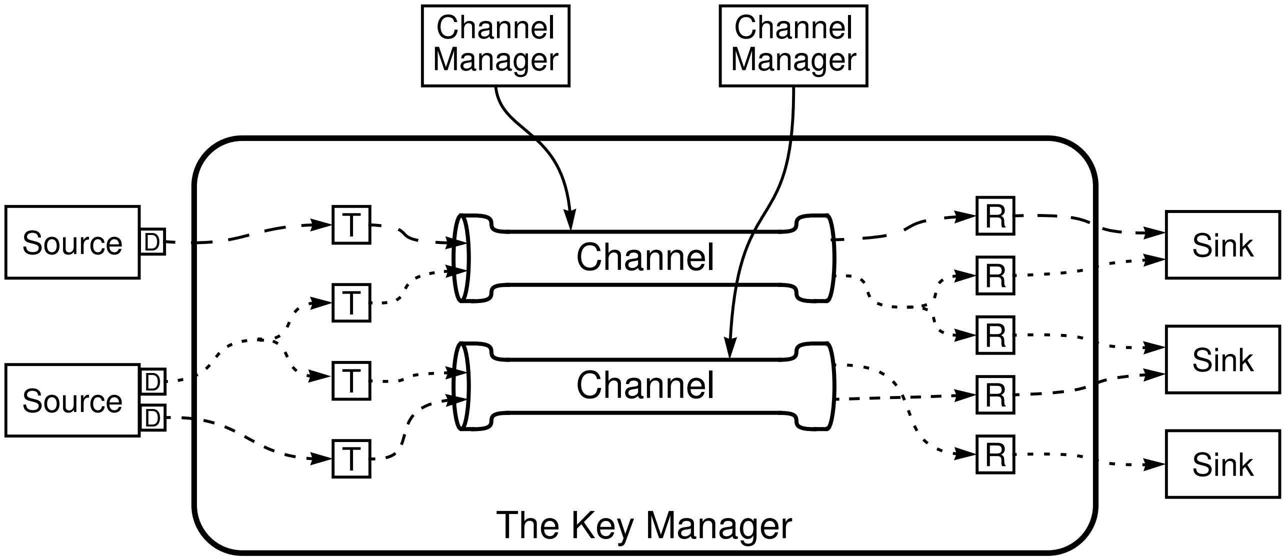

Figure 2: Communication in the coordination system; D = device, T = transmitter, and R = receiver.

manager, an object that is responsible for notifying the coordination system whenever the membership changes. Any Jupiter object can act as a channel manager; for example, each room in Jupiter might be manager for a channel whose membership contains all of the user objects within that room.

Sources, sinks, and channel managers never, in general, communicate with each other directly; instead, they all call (and are called by) a single, coordinating object called the *key manager*. The key manager is responsible for translating the abstract requests made by these objects into actual IP multicast addresses and encryption keys to use when sending and receiving audio/video data. It also coordinates passing a small amount of *in-band* data from sources to sinks.

A diagram of the communication patterns within the coordination system appears in Figure 2. The *transmitters* and *receivers* depicted there are described in the next section.

DETAILS OF THE PROGRAMMING INTERFACE

With the context provided by the previous section, we can now describe in detail the programming interface to the new media coordination system. In this section, we cover the interfaces provided to sources, sinks, and channel managers. The next section describes the implementation of those interfaces inside the key manager. Applications' interface to the four new widget types and the server-side implementations of the widgets themselves are described thereafter.

Sources

The methods that sources call on the key manager and the notifications that it sends back to them are summarized in Table 1. In all of the method calls, the source object itself is passed to the key manager as an implicit parameter (via the MOO language's built-in caller variable).

A source declares its intent to provide an A/V signal by creating a *transmitter* and associating it with one of the source's devices. A source typically creates a transmitter for each reason it might wish to send data; for users, for example, there is one transmitter for each displayed Camera or Microphone widget.

