Integrating Audio and Telephony in a Distributed Workstation Environment

Susan Angebranndt (susan@wsl.pa.dec.com)
Richard L. Hyde (rich@wsl.pa.dec.com)
Daphne Huetu Luong (luong@wsl.pa.dec.com)
Nagendra Siravara (siravara@wsl.pa.dec.com)
Digital Equipment Corporation

Chris Schmandt (geek@media-lab.media.mit.edu)
MIT Media Lab

Abstract

More and more vendors are adding audio, and occasionally telephony, to their workstations. At the same time, with the growing popularity of window systems and mice, workstation applications are becoming more interactive and graphical. Audio provides a new dimension of interaction for the user, and the possibility of a powerful new data type for multi-media applications.

This paper describes our architecture for the integration of audio and telephony into a graphics workstation environment. Our framework is a client-server model; at the heart is an audio server that allows shared access to audio hardware, provides synchronization primitives to be used with other media and presents a device-independent abstraction to the application programmer.

1. Desktop Audio

This paper describes a server designed to provide the underlying audio processing capabilities required by families of workstation-based audio applications. It is an essential component of the concept of *desktop audio*, a unified view encompassing the technologies, applications, and user interfaces to support audio processing at the workstation. This section describes the technologies underlying desktop audio, and some example applications. In addition we discuss requirements of both the applications and their user interfaces, because these dictate some of the features and performance required of an audio server.

1.1. Technologies

The basic technology for manipulation of stored voice in a workstation is *digitization*, which allows for recording and playback of analog speech signals from computer memory. At one extreme, telephone quality recording requires 8,000 bytes per second; at the other extreme the quality of a stereo compact audio disc consumes just over 175,000 bytes per second. The low end of this scale is easily within the performance of current workstations, and basic digitization is becoming commonplace. Some workstations already support CD quality coding, and this, too, will become universal within the next several years.

Text-to-speech synthesis allows computers to convert text to a digital speech signal for playback. Synthesis is usually broken into two processing steps. The first step converts the text to phonetic units; although a linguistically difficult task, this is most easily implemented on a general purpose processor. The second step is a vocal tract model capable of generating an appropriate waveform from the units generated by the first step; this has traditionally been performed on a digital signal processor.

Speech recognition allows the computer to identify words from speech. Speech recognition usually employs a digital signal processor to extract acoustically significant features from the audio signal, and a general purpose processor for pattern matching to determine which word was spoken. Although there is much talk about the "listening typewriter" which can convert fluent speech to a text document, this is well beyond the capabilities of currently available recognizers, which have small vocabularies and require both a careful speaking style as well as head-mounted microphones or an acoustically controlled environment.

Proceedings of the Summer 1991 USENIX Conference, June 10-14, 1991. Pages 419-435.

The *telephone* can be thought of as a voice peripheral, just like a loudspeaker, and is a key component to desktop audio. Voice messages and applications for remote telephone-based workstation access will likely be a primary source of stored voice used as a data type. A small amount of electronic circuitry can provide an interface to analog telephones. ISDN, the international standard for digital telephony, is driven by a data communication protocol which can be easily managed by modern workstations.

Until recently, most audio devices required special purpose hardware, resulting in the associated high cost of audio systems. But with faster workstations and plentiful memory, more and more audio processing can be implemented on the workstation itself, with little or no special hardware. The real-time requirements of managing a stream of 64 kilobit per second voice or the even slower data link of an ISDN telephone connection are well within the capabilities of existing workstation platforms. Many speech processing techniques which have traditionally been implemented on DSPs are now within the capabilities of general purpose microprocessors. The upshot of these developments is that audio is about to become universal, provided that adequate applications with well-designed interfaces can be made available to users.

1.2. Applications

Audio processing employing the technologies enumerated above will be used by a variety of applications. They will take advantage of various attributes of voice: its richness, its primacy in human communication, the ease with which we transmit it over a distance by telephone, and our ability to speak and listen while performing other, non-audio tasks.

With the ability to control the telephone, a workstation can be used to place calls from graphical speed dialers, an address book, or telephone-based "dial by name" (which allows the caller to enter a name with touch tones). Workstation-based personal voice mail allows graphic display and interaction with voice messages, and can provide the ability to move messages to other voice-capable applications, such as an appointment calendar. Voice and text messages can be merged into applications that provide for screen or telephone access to each.

Because it is rich and expressive, voice can be very useful for annotating text, such as the marginal notes on a document under review or as a quick header to a forwarded message in some other medium. Stored voice can be used more formally as an essential component of multi-media presentations.

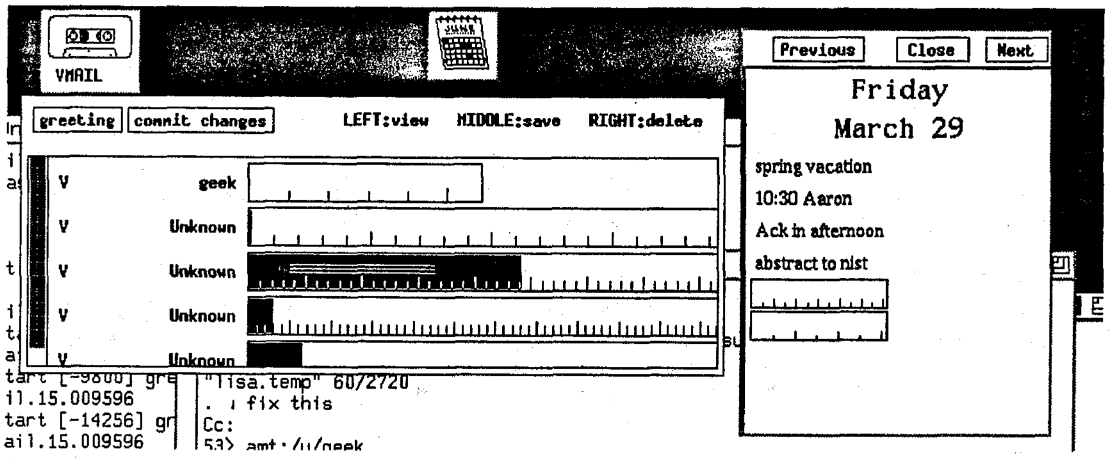

Figure 1-1: A graphical user interface for voice mail, on the left, allows telephone messages to be moved to a users calendar, on the right, in two applications developed at the MIT Media Lab.

