A Call Model for Distributed Multimedia Communications

Walter L. Hill, Audrey K. Ishizaki
Media Technology Laboratory
HPL-93-06
January, 1993

API, multimedia communications, computer-controlled communications, shared workspaces

Most experiments in building multimedia communications systems to date have adopted application architectures which are intended for communication-specific applications such as teleconferencing, and support only particular kinds of communication channels and mechanisms. This paper describes an application architecture which gives developers of diverse applications, from word processor and spreadsheet programs to large distributed multi-user systems, a means to enable users to control real-time communications directly in the context of their work. It also provides uniform application-level control of distributed multimedia resources. It is designed for eventual use in a distributed object system and provides a programming interface based on intuitive operations of a telephone call which model basic communication tasks such as placing and forwarding a call. Analogies are drawn between properties of a call system and those of a window system. The architecture and its implementation allow numerous communication services to be controlled together and can control any service with a suitable switching interface. In addition to supporting switching for traditional communications media, it is possible, for example, to control sets of connections between clients and servers in a distributed computing environment. A particularly powerful example is provided by the addition of switching control for connections between X Windows clients and servers, giving rise to *shared multimedia workspaces* which provide a seamless integration between personal work contexts and communication contexts.

Internal Accession Date Only

© Copyright Hewlett-Packard Company 1993

A Call Model for Distributed Multimedia Communications

Walter L. Hill, Audrey K. Ishizaki
Hewlett-Packard Laboratories

ABSTRACT

Most experiments in building multimedia communications systems to date have adopted application architectures which are intended for communication-specific applications such as teleconferencing, and support only particular kinds of communication channels and mechanisms. This paper describes an application architecture which gives developers of diverse applications, from word processor and spreadsheet programs to large distributed multi-user systems, a means to enable users to control real-time communications directly in the context of their work. It also provides uniform application-level control of distributed multimedia resources. It is designed for eventual use in a distributed object system and provides a programming interface based on intuitive operations on a telephone call which model basic communication tasks such as placing and forwarding a call. Analogies are drawn between properties of a call system and those of a window system. The architecture and its implementation allow numerous communication services to be controlled together and can control any service with a suitable switching interface. In addition to supporting switching for traditional communications media, it is possible, for example, to control sets of connections between clients and servers in a distributed computing environment. A particularly powerful example is provided by the addition of switching control for connections between X Windows clients and servers, giving rise to *shared multimedia workspaces* which provide a seamless integration between personal work contexts and communication contexts.

KEYWORDS: API, multimedia communications, computer-controlled communications, shared workspaces

1. Introduction

Most experiments in building multimedia communications systems to date have adopted application architectures which are intended for communication-specific applications such as teleconferencing, and support only particular kinds of communication channels and mechanisms [MM92]. This paper describes an application architecture which gives developers of diverse applications, from word processor and spreadsheet programs to large distributed multi-user systems, a means to enable users to control real-time communications directly in the context of their work. It also provides uniform application-level control of distributed multimedia resources. The architecture and its implementation allow numerous communications services to be controlled together and can control any service with a suitable switching interface. In addition to supporting switching for traditional communications media, it is possible, for example, to control sets of connections between clients and servers in a distributed computing environment. A particularly powerful example is provided by the addition of switching control for connections between X Windows clients and servers, giving rise to *shared multimedia workspaces* which provide a seamless integration between personal work contexts and communication contexts.

Modern software development relies on application programming interfaces, or APIs, which provide the operations for accessing sets of services. In C, APIs are realized as collections of functions, while in C++ they are collections of classes. System-level APIs, for example for file and window management, are used in developing most applications, while custom APIs promote modularity and reuse within particular classes of applications. As computing evolves, new system-level APIs are needed to support new functionality, as seen currently, for example, with support for speech and handwriting input. The call API described in this paper is intended as a system-level API supporting heterogeneous real-time communications services. A call system is in a sense dual to a window system (it controls arcs in a connection graph rather than nodes). However, by virtue of its ability to control contexts, it can also be used as a partial replacement for a window system when windows may not be feasible, as in the case of small displays on handheld devices.