Key manager methods called by sources
create_transmitter() ⇒ *transmitter*
set_transmitter_device(*transmitter*, *device*)
set_transmitter_channel(*transmitter*, *channel*)
add_broadcaster(*transmitter*) ⇒ {*address*, *key*}
remove_broadcaster(*transmitter*) ⇒ {*address*, *key*}
get_transmitter_address(*transmitter*) ⇒ {*address*, *key*}
broadcast_value(*device*, *value*)
transmit_value(*transmitter*, *value*)
destroy_transmitter(*transmitter*)
Notifications sent to sources
use_broadcast_address(*device*, *address*, *key*)
use_transmitter_address(*transmitters*, *address*, *key*)
broadcast_has_receivers(*device*, *yes_or_no*)
transmitters_have_receivers(*transmitters*, *yes_or_no*)
transmitters_rejected(*transmitters*, *channel*)

Table 1: The programming interface provided for sources.

The set_transmitter_channel() method aims a transmitter at a particular channel; sources may only do this if they are in that channel's current membership list. Attempts to aim anywhere else result in a transmitters_rejected() notification; this notification is also sent if the channel membership changes at some point and no longer includes the source. The *channel* argument is either the object that will act as channel manager for the channel, or a list of sources and sinks that comprises the static membership. Transmitters may be aimed at only one channel at a time, but a source can create multiple transmitters for the same device.

Sources identify which transmitters are active by calling add_broadcaster() and remove_broadcaster(). The set of active transmitters for a particular device is known as its *broadcast set*. For Camera and Microphone widgets, these calls are made as the widgets are toggled on and off. The calls return a new multicast address and encryption key to use for broadcasts from the device. The address and key might also change as channel memberships change; the new address and key are delivered to the source with the use_broadcast_address() notification.

To support the push-to-talk functionality of Microphone widgets, sources must also be able to request an address and key for sending only to a particular channel. The get_transmitter_address() method returns this information. Once the address and key are assigned, the user's client can switch in and out of push-to-talk mode without any further communication with the server, thus usually meeting our low-latency goals. If the address or key ever changes due to a change in channel membership, the key manager sends the use_transmitter_address() notification.

To allow sources to avoid sending unnecessarily, the key manager keeps track of whether receivers, sinks' equivalent of transmitters, currently exist for each source/device. The key manager sends the broadcast_has_receivers() notification whenever the set of receivers for a device's broadcast set starts or stops being empty; transmitters_have_receivers() is sent similarly for individual transmitters.

In addition to assigning addresses and keys for sending out-of-band data, the key manager provides a mechanism for sending small amounts of *in-band* data. This is currently intended primarily as a means for sources to send a URL [2] for some sound, image or video sequence that they wish recipients to retrieve and display. Values can be sent to a device's broadcast set using the broadcast_value() method, or to a specific channel using transmit_value().

Key manager methods called by sinks
create_receiver() ⇒ *receiver*
aim_receiver(*receiver*, *source*, *device*, *channel*)
unaim_receiver(*receiver*)
destroy_receiver(*receiver*)
Notifications sent to sinks
use_receiver_addresses(*receivers*, **list of** {*address*, *key*})
receivers_have_transmitters(*receivers*, *yes_or_no*)
receive_value(*receivers*, *value*)
receivers_rejected(*receivers*, *channel*)

Table 2: The programming interface provided for sinks.

Sinks

The methods on the key manager called by sinks and the notifications returned are summarized in Table 2. Again, in all method calls, the sink is passed as an implicit parameter.

A sink declares its interest in receiving a particular audio/video stream by creating a *receiver* and associating it with a specific signal being sent through some channel. For users, each displayed VideoPane widget has a single associated receiver and each Speaker widget has a set of associated receivers.

The aim_receiver() method associates a receiver with the signal from a particular source and device as sent through a given channel. As with sources, sinks can only aim a receiver at a channel if they are in its membership list. Aiming elsewhere results in a later receivers_rejected() notification. Channels are specified as for sources.

As new addresses and encryption keys are generated for particular receivers, sinks are sent the use_receiver_addresses() notification. This tells the sink the set of addresses and keys the source will use to encrypt and send out-of-band data. Since a source can switch between sending to its broadcast set address and an individual transmitter address without notifying the server, the key manager provides both addresses to receivers when both have been assigned.