Speech synthesis and recognition allow for remote, telephone-based access to information accessible by the workstation. Speech recognition can increase user performance for hand-and-eye busy applications such as CAD and management of the window system. Synthesized speech or playback of distinctive sounds can be much more effective for alerting than the universal "beep" employed in UNIX[1] applications such a "biff", "talk", "wall", editors, and broadcast system messages.

1.3. Application requirements

As inviting as these applications may sound, in general no single application can justify exclusive use of the workstation audio hardware. Instead, we envision a wide variety of applications, of which those just mentioned, all running in concert. These applications will make use of audio as a data type, and the user must be able to move audio between applications and transmit it between sites.

This is really not very different from current uses of text; a user may invoke an editor while sending electronic mail, or copy a portion of the text displayed in one window into an application running in another. For an example in the audio world, consider Figure 1-1, which shows window-based graphical user interfaces for copying sound between applications.

The various audio applications will be designed independently and run as separate processes. But they must share limited resources, such as speakers and microphones, just as window applications share screen pixels and the mouse. These applications need to communicate among themselves to coordinate the sharing of data and allow for more powerful management of the applications by the end user; such a communication mechanism must support inter-process message interchange.

1.4. User interface requirements

Although all these applications can make powerful use of audio, the medium is intrinsically difficult to employ in computer applications. Stored voice is awkward to handle since we cannot yet perform keyword searches on it. It is slow to listen to (although fast to create), and is a serial medium because of its time-dependent nature. A loudspeaker broadcasts throughout a room, disturbing others in the area and allowing them to hear possibly personal messages. Finally, the technologies themselves are currently limited; synthesized speech is difficult for unaccustomed listeners to understand, and speech recognition simply does not work very well.

Because of these limitations, graphical interaction styles will dominate when a screen is available. Graphical representations, such as the SoundViewer widgets depicted in Figure 1-1, can provide both visual cues to the duration of the sound as well as a means of interacting with playback and navigating within the sound.

Both audio playback and interactive audio systems have stringent real-time requirements. Although playback of stored voice may not require a high data bandwidth within the workstation, once playback starts it must continue without the slightest interruption; this is very different from refreshing a static text or graphic display. Since voice recognition and even touch tone decoding are quite error prone, the user needs immediate feedback that input has been recognized. In a voice-only interface this is essential but may be difficult.

Because of the difficulties building effective audio interfaces and applications, users have yet to realize its full potential. Many applications written to date have employed weak interfaces; interfaces must be designed with full appreciation of the limitations in the voice channel. For the near future we can expect, and must encourage, a great deal of experimentation in the design of audio user interfaces, both screen- and telephone-based.

2. Requirements of desktop audio

The previous section described the world of desktop audio. This view assumes access to a variety of audio processing technologies, largely implemented as software, from a number of applications running simultaneously, with demanding user interface requirements. What is needed from a software architecture to support such applications?

[1]UNIX is a trademark of AT&T Ball Laboratories

The primary need is *resource arbitration*. Applications should be written without having to worry about mechanisms to share resources with other applications (although they will have to have a strategy to deal with not gaining access to a critical resource).

Another aspect of resource management is *sharing* and allowing multiple applications to use the audio hardware. For instance, the multiplexing of output requests from a number of applications to a single speaker, to be heard simultaneously. Or distributing words detected by a speech recognizer to the proper applications.

It is also important that the software interface be *device independent* regardless of actual workstation hardware. Device independence provides for portability, which encourages application developers, and allows workstation vendors to introduce more powerful hardware devices without abandoning previously written applications.

This interface should also provide applications with *networked access* to resources. This promotes sharing of applications and data, and allows a user to more easily access applications when not at his or her own workstation. Additionally, networked access allows many workstations to share critical or expensive resources which cannot easily be replicated on every desk.

The software underlying desktop audio applications must support the *real-time* requirements of maintaining an uninterrupted stream of audio data once playback starts. Quality user interactions demand the ability to start and stop audio playback quickly, and deliver input events to applications with little latency. Audio operations must be *synchronized* to support seemless playback of multiple sounds in sequence, or to quickly transition into record mode after playback of a voice prompt while taking a message. Audio operations must also be be synchronized with other media, to support both multi-media presentations as well as the use of graphical user interfaces that control audio playback.

Because audio can be stored using a variety of encoding methods, it is useful to support *multiple data representations* at a level below the application. This is important for several reasons. First, over the next several years users will demand higher quality speech coding and workstations will become fast enough to support this; if every application must be rewritten progress will be delayed. At the same time, improved compression techniques will be used to transmit voice across local and wide area networks, reducing bandwidth at the price of data representation complexity. Applications should be sheltered from this.

Finally, audio support should be *extensible* to support new devices and signal processing algorithms as they emerge. Our approach is to provide a device subclassing mechanism in the server, allowing extension of the class hierarchy using existing protocol capabilities.

3. Why a client - server model?

In order to satisfy the requirements of the types of applications and technologies described above, we have built an audio server. A server, running as a single, separate process to support a number of clients (applications) simultaneously, is advantageous for several reasons. A server approach has been utilized successfully in the past, across a wide range of applications, technologies, and operating systems [2], [1], [4].[2] The server concept has been widely accepted for window systems, and in fact we can apply much of what we have learned from our experience with the X Window System[3] [3] to the audio domain.

A server is required for resource sharing and arbitration across multiple applications; there must be some point at which all application requests meet so that resource contention can be arbitrated. Use of a well-defined protocol for communication with the server allows the application to remain compatible across multiple vendors' server implementations, and its device-independent nature shelters the application from hardware differences.

[2]The Olivetti VOX audio server introduced the concepts of typed server-side entities (corresponding to devices) and application compositing of devices, and also borrowed heavily from X. There are a number of similarities between VOX and our audio protocol. The design of the prototype VOX server was more oriented towards control of analog audio devices.

[3]X Window System is a trademark of The Massachusetts Institute of Technology.