As the name suggests, the *calls* in a call system are motivated by intuitions about telephone calls, and specifically by the intuitive set of operations such as place, hold, forward, and hang-up, associated with them. One must be careful, however, as with windows, about taking the analogy too literally. The use of the word *call* here is with reference to the set of operations, which in fact was the starting point for the call API. The choice of the word seems appropriate, as with window, despite its other uses. The idea of basing a communications API on notions associated with a telephone call seems natural and potentially congenial for

programmers, and indeed has also influenced the design of other systems, notably EXPANSE [Minzer91] and Touring Machine [Arango92,Gopal92] at Bellcore. In addition to providing a programming model based on communication tasks, calls are also effective as a mechanism for organizing and distributing communication information and control. Along with calls, a call system has *parties* which can belong to calls and whose media connections are controlled through calls. As with systems like Etherphone [Swinehart87, Vin91] and Touring Machine, the notion of party is construed broadly so that a wide range of sources and sinks of information can be represented and treated uniformly. For example, the shared workspaces mentioned above allow parties to be associated with arbitrary X Windows clients. A basic design objective for the call system is robust support for multiple parties and media which can be added and removed dynamically from a call.

The call system was designed for eventual implementation in a distributed object system and will be described here in terms of classes of objects. The notions of call and party objects strongly influence the design of the call system and the ways in which it can be used. For example, calls can be longer-lived than conventional telephone calls. The same call object might be used repeatedly for a series of related communications over time and might be stored and accessed, for example, in an on-line appointment book. Of course, connections controlled by the call may be broken when the call is not actually in use.

In providing a set of system-level services, it is important to distinguish general mechanisms which the call system must provide from those which must be provided by an application or some other system component. For example, the call system provides indirect support for controlling media devices but does not provide a user interface for controlling the devices. The call system provides a set of mechanisms which allow an application to implement policies appropriate to it. For example, in some cases it is important to distinguish a caller party among the parties to a call; other applications distinguish a current speaker or moderator party which can change over the life of the call; still other applications don't have a distinguished party at all. The call system can provide support for distinguished parties, but does not require them to be present. There are numerous examples of policies which the call system attempts to support without imposing them on the general application developer.

The following section of this paper presents the actual call system API and discusses some of the issues which its design addresses. Section 3 discusses certain aspects of the call system implementation, in particular its interface to underlying switching mechanisms. Section 4 describes real applications, including shared workspaces, in various stages of design and development, which use the call system. This section also identifies design ideas and issues which need to be explored in future work. Section 5 discusses the connections between the call system and related work. Finally, Section 6 provides a summary and draws conclusions from the work presented here.

2. Call System API

This section describes some of the important operations on Call, Party, and Resource Manager objects which are basic to the call API. The API emphasizes operations which model communication tasks such as creating, initiating, suspending, and resuming Calls. While the API does provide operations for direct manipulation of media connections for finer control, the API's higher-level operations can control sets of connections automatically and are sufficient in many applications. Developers can add new state and behavior to Calls, Parties, and Resource Managers using object-oriented inheritance, for example, to allow different Parties to have different roles in a Call or provide Resource Managers which monitor resource usage.

While it is important that the objects in the call system be intuitive for general application developers, their primary purpose is to organize information and control for communications effectively. A Resource Manager, for example, is the locus of all information about the communication resources, such as logical and physical devices, of the Parties it supports. It also holds such things as preferences of Parties for selecting resources to use in various communications. Resource information is not held in Parties, and is not available to Calls, for security reasons among others. Parties contain all information about their communication activities, for example which calls are active or on hold. A Call controls all the connections between its Parties.

There is a notion of abstract connection which underlies the Call System. Parties control abstract connections between their resources and a Call, while a Call manages abstract connections between Parties. A concrete connection enabling communication is made only when a complete series of such abstract connections from one resource to another is made. Different kinds of control are needed in connecting Parties in Calls. For

example, putting a Call on hold is an operation primarily local to a Party while terminating a Call is an operation on the Call itself. Whenever abstract or concrete connections are made or broken, messages are sent informing the Call and all Parties, which in turn can inform a controlling application of the state change. There are a large number of possible policies which an application could use in controlling media connections. In some applications, centralized control is desirable while in others control will be mainly local. For example, using the mechanisms provided by the Call System, one application could make sure that all Parties were viewing the same video source, while another could allow each Party to choose from all the available video sources in the Call.

Parties provide a uniform interface for communicating with diverse sources and sinks of information, including people. Calls can be placed to Parties which represent offices, hospital operating rooms (with many audio, video, and instrument data resources), movies, telephones, CATV channels, multimedia documents, X Windows applications, etc., just as they are placed to Parties representing human beings. Thus, from the point of view of object-oriented programming, any object can be made into a Party if it can meaningfully inherit from the Party class described in this section.

