Session VII: Multimedia Abstractions I

Chair: Jonathan Rosenberg, Bellcore

The session "Multimedia Abstractions I" comprised three presentations about abstractions that provide a framework for supporting software that manipulates digital audio and video. These abstractions serve the same purpose as existing system abstractions, such as I/O libraries and windowing systems. They hide the complexity of underlying hardware and software and provide organized mechanisms for performing common manipulations. These abstractions also ease the design, implementation and debugging of software and promote portability. As we begin to see progress on the fundamental systems issues for supporting digital media (such as operating systems and networks), we can expect such multimedia abstractions to become more important.

The first presentation was by David Anderson from the University of California at Berkeley ("Toolkit Support for Multiuser Audio/Video Applications," David Anderson and Pamela Chan). David discussed COMET, an extensible toolkit providing a set of abstractions designed for distributed, multi-user applications that use continuous digital media. An example application is a teleconferencing system that provides multiple users with video, audio and graphics communications.

The set of abstractions provided by COMET is implemented by a set of object-oriented classes. There are classes for abstract I/O devices and audio mixers and a mechanism for programs to link objects of these classes to form the nodes of a graph. It is the specific linking of objects that defines the architecture of an application. Once an application has created and linked objects as desired, the application calls a COMET-supplied setup routine. This routine realizes the graph by creating processes and establishing communications as necessary.

In COMET, nodes of the graph must know about their neighbors to allow the calculation of delay requirements and the determination of data conversions. The nodes communicate this information among themselves by obeying a set of conventions defined by COMET. The extensibility of COMET comes from allowing programmers to define new classes and include objects of these classes within the graph. As long as the objects obey the communications conventions, they participate as full-fledged COMET objects.

Data conversions along a path are needed to match the source data format to the capabilities of processing and presentation devices. For example, an input source might provide a stereo audio data stream, while the system on which it will be played supports only mono audio. In this case, the stream must be converted from stereo to mono before it enters the speaker. The places at which data conversions are performed is important because they determine the delays along paths and the amount of network traffic.

The second presentation was by Duane Northcutt of SUN Laboratories ("System Support for Time-Critical Applications," J. Duane Northcutt and Eugene M. Kuerner). Duane began his presentation by giving his personal views about the state and future of multimedia. In his view, multimedia is currently a lot of hype with little substance. He suggested that multimedia was the application area that would evolve to support what people will do with the next generation of workstations. What will people be doing with these workstations? Duane believes that we will use the machines primarily to support communications, not to support computation as is done today.

The bulk of Duane's presentation was about abstractions for supporting digital media within a workstation. The major change required is the introduction of time as a first-class notion within workstation operating systems. To this end, Duane and his colleagues are working on a high-level operating system model for supporting time.

An operating system for supporting multimedia is composed of two parts. The "kernel" of the operating system provides fundamental resource management for the machine (such as bus access scheduling and process allocation). Sitting on top of this is the "system," which augments the kernel's capabilities to provide appropriate multimedia support to user processes. It is, however, important to note that the system level is not expected to be exported directly to application programs. Rather, libraries will be built to present a more appealing interface to the machine's capabilities.

A central notion for the system level is resource management, which must be based on time constraints to support multimedia. This will require new operating system abstractions. Duane made an analogy to network support in operating systems, which necessitated new abstractions, such as ports and sockets.

In supporting time, the researchers feel it is necessary for the definitions of time constraints to be modular. This is necessary to isolate the effects of constraints to make their implementation more manageable. In addition, this modularization aids software reuse. Furthermore, they decided to concentrate all notions of time within the new programming abstractions as opposed to distributing the mechanisms throughout the operating system interface. This avoids the need to augment existing operating system functions with additional parameters to express time constraints.

Duane closed by stating that a user-level implementation of the system exists and a kernel-level implementation has begun.

The third presentation was by Daniel Ingold of ETH in Zurich ("An Application Framework for Multimedia Communications," Stefan Frey and Daniel Ingold). Daniel discussed the design of an object-oriented framework to support multimedia applications. This work is part of the ETHMICS project, which is building a testbed for integrated multimedia communications. Their design provides abstractions for controlling multimedia devices, signaling among such devices, and controlling the presentation of media.

