The Design of Audio Servers and Toolkits for Supporting Speech in the User Interface

Barry Arons
Media Laboratory
Massachusetts Institute of Technology

1 Abstract

An audio server is a software platform that oversees the sharing of audio resources in a distributed computing environment, and simplifies the task of integrating speech into the user interface. An audio toolkit layered on top of an audio server further simplifies the creation of voice interfaces, by providing a higher level of abstraction to the application programmer.

This article explores the design of audio servers, toolkits built on audio servers, and some applications that use these facilities to integrate voice into our everyday computing environment [14, 18]. It discusses the implementation of an audio server, the impact of a software-based digital audio system on the design of a server, proposes an architecture for a toolkit, and a minimal set of speech interface components. The interaction of the toolkit with the audio server and the window server is investigated by drawing parallels with window systems and their graphical toolkits.

2 Why Audio Servers?

Most current interactive voice applications assume a particular hardware configuration and control that hardware directly (and few such applications run on general-purpose workstations). This approach is easiest from the application writer's point of view; the program communicates with the hardware in an efficient device-dependent manner, and the exact configuration of the hardware is predefined. When incompatible hardware from different vendors must be supported, or when more than one application requires access to a voice I/O device at a given time, this approach breaks down. These limitations are acceptable for stand-alone applications, but impose severe restrictions if the audio resources are to be shared by several applications. For example, all the voice-based applications running concurrently within different windows on a multi-tasking operating system might need to share a single microphone or speech recognizer.

One attractive solution is a client-server model. This paradigm logically separates the application from the hardware; client processes make requests to a single server process that controls the hardware resources for all applications (and thus is somewhat analogous to an operating system). This separation is provided by a standard communication protocol between the clients and server. The protocol defines the stream of bytes that represent the request, reply, and event messages that are transported over a network. The protocol is a well defined reference point between server and client. Client applications and servers can be developed independently using different programming languages, operating systems, and underlying hardware; the server is not a subroutine library that is linked in by the application. A variety of servers that obey the protocol specification can be implemented—one vendor, for example, can built a server on simple analog hardware, while another can provide a highly optimized server that does all audio processing in software.

The intended use of an audio server is on a per-workstation basis—to control the digitizing, telephony, mixing, text-to-speech synthesis, and speech recognition components envisioned in next generation personal computers and workstations. This is analogous to the X Window System, where a window server runs on each workstation and manages the local user interface I/O devices—the mouse, display, and keyboard.

Client applications can also communicate with audio servers located anywhere within a network to control remote audio hardware resources. This distributed configuration is useful where there are limited resources that must be shared, or where there is specialized hardware cannot be plugged into a particular workstation bus.

3 Prototype Audio Servers

A simple audio server that emphasizes the support of conversational voice applications has been evolving at the M.I.T. Media Lab since 1984 [17]. This server originated with the desire to use a general purpose workstation to control audio equipment built into a customized minicomputer. The approach taken is simplistic; the "network" is a serial line (RS-232) that carries character-oriented commands between the workstation and the server. Low-level functions supported by the server included digital recording, telephone control, and speech recognition. This architecture is advantageous from an interface design perspective; it allows a clean separation of audio hardware and functionality from the remainder of the user interface. Support for this device-independent control protocol has been superseded by PC-BUS machines populated with voice/telephone boards from several vendors.

This serial server is limiting in several respects: (1) an entire PC server is dedicated per workstation, (2) only a single application can access the server at a time, and (3) the serial server only functions in a synchronous manner—when the client application makes a request to the server, the client must wait for a response, and no other processing can take place in the client.

The Etherphone system [20, 21] developed at Xerox PARC uses a centralized voice storage server. Etherphone successfully integrates audio into the Cedar programming environment, making voice accessible from the Tioga editor, and hence a range of other applications. This system is robust, and a strong emphasis was placed on issues such as reliability and security.

4 A Workstation-Based Audio Server

The VOX Audio Server [2] was designed specifically to address the types of applications envisioned in a “desktop audio” workstation environment, and to remove the limitations cited in previous sections. Many of the core ideas for the audio server are inherited from the M.I.T. serial server and the X Window System[8]. VOX was developed to demonstrate the utility of an audio server, for building research prototypes, and to explore the usefulness of desktop audio applications.