One of the benefits for developers from the call system's object-oriented design is the ability to use objects which represent *sets* of Calls, Parties, MediaLines, Resources, etc. These sets arise naturally throughout design of the call system; much of the application code and implementation code for the call system is written in terms of set operations.

Call Objects

Creation of a Call object is separate from its being initiated; Calls can exist, with or without Parties, independent of any connections being made. Each Party in a Call has a set of abstract MediaLines which represent media sources and destinations which can be connected by the Call. The MediaLines may differ for different Parties in a single Call (e.g., some Parties may only have output lines) and for different Calls (e.g., some Calls may only have audio lines). Each MediaLine has a name, a media type, and is a source or destination or else bidirectional. A PartyInfo object is defined to be a Party object, together with a set of MediaLines. PartyInfos and PartyInfoSets are convenient for adding and removing Parties and MediaLines from Calls, and are also used in creating Calls.

```
call emptyCall()
call newCall(partyInfoSet partyInfos)
int callAddParty(call c, partyInfo pi)
int callRemoveParty(call c, party p)
int callAddMedia(call c, partyInfo pi)
int callRemoveMedia(call c, partyInfo pi)
```

Notice that Calls don't have a distinguished Party which is the "caller". Indeed, a Call need not be created or initiated by a Party participating in the Call. It is up to an initiating Party to know that it placed the Call. Using inheritance, one can add a distinguished Party to a Call.

A Call can be queried for its Parties and their associated MediaLines.

```
partySet callParties(call c)
mediaLineSet callMediaInfo(call c, party p)
```

When a Call is initiated, its Parties are notified and it is their Resource Managers which negotiate their participation in the Call at that point. The negotiation behavior will depend on the Party. If a Party rejects the Call when it is initiated, it is removed from the Call. Parties added to the Call after it is initiated must negotiate participation immediately. Similarly, removing a Party from a Call after it has been initiated is the same thing as hanging up. When a Party accepts participation in a Call, all other Parties are notified.

```
int callInitiate(call c)
```

To describe connections in a Call, we refer to Figure 1 which shows a very simple two-party call in which one Party just has an audio source and the other just an audio destination. Each Party's Resource Manager controls the connection between a MediaLine and appropriate Resource objects. The Call itself controls whether the two MediaLines are connected. Only when the Call's connection and each of the Parties'

connections are made, is an actual connection made between a microphone and a speaker. It is worth noting that Party2 could connect its MediaLine to multiple speakers, in which case the single abstract connections controlled by Party1 and the Call correspond to multiple actual connections (or to a single multicast connection). If there are additional Parties with speakers connected then a lot of switching activity results from Party1's single connection. Calls themselves extend this power; when a Party puts a Call on hold, it affects in a single operation all of its connections to MediaLines in the Call. An even more important observation is that the set of MediaLines in a Call can be thought of as a kind of "local coordinate system" for switching where the Parties' connections determine the mapping between the local entities, i.e., the named MediaLines, and actual Resources. This allows applications to operate on Calls independent of particular Resources, and even independent of particular Parties, by just referencing the named MediaLines. This is analogous to the independence which window applications enjoy with respect to the location of the window.

```
int callConnect(call c, party fromParty, mediaLine fromML,
                        party toParty, mediaLine toML)
int callDisconnect(call c, party fromParty, mediaLine fromML,
                           party toParty, mediaLine toML)
```

There are two merge operations on Calls. The first, called `merge`, is applied to a set of Calls. It creates a new Call whose Parties consist of all the Parties to any Call in the set. The MediaLines of each Party to the merged Call are the union of the sets of MediaLines of that Party in all constituent Calls with duplicate names being distinguished by suffixes. A second form of merge, called `mergeInto`, takes a "primary" Call and merges a set of "secondary Calls into it without creating a new Call. The identity of the first Call is preserved.

```
call merge(callSet calls)
call mergeInto(call c, callSet calls)
```

Forwarding a Call from one Party to another can be accomplished by setting up a second Call between two Parties, merging it into the first Call, and then removing the forwarding Party.

When a Call is terminated, all of its connections are broken. Each of its Parties and any controlling application are notified. The terminate operation must be used explicitly to end a Call. Since a Call is allowed to have no current Parties, it is not enough for all Parties to hang up, as would be the case with a traditional telephone call. Of course, a particular application may enforce a termination policy for its empty Calls.

```
int terminate(call c)
```

Party Objects

As we have mentioned, the notion of Party object is very general. It is worthwhile to think abstractly about a Party as any object which could support the operations and behavior described here. The notion of MediaLine should be thought of similarly so that connections between many kinds of resources can be controlled by the call system. We again mention client-server systems as an important source of non-traditional examples of communication channels where the connections between clients and servers are controlled through MediaLines in a Call.