Use of a separate process for control of a real-time medium such as stored voice simplifies application design in many respects. Audio playback requires repeated calls to buffer management routines to move digital audio data from disk storage to audio hardware. Audio streams may be mixed, or effects added, by introducing more buffers and operators upon them. Continual feeding of buffers is a distraction for application software, which is more easily written and maintained as a set of routines, each responding to some user stimulus or server-generated event such as sound playback completion.

Although many parts of the audio server internals can be modeled after the X server, it is important to keep the two separate. The server for a window system has very different real-time requirements than an audio server; they cannot be merged without serious design compromises for both. Adding audio to a graphics server adds needless complexity to components supporting each medium. Telephone-based audio applications may well run on a workstation not running any window system, and should not be burdened by many lines of display support code.

The strongest argument for merging media into a single server is *synchronization* between media. A single server can provide internal methods of controlling one medium as a function of progress though another, which is useful if the media have varying latencies or throughput characteristics. This is a minor problem; most synchronization will happen in response to some user input rather than the internal state of the server. This entails round trips for messages between client and server, at which point it is much less critical whether there is a single server or many.

In fact, servers for multiple time-dependent media such as audio and video will almost certainly employ a multi-threaded architecture internally, with all the associated problems of state and communication between threads. So the cost of multiple servers for synchronization can be reduced to the cost of the context switch between server processes and data sharing across server address spaces. With vastly improving processor speeds and increased hardware support for context switches, these differences are probably minor.

4. System model

Our audio architecture consists of five main components: the audio network or *protocol*, an audio server that implements the protocol, a client-side library (Alib), a user-level toolkit, and applications. The relationship between the components is shown in Figure 4-1. The structure of each of these components is briefly described in the following sections, with the remainder of the paper focusing on the protocol, the server implementation, and communication with the server.

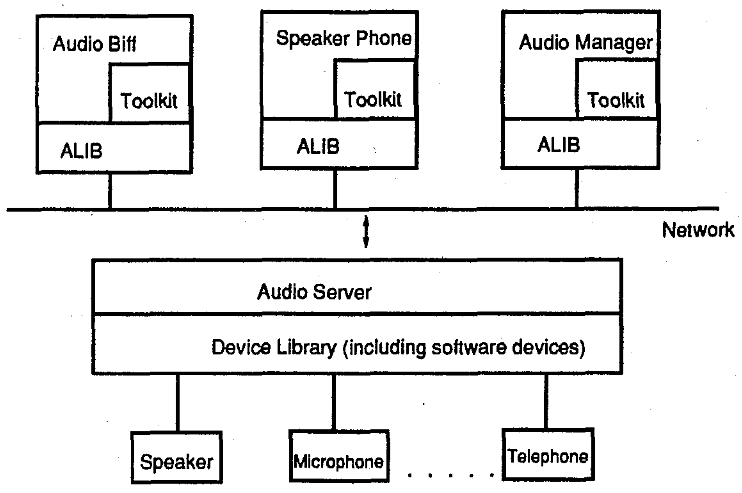

Figure 4-1: Audio System Model

4.1. The Audio Server and the Audio Protocol

For each workstation, there is a controlling server. The server implements the requests defined in the protocol and executes on the workstation where the audio hardware is located, providing low-level functions to access that hardware and coordination between applications. Clients and a server communicate over a reliable full duplex, 8-bit byte stream. A simple protocol is layered on top of this stream. The audio server can service multiple client connections simultaneously, and a client can have multiple connections to one or more audio servers. This protocol is a very precisely defined interface. The tight definition of the semantics of the protocol make it independent of the operating system, network transport technology and programming language. The "Alib" library provides clients with a procedural interface to send and parse the protocol messages.

Requests are asynchronous, so that an application can send requests without waiting for the completion of previous requests. Some requests do have return values (state queries, for instance), which the server handles by generating a reply which is then sent back to the application. The client-side library implementation can block on these requests or handle them asynchronously. Blocking on a request with a reply is tantamount to synchronizing with the server. Errors are also generated asynchronously, and applications must be prepared to process them at arbitrary times after the erroneous request.

The audio protocol describes several major pieces :

1. *connections* that provide the communication between server and client
2. *virtual devices* that specify a device-independent abstraction of the actual hardware. These are combined to build audio entities that can play, record or otherwise interact with the user.
3. *events* that notify the client of changes in state (for instance that a play command has completed).
4. *command queues* that control the use of the virtual devices and provide synchronization among devices and commands.
5. *sounds*, or audio data repositories, that can be played or recorded.

4.2. Alib and the Toolkit

Alib is simply a procedural interface to the audio protocol. It is a "veneer" over the protocol and is the lowest level interface that applications will expect to use. Applications should not use the workstation hardware interface directly or bypass the library.

We have built a toolkit that sits on top of Alib. The goals of the toolkit are to: hide or automate wiring of devices for greater portability, hide the location and format of sound data, hide and manage device queue management, and provide mechanisms for synchronizing audio with other media (for example, X graphics). Clients use the toolkit to construct audio user interfaces, such as an audio dialogue or touch tone-based menu. However, the toolkit is "policy free" in that it does not enforce a particular style but attempts to provide a mechanism for interaction. Further discussion of the toolkit is beyond the scope of this paper.

4.3. The Audio Manager Client

In a window system, a special application called a "window manager" mediates the competing demands for scarce resources such as screen space, input focus, and color selection. The window manager sets the policy regarding the input focus of the pointing device and keyboard; it keeps track of windows and sets the policy for moving and resizing them. Because the audio protocol allows multiple clients to access the audio hardware simultaneously, an application similar to a window manager is needed to enforce contention policy. We call this the *audio manager*.

5. Protocol Overview

5.1. Audio device abstraction

The protocol provides applications with a device-independent interface to the audio capabilities of the workstation. The device-independent objects, *virtual devices*, are the basic building blocks of "audio structures" in the protocol. Each virtual device can be described by a class name, a set of attributes, a set of controls, and a set of device ports. Device ports represent audio inputs and outputs, known respectively as *sink* ports and *source* ports. The ports are used to connect virtual devices together and define the audio data path between them.

Audio structures are constructed by organizing one or more virtual devices within containers called *logical audio devices* or *LOUDs*. LOUDs can then be constructed into a tree hierarchy. This hierarchy is used to logically group virtual devices into manageable substructures, such as a tape recorder that plays and records, or an answering machine.