The design provides an object hierarchy rooted at the class Stage, which defines a means for signaling and event propagation among objects. A tree of Stage objects controls the multicasting of a single media stream. The branches of the tree are made up of alternating objects of class Device and Channel.

The Device class specifies common properties of physical resources such as input and output actions. A Device instance connects an incoming Channel to an outgoing Channel. The Device class provides methods for specifying bandwidth demands and jitter requirements.

Device objects, used to connect Channel objects, support 1-to-many transmission of a single medium stream. Device objects are responsible for requesting bandwidth and propagating information concerning synchronization requirements.

The mapping from an object to its presentation is specified by MediaView objects. For example, the VideoView subclass defines the destination area for video on a display and allows the video to be scaled and moved as any window. As another subclass of MediaView, the AudioControl object provides an interface for audio manipulations as well as for mixing multiple audio channels.

Daniel concluded his presentation by presenting an example object hierarchy that implements a simple picturephone application. This application has been partially implemented.

At the end of the presentations, there was an open discussion among the workshop attendees. The discussion was sparked by Ralf Guido Herrtwich's comment that dataflow models appear to be the appropriate basis for multimedia abstractions, but it was not clear how to turn such models into efficient low-level scheduling decisions.

One group of participants suggested that it was more important to define proper abstractions than to worry upfront about performance. Someone commented that this was an instance of a common, but mistaken view: if one gets the abstractions correct, then appropriate performance "magically" follows. The response to this was that there is no substitute for basic engineering competence. If an abstraction cannot be implemented efficiently, it is due to poor design engineering.

This led to a heated discussion about the importance of obtaining experience from building applications as opposed to pushing on fundamental technology. This is the classic "technology push" versus "demand pull" argument. On the side of demand pull, it was argued that proceeding without applications experience was likely to produce "solutions looking for a problem," technology with no apparent practical application. The other side, technology push, stated that multimedia was a field that was fundamentally limited by technology. Therefore, it was virtually impossible today to build many of the promising multimedia applications, or even to conceive of them given today's technology.

TOOLKIT SUPPORT FOR MULTIUSER AUDIO/VIDEO APPLICATIONS

David P. Anderson
Pamela Chan

Computer Science Division, EECS Department
University of California at Berkeley
Berkeley, California 94720

October 25, 1991

ABSTRACT

Comet is a UNIX/C++ toolkit for writing programs that involve multiple users and that use digital audio and video. Comet provides a simple programming interface: the application builds a graph of objects representing speakers and microphones, mixers, files, and so on. Comet then realizes the graph by creating processes to handle mixing and file I/O if needed, and linking them by network connections to audio/video I/O servers. In addition, Comet addresses the interrelated issues of client requirements and resource management. It determines delay and throughput requirements, process placement, and data type conversion; it deals with resource managers on the application's behalf. These mechanisms are based on a negotiation protocol among the components of the object graph.

1. INTRODUCTION

''Multiuser CM applications'' are those that 1) use audio/video media (*continuous media*, or CM), and 2) are run concurrently by multiple users at different locations, allowing real-time interaction and collaboration. Such applications typically provide conferencing (each participant hears and perhaps sees the other participants) and may also involve storage and playback of CM material. However, applications may differ in their system-level requirements and preferences: For example, distributed music rehearsal needs low delay while distributed music recording needs high data quality and low loss.

Integrating CM data in the standard framework (operating system, networks, file system, user programs) provides many advantages, but it makes multiuser CM application difficult to develop. First, they are highly distributed: a typical application might involve a graphical user interface program running on each user's workstation, servers on each workstation for discrete- and continuous-media I/O, processes for audio mixing and file I/O, and a central process to manage conference membership. Second, the application must determine its performance requirements and inform the underlying system, perhaps to reserve resources. Finally, since many implementation details depend on hardware and network properties (such as the availability of multicast), it is hard to write portable programs.

We believe that many of the implementation issues of multiuser CM applications should be handled by a software layer that we call a *multiuser CM toolkit.* Applications based on such a toolkit specify the desired CM functionality in high-level terms, and the toolkit manages the details. In a distributed system whose components provides real-time semantics (network channels, CPU scheduler, real-time file systems, *etc.*) the CM toolkit layer acts as an "overseer" that interacts with these system components to determine an application's implementation structure and the corresponding performance requirements.