Each Party has a name which is used to identify it in a Call. The name may not be unique to the Party; applications distinguish Parties by their identity as objects. A Party has a Resource Manager which manages all of its resources and also manages the Party's media connections in a Call.

```
party newParty(char *name, resourceManager rm)
char *partyName(party p)
resourceManager partyRM(party p)
```

When a Party receives a request notification from a Call, its Resource Manager may negotiate its participation directly or pass control on to a controlling application. In the latter case, the Party must be told whether to accept or reject.

```
int acceptCall(party p, call c)
int rejectCall(party p, call c)
```

If the Call is accepted, the Party adds the Call to its set of Calls, and its Resource Manager attempts to connect default resources to it automatically, notifying the Party if it fails either because resources are busy or it doesn't know how to make the connection. To defer connecting resources, a Call can also be accepted on hold.

```
int acceptCallOnHold(party p, call c)
```

A Party can place a Call. This operation creates the Call which includes the calling Party with associated MediaLines. It then initiates the Call. The Party object placing a Call accepts the Call automatically.

```
int placeCall(party p, partyInfoSet partyInfos, mediaLineSet mediaLines)
```

A Party can put a Call on hold. In doing so, it informs its Resource Manager to disconnect all of that Party's resources which are connected to the Call. While the Call is on hold, those resources can be used for other purposes. It is intended that putting a Call on hold be analogous to the conventional operation with telephone calls. That operation frees local resources while maintaining others (phone lines). It is useful for a Party also to be able to free non-local resources, which can have high connect costs, without terminating the call. This is accomplished by the suspend operation, which frees both local and non-local resources which the Party uses in the Call. The Party can resume a Call which has been put on hold or suspended (the latter may take longer to resume and incur an additional setup cost). If the original resources are not available when the Call is resumed, the Resource Manager allocates new resources, possibly communicating with a controlling application, as it would for a new Call. In addition to the hold, suspend, and resume operations which affect all of a Party's MediaLines, there are selective versions which would allow, for example, "putting audio on hold" while keeping a Call active.

```
int holdCall(party p, call c)
int suspendCall(party p, call c)
int resumeCall(party p, call c)

int holdMedia(party p, mediaLineSet mediaLines, call c)
int suspendMedia(party p, mediaLineSet mediaLines, call c)
int resumeMedia(party p, mediaLineSet mediaLines, call c)
```

Of course, a Party can hang up. This operation removes all connections between the Party's resources and the Call. It also removes all references from the Party to the Call and vice versa. All other Parties to a Call are notified when a Party hangs up.

```
int hangUp(party p, call c)
```

A Party can participate in multiple Calls, which are active, on hold, or suspended, and can be queried for the corresponding sets of Calls.

```
callSet partyCalls(party p)
callSet partyActiveCalls(party p)
callSet partyHeldCalls(party p)
callSet partySuspendedCalls(party p)

char *partyCallStatus(party p, call c)
```

Resources and Resource Manager Objects

Compared with Call and Party objects, Resource Manager objects play a supporting role in the call system. Typical applications will deal mainly with Call and Party objects, while the work of the Resource Managers and with it much of the complexity of the call system implementation remains hidden, just as much of the complexity of a window system is hidden in the window manager.

Resource Managers hold all information about Parties' media resources which are represented by Resource objects. They also support the actual negotiation of connections. Resource objects include MediaPorts which correspond to ports on physical and logical switches (including, for example, client and server processes whose connections are managed by the call system), and also composite resources which can have a set of associated MediaPorts and also a control interface. Composite resources include devices like VCRs and also

recorded media, such as film clips on a laserdisc, which have multiple associated MediaPorts for audio, video, etc., when mounted on a suitable device. Composite resources can provide alternative realizations of the same Party. For example, a single Party representing the movie "Gone With the Wind" might have many resources which are laserdisc copies of the movie. In a video-on-demand application, a user's request to see "Gone With the Wind" would create a Call to the unique "Gone With the Wind" Party. The Party's Resource Manager is responsible for selecting and connecting an available (mounted and not busy) laserdisc to the Party's MediaLines in the Call. In our UNIX implementation, one of the MediaLines would be connected to a resource representing an X client program presenting play, pause, stop, etc., controls to the user for viewing the movie. At the other end, the user Party's Resource Manager selects resources to connect to that Party's MediaLines for viewing the movie. To make such connections intelligently, Resource Managers store information on default preferences for Parties. In the current implementation this is done by a simple ordering of each Party's Resources.