In the same way that graphical interfaces can be built on top of the X Window System, it is desirable to build audio user interfaces on top of an audio server platform rather than directly on the underlying hardware. The server supports the sharing of audio resources between multiple clients, dynamic routing, and a queuing mechanism that helps to provide real-time response for audio events within a Unix operating environment.

4.1 Server Design Goals

The server architecture supports a wide range of potential interactive voice applications by emphasizing:

- Resource management: The architecture supports the transparent sharing of audio resources by multiple applications. By default, resources are shared with a preemptive scheme; however, applications can gain exclusive access to audio resources for a limited time. For example, control of the telephone cannot be taken from an application if a call is already in progress.
- Real-time behavior: The architecture addresses the issue of real-time behavior in the handling of audio events by supporting a queuing mechanism that minimizes the overhead of processing audio requests. Note that this service is provided in a non-real-time Unix environment.
- Routing: The architecture enables applications to create dynamic routings between audio components. For example, in conversational applications it is often desirable to switch rapidly from a speech recognition to a recording configuration.
- Network transparency: The architecture permits clients and servers to communicate asynchronously within a distributed computing environment.
- Device independence: The architecture shields applications from the idiosyncrasies of the underlying audio hardware.
- Extensibility: The architecture allows for the unforeseen uses of voice in the interface, and for the integration of new types of speech, audio, or telephony devices into the server.

4.2 Server Architecture

The software architecture needed to achieve these goals is similar to that of many contemporary window systems (Figure 1). Multiple client applications can connect to a single audio server process that provides voice computing and voice I/O services.

Figure 1: Audio and window server architecture.

Dynamic routing between audio devices is analogous to assembling audio circuits out of electronic components—low-level components are grouped into higher-level assemblies of increased functionality. The lowest level building block is called a LAUD (pronounced "loud") for Logical AUdio Device (perhaps a "virtual" device would have been a better name than "logical"). Examples of LAUDs include abstractions for playback, recording, or mixing. LAUDs can be combined into a composite LAUD, called a CLAUD. LAUDs have audio ports that can be connected during the compositing process.

Input and output to CLAUDs is asynchronously event-driven—the client submits output requests to, and may receive input events from, a CLAUD. For example, the client may submit dialing requests to a telephone, or receive tokens from a speech recognizer. When enqueuing output requests, the server attempts to prepare the request as much as possible before the activity is to take place. This may involve actions such as prefetching a recorded sound, or establishing the state of a speech recognizer. These preparation activities can be executed before servicing the primary request, thereby reducing the execution latency inherent in a client-server model.

4.3 Hardware Testbed

A multi-station testbed was built on 80386-based PCs running a variant of the Unix operating system. The workstations were equipped with a mixer, microphones, loudspeakers, a full duplex telephone echo cancellation unit, and a codec board that provided a telephone interface.

Because it is difficult to predefine the configuration of audio devices for all possible client applications in a research environment, devices on each workstation were interconnected with a crossbar switch. The crossbar provides a straightforward software controlled interconnection mechanism between all components, permits inter-workstation conferencing, and makes it easy to connect new audio components, experimental speech processors, or other devices to the workstation audio environment.

An analog crossbar switch was chosen as a central component of the research testbed (Figure 2). A full 16×16 non-blocking switch is perhaps excessive to require on every desktop, but this design allows maximum flexibility in creating complex client applications. This physical crossbar switch can be eliminated in a software-based all-digital implementation of an audio server.

Figure 2: Audio workstation configuration.

5 Server Applications

Application areas that the audio server supports include: telephone management [4, 16], voice annotation, real-time teleconferencing, conversational answering machines, and, more generally, voice-based tools to support collaborative work. The server mechanism allows a range of such applications to work together synergistically.

5.1 Telephony

To implement a simple auto-dialing mechanism, a client application accesses and manipulates a single LAUD. This telephone interface LAUD is the low-level device abstraction used by the audio server that provides dialing, call progress information, ring detection, and other telephony functions to the application.

Implementing basic answering machine capabilities (e.g., *"Please leave your message... BEEP!"*) with an audio server involves the playing and recording of sounds in addition to a telephone interface. The queuing mechanism of the audio server is required for this application to ensure that the recording is started immediately after the greeting is played. This provides a smooth interaction with the caller, and prevents the beginning of the recording from inadvertently getting clipped.