Once a LOUD tree is built, two or more virtual devices can be combined to create more complex device abstractions by connecting *wires* between them. The wires specify the flow of data between the devices. The root of the LOUD tree is used to control and coordinate the audio streams to the LOUDs in the tree. A command queue is provided for each root LOUD. Figure 5-1 shows the hierarchy and wiring for an answering machine LOUD tree.

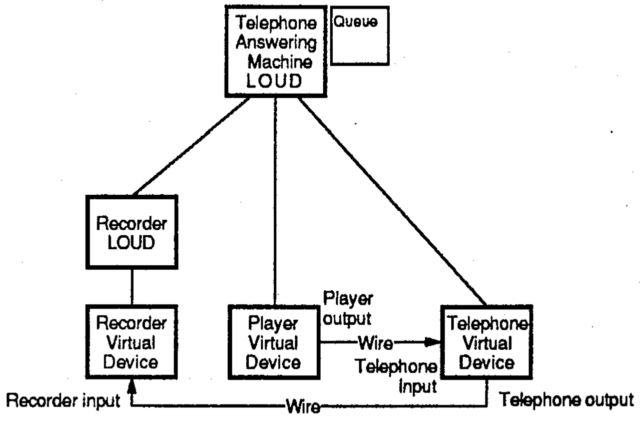

Figure 5-1: Answering Machine Loud

Virtual device classes. The devices supported by the protocol are divided into classes, which define generic audio functions that are supported by a set of device-independent *commands*. Below we have enumerated the classes supported by the protocol, and their commands. A device command is issued in either *queued* or *immediate* mode. Some device commands, such as `Play` or `Record`, must be synchronized with other commands, and can be issued only in queued mode. Other commands, such as `Stop` or `ChangeGain`, can be issued in either immediate or queued mode.[4] In immediate mode, a command takes effect instantaneously, and can stop processing of a queued command.

Inputs and *outputs* provide connections to external devices, such as speakers and microphones. The are used as wiring constructs to attach to the other classes. The base command is `ChangeGain`, which adjusts the volume.

Players have one or more output ports, typed according to a speech encoding format. They convert sound data to the output port type and then transmit the data out the port. The ports can be wired to an output device. The commands `Play`, `Stop`, `Pause`, and `Restart` control the transmission of the data on the ports.

[4]An application can issue a queued `ChangeGain`. An example would be a client that wants to play one sound after another but change the gain in between. In this case the client would issue a `Play`, a `ChangeGain`, and a `Play`, all in queued mode.

Recorders have one or more input ports, typed according to a speech encoding format. They store sound data received on the input ports. To digitize and store data from an external microphone, a recorder and an input of class microphone are wired together. The commands `Record`, `Stop`, `Pause`, and `Restart` control the flow of data from the ports.

Telephones are combined input and output devices with the commands `Dial`, `Answer`, `SendDTMF`, `Stop`, `Pause`, `Resume`.

Mixers take data on multiple inputs, combine the streams and then present the combined data on one or more output ports. The relative combination is determined by a percentage assigned to each input. The command `SetGain` sets the percentage to be mixed on a given input.

Speech synthesizers speak text strings. They have a single output for the synthesized audio. The commands `SetTextLanguage` and `SetValues` control interpretation of the text and acoustical characteristics of the vocal tract model used for synthesis. `SetExceptionList` allows applications to override the normal pronunciation of words, such as names or technical terms. `SpeakText` accepts commands to speak text strings.

Speech recognizers detect words spoken by a user. A recognizer has a single input, and produces recognition results as events. The commands `Train`, `SetVocabulary`, `AdjustContext`, and `SaveVocabulary` control which words a recognizer will detect, based on application and user.

Music Synthesizers process note-based audio. They accept commands, and produce audio data on their single output. The commands `SetState`, and `SetVoice` control music generation parameters. `Note` makes a sound.

A *Crossbar* is a switch to control routing of a number of inputs to a number of outputs. Each input can be connected to one or more of the outputs, as controlled by the command `SetState`.

A *Digital Signal Processor* is a set of software to manipulate one or more audio data streams. It may have several inputs and outputs. Commands have not yet been specified. .

Device Attributes Both virtual and physical devices have *attributes* which describe specific features of the device. Attributes are used for virtual devices to constrain their mapping to physical devices, or to get information about the physical devices with which they are associated. Attributes of a physical device describe its actual capabilities.

To facilitate device-independence, an application specifies the desired virtual device by a list of attributes. The attributes can specify a device either tightly or loosely. For instance, a loose specification might be "give me a speaker". A more tightly specified list of attributes might be "give me the left speaker". The audio protocol maps the virtual devices created by the application onto the specific instances of those functional devices. When creating a virtual device, the application need only specify the class and other attributes of the device, rather than the specific hardware required to accomplish the operation.

The attributes for a specific device are dependent on the actual hardware. For example, the attributes of a recorder device include

1. sound encoding formats supported
2. whether the recorder supports automatic gain control (AGC) during recording
3. whether the recorder can compress the recorded audio by removing pauses
4. whether the recorder supports pause detection to terminate recording.

Attributes of a telephone device are largely constrained by the telephone equipment (digital or analog) and network (public or private) capabilities. Every telephone will have one or more numbers and area codes associated with it. Telephones may have multiple lines. Telephones may report information about incoming calls, such as the identity of the caller and whether the call was forwarded from another number.

An answering machine might wish to use calling party information to choose an outgoing message, or to label a message it takes. The answering machine client needs to query the device attributes of the telephone in order to determine whether this information will be available as part of the incoming call notification event.

What does the hardware do, really? Providing a device-independent interface is not specific enough for some applications and some hardware. Some devices are connected via physical wires that cannot be broken. Users will want global control over some devices, such as volume on a speaker, rather than the local control that virtual devices provide. A special LOUD tree, called the *device LOUD*, encapsulates all of the available functions in every device controlled by the server. The device LOUD tree contains a LOUD for every physical device, and if two devices are hard-wired, they are wired in the device LOUD. Each LOUD in the device LOUD is given a unique id that can be used by an application to monitor the device. The example in Section 5.9 monitors the telephone using the device LOUD.