To allow a sink to discard useless receivers, the key manager sends the receiver_has_transmitters() notification as the set of corresponding transmitters starts and stops being empty.

Sinks receive in-band information from sources via the receive_value() notification.

Channel Managers

The third and final kind of participant in media coordination is channel managers. Any object may be a channel manager; to do so, it must support the following key manager notification:

channel_in_use(*yes_or_no*)

This is sent whenever the set of transmitters and receivers aimed at this channel manager starts or stops being empty. When *yes_or_no* is true, the channel manager should begin informing the key manager of the membership of the channel, both immediately and whenever it changes in the future. The method used for this is

set_channel_membership(*members*)

A given object can be a channel manager for at most one channel, since the object serves as the channel's unique name in calls to set_transmitter_channel() and aim_receiver().

A member of a channel is presently allowed to be either a source or a sink or both on that channel. Providing independent source and sink membership lists might be useful, though, and we are considering this change.

Key manager methods called by sources and sinks
watch_channel(*channel*, *aspects*)
channel_membership(*channel*) ⇒ *members*
channel_transmitters(*channel*) ⇒ **list of** {*source*, *device*}
Notifications sent to sources and sinks
channel_changed(*channel*, *aspect*)
watching_rejected(*channel*)

Table 3: Programming interface for both sources and sinks.

Other Key Manager Services

The remaining methods and notifications provided by the key manager are summarized in Table 3. These methods are typically used by both sources and sinks.

The watch_channel() method provides a way for sources and sinks to be notified of changes to various aspects of a channel's state, including the channel's membership and the set of transmitters aimed at the channel. Watchers are sent the channel_changed() notification whenever any of their specified *aspects* changes. As in other methods, sources and sinks can only watch a channel when they are in its membership. Otherwise, or should they later become excluded from the membership, they receive a watching_rejected() notification.

Any source or sink that is a member of a channel can use the channel_membership() method to discover the complete membership; this method is on the key manager and not the channel manager to prevent the latter from maliciously giving different answers to different sources and sinks. Members can also use channel_transmitters() to find out what signals are available through a given channel.

IMPLEMENTING THE KEY MANAGER

Fundamentally, the key manager must maintain several mappings that represent the current state of channels, transmitters, devices, and receivers:

$$\mathcal{C}[\mathit{channel}] \Rightarrow \langle \mathit{members}, \mathit{address}, \mathit{key}, \mathit{watchers} \rangle$$
$$\mathcal{T}[\mathit{source, xmtr}] \Rightarrow \langle \mathit{device}, \mathit{channel}, \mathit{is\_broadcaster} \rangle$$
$$\mathcal{D}[\mathit{source, device}] \Rightarrow \langle \mathit{address}, \mathit{key} \rangle$$
$$\mathcal{R}[\mathit{sink, rcvr}] \Rightarrow \langle \mathit{source}, \mathit{device}, \mathit{channel} \rangle$$

In most cases, it is easy to see how to update these mappings for each of the key manager methods described in the previous section. The one exception is the *key* fields, which should be changed only when necessary, a condition that would be difficult (or at least inefficient) to test given only the obvious representations of the mappings above.

It is more difficult to characterize the conditions under which the various key manager notifications should be sent, especially as a function of the various method calls in the public interface. For example, the broadcast_has_receivers() notification should be sent for a given $\langle \mathit{source}, \mathit{device} \rangle$ pair exactly when the set

$$\{\langle \mathit{sink}, \mathit{rcvr} \rangle \mid \exists \mathit{channel}, \mathit{xmtr} \,.\, \mathcal{R}[\mathit{sink, rcvr}] = \langle \mathit{source}, \mathit{device}, \mathit{channel} \rangle \wedge \mathcal{T}[\mathit{source, xmtr}] = \langle \mathit{device}, \mathit{channel}, \underline{\mathbf{true}} \rangle \}$$

transitions between empty and non-empty (in either direction). The simple mappings given above do not make it easy to read off the current value of this set; worse, several different method calls can trigger the important transition, each in a different way.