As a proof of concept and to explore the issues, we are developing a multiuser CM toolkit called *Comet.* Comet supplies a set of C++ classes representing abstract CM components: microphones, speakers, files, audio mixers, and so on. The application creates instances of these classes and links the objects to form a dataflow-style graph. Comet then implements this abstract graph by 1) creating processes, if needed, to handle CM data, and 2) connecting existing server processes to each other and to the new processes. It determines the delay and throughput requirements of each component, and informs resource managers accordingly. Hardware and network properties are encapsulated within Comet classes, so they are transparent to the programmer and can be changed easily.

A prototype version of Comet has been completed, and has been used as the basis for a transcribed conferencing application. The implementation of Comet uses ACME [2], a server that provides shared network-transparent access to workstation CM I/O devices. Comet also uses the IPC and I/O multiplexing features of InterViews [5], a C++ toolkit for X11. However, the principles of Comet do not depend on ACME or InterViews.

2. COMET FEATURES

2.1. Master/Slave Architecture

Comet applications have a master/slave structure. Each participant runs a separate instance of the slave program, and there is a single instance of the master program. The master program can run as a daemon at a well-known host, or can be run on demand by the slave program. The hosts must run some version of UNIX, but need not share a common file system; Internet TCP connections are used for all communication.

The slave program provides the user interface (handling mouse/keyboard events and generating window system requests) while the master program maintains global state and issues CM-related commands to Comet. In a basic conferencing application, for example, the slave provides mouse-based controls for joining and leaving conferences, while the master maintains conference membership and tells Comet how CM I/O devices and files are to be interconnected. In a collaborative editing program, editing may be local to the slave, but any globally visible changes must be propagated through the master.

Communication between the master and the slaves is done using remote procedure calls (RPCs). RPCs may be initiated at either end, and include application-defined RPCs as well as Comet's internal RPCs. The master and slave programs are single-threaded. Comet uses the InterViews Dispatch library to multiplex among I/O sources (window system events and RPCs on the slave, RPCs from multiple slaves on the master).

2.2. Continuous Media Objects

The master program manages the continuous media (audio/video) components of the application. The Comet toolkit on the master side provides a set of C++ classes, derived from a base class `CM_NODE`, that represent these components. The current set of `CM_NODE` types is as follows:

- `MICROPHONE` and `SPEAKER` represent CM I/O devices. These devices are "abstract" in the sense that several of them (perhaps from different Comet applications) may be mapped simultaneously to the same physical I/O device. Constructor arguments specify the slave where the device is located.
- `INPUT_FILE` and `OUTPUT_FILE` represent disk files storing CM data. Constructor arguments specify the file name, the slave whose file system stores the file, and (for `OUTPUT_FILE`) CM data type preferences.
- `AUDIO_MIXER` represents a component that takes in N audio streams and generates as output the sums of each $N-1$ of these streams, as well as the sum of all N. Each input and output has a corresponding `PORT` (see below).
- `SOURCE_PROCESS`, `SINK_PROCESS`, and `FILTER_PROCESS` represent processes executing user-defined programs that act as a CM data source, sink and filter respectively. Constructor arguments specify the program name and the slave where the program is stored.

2.3. The `CM_NODE` Graph

A `PORT` object represents a source or sink of CM data. A `CM_NODE` has one or more associated `PORT`s. The operation

```
p->join(PORT* q)
```

creates an (abstract) connection from output port `p` to input port `q`. Each port can have at most one such connection.

Using `join()`, the application builds a directed graph of `CM_NODE`s. When the `CM_NODE` graph has been constructed, the application calls `setup()` on any node of the graph. This call causes Comet to realize the graph (see Section 3) and starts the flow of data between the devices, processes, and files represented by the graph.

The `CM_NODE` graph may be modified (by adding or removing nodes or links) during program execution. For example, as new participants join a conference, new `MICROPHONE` and `SPEAKER` objects are created and connected to the `AUDIO_MIXER`. After an addition is complete, the application must call `setup()` on a `CM_NODE` in each modified subgraph.