Figure 3: Answering Machine CLAUD.

Figure 3 represents a telephone answering machine built within the server. Simple playback, recording, and telephone control LAUDs are grouped into a single telephone answering CLAUD. Implementing such a mechanism with the audio server consists of the following basic steps:

1. Create a new composite CLAUD
2. Initialize play, record, and telephone LAUDs
3. Insert the LAUDs into the CLAUD
4. Prepare play and record events
5. Map the composite LAUD onto physical devices
6. Wait for an incoming ring event
7. Enqueue the play and record events
8. Wait for the recording to finish
9. Hang up

All control functions are parameterized so, for example, it is possible for the application to set the termination conditions for recording, select the sound compression algorithm, or select the number of rings before the call is answered. A client application program that implements such a server-based answering machine using the C programming language consists of about two pages of source code.

More advanced answering machines can be created by adding sophisticated interactions to the user interface. The ability to detect touch-tones allows a user to interact with the software-based answering machine from a standard telephone set. Adding text-to-speech synthesis, speech recognition, and silence detection allows for conversational interactions, as explored in the *Phone Slave* project [10, 11].

5.2 Workstation Control

Speech can be used to control the workstation, augmenting (not replacing) the keyboard and mouse. The *Conversational Desktop* [13, 12, 15] provided a speech-based interface to functions such as scheduling, audio reminders, teleconferencing, and voice mail. The hardware configuration for the system was complex and dynamic—audio was routed to and from recognizers, recorders, synthesizers, and customized audio processing equipment. When playing a sound, for example, audio input to the speech recognizer was switched off to prevent spurious insertion errors. This style of run-time configurability of the audio hardware (as handled by the crossbar switch), is essential for many applications, particularly those supporting multiperson collaboration.

Another interesting application for an audio server is to provide a speech interface to a window system. Most current generation Unix applications are "mouse illiterate"—the users hands only leave the keyboard to move between applications. Xspeak [9, 1] provides a voice interface to controlling window management within the X Window System; such an application could easily be implemented on top of an audio server platform.

6 Software-Based All-Digital Audio Servers

Window servers control a limited set of hardware: the mouse, display, and keyboard—a hardware-based audio server (section 4.3) has a much richer set of primitive hardware components: speech recognizers, synthesizers, digitizers, mixers, filters, etc. This implies that the protocol interface to the server must support all these devices, and a toolkit must likewise reflect this underlying complexity.

In the near future, a software-based all-digital audio environment will exist on general purpose workstations. Rather than supporting a wide range of hardware components, the server can be structured so that it handles only low-level input and output—all processing beyond this analog-to-digital or digital-to-analog conversion is done by software modules in the server. This design can significantly reduce the cost of the audio server hardware, and can provide long-term flexibility over a hardware-based analog approach.

Software modules for text-to-speech synthesis, speech recognition, or time-scale modification of recorded sounds, are loaded on demand into server. This technique also may act to unify the protocol interface to the audio processing components (LAUDs) and reduce the complexity of the software interface. Crossbar-style switching can easily be implemented by such a software-based system by simply routing, or copying, streams of data between the various software modules in the server.

For abstraction and information-hiding reasons, as well as for efficiency, it is desirable to keep all digitized audio data entirely within the server. Some applications, however, will need access to the data for specialized audio processing, or to display the data in graphical form. One attractive alternative to having all the data cross the client-server interface is to have an interpreter in the server [19]. This permits a client application to download programs to the server that manipulate the data directly in the server.

7 Software Toolkits for Audio

While an audio server may provide great flexibility, it can increase the complexity of application programs. A software layer built on top of an audio server can provide high-level abstractions to hide the intricacies of the server, and thus simplify the applications. The application writer is thus freed of low-level interrupt and device handling, and can concentrate on writing interactive voice-based applications. A "toolkit" approach to designing such an interface is explored, using the architecture of existing graphical toolkits as a point of departure.

7.1 Graphical Toolkits

A graphical toolkit is a high-level software interface built on top of a window server. An analogous architecture is proposed for a toolkit built on an audio server.