We attacked these problems by maintaining extra, redundant information in the mappings that make it easier to read off the important derived sets and relations. Each derived set corresponds to a specific set of notifications; when certain key transitions occur in the values of these sets, the appropriate notifications are sent. In our implementation, we maintain the following mappings differently:

$$\mathcal{C}[\mathit{channel}] \Rightarrow \langle \mathit{members}, \mathit{xmtrs}, \mathit{rcvrs}, \mathit{address}, \mathit{key}, \mathit{watchers} \rangle$$
$$\mathcal{T}[\mathit{source, xmtr}] \Rightarrow \langle \mathit{device}, \mathit{channel}, \mathit{is\_broadcaster} \rangle$$
$$\mathcal{D}[\mathit{source, device}] \Rightarrow \langle \mathit{has\_rcvrs}, \mathit{address}, \mathit{key} \rangle$$
$$\mathcal{Z}[\mathit{source, device}, \mathit{channel}] \Rightarrow \langle \mathit{xmtrs}, \mathit{rcvrs} \rangle$$

To the $\mathcal{C}$ mapping we added lists of the transmitters and receivers currently aimed at each channel. This makes it easy to generate the appropriate use_transmitter_address(), use_receiver_addresses(), transmitters_rejected(), receivers_rejected(), and watching_rejected() notifications when the channel membership changes. The various channel_changed() notifications are also now trivial to implement.

The $\mathcal{Z}$ mapping was added to enable easy implementation of the transmitters_have_receivers() and receivers_have_transmitters() notifications.

The efficient implementation of the broadcast_has_receivers() notification was perhaps the most difficult design task; it may be called for after changes to any of four mapping components: receiver aiming, transmitter device, transmitter channel, or transmitter membership in the associated broadcast set. We added a *has_rcvrs* flag to the $\mathcal{D}$ mapping to remember the current empty/nonempty state of the set described earlier; this allows us to avoid useless work in cases where that state is obviously not changing, such as when a new receiver is added to a device that already had receivers listening. The new $\mathcal{Z}$ mapping provides the rest of the information used to trigger possible transitions; unfortunately, some searching is still required in order to make the final determination of whether a true transition has occurred.

It is worth mentioning two other optimizations. The specification calls for a change in broadcast-set key whenever the set of allowed sinks changes; that is, for each change in the union of the memberships of the channels aimed at by transmitters in the broadcast set of a given source and device. This union is sufficiently expensive to maintain that we chose instead to change the key whenever there is a change either in the set of channels in the broadcast set or in the membership of any one of those channels. This could lead to more key changes than strictly necessary, but seems a worthwhile trade-off for implementation efficiency.

The other optimization is simpler; whenever there is only one channel in a broadcast set, we use that channel's address and key for the broadcast set as well. This reduces both key generation work on the server and cryptographic and network expenses on the client.

IMPLEMENTING THE USER INTERFACE

In this section, we describe applications' interface to the new widgets and the ways in which the widget themselves are implemented in terms of the coordination system.

All widgets in the Jupiter window system have their implementations split between core objects in the database and code in each user's client. The server-side code is responsible for coordination with other parts of the system and for communicating with the client. The client-side code is responsible for actually interacting with the user and communicating with the server. In the rest of this section, we often elide the server/client communication step; it does not usually hold much interest.

The server-side implementations of the four new user interface widgets in our design use the facilities described in the previous section. The implicit source or sink argument to all key manager method calls, however, is overridden by the widget implementations; from the key manager's perspective, the calling source or sink always appears to be the object representing the user on whose screen the widget appears. This approach allows the key manager to treat user objects in the same way as other sources and sinks.

Camera and Microphone Widgets

When a Microphone widget is created, it makes a call to create_transmitter(). Similarly, the creation of a Camera widget causes a call to create_transmitter(). The resulting transmitter is saved as part of the widget's private state.

