AudioFile: A Network-Transparent System for Distributed Audio Applications

Thomas M. Levergood, Andrew C. Payne,
James Gettys, G. Winfield Treese, and Lawrence C. Stewart***

Digital Equipment Corporation
Cambridge Research Lab

Abstract

AudioFile is a portable, device-independent, network-transparent system for computer audio systems. Similar to the X Window System, it provides an abstract audio device interface with a simple network protocol to support a variety of audio hardware and multiple simultaneous clients. AudioFile emphasizes client handling of audio data and permits exact control of timing. This paper describes our approach to digital audio, the AudioFile protocol, the client library, the audio server, and some example client applications. It also discusses the performance of the system and our experience using standard networking protocols for audio. A source code distribution is available by anonymous FTP.

This is a reprint of a paper that appeared in the Proceedings of the USENIX Summer Conference, June, 1993.

* Also with the MIT Laboratory for Computer Science.
** The authors' names are in random order.

1 Introduction

Audio hardware is becoming increasingly common on desktop computers. In 1990, the authors began a project at Digital's Cambridge Research Laboratory to explore desktop audio. [1] We began by designing a flexible I/O device for audio and telephony. Once that hardware was available, we began work on software. The result of our efforts is the AudioFile System.

Similar to the X Window System [13], AudioFile was designed to allow multiple clients, to support a variety of audio hardware, and to permit transparent access through the network. Since its original implementation, AudioFile has been used for many applications and experiments with desktop audio. These applications include audio recording, playback, video teleconferencing, answering machines, voice mail, telephone control, speech recognition, and speech synthesis. AudioFile supports multiple audio data types and sample rates, from 8 KHz telephone quality through 48 KHz high-fidelity stereo.

Currently, AudioFile runs on Digital's RISC and Alpha AXP systems, on Sun's SPARC systems, and on Silicon Graphics' Indigo workstations. A source code distribution is available by anonymous FTP.

Like the X Window System, AudioFile has four main components:

- The Protocol. The AudioFile System defines a wire protocol that links the server with client applications over local and network communication channels.
- Client Library and API. The client library and applications programming interface (API) provide a means for applications to generate protocol requests and to communicate with the server using a procedural interface.
- The Server. The AudioFile server contains all code specific to individual devices and operating systems. It mediates access to audio hardware devices and exports the device-independent interface to clients.
- Clients. The AudioFile distribution includes several out-of-the-box applications which make the system immediately usable and which serve as illustrations for more complex applications.

The parts of the implementation of AudioFile that are not specific to audio, such as client/server communications, are based on X11 Release 4. [2] We should emphasize that AudioFile is *not* an addition to the X Window System; it is a separate entity which borrowed some source code. We feel quite strongly that audio services should be separate from graphics.

AudioFile was designed with several goals in mind. These include:

- Network transparency. Applications can run on machines scattered throughout the network. Network transparency allows applications to run anywhere but still interact with the user. Network transparency also enables applications such as teleconferencing that need to use audio on several systems simultaneously.
- Device independence. Applications need not be rewritten to run on new audio hardware. The AudioFile System provides a common abstract interface to the real hardware, insulating applications from the messy details.
- Support for multiple simultaneous clients. Applications can run concurrently, sharing access to the actual audio hardware. [3]
- Support a wide range of clients. It should be possible to implement applications ranging from audio biff to multiuser teleconferencing. We chose to implement a few very general-purpose mechanisms that permit a wide variety of applications.
- Simplicity. Simple applications should be simple; complex applications should be possible.

This paper begins with a discussion of the historical context of AudioFile. We discuss the key abstractions, the network protocol, the client library, and the server implementation. Next, we describe some sample applications and analyze AudioFile's performance. We conclude with a brief discussion of our plans for future work.

[1] And video, but that is another story.

[2] Why start with a clean slate? For more information on how to steal code, consult Spencer [14].

[3] We think that something like an "audio window manager" might be useful, but so far we have not found it necessary to implement one.