Just as a Party has a set of MediaLines in a Call, it has a set of Resources in a Resource Manager. The creation of Resource Managers uses the same PartyInfo structures, with Resources instead of MediaLines, as were used in the creation of Calls.

```
resourceManager newResourceManager(partyInfoSet partyInfos)
int rmAddPartyResources(resourceManager rm, partyInfo pi)
int rmRemovePartyResources(resourceManager rm, partyInfo pi)
```

Each resource is owned by one or more Parties. Parties control adding and removing resources.

```
int partyAddResources(party p, rmResourceSet rmResources)
int partyRemoveResources(party p, rmResourceSet rmResources)
```

Connections between a Party's resources and MediaLines in a Call are usually handled automatically by its Resource Manager using default preferences. However, direct control is provided for connecting and disconnecting MediaPorts on media resources to MediaLines.

```
int partyConnectLine(party p, call c, mediaLine ml, MediaPort mp)
int partyDisconnectLine(party p, call c, mediaLine ml, MediaPort mp)
```

3. Call System Implementation

The call system has evolved from a high-level control layer for a prototype desktop videoconferencing application written in object-oriented Scheme. In order to improve its modularity and robustness as an object-oriented application framework, the call system was later moved to Smalltalk. It was at that point that its usefulness for general application development and some of the analogies mentioned with window systems became evident. The current complete implementation of the call system is in C both to support application development in C and also to facilitate interfacing to existing underlying switching software. It uses the same object-oriented design, which would be expressed more naturally and succinctly in C++.

One of the lessons learned from the earlier experiments with the call system was that care must be taken in using object-oriented inheritance to create different classes of calls. Much of the generic structure of calls is dynamic while its class is fixed in most object systems. If one had classes of calls with a fixed number of parties or a fixed set of media, for example, then those calls would have to be replaced with instances of other call classes to allow parties or media to be added or removed. However, it should not be necessary to change the identity of the call object in order to change parties or media. Nevertheless, inheritance can be used profitably to create new call classes which add state and behavior to support a particular application's policies, for example, for providing distinguished parties or party roles in a call, and for logging calls.

There are two layers of switching in the call system. The first consists of controlling the abstract connections between MediaPorts and MediaLines for Parties and between pairs of MediaLines for Calls. The set of all such connections is treated as a connection graph. Each time connections are made or broken, part of the graph is traversed to determine changes in connectivity between the MediaPorts. The second layer of switching controls "actual" connections between "actual" things which those parts represent. It is the changes in connectivity in the first layer which control the connections and disconnections in the second.

Referring back to Figure 1, a physical switch will connect Party1's microphone to Party2's speaker only when three abstract connections are made: from the MediaPort for the microphone to Party1's MediaLine in the Call, from Party1's MediaLine to Party2's MediaLine, and from Party2's MediaLine to the MediaPort for the speaker. The microphone and speaker will be disconnected if any one of the abstract connections is broken.

The resulting distribution of control between Calls and Parties is an important feature of the call system. The call system can control many "actual" switches which support connect and disconnect operations for some meaningful set of ports. With slight extensions, it is possible to control multicast and broadcast mechanisms as well. In order for the two switching layers to determine equivalent sets of connections, it is necessary that the MediaPorts represent distinct "actual" ports and that the "actual" connect and disconnect operations affect only the ports determined by their arguments.

4. Applications Using the Call System

The applications presented here represent work to date in using the call system. The videoconferencing prototype is a running application while the others are in various stages of design and implementation. The range of examples so far is much narrower than the possible applications we envision for the call system. Additional examples of particular interest include enabling personal applications such as spreadsheet programs for real-time communications. For example, a communication-enabled spreadsheet program might allow a user to place calls automatically to get information for particular cells in a spreadsheet. Another interesting application would use the notion of shared workspaces described here with handheld devices which have a small display and no window system.

Desktop Videoconferencing

The call system has been used in a prototype desktop videoconferencing application developed by the Interactive Media Group at the University of Massachusetts at Lowell for the Hewlett-Packard Media Technology Laboratory at Chelmsford, Massachusetts. In this application, the call system is used to control switching of a number of analog audio and video devices through the Distributed Media Control System [UML91] developed earlier at Lowell. Parties in the application represent either end users or abstract media objects such as video clips on a laser disk.