Two methods are available on Microphone and Camera widgets. They let applications set from which of the user's local devices to get the signal and to which channel to send it:

set_device(*device*)

set_channel(*channel*)

When these methods are called, the widget calls set_transmitter_device() or set_transmitter_channel(), as appropriate. It is expected that when users' client programs receive a set_device() for widgets that are "on," they will generally send back an event indicating that the widgets have been turned off; this ensures that an application program can't cause a user to start sending from some different device without an additional, explicit user action.

When a Camera or Microphone widget is turned on by the user, its associated transmitter is added to the broadcast set for its designated device. When it is later turned off, the transmitter is removed again. These are the operations taking place while the button icon's background is in its intermediate, stippled state.

When a Microphone is used as a push-to-talk button for the first time, the widget calls get_transmitter_address() to have an address assigned just for that channel. On future uses of the widget, the client can respond instantly, already having all of the necessary information.

The key manager's use_broadcast_address() and use_transmitter_address() notifications are simply passed through to the user's client, to update the addresses and keys assigned to existing widgets. Similarly, the broadcast_has_receivers() and transmitters_have_receivers() notifications are passed through, so that the client can avoid sending unnecessary data. When a transmitters_rejected() notification is received, a message is sent to the client clearing the appropriate address and key information.

Speaker Widgets

Applications set which sources a Speaker should listen to by calling the method:

set_sources(**list of** {*source*, *device*, *channel*, *volume*})

For each tuple in the list, a receiver must be created and aimed at the specified source, device, and channel. The *volume* arguments allow an application to specify the relative level of the incoming audio, so that certain streams can be played more softly than others. This is useful, for example, when trying to play audio from sources that are virtually "farther away," such as in a nearby room.

When set_sources() is called, the widget creates a set of new receivers, one aimed at each of the given sources; the *volume* arguments are simply passed on to the client.

The key manager's use_receiver_addresses() notifications are passed on to the client. When a receivers_rejected() notification is received, a similar message goes to the client, setting the empty list of addresses for those receivers. Finally, on a receive_value() notification, the value received is passed to the client as a URL naming audio data for the client to retrieve and play.

VideoPane Widgets

VideoPane widgets are implemented almost identically to Speakers. The one difference is that, since a VideoPane can only receive at most one signal at a time, a single receiver can be created and saved when the widget is created; the widget can simply re-aim the receiver whenever the application changes the source to which the widget listens. To make this change, applications call this widget method:

set_source(*source*, *device*, *channel*)

VideoPane widgets also support the receive_value() notification; at present, this is used to pass a URL containing a static image to load. We may later extend the usage to allow playing a video clip.

IMPLEMENTING THE APPLICATIONS

Earlier in the paper, several applications were described as motivating the audio/video architecture design. In this section, some of these applications are re-examined, to see how their implementations could take advantage of the facilities provided.

Communication Within a Room

Communicating with other users in the same room is the most common use of audio and video within Jupiter. Typically, support for this is provided one of the windows that the system opens when a user first connects. An example of such a window appears in Figure 3.

This window contains Microphone and Camera widgets to control sending to everyone in the room, and a collection of VideoPanes, each surrounded by a Speaker, to receive from others in the room. Below each VideoPane is a Microphone that serves as a push-to-talk button for whispering to the