2 Background

In the early 1980's, Xerox PARC built an Ethernet-based telephone system called Etherphone [15]. The system also had capabilities for workstation control of recording, playback, and storage. The Etherphone system was used primarily to explore issues of multimedia documents and computer-telephone integration.

In the mid 1980's, the Firefly multiprocessor [17] developed at Digital's Systems Research Center incorporated telephone-quality audio. An audio server buffered the past input and future output and exported a remote procedure call (RPC) interface to clients. This system pioneered explicit client control of time (as described in Section 3) and was primarily used for applications such as teleconferencing and multimedia presentations.

In the mid to late 1980's, the MIT Media Lab and Olivetti Research collaborated on a project called VOX [2]. In this system almost all audio functions were implemented inside the server, with the client merely controlling those functions. VOX was constrained by the view that clients would control the flow of audio between external devices, rather than handling the data themselves.

Other projects were underway at about the same time as AudioFile. Digital's XMedia [1] and Bellcore's Sonix [12] are similar to VOX in their emphasis on handling audio within the server. The conferencing system described by Terek and Pasquale [16] was based on a modified X server. In contrast, we think audio and graphics should be kept separate for ease of implementation and so that non-graphics equipped machines can still use audio.

3 Audio Abstractions

This section describes the fundamental abstractions of AudioFile. These provide the view of audio available to clients and guided the design of the protocol, client library, and audio server.

3.1 Devices

Abstract audio devices are Analog-to-Digital (ADC) and Digital-to-Analog (DAC) converters which produce and consume sample data at a regular rate known as the sampling frequency. The sample data are one of several predefined types and consists of one or more channels.

3.2 Time

The concept of audio device time is critical to understanding the design of all AudioFile components. We expose audio device time in the protocol and at the client library API. All audio recording and playback operations in the AudioFile System are tagged with time values that are directly associated with the relevant audio hardware.

There are a remarkable number of clocks in a modern distributed computer system. A simple desktop system might have four different clocks: time-of-day, interval timer, display refresh, and audio. Each computer system in a network has its own clocks. Time-of-day clocks may be synchronized with protocols such as NTP [9], but we are not aware of any systems that keep the other clocks synchronized. In principle, it is possible to use any clock for audio. Because we wanted to be able to specify audio data down to the individual sample, we chose to use the audio sample rate clock. The server maintains a representation of this clock in a "time register" for scheduling all audio events for the particular device. AudioFile does not provide a complete infrastructure for synchronization; rather, it supplies low-level timing information to its clients. Applications can build synchronization mechanisms suitable to their own needs. [4]

Audio device time is represented by a 32-bit integer that increments once per sample period and wraps on overflow. There is no absolute reference value for a device time; the value is set to 0 when the server is initialized and advances thereafter. Time comparisons are easy to implement, as illustrated by the following code fragment for a device running at 8000 samples per second.

```
if ((b - a) > 0)      /* time b is later than time a.            */
if ((b - a) < 0)      /* time b is earlier than time a.          */
if ((b - a) == 8000)  /* time b is one second later than time a. */
```

This method breaks when the difference approaches 2^{31}. Programs that deal with time must be careful not to make comparisons between widely separated time values. Even at a 48 KHz sampling rate, however, 2^{31} samples represent about 12 hours worth of audio.

[4] We envision adding standard mechanisms for providing clock conversion services, but have not yet encountered a compelling need to do so.

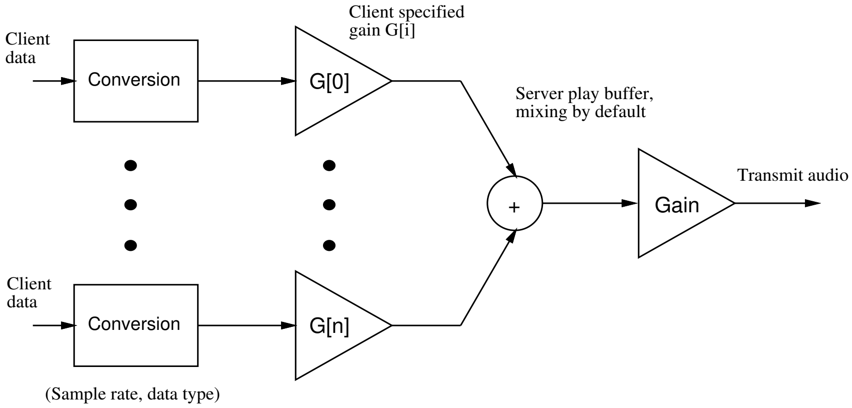

Figure 1: AudioFile server output model

Each play and record request carries with it an exact timestamp. The abstraction is implemented by buffering future playback and recent record data in the server. Continuous recording or playback is done by advancing the requested device time by the duration of the previous request.