Devices are controlled by applications through commands or through the manipulation of *device controls.* The commands provide a portable interface to abstract audio devices. Device controls allow for more complete access to devices at the cost of portability. Device controls have a format similar to X properties. Device controls should be necessary only for extensions, such as new subclasses of devices, or to take advantage of a particular implementation of a device at the cost of portability.

5.2. Wires

Wires establish the flow of data between virtual devices. They are used to construct complex devices by specifying connections between source and sink ports. A wire connects a source port of a virtual device to a sink port of another virtual device. This is how the internal connections of a complex device are established.

Wires have type information so that an application can query the type of the path, be it analog or a digital sample format. It is possible to query a virtual device for its wires, or a wire for its virtual devices and their corresponding port indexes. If it is necessary to constrain the nature of the path between two virtual devices, the desired wire type can be specified when the wire is created. The server checks that data on the wire matches the wire type.

In the device LOUD, the existence of a wire between two virtual devices indicates that there is a permanent connection between their respective devices. Unfortunately, not all hardware is as general as might be desired. If the capabilities and constraints specified for a virtual device require the devices to have permanently wired connections, special wiring rules apply. If an application attempts to attach a wire between a virtual device and any other virtual device, the capabilities and constraints for the connected virtual devices must match those of the devices that are physically connected, or an error will result. An example of such a mismatch might be an outboard speaker-phone that has hard-wired connections between a telephone line, a microphone, and a speaker. Attempts to wire a LOUD that requires use of one part of the speaker-phone with a device that cannot be implemented by another piece of the speaker phone is not allowed and will generate an error.

5.3. Mapping: associating a virtual device with an actual device

There is not necessarily a one-to-one correspondence between an actual piece of hardware and a virtual device. If a device can be used by more than one virtual device at the same time, the functional device will appear as multiple active virtual devices. For example, a speaker or output device, through which the sounds from multiple applications are simultaneously mixed, would be represented by multiple active virtual devices.

The server does not bind a virtual device to a physical device until the LOUD has been *mapped.* At this point, the server examines the attributes given when the LOUD was created to find a matching device. Most applications do not care which device they use, only that they can get the services they require. If an application must use a particular device, the id of the device in the device LOUD can be passed as an attribute to force the binding.

It is possible to augment the virtual device's attributes to tighten the device constraints. Such augmentation capability is useful when, for example, an application does not care which speaker it uses, but does not want to change speakers once the output has commenced. If an application requests a device by its class, the actual device used may change between activations. If it is desirable to use the same device for each request, an application can create a virtual device in a LOUD, and then map the LOUD. At this point, a `QueryVirtualDeviceAttributes` request will generate a list of device attributes that contains, among other things, the device ID selected by the server. This device ID can then be specified in an `AugmentVirtualDevice` request, so that it becomes an application-specified constraint.

5.4. Activation: who gets the device

LOUD access to shared resources is controlled by an *active stack*, which is the fundamental scheduling mechanism in the server. When a LOUD is mapped, it is put on the active stack. Unmapping a LOUD removes it from the active stack. The LOUD at the top of the active stack controls the functional devices represented by all of its virtual devices. Lower priority LOUDs can be put on the bottom of the stack to yield to higher priority LOUDs.

The server activates as many LOUDs as it can at one time. It does this by starting at the top of the active stack and activating all LOUDs that do not requires a resource that is being used exclusively by another active LOUD.

The activation and deactivation of a LOUD occurs dynamically. The state of the functional devices controlled by the LOUD are stored in its virtual devices, so that the server can restore the LOUD's devices to their state prior to the moment the LOUD was deactivated. Applications can request that a LOUD be activated or deactivated, and receive notification of these transitions. A mapped LOUD can be activated by the server or an audio manager at any time, so an application must treat a mapped LOUD as if it were active.

5.5. Synchronizing audio streams: command queues

Each root LOUD has a *command queue* to synchronize the actions of the virtual devices contained in the LOUD tree. Queues allow for the sequential processing of commands within the server, without requiring application notification and the associated round-trip communication. For example, an application may play a prompt and then record the user's response. In this situation, an application could first issue a `Play` to a queue and then issue a `Record`. The queue would process the `Play` and, on completion, start the record operation. Another case might be an application that wants to play several sounds back-to-back. The application could issue three `Play` commands, one right after another, and the server would play them in order, one after another, with the minimum possible space in between. For a set of digital sounds, there should be zero delay between them.

Queues have four possible states: *started, stopped, client-paused,* and *server-paused.* Time in a queue is relative to the activity of its LOUD tree. When a queue is paused, command queue relative time is suspended for that queue. If a LOUD is made inactive while processing a command, the server pauses the queue. Upon activation of a LOUD, a queue in the server-paused state is automatically resumed. Pausing a queue pauses the virtual device on which the current command is operating. All other devices in the LOUD are not affected.

The application can explicitly pause a queue by placing it in a client-paused state. The client-paused state allows a LOUD's queue to be paused, preempted, and re-activated without losing track of the queue state. The pausing and resuming of a queue are also propagated to all virtual devices affected by the current command. If the application issues a request to pause a queue in which the current command is operating on a device that cannot be paused, the queue is stopped.

There are four queue commands that allow device synchronization, but do nothing to devices. These commands are `CoBegin`, `CoEnd`, `Delay`, and `DelayEnd`. These queue commands are not meant to provide a programming language but to facilitate synchronization. There are no conditionals or branches and the queue is not an interpretor.

The CoBegin command causes all of the commands up to the bounding CoEnd command to be started simultaneously. The command after the CoEnd is not started until all commands within the CoBegin/CoEnd bracket are completed. These commands exist for the synchronization of complex audio configurations. A CoBegin command is useful when an application wants two operations to start at the same time. If, for example, an application is playing two sounds through a mixer, a CoBegin is necessary for the sounds to be started at the same time. The following example starts playing A and B at the same time. When both A and B are finished, sound C is started.

```
cobegin
    play A on device 1
    play B on device 2
coend
play C on device 1
```

The Delay command waits some interval time before processing. Commands contained within a delayed segment are processed sequentially, unless a CoBegin command is encountered before the DelayEnd command. The following example plays sound A, waits 5 seconds and then starts playing B. When B is finished, sound A is stopped.

```
cobegin
    play A on device 1
    delay 5 seconds
        play B on device 2
        stop device 1
    delayend
coend
```

5.6. Audio sound abstraction

Once a LOUD tree is created and the virtual devices are wired together, applications will want to pass audio data between the devices via wires. A *sound* is a typed object that represents digitized audio data. Its type is represented by the tuple (encoding, samplesize, samplerate). While the contents of a sound must be on the server side to be manipulated, the data can be supplied by the application or it may be supplied and controlled by the server.