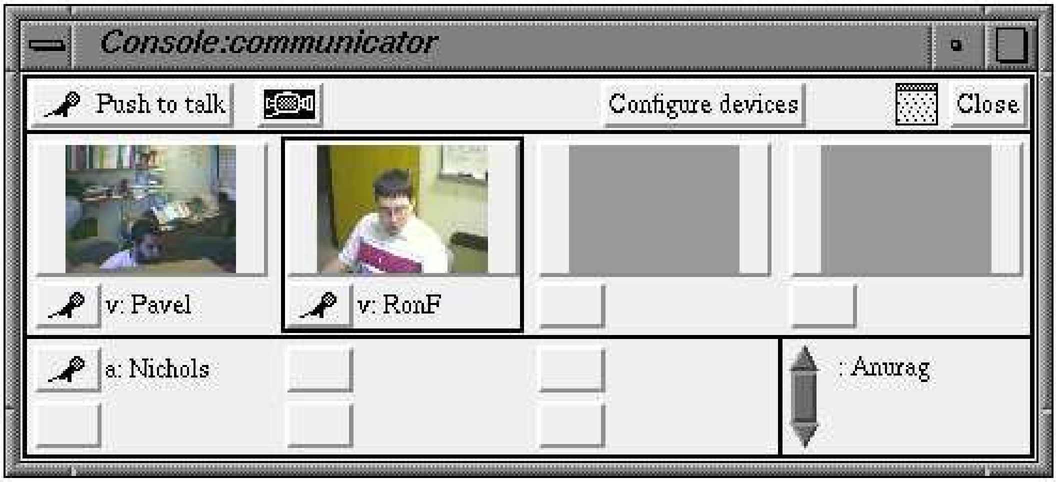

Figure 3: An example of a tool for in-room communication

indicated user. Below this are Speakers and Microphones for users who are not providing video. In the lower right, there is a list of users who either won't fit in the rest of the window or else are not providing either audio or video. Finally, there is also a large Speaker widget surrounding the entire window.

Whenever any user talks, their associated Speaker widget highlights. If they are speaking to the whole room (as opposed to whispering), the large outer Speaker also highlights. The figure shows user RonF speaking to the room.

This application takes advantage of two different kinds of channels provided in the system. For sending to the room, the Jupiter object representing the room acts as a channel manager, dynamically updating the channel membership as its contents change. The membership is not limited to the users in the room; it includes all of the objects present, to allow for the participation of non-user sources and sinks. For whispering, the tool specifies a channel with a static membership containing just the two users involved.

When the user first enters a room, the tool clears out all of the Speakers and VideoPanes; they were pointing at users in the previous room. The primary Microphone and Camera widgets are re-aimed at the new room's channel, and the application starts watching for transmitters on that channel. For every transmitter found, one of the VideoPane/Speaker pairs is directed at the source or, if no video is available, one of the audio-only widgets is used instead.

Supporting audio from other rooms is a straightforward extension of the preceding. It only requires that the room objects acting as channel managers include in their membership the objects in "nearby" rooms. The application then also watches these nearby room channels for sources, and any found are added to some Speaker, perhaps labelled with the name of the appropriate room. Using Speakers' *volume* feature, this audio is played more softly than it would if the source were in the same room as the user of the application.

To "glue together" a virtual and a physical place, a machine must exist somewhere on the network with physical microphones and cameras pointing into that room and a server capable of capturing and delivering this audio and video as a multicast stream. The Jupiter application performing the gluing would then contact this server, create a transmitter pointing at the virtual room, and pass on address and key information in much the same way the Microphone and Camera widgets do for client programs. Allowing virtually-present users to be audible in the physical room is a symmetric problem; the room's server must receive and play audio streams under the direction of the Jupiter application.

Media Spaces, Public Address, and Telephony

As far as the media coordination system is concerned, supporting an audio/video tunnel or media space is exactly the same as supporting a room. The only difference is in how channel membership is determined. All of the mechanisms for discovering new transmitters and setting up whispering work in exactly the same manner in both cases.

A public address system can be supported in a few different ways. If it should be able to reach all users at all times, then it might make sense to leave some space for an associated Speaker in the main client window. The channel manager for the public address channel then sets a membership containing all connected users. As another option, the act of sending a public address could pop up a new window on each connected user's screen, with a Speaker (and possibly VideoPane) for the public address channel included in it. In either case, if limits are desired on the use of this facility, the application could arrange to include only authorized sources in the list associated with the Speaker widget.

A virtual videophone object once again has many things in common with sending to rooms or a media space. However, the user interface would need to provide some sort of pop-up window allowing users to be invited into the conversation, only then adding appropriate Camera, Microphone, VideoPane, and Speaker widgets. Also, the videophone might create its own audio transmitter, for use in sending sound effects to the users placing or receiving calls; the transmit_value() facility could be used for this.