A graphical toolkit hides the details of input, event, and display handling from the application, allowing the programmer to develop an effective user interface without being concerned with details such as tracking the mouse, or updating the display. Equally important, a graphical toolkit provides a set of commonly used presentation components such as scrollbars, text editors, selection menus, and command buttons, that can easily be incorporated into graphical applications and interfaces.

For example, the software hierarchy of the X Window System is shown in the Figure 4. The window server is directly accessible to applications at the protocol level through a library (Xlib). A toolkit (Xt Intrinsics) is built on this library, and several sets of presentation components or "widget sets" are built on top of the toolkit (e.g., Athena Widgets, Open Look Widgets, Motif Widgets).

A toolkit is usually implemented as a library; however, it provides functionality well beyond a conventional subroutine library. Besides enabling access to the server and underlying hardware, it provides memory management, a comprehensive set of options that be can overridden by the user at run-time, and mechanisms for input, output, and event handling. The X Toolkit [3] is built on an object-oriented framework with simple base classes and multiple levels of inheritance. This development environment makes it easy to create new user interface components since many attributes can be inherited from existing components, thus substantially reducing the complexity of the software.

The handling of input and output events is particularly important in a multitasking environment where many applications can be active concurrently (unlike the single process nature of the MS-DOS and Macintosh operating systems). The X Toolkit is structured to gather

Figure 4: Window system and audio system hierarchies.

input from the user, and asynchronously call specific routines in the application program. The toolkit also provides a simplified procedural interface to the low-level asynchronous protocol layer provided by the X server. A client application can query the server by making subroutine calls to the toolkit—the query request to, and asynchronous reply from, the server are hidden from the application. Between the query and reply, other asynchronous events may have been received by the toolkit from the server.

7.2 Audgets

In X Window System terminology, a "widget" is a graphical user interface component. The term "audget," for "audio widget," will be used to represent an audio interface component analogous to a graphical widget (e.g., scrollbar or button). An audget set predefines common audio configurations to reduce the complexity of application programs, and to provide components for creating uniform user interfaces.

An application can create a combined sound and graphical interface by using both an audget and a graphical widget. A "multimedia toolkit" can be create multimedia interface components by encapsulating several audgets and widgets. Note that a graphical interface to an audget (or an audio interface to a widget) should be completely isolated from its voice interface so that a single audget can be used from a variety of user interfaces, and from more than one graphical widget set.

8 An Audget Set

Until a complete audio server, toolkit, and a range of applications have been developed it is difficult to envision a proper base set of voice interface components. However, experience with audio server designs and previous interactive voice-based applications leads to an initial set of audgets as proposed here. Presentation audgets, for example, include interfaces to digital recorders, sound editors, or voice menus.

Note that there is nothing magic about the presentation audgets described in this section, or applications that use them. Any application, such as the telephone answering machine described in section 5.1, can be built using a collection of audgets, or directly on top of an audio server. However, the program using the toolkit and audgets will be shorter, simpler, and more concise than the server-only version (e.g., see [5, 7]).

8.1 Voice Menu Audget

Menus are commonly used in voice mail systems, interactive voice response systems, and a range of other speech applications. A voice menu is similar to a graphical pull-down or pop-up menu; it presents a set of choices to, and accepts input from, the user. Voice menu items are presented with recorded or synthesized speech rather than in textual form, and input is gathered from a speech recognizer or telephone keypad, instead of a mouse or keyboard.

The programmatic interface to a voice menu minimally includes a list of text strings that represent voice prompts for use with a text-to-speech synthesizer, or a list of file names for a digital playback system. These prompts are then used by the audget to present options to the user. A voice menu audget also could handle more complex interactions, for example, with additional lists of second or third round prompts if the user fails to respond. The audget can have a built-in help facility that is derived directly from the prompt strings and the menu semantics, and the structure of the interactions may be customizable so that a variety of user interfaces can be rapidly prototyped. For example, the audget can sort the menu items alphabetically, or break the items down into submenus if the list is greater than a specified length.

8.2 Editor Audget

Applications often need to manipulate and edit audio buffers or sound files. The application programmer may want to use high-level cut, copy, and paste functions, while internally (in the server) this is accomplished through the manipulation of linked lists of pointers to audio buffers, reference counts, etc. An editing audget could manage these internal data structures, and provide a device-independent interface to a range of sound formats and encoding styles.