The server provides a collection of sounds in its data space. Applications reference these sounds by name. The sounds are grouped into libraries or catalogues. Most sound data will be stored in files. However, some sound data will not be available to the server directly, but rather through an external device controlled by the server. Consider, for example, a CD that plays directly to its own speaker. In this situation, the sound data is supplied by the CD, rather than the application. Many CDs do not provide a digital data path to the computer so applications cannot read the audio data. Since the server controls the connection between the CD and the speaker, it must also control the sound data.

In addition, sound data can be supplied in real-time by an application, such as a networked-based audio process. The protocol provides a mechanism to supply or retrieve data from an active device. The application supplies the data to the server; it can then be used in the same way as server-side data.

5.7. Events

An *event* is data generated asynchronously by the audio server as a result of some device activity or as a side-effect of a protocol request. Events are the primary mechanism for synchronizing audio with other workstation services and media. The server generally sends an event to an application only if the application specifically asked to be informed of that event type.

There are 3 major event categories: *command queue*, *device* and *synchronization*. When a device or a command queue changes state, an event can be generated. For the queue, these are such things as QueueStarted, QueueStopped, and CommandDone. For devices, events are class specific. For the telephone class, they are "a dial request has been issued", "the telephone has been answered", "the phone is ringing". For the recorder class, they are "start" and "stop".

The synchronization events are used to coordinate the audio stream with other media or services. For example, consider an application displaying a set of images while playing a stored digital sound track. The images are displayed using the window system, and the audio track is played using the audio server. This application wants to display the images at some fixed rate. The application monitors the audio server synchronization events on the sound track, and uses them to time the update of the display.

5.8. Audio Manager support

The audio protocol provides several mechanisms for audio managers. It also specifies sensible defaults in the absence of an audio manager. These mechanisms directly parallel those provided by X. The mechanisms are: ambient domains, device exclusion, properties and redirection of mapping requests.

An *ambient domain* indicates a relationship between devices and the acoustic environment. A server supports at least one domain. For example, two speakers and one microphone on a user's desk are one ambient domain, called the *desktop domain*; sound from the speaker will be audible by the microphone. A telephone line is another ambient domain, as it does not interfere with the desktop domain. A speaker-phone is in both domains.

Ambient domains allow a user to activate a microphone, a device of class input, and exclude all devices of class output that might interfere with the same ambient environment. To accomplish this, in addition to the ambient domain attribute, devices of class input or output can request the attributes of *exclusive input* and *exclusive output*. Requesting a device with the exclusive input attribute preempts all other devices of class input in the same ambient domain. The exclusive output attribute performs similarly, but affects only devices of class output.

A *property* is a (name, value, type) triple. Properties can define any arbitrary information and can be associated with any LOUD or sound data. Properties can be used to communicate information between applications. In the case of an audio manager, they can be used to indicate application or user preferences. For instance, applications might attach a property named DOMAIN to the root LOUD. The value of this property would be the user's ambient domain preference for that application.

Another way to enforce policy is through *redirection control*. When an application attempts to map or restack a LOUD, the request may be redirected to a specified client rather than the operation actually being performed. In this way, the audio manager can override the application.

5.9. Example: an answering machine

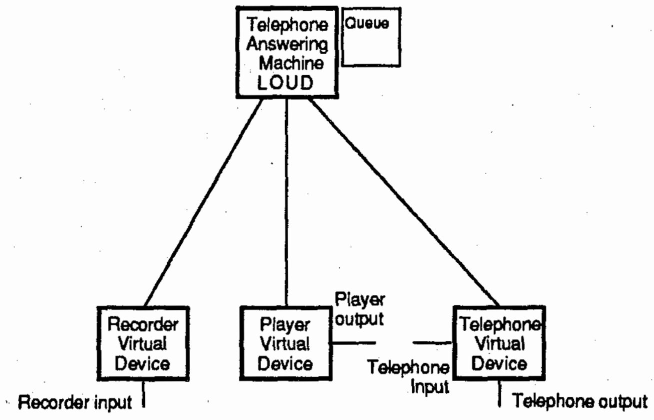

Figure 5-2: Answering Machine LOUD tree

To give a feel for how an application would use the protocol, this section contains an example of building and using an answering machine. An answering machine plays an outgoing or greeting message after answering a call, records an incoming message and then hangs up. The protocol entities needed to build the answering machine are a telephone, a player, a recorder and two sounds. The LOUD that the application would construct is shown in Figure 5-2.

When creating the virtual devices, the answering machine application specifies only the class of the device and the type of data that will be used. In this example, the greeting message is stored in an 8-bit μ-law encoding. Therefore, the attribute specification for the player is 8-bit μ-law. The application need not give any more information to the audio server. In this example, the incoming message is also an 8-bit μ-law sound.

The telephone will probably be an actual hardware device connected to the workstation. The player and recorder will be software devices, or algorithms. The player is responsible for reading cached sound data (probably from a file) and presenting that data on its output source. The recorder does the reverse; it takes data from its input sink and stores it as a sound (again, probably a file). At creation time, the three devices are unmapped and not activated. Any commands sent to them will be ignored until they are activated.