Explicit control of time provides the mechanism needed for real-time applications. As long as play requests reach the server before their start times, playback will be continuous. A leisurely application will schedule playback well in the future, while a real-time application will schedule for the very near future.

Recording is a much easier problem than playback. The server buffers all audio input, typically for several seconds. No data will be lost unless a request fails to reach the server until well after its start time. Because the server buffers all device input, clients can request recording at times "in the past" and deliver instantaneous response. For example, an application can begin recording at the exact moment the "Record" button is pressed, even though there is a delay from pressing the button to scheduling the record. This is a more natural interaction than requiring the user to wait for an audible beep.

This explicit use of time means that clients operate on blocks of audio data. The alternative to AudioFile's block-oriented design would be streams, which would cause problems that AudioFile avoids. It is difficult to determine how much data is buffered in a stream or to find out if a stream is running or blocked. Streams tend to obscure issues of bandwidth and latency that are critical to real-time applications. Finally, in practice, applications deal with data in blocks anyway.

3.3 Output Model

AudioFile's output model is shown in Figure 1. Clients can schedule playback at any time from the present to four seconds into the future. Requests that fall in the past are silently discarded. Requests that fall beyond the four-second buffer are suspended until time advances to within four seconds.

After a playback request is received by the server, the data are passed through an optional module that converts the client's data type to the device's preferred representation. [5] After conversion, a client specified gain is applied, and the data are combined with the data from other clients in the playback buffer. The server will mix client data by default, but preemptive playback is possible. Finally, a master volume control is applied.

The output model specifies that silence is emitted during periods of time in which no data have been written to the output buffer. This may reduce network bandwidth requirements, since clients need not transmit silence data to the server.

3.4 Input Model

AudioFile's input model is shown in Figure 2. As in the output model, the server buffers four seconds of data. The recorded data are modified by a master input gain and placed into the server buffer. Clients requesting input data

[5] Conversions could include sample rate changing as well as data type conversion.

older than four seconds in the past are given silence. Requests within the past four seconds return buffered data, and requests in the future block until time advances. The system also supports a non-blocking record request, which will return as much data as possible without blocking. The input model also supports optional modules which convert the native audio hardware data type to the client requested type.

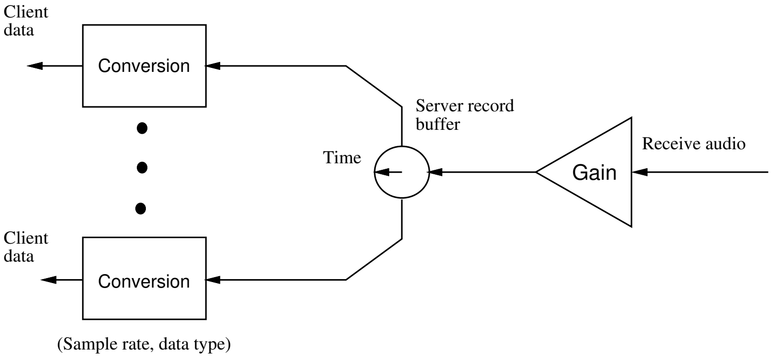

Figure 2: AudioFile server input model

3.5 Events

Events are asynchronous messages from server to client. Events may be generated by a device or as a side effect of some client's request. Clients must register their interest to receive various classes of events, such as a telephone device ringing or a change in a property used for interclient communications.