Editing functions can be controlled by touch-tones, mouse interactions, or possibly voice commands. With a sound editing audget, it is possible to create an audio-only user interface, or a combined audio and graphical user interface. The graphical interface can be a mouse-based selection mechanism, or keystroke-oriented, if integrated with a conventional keyboard-based text editor. An audio editor audget thus can be connected to a range of graphical components to create a variety of interactive editors each with its own look and feel.

Figure 5: A prototype graphical interface to an audio editor.

A prototype graphical interface to a voice editor is shown in Figure 5 (this application was built using the X Toolkit directly on the audio server—not using an audio toolkit). In this example, up to four sound buffers can be loaded and edited concurrently. The energy content of the sound is represented by black (high energy, sound present) and white (low-energy, silence) bars. The vertical dotted line in the first and third buffers show the current selection/insertion point in the buffer. The three horizontal lines in the second buffer indicate a user selection that can be cut or played. When a buffer is played, a cursor moves through the graphical representation of the sound to show the current position within the buffer. A graphically-based sound editor such as this may be written as a stand-alone client application, but it is expected that such sound editing capabilities will be used as a generic tool by other applications.

8.3 Extensible Audgets

Other audgets include interfaces to stand-alone players and recorders. A recording audget, for example, can notify the application after reaching a maximum length, on detection of silence, or on receiving other user input. The application controls the audget—it can set a file name associated with the recording, the state of the recorder, or the silence detection sensitivity. The audget can be queried for its status, position within a file, or recording parameters.

Basic record and play audgets can be extended via the inheritance mechanism, to operate on a sequence of sounds rather than individual sound files. Building on the queuing ideas of the server, a play-list or record-list audget can be programmed to analyze the items in the list. The audget can pre-fetch sound files while another event is active, to reduce operating system and audio server latencies. Such an extension increases the interactivity of the application and may further simplify the programming interface.

9 Other Technologies Required

The development of audio servers and audio toolkits is a significant step forward; however, several critical components are still missing before voice can be completely integrated into the desktop. As mentioned in section 6, a software-based implementation of all audio and speech processing components would not only bring down the cost of realizing audio technologies in the workstation, but it would simplify many aspects of the server.

To make the audio server and toolkit idea pervasive, there must be standard protocols and software interfaces—the wide acceptance of the X Window System is due in part to defining a low-level protocol specification [6] that is supported on a wide range of computer systems. Note that successful, and profitable, proprietary implementations of servers based on public standards can be built. In the fast-paced environment of "open systems," it is unlikely that a fully proprietary audio server will survive in the marketplace.

Another critical component is a mechanism for synchronizing events across media and servers. For example, the graphics-based audio editor described in section 8.2 needs to synchronize a moving cursor with the current location within the sound buffer as the sound is played. The exact correspondence of the audio with the graphics display cannot be guaranteed since both the audio and window servers work in an asynchronous manner. While a synchronizing mechanism can be built into a graphics or audio server, this does not solve the general problem—it also may be necessary to synchronize audio with video, and video with graphics. An independent synchronization service that can be used by all types of media servers is required.

10 Discussion

There are many ways to integrate audio into the workstation, and many levels of integration. Audio servers and toolkits directly address the issue of incorporating speech and audio I/O into a workstation user interface. Audio servers and toolkits, as presented here, take ideas developed by the window system community and successfully merge them with practical experience gained from building interactive audio and voice systems. The server and toolkit architectures discussed provide the underlying system support so that future workstations can take full advantage of voice in the interface.

This paper raises as many design issues as it answers; it is intended to encourage future research and development efforts in this area. The development of window systems and toolkits has accelerated the pace of graphical application development by allowing programmers to quickly and efficiently harness the power of the graphical interface. Servers and toolkits for audio, and easy-to-use audio interface components, should help bring about a similar revolution in the design of speech interfaces.

11 Acknowledgments

Many of these ideas originated from collaboration with Chris Schmandt, Keith Lantz, and Carl Binding. Toolkit design ideas further evolved in discussions with the Audio Group at Sun Microsystems, particularly Wayne Yamamoto, Don Jackson, Ben Stoltz, Daniel Steinberg, and Jeff Peck. Lisa Stifelman assisted in proofreading a version of this document.