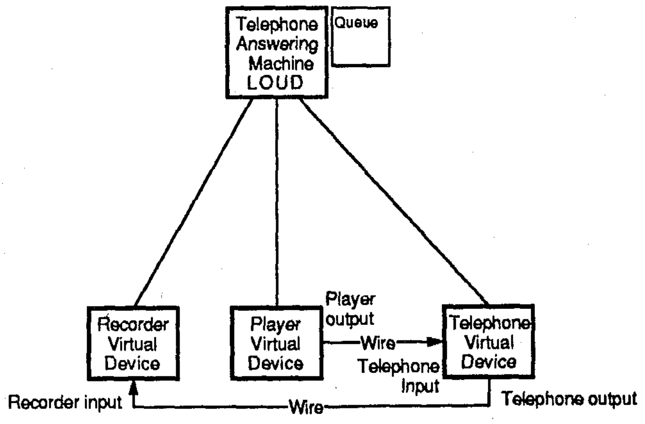

Figure 5-3: Answering machine: wired

The next step is to wire the devices to indicate the data paths. The results of wiring are shown in Figure 5-3. The output sink of the player is connected to the input of the telephone. This allows the greeting to be played to the caller. The output of the telephone is connected to the recorder's input source. This wire allows the caller's message to be recorded. As each wire is created and connected, the server checks the data types available at the two ends of the wire. If one end can only produce 8-bit μ-law and the other can only take ADPCM,[5] a protocol error will be generated.

Now that the LOUD is wired, it can be mapped and activated. Once mapped, the LOUD will accept and execute queue commands. To map and activate the LOUD, the audio server assigns virtual devices to actual devices, if possible. The assignment is qualified with an "if possible" because a device that is needed might be in exclusive use by another application. Or the LOUD may contain an impossible configuration, such as two virtual devices that specify the same actual device. In the later case, the map request will generate an error. If successful, the application gets an `ActivateNotify` event.

At last the LOUD is ready to accept commands. Commands are given to the LOUD, rather than the virtual devices (or, if there were any, sub-LOUDS) so that operations can be coordinated by the command queue. Playing the greeting before the phone is off-hook will result in a confused caller. The command queue for the answering machine is shown in Figure 5-4. First the phone is answered. When that command is completed, the greeting is played, followed by playing a "beep" sound. Once the beep is completed, the message recording begins.

Since most of the time the phone is not ringing, the LOUD can stay unmapped. The queue commands can be preloaded before the phone is answered. When the phone rings, the application would raise the LOUD to the top of the active stack, map it and start the queue.[6] The `Record` command has a termination

[5]Adaptive Delta Pulse Code Modulation, a compression algorithm, can reduce audio data rates by about one half.

[6]Because the answering machine LOUD is unmapped, the application cannot tell, from the LOUD, if the telephone rings. Therefore it monitors the device LOUD telephone.

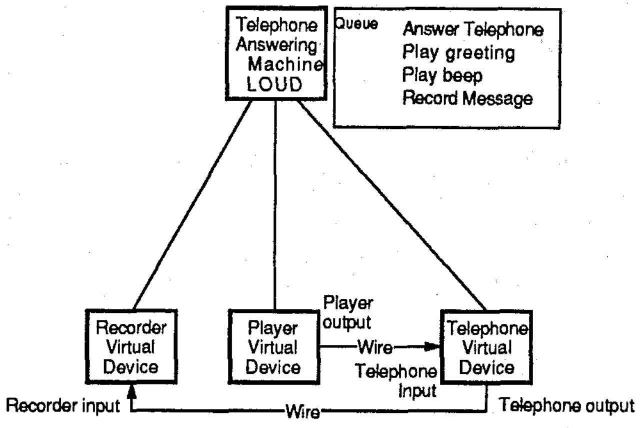

Figure 5-4: Answering Machine with command queue

condition, which can be either after a pause or when the caller hangs up.

Of course there are exceptions that must be handled. For instance, what happens when the caller hangs up? The caller may hang up before the beep is played. The LOUD can ask for `TelephoneNotify` events from the server. The application will get a `CallProgress` event that says that the phone is now hung up, and can then stop the queue and get ready for the next call.

6. A Prototype Implementation

A prototype audio server has been written. At the moment it supports playing, recording, and mixing, with the underlying LOUD creation, wiring, and mapping as described in the previous section. Multiple clients can play to the speaker simultaneously. The prototype server is written in C++ and is multi-threaded; it runs on a DECstation 5000, a 20 MIPS workstation, under UNIX. The audio hardware required is a simple CODEC with memory-mapped buffers.

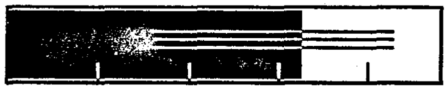

Figure 6-1: The SoundViewer widget supports audio playback and recording using several display modes

To test synchronization with other media, we have implemented a graphical sound viewer widget using the X toolkit and Alib. The widget displays a continually updated bar graph as a sound is played. Audio server synchronization events are used to control the graphics; the bar chart is updated in response to the these events. Figure 6-1 shows such a bar chart. The darkened area is the part of the sound that has already been played. The tick marks give an indication of the sound length. The dashes in the middle denote a part of the sound that has been selected, to be pasted into another application.

A working audio server and a few test applications have increased our confidence in the protocol and demonstrated the feasibility of a software architecture to support the server constructs described in the previous section. Although it would be premature to quote hard performance statistics, day-to-day use of the server indicates that performance will be well within our goals. We would like to be able to start playback of a sound, using an existing server connection, in less than several hundred milliseconds and support continuous playback without gaps, using well under 10% of the CPU.

But such numbers do not tell much about how well the audio medium will integrate with existing and future workstation application environments. With increased processor speeds and peripheral bandwidths, higher quality audio will become increasingly practical, but the limiting factor will be how well audio-based applications will perform in the context of many user applications running concurrently on top of a window system. Users are unlikely to tolerate system performance degradation for the sake of audio, as they have yet to be convinced that audio has real utility. But until audio becomes universal, software vendors will have little motivation to write applications.

In order to meet what will be demanding requirements on generic multi-application support of audio, several aspects of our server have received particular attention. One is the use of a multi-threaded approach for modularity of design and management of multiple simultaneous audio data streams. The other is a set of design considerations to support the real-time nature of audio; some of these are exposed in the protocol while others are embedded in the server. These will be described in the remainder of this section.

6.1. Threads

We used threads in our implementation to allow concurrent use of the various audio devices. In addition, we wanted to be able to start and stop devices out-of-band. This section describes the various threads and what they do.

The *connection manager* detects and manages incoming connections. It is a daemon at a well-known port that detects incoming client connection requests and creates new connections for the clients. The connection manager runs as a thread. It is also responsible for initiating shut-down for a connection in case of errors. The connection manager keeps a container object for each client connection. The container objects hold everything that is related to a particular client connection.