3. IMPLEMENTATION

Comet must convert a graph of `CM_NODE`s (which are simply C++ objects in the master program) to a set of processes and network connections that realize the graph. Our approach is modular: each `CM_NODE` manages its own implementation (this facilitates adding new `CM_NODE` types). To implement itself, a `CM_NODE` needs information about its neighbors. For this purpose, Comet defines a C++ interface, which we call the "CM_NODE Protocol"[1], between CM_NODEs.

The CM_NODE Protocol addresses four issues: 1) whether data streams require low delay, 2) the data representation on each stream, 3) the message size on each stream, and 4) connection establishment. The protocol defines a set of functions that each CM_NODE must provide. These functions, enumerated in the following sections, are implemented differently in each derived class of CM_NODE.

3.1. Delay Bound Determination

If end-to-end delay exceeds 200 milliseconds or so in a conference application, conversation becomes difficult. When CM data is being written to a file, however, the end-to-end delay is unimportant. We define a CM data stream to be *low-delay* if it involves data being sent between human users (*e.g.*, from a microphone to a speaker) and *high-delay* otherwise. For many CM components, the optimal handling of CM data depends on whether it is low- or high-delay: Low-delay streams must be handled by high-priority processes and cannot tolerate the delay of buffering, while high-delay data streams can be buffered and can tolerate high processing delays.

The CM_NODE Protocol allows each CM_NODE to learn whether the data streams it handles are low- or high-delay. The function from_rt_source(PORT*) returns True if the data stream from the given output PORT includes data originating from a "real-time" source (a CM input device). Similarly, to_rt_sink(PORT*) returns True if the data stream entering the given input PORT is destined for a real-time sink (a CM output device).

The implementation of these operations is type-specific. For example, AUDIO_MIXER implements to_rt_sink() by examining the outputs to which the given input stream contributes, calling to_rt_sink() on each of the input ports connected to these outputs, and returns True if any of these calls returns True. The result is then cached in the PORT object for subsequent calls (this is necessary to avoid infinite recursion).

3.2. Data Type Negotiation

The "type" of an audio stream is determined by the number of samples per second, the number of bits per sample, and possibly a logarithmic compression of samples. For video, the type includes the image size, the number of frames per second, and so on. We assume that

- There is a fixed finite set of CM data types.
- There is a partial order $<$ on the set of data types. $S < T$ means that S has less information, and typically a lower data rate, than T.
- For any types S and T there is a least type $U = sup(S, T)$ such that $U > S$ and $U > T$.

Some pairs of types, such as 44 KHz mono and 22 KHz stereo, may be incomparable; the preference of one over the other is then application- or user-specific (see Figure 1).

[1] We call it a "protocol" because function calls must be made in a certain order; it is not a network protocol.

The CM_NODE Protocol allows CM_NODEs to negotiate the data types of the CM data streams that connect them. The goals of the negotiation is to find an assignment of types to streams that 1) is feasible (the conversions are implementable by the CM_NODEs); 2) provides the maximum possible quality at the outputs; and 3) minimizes network traffic by doing conversions as far "upstream" as possible.

A CM_NODE provides functions

```
    TYPE_SET feasible_types(PORT* p);
    TYPE_LIST prime_types(PORT* p);
```

where p is an input PORT of the CM_NODE. feasible_types() returns the set of types the CM_NODE is able to accept on the port. prime_types() returns the list of feasible types that have no redundant information; *i.e.*, for which no lesser type will produce identical output. The list is sorted by decreasing preference, allowing the CM_NODE to rank incomparable types.

For an output device such as SPEAKER, prime_types() depends on the speed and width of the DAC. For example, the prime types for a 22 KHz monoaural DAC might be 22 KHz mono and 8 KHz mono (in that order). 44 KHz mono is not prime because it would produce the same output as 22 KHz mono. Incomparable types can be ranked, if desired, by the application. feasible_types() depends on the conversion capabilities of the I/O server. If the server can convert 44 KHz stereo to 22 KHz mono in real time (along with its other tasks) it would list 44 KHz stereo as a feasible type.