References

[1] M. S. Ackerman, S. Manandhar, and C. Schmandt. Xspeak: A use for a speech interface in a window system. In *Proceedings of 1990 Conference*. American Voice I/O Society, 1990.

[2] B. Arons, C. Binding, K. Lantz, and C. Schmandt. The VOX audio server. In *Proceedings of the 2nd IEEE ComSoc International Multimedia Communications Workshop*. IEEE Communications Society, Apr. 1989.

[3] P. J. Asente and R. R. Swick. *X Window System Toolkit*. Digital Press, 1990.

[4] R. Kamel, K. Emami, and R. Eckert. PX: Supporting voice in workstations. *IEEE Computer*, Aug. 1990.

[5] J. McCormack and P. Asente. Using the X toolkit or how to write a widget. In *Proceedings of Summer USENIX '88*. USENIX, Summer 1988.

[6] A. Nye, editor. *X Protocol Reference Manual*, volume 0 of *X Window System Series*. O'Reilly and Associates, 1989.

[7] D. S. Rosenthal. A simple X 11 client program, or, how hard can it really be to write 'hello world'? In *Proceedings of Winter USENIX '87*. USENIX, Winter 1987.

[8] R. W. Scheifler and J. Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–106, Apr. 1986.

[9] C. Schmandt, M. S. Ackerman, and D. Hindus. Augmenting a window system with speech input. *IEEE Computer*, Aug. 1990.

[10] C. Schmandt and B. Arons. A conversational telephone messaging system. *IEEE Transactions on Consumer Electronics*, CE-30(3):xxi–xxiv, Aug. 1984.

[11] C. Schmandt and B. Arons. Phone Slave: A graphical telecommunications interface. *Proceedings of the Society for Information Display*, 26(1):79–82, 1984.

[12] C. Schmandt and B. Arons. A robust parser and dialog generator for a conversational office system. In *Proceedings of 1986 Conference*, pages 355–365. American Voice I/O Society, 1986.

[13] C. Schmandt and B. Arons. Conversational Desktop. ACM SIGGRAPH Video Review, Volume 27, 1987.

[14] C. Schmandt and B. Arons. Desktop audio. *Unix Review*, Oct. 1989.

[15] C. Schmandt, B. Arons, and C. Simmons. Voice interaction in an integrated office and telecommunications environment. In *Proceedings of 1985 Conference*. American Voice I/O Society, 1985.

[16] C. Schmandt and S. Casner. Phonetool: Integrating telephones and workstations. In *Proceedings of GLOBECOM '89*. IEEE Communications Society, Nov. 1989.

[17] C. Schmandt and M. McKenna. An audio and telephone server for multi-media workstations. In *Proceedings of the 2nd IEEE Conference on Computer Workstations*, pages 150–160. IEEE Computer Society, Mar. 1988.

[18] C. R. Strathmeyer. Voice in computing: An overview of available technologies. *IEEE Computer*, Aug. 1990.

[19] Sun Microsystems Inc., Mountain View, California. *NeWS: Technical Overview*, 1987.

[20] D. Swinehart. Telephone management in the Etherphone system. In *Proceedings of GlobeCom '87*. GlobeCom, Nov. 1987.

[21] D. Swinehart, L. Stewart, and S. Ornstein. Adding voice to an office computer network. Technical Report CSL-83-8, Xerox Palo Alto Research Center, Feb. 1984.

Biography

Barry Arons is a doctoral candidate in the Speech Research Group at the M.I.T. Media Laboratory. His research interests include highly interactive systems with emphasis on conversational voice communication. Before returning to academia, he was leader of the Desktop Audio Project at an industrial research laboratory, and a member of the technical staff at Hewlett-Packard Laboratories. He holds a BS in Civil Engineering from M.I.T., and a MS from the Architecture Machine Group at M.I.T. where he co-designed *Phone Slave* and the *Conversational Desktop*.

The author's address is:
M.I.T. Media Laboratory
20 Ames Street, E15-353
Cambridge, MA, 02139

e-mail: arons@media-lab.media.mit.edu