The call system is controlled by another system component called the session manager, developed at Lowell for this application, which supports a user interface for placing and modifying calls. The session manager also provides integration with video windows using video overlay and compression boards from Fluent Corporation which Lowell has interfaced to Hewlett-Packard workstations. Part of the session manager user interface is shown in Figure 2. The session manager uses a single live video window to display the current speaker while other participants can express themselves by selecting from a set of still images which present gestures or moods.

As a call manager application, the session manager is responsible for its own floor control policies supporting selection of the current speaker who is viewed by all users. In the implementation, this is realized by connecting a single source MediaLine to all of the destination MediaLines. Using the call system API to manage the participation of users in a Call is very simple once Parties have been created for them and preferences have been provided for selecting devices for them to use in communicating.

Remote Consulting in Neurophysiology

Multimedia MedNet [Sclabassi91] is a distributed application under development at the Center for Neurophysiology at the University of Pittsburgh which allows specialists to consult on multiple neurosurgical procedures from remote locations. It enables the remote consultant to observe the operation in progress visually and communicate verbally using a cable television connection, and also to monitor neurophysiological data which are transported in real time over a computer network. Using Multimedia MedNet, it is possible for one consultant to participate in several surgeries at the same time. Figure 3 shows a prototype user interface for this system.

In this application of the call system, each consultant and each operating room is represented by a Party. A Call contains a single operating room Party but possibly more than one consultant Party. Consultants can independently select among video sources in the operating room and discuss the case over an audio channel which is shared with the operating room. The consultant can change from one operating room to another in a single action and even view multiple operating rooms simultaneously.

One of the distinguishing features of this application is the use of real-time medical data as a communications medium. It is supported in Multimedia MedNet by connecting remote clients that display the data with servers which manage the acquisition of data from the operating room. Using the call system, MediaLines can be created to control the connections between the remote display clients and the data acquisition servers. These allow a call to control the data connections in just the same way that they control connections for audio and video, which in this application involve modulating signals onto a cable television network.

Shared Workspaces

Shared multimedia workspaces are an application of the call system which we have pursued to extend and merge functionality of Hewlett-Packard's Shared X and HP VUE products for UNIX workstations. The resulting integration of VUE's support for personal workspaces with the call system's control of real-time communications services including Shared X gives rise to functionality which is qualitatively new and seems natural for supporting group work. The example in Figure 3 shows shared workspaces for consulting in brain surgeries using Multimedia MedNet.

Shared X allows multiple users to see and interact with the same X window from different displays. While this is patently a communication service, it is not yet integrated with other forms of communication. On the other hand, HP VUE provides users with multiple personal workspaces, each of which contains its own set of windows and constitutes a work context. One might use different workspaces, for example, to hold work related to different projects. At the center of the VUE dashboard shown in Figure 3 are six buttons which allow the user to select the current workspace. VUE's buttons for selecting a workspace are analogous to buttons on a telephone which select between phone calls using a hold-and-resume mechanism. With shared workspaces which are implemented using the call system, those two analogous operations become literally the same thing. Of course, the static set of buttons is an artificial limitation. Support of multiple simultaneous multimedia calls is desirable, and is provided by the call system. There are, however, design problems which need to be solved in order to support them effectively in the user interface.

The key to implementing shared workspaces as calls is to interpret Shared X as a multicasting switch which controls connections between X Windows clients and displays. An important implication of this level of control is that the connections are transparent to the client application which assumes it is running on a single display; any X client can be connected. Shared X provides the mechanisms needed to implement support for interaction with multiple users.

In terms of the call system, the notion of workspace in VUE would suggest taking a shared workspace to be a Call with Parties which have MediaLines controlling connections between X clients and displays. However, it turns out to be better not to distinguish those Calls from others, so that shared workspace becomes a synonym for Call, giving rise, for example, to shared audio workspaces. The important thing is the natural way in which personal workspace usage is extended for multimedia communications. A more complete integration awaits suitable support in X Windows for digital audio and video. The interpretation of a call as a shared workspace suggests further extensions to represent virtual meeting rooms and other computer-supported communication contexts.