The determination of prime and feasible types for AUDIO_MIXER is more complex. Suppose I is an input stream of the mixer, and let S be the set of output streams to which I contributes. A type T is feasible for I if, for each $s \in S$, the mixing agent[2] is able to mix input in format T with the other inputs of s, and convert the result to a type that is feasible for the destination port of s. The prime type list for I might be determined as follows. Fix a type T and a stream $s \in S$. Let $<P_1 \cdots P_n>$ be the prime types of the destination port of s. Let $V(s)$ be the least i such that $Pi < T$ ($V(S)$ is the "value" of type T for stream s). Define $R(T)=\sum_{s \in S} V(S)$; $R(T)$ is the aggregate value over all output streams to which I contributes. The prime_types list is then formed as follows: enumerate all types by increasing value of $R(T)$; delete from this list types T that are not feasible or for which there is a type U such that $R(T)=R(U)$ and $U < T$. An example is shown in Figure 2.

Each CM_NODE supplies a function

```
    TYPE actual_type(PORT* p);
```

where p is an output PORT of the CM_NODE. This returns the actual type to be output on the given PORT. The policy is type-specific. For example, an AUDIO_MIXER might use the following policy to determine the actual type for an output stream R. Let $S = s_1 \cdots s_n$ be the input streams that contribute to R. Call actual_type() on the nodes that generate these streams to learn their types $T_1 \cdots T_n$, and let $\overline{T} = sup(T_1 \cdots T_n)$. Let U be the set of types S in the feasible set of the destination

[2] This agent may be an I/O server or a separate mixing process; see Section 3.6.

such that the mixer can convert the output stream to S. Let $T_0 \cdots T_n$ be the prime types of the port to which R is connected. Let i be the least such that $T_i < \overline{T}$ and there is a type $T \in U$ with $T_i \leq T$. T is then the actual output type of R. An example is shown in Figure 3.

3.3. Message Length Negotiation

The length of messages (the units in which data is written on connections) is an important issue for low-delay connections. There is a tradeoff between packetization delay and per-message overhead. To minimize delay and overhead simultaneously, the temporal message size on data stream should approximate the maximum of the I/O interrupt periods of the input and output devices (see Figure 4).

To allow negotiation of message size, each `CM_NODE` provides a function `message_length(PORT* p)`. If `p` is an input `PORT`, this returns the largest message length (measured in milliseconds) that will minimize packetization delay. For a `SPEAKER`, this is determined by the device interrupt period. For an input stream S of an `AUDIO_MIXER`, it is the minimum of the `message_length()` values of the input ports to which S contributes.

If `p` is an output `PORT`, `message_length()` returns the actual message length to be sent on `p`. In general, this is computed as $max(N, M)$, where N is the message length of the connected input `PORT` and M is the minimum of the message lengths of data streams that contribute to `p`.

3.4. Connection Establishment

Comet uses TCP connections to convey CM data. Each TCP connection is used as a simplex channel. By convention, the sending end plays the active role in connection establishment (in BSD UNIX terminology, the sender does the `connect()` and the receiver does the `listen()` and `accept()`). Each `PORT` object includes a network address (host Internet address and port number) for the corresponding TCP socket. Each `CM_NODE` must provide the following operations:

```
bind();
connect();
accept();
```

`bind()` creates listening sockets for all input `PORT`s, and stores their addresses in the `PORT` objects. `connect()` sets up outgoing connections for the object's output ports. `accept()` accepts incoming connections, finishes setup in a class-specific way. Each function then performs the same operation on all neighboring objects. The `setup()` function (called by the application to "activate" a `CM_NODE` graph) simply calls `bind()`, `connect()` and `accept()` on the target `CM_NODE`; these calls eventually propagate throughout the graph.

3.5. Collapsing Subgraphs

It is sometimes useful to "collapse" portions of the `CM_NODE` graph, and have one `CM_NODE` assume the responsibility for implementing some of its neighbors. For example, an `AUDIO_MIXER` object handles the implementation of any `MICROPHONE` or `SPEAKER` objects to which it is connected (see Section 3.6). This is accomplished by having a type field in the CM_NODE base class. The setup() function of MICROPHONE checks if it is connected to an AUDIO_MIXER, and if so it simply forwards the call there.