The Tape Recorder and the Archive Server

A virtual tape recorder's audio or video data must be transmitted and received by an auxiliary server, under the control of a Jupiter application, as in the case of gluing together physical and virtual rooms. In contrast, however, such an archive server should probably be shared among a number of applications, recording and playing back several different media streams simultaneously. A Jupiter object, the *archive manager*, coordinates the various requests from applications and relays them to the archive server. These requests could include such services as seeking to a new position in a recorded stream or changing the speed of playback. These aren't directly tied to the audio/video distribution system, however; they only cause the archive server to modify the signal it actually delivers to the receivers.

An individual tape recorder, then, simply maintains a list of the recordings it has stored on the archive server. For playback, it creates a transmitter aimed at the room and passes the address/key information it receives on to the archive manager. For recording, it watches the room channel, creating a receiver for every source and again passing on the addresses and keys to the archive manager.

Other Applications

The other applications we described earlier can be implemented as straightforward variations on the examples given above. The 'news on the hour' service is just another client of the archive server, for example. Applications wishing to play sound effects just create a transmitter themselves from which to send URLs, aiming it either at individual users, rooms, or channels whose membership they directly manage, as appropriate.

CONCLUSIONS

The Jupiter system supports multimedia communications and long-term collaboration, using a "real-world" metaphor including people, places, and things. In this environment, we wanted to allow experimentation with a wide range of communication metaphors, such as talking in a room, using a telephone, making announcements on a public address system, and using a tape recorder.

It is challenging in such a system to keep our users from being confused by the variety of sounds and images emanating from their workstations. Another challenge is providing a programming library for controlling the flow of audio and video that is powerful enough to support our list of applications, yet simple enough to allow users to build new ones. Finally, to allow use of our system in insecure contexts, it is important to protect the privacy of all communications and the integrity of the coordination system from both misbehaving users and other possible attackers.

Based on experience with an earlier prototype, we designed and implemented a flexible new programming interface for secure media coordination. We also designed a small set of new widgets that provide users with the information and control they require. We are in the process of reimplementing our motivating applications on top of the new facilities.

REFERENCES

1. Adler, Annette, and Austin Henderson, "A Room of Our Own: Experiences from a Direct Office-Share," *Proceedings of the 1994 ACM Conference on Human Factors in Computing Systems*, May, 1994.
2. Berners-Lee, T., L. Masinter, and M. McCahill, "Uniform Resource Locators (URL)," IETF Request for Comments RFC 1738, available from ftp://ftp.internic.net/rfc/rfc1738.txt.
3. Bly, Sara, Steve Harrison, and Susan Irwin, "Media Spaces: Bringing People Together in a Video, Audio, and Computing Environment," *Communications of the ACM*, 36(1), January 1993.
4. Curtis, Pavel, *LambdaMOO Programmer's Manual*, available as ftp://ftp.parc.xerox.com/pub/MOO/ProgrammersManual.ps.
5. Curtis, Pavel and David A. Nichols, "MUDs Grow Up: Social Virtual Reality in the Real World," in the Proceedings of the 1994 IEEE Computer Conference, pp. 193--200, January 1994. Also available as ftp://ftp.parc.xerox.com/pub/MOO/papers/MUDsGrowUp.ps.
6. Deering, S. and D. Cheriton, "Multicast routing in datagram networks and extended LANs," *ACM Transactions on Computer Systems*, May 1990.
7. Hickman, Kipp E. B., "The SSL Protocol," available from http://home.mcom.com/newsref/std/SSL.html.
8. Minneman, Scott L., and Steve R. Harrison, "Where Were We: making and using near-synchronous, pre-narrative video," *Proceedings of the 1993 ACM International Conference on Multimedia*, pp. 207--214, June 1993.
9. Nichols, David A., Pavel Curtis, Michael Dixon, and John Lamping, "A Low-Bandwidth Window Toolkit," in preparation.
10. Schulzrinne, H., S. Casner, R. Frederick, and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications," IETF Internet Draft, available from ftp://ftp.internic.net/internet-drafts/draft-ietf-avt-rtp-07.txt.
11. Zellweger, Polle T., Douglas B. Terry, and Daniel C. Swinehart, "An Overview of the Etherphone System and its Applications," *Proceedings of the 2nd IEEE Conference on Computer Workstations*, March1988.