4 Audio Hardware

AudioFile is designed to support a variety of audio hardware. Currently supported devices include:

LoFi LoFi,[6] designed at CRL, is a workstation peripheral later released as the DECaudio product [7]. LoFi supports two 8 KHz CODECs, one connecting to a telephone line. LoFi contains a digital signal processing chip with 32K words of memory shared with the host. The DSP supports a 44.1 KHz stereo DAC and can also be used with external "DSP port" devices operating at up to 48 KHz. The telephone interface on LoFi enables applications such as voice mail and remote information access.

We see no difficulty in supporting other kinds of telephone interfaces, such as ISDN or PhoneStation [19].

JVideo JVideo is an experimental desktop video peripheral. Like LoFi, JVideo includes audio hardware based on a DSP and shared memory. However, JVideo has neither telephony capability nor an external DSP port.

Baseboard The Personal DECstation series, the Alpha AXP-based systems, and the SPARCstation 2 all include 8 KHz CODEC devices on their system modules.

Indigo The Silicon Graphics Indigo workstations support stereo audio at rates up to 48 KHz.

LineServer LineServer is an Ethernet peripheral used within Digital's research labs for remote bridging and routing. LineServer includes an 8 KHz CODEC. The LineServer version of the AudioFile server is interesting because the server runs on a nearby host, not on the LineServer itself.

5 Protocol Description

The AudioFile protocol is modeled on the same basic principles as the X Window System protocol. AudioFile can be used over any transport protocol that is reliable and which does not reorder or duplicate data. The current version supports TCP/IP and UNIX-domain sockets. At connection setup, the client and server exchange version information, the server sends audio device attributes, and clients provide authentication information.

[6] We called it LoFi primarily because it wasn't. LoFi always included high fidelity audio capability.

Audio and Events	SelectEvents	Select which events the client wants
	CreateAC	Create an audio context
	ChangeACAttributes	Change the contents of an audio context
	FreeAC	Free an audio context
	PlaySamples	Play samples
	RecordSamples	Record samples
	GetTime	Get the audio device's time
Telephony	QueryPhone	Get telephone state
	EnablePassThrough	Enable telephone passthrough
	DisablePassThrough	Disable telephone passthrough
	HookSwitch	Control hookswitch
	FlashHook	Flash hookswitch
	EnableGainControl	Not for general use
	DisableGainControl	Not for general use
	DialPhone	Obsolete, do not use
I/O Control	SetInputGain	Set input gain
	SetOutputGain	Set output gain (volume)
	QueryInputGain	Find out current input gain
	QueryOutputGain	Find out current output gain
	EnableInput	Enable input
	EnableOutput	Enable output
	DisableInput	Disable input
	DisableOutput	Disable output
Access Control	SetAccessControl	Set access control
	ChangeHosts	Change access control list
	ListHosts	List which hosts are permitted access
Atoms and Properties	InternAtom	Allocate unique ID
	GetAtomName	Get name for ID
	ChangeProperty	Change device property
	DeleteProperty	Remove device property
	GetProperty	Retrieve device property
	ListProperties	List all device properties
Housekeeping	NoOperation	Non-blocking NoOperation
	SyncConnection	Round-trip NoOperation
	QueryExtension	Not yet implemented
	ListExtensions	Not yet implemented
	KillClient	Not yet implemented

Table 1: AudioFile protocol requests

The AudioFile protocol defines 37 request types. [7] Many of these are for housekeeping purposes, including access control, interclient communications, and extensions (though no extensions are implemented today). Other requests support device control, telephony, and audio data handling. Table 1 summarizes AudioFile's protocol requests. All protocol requests have a length field, an opcode, and an opcode extension. Header fields are kept naturally aligned by padding request data out to a 32-bit boundary.

There are five fixed-size events supporting telephone control and interclient communications. All events contain the audio device time and the clock time of the host of the server. The host clock time may be needed when synchronizing with other media.

Rather than specifying all parameters for play and record with each request, a client uses an "audio context" (AC) to encapsulate most of these parameters. The audio context includes the play gain, number of channels, sample type, and byte order. ACs simplify the programming interfaces for play and record.