3.6. Implementation of Some CM_NODE Types

Abstract CM I/O devices (MICROPHONE, *etc.*) are implemented using ACME. When a slave arrives, the master sets up an RPC connection to the ACME server on the user's workstation. If a CM I/O device is connected to an AUDIO_MIXER, then the graph is collapsed and the implementation is left up to the AUDIO_MIXER (see below).

File I/O (INPUT_FILE and OUTPUT_FILE) is implemented using the Comet slave library. The bind() and accept() operations for an OUTPUT_FILE, for example, makes RPCs to the Comet library in the appropriate slave. The accept() handler accepts a CM connection, then creates an I/O activity (using the InterViews I/O multiplexer) that reads data from the socket and writes it to a disk file.

An AUDIO_MIXER object can implement itself in either of two ways (see Figure 5). Both implementations use an *audio mixer program* that runs on the master host. The mixer program takes an arbitrary number N of digital audio input streams, and produces output streams for the total sum and for each $N-1$ sum. It performs conversions between different data types, and it distinguishes between low- and high-delay streams, dealing appropriately with each one.

The constructor for AUDIO_MIXER takes as an argument the estimated number K of MICROPHONEs and SPEAKERs to be directly connected. If K is below a system-dependent threshold, a *distributed* implementation is used: AUDIO_MIXER examines its neighboring nodes and, for MICROPHONEs and SPEAKERs, connects the corresponding ACME servers directly. Each ACME server receives audio streams from the other ACME servers, and mixes them itself. If other objects (*e.g.*, files) are also connected to the mixer, then an audio mixer process is created to handle all such objects.

The above approach (direct interconnection of ACME servers) minimizes delay, but does not scale well: it generates $O(N^2)$ network traffic, and causes each ACME server to do $O(N)$ work for mixing. Therefore, if K exceeds the threshold, AUDIO_MIXER implements itself using a central mixing process only; this reduces network traffic to $O(N)$ and ACME workload to $O(1)$, at the cost of roughly doubling the delay.

Other implementations are possible. Network multicast capabilities could be exploited. If a mixer joins devices clustered in two LANs connected by a WAN, it could by implemented by a central mixer processes in each LAN, with the two processes linked by a single WAN connection.

4. RELATED WORK

Comet was inspired by graphical user interface toolkits such as InterViews [5]. These toolkits provide abstractions (menus, editors, *etc.*) for displaying and interacting with discrete data such as graphics and text. Comet is concerned purely with continuous media.

A *centralized CM toolkit* provides access to CM I/O at a single workstation. Such a toolkit, in combination with an existing window-system toolkit, makes it easy to define objects for acquiring and playing sound; these objects can then be added to mail or document systems. The Andrew Toolkit [7] takes this approach; we also developed a centralized toolkit for ACME that provides similar functionality. While this approach is useful for some applications, it has two fundamental limitations. First, it does not support multiuser applications well because management of global state, such conference membership and floor control, requires complex protocols for synchronization and failure handling. Second, handling CM data in the client program is nonoptimal in some situations: the telephony application directly connects the I/O servers, and file playback may be done more efficiently by sending data directly from a file server to an I/O server rather than having data pass through the client program.

A system described by Bates and Segal [3] offers object-oriented software layers for writing multi-user CM applications. The abstractions are similar to those of Comet: I/O, storage, mixing and multicasting. Ludwig [6] describes an analogous system based on a hierarchical dataflow model. These systems differ from Comet in that they are grounded in the telecommunications world. Because the underlying hardware is a dedicated network or crossbar switch, they have no mechanisms corresponding to the `CM_NODE` protocol. They are targeted at developing "services" rather than user-defined applications, so they do not emphasize integration with existing programming environments.

Other distributed CM toolkits are based on a "conference server", reflecting the viewpoint that the CM portion of most multiuser CM applications is simply generic audio/video conferencing. Examples include MMConf [4] and CoLab [8]. Comet takes a different viewpoint: we think of conferencing as itself being a class of applications with a broad range of requirements (scalability of the number of speakers and listeners, delay bounds, synchronization of recorded material playback, floor control policies, security, *etc.*).