5. Related Work

The Etherphone project [Swinehart87, Vin91] at Xerox PARC has done pioneering and influential work both in using LANs to transport and control real-time communications media and also in exploring highly innovative user services which take advantage of computer-controlled communications. The software architecture for the Etherphone system has some similarities and differences with the call system. It has similar intuitive notions of calls (called conversations) and parties, but information and behavior are centered in a connection manager and in agent processes which support a party in a conversation. Rather than communicating directly with the representation of a conversation, an agent communicates with the connection

manager. This provides for more centralized control. From the point of view of supporting general applications, it may be better to view such control as policy rather than mechanism, and provide for Call objects in the API as we have done. While Etherphone's conversation manager has coarser granularity than Calls, its agents for parties have finer granularity than Parties in the call system. Parties and their Resource Managers in the call system could probably be implemented easily using Etherphone agents. For a higher-level API, it may be more effective to expose Party objects with a suitable set of operations, but keep the behavior of the individual agents hidden from the application developer.

The EXPANSE [Minzer91] project at Bellcore provides a software architecture containing call and party objects for controlling multimedia communications over high-speed networks. EXPANSE's call objects are composite structures built up directly from finer-grained channel and connection objects. In directly modeling network services and capabilities, EXPANSE's software architecture is intended for controlling network infrastructures and seems better suited for that than for directly supporting general application development.

The Touring Machine [Arango92, Gopal92] project at Bellcore is distinguished for its emphasis on supporting development of communications applications through an API. Touring Machine has some strong similarities with the call system, both in its goals and in its software architecture; however, the resulting APIs are quite different. One of the important commitments for both systems is support for dynamically changing calls with multiple parties and media which users of the system can control effectively. The Touring Machine architecture contains a greater variety of objects than the call system since it supports software layers all the way from switch and network interfaces to the user interface. Touring Machine has Session objects which are similar to Call objects. However, the Session object provides support for negotiation of requests while the call system distributes that negotiation out to the Parties involved in the requests, which seems better as the variety of Parties, requests, and negotiation mechanisms in the system increases. Touring Machine has an implicit notion of a user which corresponds to a Party object in the call system and which is similarly intended to generalize the intuitive notion of user to include other media sources. The abstract switching mechanisms in Session objects and Call objects are strikingly similar. There are direct counterparts to the call system's abstract MediaLines and MediaPorts. In the Session object, connections are represented explicitly using connector values, while the corresponding connection objects in Calls are hidden in the implementation. The call system's architecture emphasizes greater generality in the notion of communication channel to provide better integration of communications with computing.

Despite these architectural similarities, the APIs of Touring Machine and the call system emphasize different things. The Touring Machine API gives more direct control of a client-server implementation and explicit support for creating and negotiating connections. To make applications easier to write and to understand, the call system emphasizes instead the operations which represent communication tasks at the level of Call and Party objects such as adding and removing Parties from a Call and suspending and resuming Calls. Much of the connection management is hidden in the call system's Resource Managers.

A rather different area of activity which deserves brief mention here is the current work on *messaging APIs* which provide general application support for electronic mail. There are currently several competing messaging API standards including X.400 [CCITT88], VIM [VIM92], and Microsoft's MAPI. These APIs enable applications to support sending and receiving mail and to do so intelligently. The call system's objective is to provide complementary support for real-time communications. It is possible that there are interesting ways to unify the asynchronous and synchronous services.

6. Summary and Conclusions

The call system's API tries to present to general application developers an intuitive set of components and operations which model basic communications tasks and to hide where possible the structural and operational features of any particular communications system. The central components of the call system are Call and Party objects. Call objects can model diverse communications contexts, from a brief telephone call to a persistent meeting room or project workspace. Parties and media can be added and removed dynamically from Calls and sets of Calls can be merged. Party objects are very general, allowing the call system to be used broadly and to support new notions of communication channels and media derived from distributed computing. The call system has been implemented and used in a few experiments so far. It needs to be subjected to further testing in practice to determine its adequacy.

The interpretation of Call objects as "windows in a communication space" is provocative. Just as graphics hardware and underlying graphics software formed the enabling technology for window systems, networking and distributed computing make possible a call system, which controls virtual rather than graphical contexts. As shown by the example of shared workspaces, it is the call system's ability to control the management, navigation, and integration of virtual contexts which gives it a distinguished role. However, there is a great deal more to graphical user interfaces than a window system. It is interesting to ask whether in the future there will be richer sets of software components, possibly including Call objects, which will yield more powerful application frameworks for computer-controlled communications to support development of applications for inter-personal computing.