The *device layer* contains devices that talk directly to the physical hardware. Each device runs as a separate thread. Devices provide command methods for others to call. Commands are handled synchronously. Events or status are reported by each device synchronously to its virtual devices.

The *virtual device layer* implements the virtual devices, each of which runs as a thread. Each virtual device, when active, is associated with an physical device. The different classes of virtual devices are subclasses of a common virtual device object class. Virtual devices process requests from the client input handler and associated command queue. Multiple virtual devices share a single device when possible; the server creates mixer-like virtual devices when it can, transparent to applications.

Data source and *data sink* implement the server version of wire, manipulating data inside the server. They create an efficient data path by by-passing in-between devices and connecting the source and sink objects directly. Each source and each sink runs as a separate thread.

6.2. Copying and Caching Data

Implementing an audio server on a non-real time, general-purpose UNIX system poses interesting challenges. The biggest and most obvious problem is the real-time nature of audio. Although medium quality audio data rates are not excessive, substaining the rate may be difficult. Sub-second delays in the middle of speech are annoying and damage the intelligibility of the speech. Another challenge is supporting seamless transitions between commands in the queue, so that playback appears continuous.

Our design alleviates some of the real-time issues and allows the client to deal with those cases where the system falls short. Previous experience with X has shown that we can often achieve real-time appearance with sufficient buffering and an intelligently designed protocol.[7] Telephone or voice quality audio requires a 8000 bytes per second data stream. Higher quality audio can involve data rates up to 175,000 bytes per second. The lower data rates are usually adequate for telephone quality speech and within the ability of current technology, so this is our focus. As coding standards emerge, higher bandwidth voice coding with only modest increases in data rate will become more common and will be supported by the server; these data rates are already supported by the protocol.

We have addressed the real-time constraints in several ways. The most common operation is the playback of recorded audio data. If the data is cached by the server, for instance in the file system, the data transfer is local, the number of copies small, and the performance should be acceptable. If the application

[7]It was originally believed by many that rubber-banding and menu tracking require extensive system changes. In the early days of X it was shown that this is not the case.

wants to supply real-time data to the server, the constraints are harder to satisfy. When an application is providing data in real-time there is the possibility that the application or the application's source, maybe a network connection, will not have the data when it is needed. To deal with this, the protocol provides client-side reading and writing of data. This type of interface allows an application to implement its own policy and has been proven effective in the past [1]. This approach also allows the application to make trade-offs with respect to latency and memory.

One requirement of the protocol is to support seamless transitions between commands in a queue. There are two interesting cases to consider: playing back to back sounds, and a play followed by a record. These can be illustrated by considering the answering machine example in Section 5.9. When the phone rings, the application starts the queue and the server executes the queued commands. Once the outgoing message play is started, arrangements must be made so that the beep sound starts instantaneously after the message.

Once the queue starts, the server answers the phone and starts the play. When the first play command is about to finish, the player device informs the queue of the time at which the last sample will be played. The queue can then issue the next play command specifying that the play should start when the first command is scheduled to terminate.[8] Pre-issuing commands allows plays to occur without a single dropped or inserted sample. Recording back-to-back with a play is accomplished in the same manner.

7. Conclusion

This paper has discussed a server to control audio and telephony hardware. We described the requirements of an audio server, a protocol to communicate with the server, and a set of constructs which an application must manipulate to use the server. We described an implementation of a prototype server, with emphasis on its multi-threaded architecture and design goals for real time performance.

This work is in its early stages, but we are encouraged by our progress so far, and are confident of the utility of the server model to support time dependent media.

8. Acknowledgements

Our protocol owes many ideas to pioneering work done on the VOX project at Olivetti Research Labs. We would also like to thank Chris Kent, Tom Levergood, Larry Stewart and Jim Gettys for their comments and review of our protocol. Mark Ackerman of M.I.T. wrote the original graphical SoundViewer widget software. Chris Kent and Gordon Furbush provided valuable comments on drafts of this paper.

References

1. P. Zellweger, D. Terry and D. Swinehart. "An overview of the Etherphone system and its applications." *Proceedings of the 2nd IEEE Conference on Computer Workstations* (March 1988).

2. C. Schmandt and M.A. McKenna. "An audio and telephone server for multi-media workstations." *IEEE Proceedings of the 2nd Workstations Conference* (March 1988), 150-159.

3. Robert W. Scheifler, James Gettys, and Ron Newman. *X Window System.* Digital Press, Bedford, MA, 1988.

4. Barry Arons, Carl Binding, Keith Lantz, and Chris Schmandt. "A Voice and Audio Server for Multimedia Workstations." *Proceedings of Speech Tech '89* (May 1989).

[8]Note the queue does not calculate the ending times itself, because the server's CPU may not use the same time base as the CODEC's, and clock skew is a problem.

Susan Angebranndt received a B.S. in Mathematics from Carnegie Mellon University in 1980. She is currently a Consulting Engineer at Digital Equipment Corp. and technical project leader for the Multimedia software development group.

Richard L. Hyde received a B.S. in Computer Science from Brigham Young University in 1982 and an M.S. in Computer Science from Brigham Young University in 1983. In 1984 he join Digital's UNIX engineering group. In 1987 he moved to Digital's Western Software Labs to help with the X11 development effort. He currently is the technical project leader for the audio server work in the Multimedia software development group.

Daphne Huetu Luong received her degree in Electrical Engineering and Computer Science from the University of California, Berkeley in 1987. She is a software engineer at Digital Equipment Corporation's Western Software Laboratory in Palo Alto, California. She works in the Multimedia software development group. She previously worked at Xerox, doing network communication software development.

Chris Schmandt received his B.S. in Computer Science from MIT and an M.S. in computer graphics from M.I.T.'s Architecture Machine Group. He is currently a Principal Research Scientist and director of the Speech Research Group of the Media Laboratory at M.I.T.

Nagendra Siravara recieved an M.E. in Electrical Engineering from Indian Institute of Science, Bangalore, India. Since 1990, he has worked in the Multimedia software development group at Digital Equipment Corp. He worked previously on software for voice response applications, speech and image processing and error correction coding.