5. CONCLUSION

Like other systems for multiuser audio/video applications, Comet provides a set of classes representing CM resource types; low-level details and hardware dependencies are hidden within the implementation of these classes. Comet makes two additional contributions. First, the `CM_NODE` Protocol, embodied in the C++ interface between `CM_NODE` objects, addresses issues that are crucial for handling CM data in general-purpose distributed systems: delay bounds, data type, message size, and connection establishment. Because these issues are dealt with in a uniform way, new `CM_NODE` types can be added easily. Second, Comet's master/slave model simplifies the management of global state (handled by the master) while providing fast response to local GUI interactions (handled by the slaves).

The Comet prototype lacks some key features. Comet provides audio capabilities only; we expect that many of the ideas will extend to video. Since the CM data graph is closed, cross-linking between discrete and continuous media (as would be needed for speed recognition or synthesis) is not possible. Comet should be combined with a system for CM data storage, linkage and indexing more sophisticated than the UNIX file system. Finally, it would be convenient to have classes that combine GUI and CM (*e.g.*, a control panel for playing a sound file).

Similarly, the Comet implementation might be extended in several ways. Currently, mixer processes are run on the master host, and user-defined processes and file I/O on the slave host. More generally, these processes might be placed on faster or less loaded machines, perhaps taking network communication costs into account as well. Beyond load-balancing, one might ask that resources (CPU time, network and disk bandwidth) be reserved so that the application will receive a guaranteed performance level. Issues of distributed resource reservation are discussed in [1].

REFERENCES

1. D. P. Anderson, "Meta-Scheduling for Distributed Continuous Media", UC Berkeley, EECS Dept., Technical Report No. UCB/CSD 90/599, Oct. 1990.
2. D. P. Anderson and G. Homsy, "A Continuous Media I/O server and its Synchronization Mechanism", *IEEE Computer*, Oct. 1991.
3. P. C. Bates and M. E. Segal, "Touring Machine: A Video Telecommunications Software Testbed", *First International Workshop on Network and Operating System Support for Digital Audio and Video*, Berkeley, CA, November 8-9, 1990.
4. T. Crowley, P. Milazzo, E. Baker, H. Forsdick and R. Tomlinson, "MMConf: An Infrastructure for Building Shared Multimedia Applications", *Proc. 1990 CSCW Conference*, Oct. 1990, 329-342.
5. M. Linton, J. Vlissides and P. Calder, "Composing User Interfaces with InterViews", *IEEE Computer 22*, 2 (Feb. 1989), 8-22.
6. L. Ludwig, "A Threaded/Flow Approach to Reconfigurable Distributed Systems and Service Primitives Architectures", *Proc. of ACM SIGCOMM 87*, Stowe, Vermont, Aug. 1987, 306-316.
7. A. J. Palay, "The Andrew Toolkit: An Overview", *Proceedings of the 1988 Winter USENIX Conference*, Dallas, February 9-12, 1988, 9-21.
8. M. Stefik, G. Foster, D. Bobrow, K. Kahn, S. Lanning and L. Suchman, "Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings", *Comm. of the ACM 30*, 1 (Jan. 1987), 32-47.

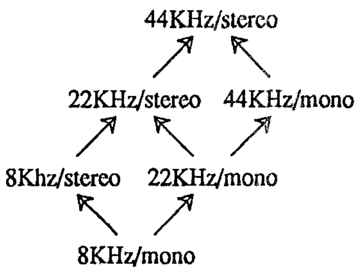

Figure 1: Comet assumes that the data types for a particular medium (audio or video) are partially ordered by <; $S < T$ means that S contains less information than T. Audio types might be ordered as above (an arrow from S to T means $S < T$).

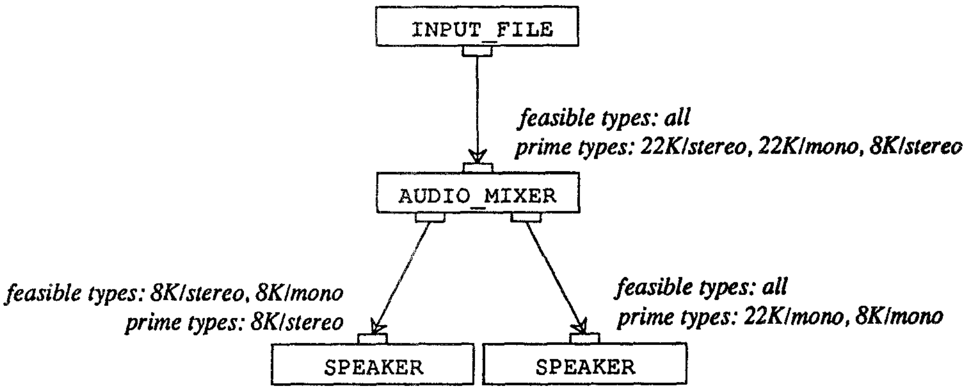
INPUT_FILE
feasible types: all
prime types: 22K/stereo, 22K/mono, 8K/stereo
AUDIO_MIXER
feasible types: 8K/stereo, 8K/mono
prime types: 8K/stereo
feasible types: all
prime types: 22K/mono, 8K/mono
SPEAKER
SPEAKER