Acknowledgments

We would like to thank Riccardo Gusella, Shiz Kobara, Bob Leichner, and Aaron Oppenheimer of Hewlett-Packard, Prof. John Koegel and John Rutledge of the University of Massachusetts at Lowell, and Bob Simon of the University of Pittsburgh for technical discussions and collaboration on the design, use, and implementation of the call system. We would also like to thank Allan Kuchinsky, John Limb, and Robert Wu of Hewlett-Packard and Dr. Robert Sclabassi of the Center for Clinical Neurophysiology at the University of Pittsburgh for providing support for the various parts of this work.

Bibliography

[Arango92] M. Arango, P. Bates, et al. Touring Machine: A Software Platform for Distributed Multimedia Applications. IFIP '92, May 1992.

[CCITT88] Message Handling System and Service Overview, CCITT X.400. CCITT, 1988.

[Gopal92] G. Gopal, G. Herman, and M.P. Vecchi. The Touring Machine Project: Toward a Public Network Platform for Multimedia Applications. Proceedings of the Eighth International Conference on Software Engineering for Telecommunication Systems and Services, 1992.

[Minzer91] S.E. Minzer. A Signaling Protocol for Complex Multimedia Services. IEEE Journal on Selected Areas in Communications, Vol. 9, No.9, pp. 1383-1394, December 1991.

[MM92] Proceedings of the 4th IEEE ComSoc International Workshop on Multimedia Communications, 1992.

[Sclabassi91] R.J. Sclabassi., R.L. Leichner, et al. The Multi-media Medical Monitoring, Diagnosis, and Consultation Project. Proceedings of the 24th Annual Hawaii International Conference on System Sciences, 1991.

[Swinehart87] D.C. Swinehart. Telephone Management in the Etherphone System. IEEE GlobeCom '87, November 1987.

[UML91] Distributed Multimedia Control System User Manual. Interactive Media Group, University of Massachusetts at Lowell, 1991.

[VIM92] VIM Interface Steering Group. Vendor-Independent Messaging Interface Functional Specification, Version 1.0, March 1992.

[Vin91] H.M. Vin, P.T. Zellweger, D.C. Swinehart, and P.V. Rangan. Multimedia Conferencing in the Etherphone Environment. IEEE Computer, October 1991.

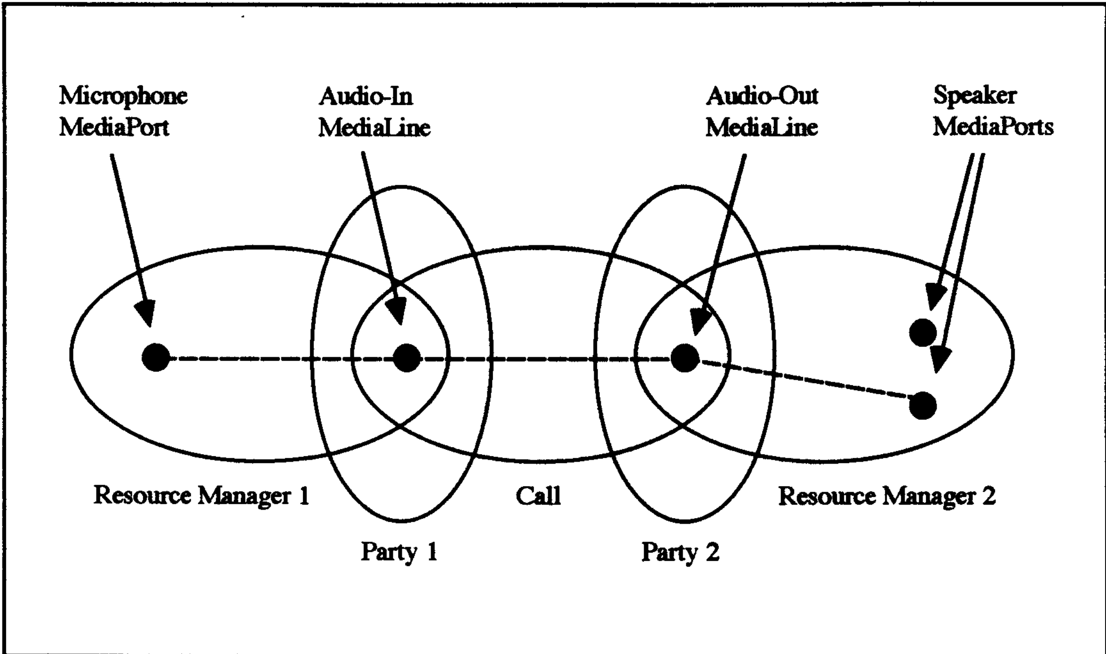

Figure 1