Figure 2: Each input port is assigned a set of *feasible* types and a list of *prime* types. For output devices, these are determined by the hardware of the I/O server. For other CM_NODE types, they are determined by the corresponding lists from the ports to which the data is destined, and by the conversion capabilities of the CM_NODE.

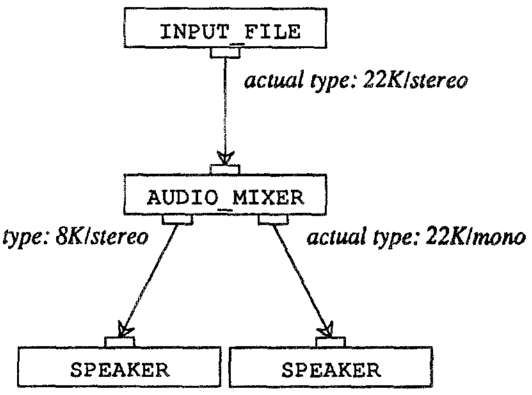

Figure 3: A CM_NODE selects the actual data type to be sent on each output PORT based on the feasible and prime types of the corresponding input PORT. In this example (continued from Figure 2), the INPUT_FILE stores 44KHz stereo data. Because this high quality is not usable, it converts the data to 22 KHz stereo before sending it to the AUDIO_MIXER.

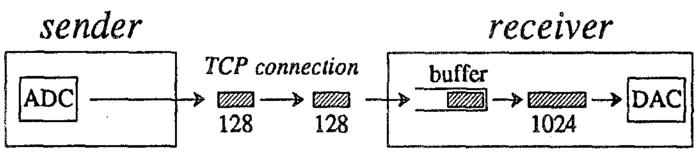

Figure 4: In this example, 128-sample messages are being sent to an output device with a 1024-sample interrupt period. Since the device must receive 8 messages before it can output, it would be more efficient to use 1024-sample messages. Increasing the message size beyond 1024, however, would increase packetization delay.

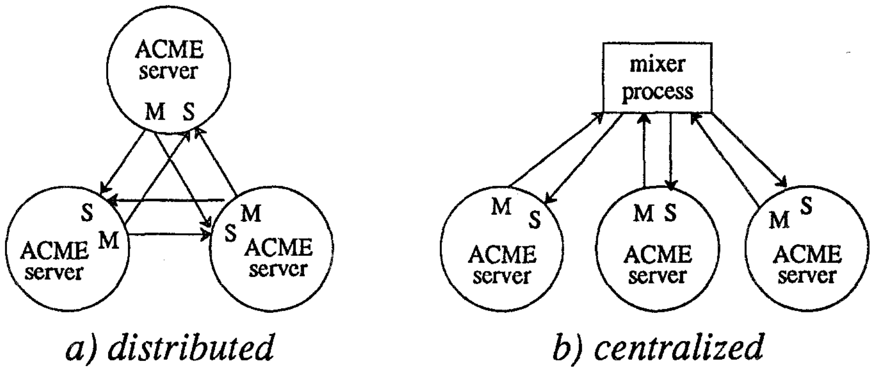

Figure 5: An AUDIO_MIXER object can implement itself in either of two ways. If the number N of participants is small (a), it directly interconnects the microphone (M) of each ACME server to the speakers (S) of the other servers, and has each server do its own mixing. If N is large (b), it creates a separate mixer process; each ACME server sends its input to the mixer process and receives a mixture of the other $N - 1$ inputs.