AudioFile adopted from X the idea of property lists to enable clients to communicate. Properties are associated with a device, and can be read and written by clients. Clients can register to be notified when properties change. These facilities can be used to coordinate use of resources and to share information, such as the last phone number dialed.

[7] For comparison, the X Window System has 119 requests in the core protocol.

6 Client Libraries

We have developed two client libraries: a standard interface to the AudioFile server and a utility library of common functions required by many clients.

6.1 Core Library

The core client library is the standard interface for AudioFile clients. Some of its functions provide interfaces to the AudioFile protocol; others provide an interface to the library's internal data structures. Table 2 summarizes these library functions.

Some library functions, such as AFGetTime(), require an immediate response from the server; others, such as AFCreateAC(), do not. In the former case, the library blocks until a reply is received. Otherwise, the library may defer sending the request and will return to the client immediately. Certain operations, including the synchronous functions, flush any deferred requests.

AudioFile provides a simple access control scheme based on host network address. The access control functions allow hosts to be added or removed from the access list and allows access control to be disabled entirely.

6.2 Client Utility Library

The AudioFile distribution also includes a utility library to provide a number of facilities that are used by several clients. Two kinds of facilities are provided: tables and subroutines.

The AudioFile system handles a variety of digital audio data formats, such as μ-law and A-law. The utility library includes tables for converting these formats to and from linear encoding. The library also includes tables for computing signal power, gain control, and generating sine waves at various frequencies.

The utility library gathers together a number of useful subroutines. Most do not directly interact with the AudioFile protocol. They include subroutines for generating on-the-fly gain translation tables for μ-law and A-law samples, tone generation procedures, and some miscellaneous functions. One subroutine, AFDialPhone(), encapsulates the operations necessary to generate Touch-Tone dialing sequences on a telephone device.

7 Server Design

The AudioFile server is responsible for managing the audio hardware and presenting abstract device interfaces to clients via the AudioFile protocol. This section discusses some of the important issues in the server's design, the implementation of buffering to provide the audio device abstraction, and some other details of the server's implementation.

7.1 Implementation Considerations

Our primary concern for the implementation of an AudioFile server was performance. We wanted the server to run continuously in the background, so we felt that the quiescent server should present a negligible CPU load. Further, load due to the server with a few clients running should leave most of the CPU available for applications.

We considered using threads to implement the server, but were apprehensive about the performance and portability of existing thread packages. Although the internal structure of the server might be slightly cleaner with threads, we took the safer route and designed the server as a single-threaded process. The server must handle requests from multiple active clients, so we designed it so that one client cannot dominate the available processing time.

7.2 Buffering

The server maintains input and output buffers for each audio device. A periodic update task moves samples between the server buffers and the audio hardware. Figure 3 illustrates the server record and play buffers before and after the update task executes. At each invocation, the update task moves new record data (since recLastUpdate) from the hardware buffer to the server buffer, and moves the next batch of playback data (starting at the "before" timeNextUpdate) from the server buffer to the hardware buffer. Finally, the update task initializes the end of the server buffer with silence.

Connection Management	AFOpenAudioConn	Open a server connection
	AFCloseAudioConn	Close a server connection
	AFSynchronize	Synchronize with the audio server
	AFSetAFterFunction	Set a synchronization function
Audio Handling	AFPlaySamples	Play digital audio samples
	AFRecordSamples	Record digital audio samples
	AFGetTime	Get the device time of a device
Audio Contexts	AFCreateAC	Create a new audio context (AC)
	AFChangeACAttributes	Modify an AC
	AFFreeAC	Free resources associated with an AC
Event Handling	AFEventsQueued	Check for events
	AFPending	Returns number of unprocessed events
	AFIfEvent	Find and dequeue a particular event (blocking)
	AFCheckIfEvent	Find and dequeue a particular event (nonblocking)
	AFPeekIfEvent	Find a particular event (blocking)
	AFNextEvent	Return the next unprocessed event
	AFSelectEvents	Select events of interest
Telephone	AFCreatePhoneAC	Create an AC for a telephone device
	AFFlashHook	Flash the hookswitch on a telephone device
	AFHookSwitch	Set the state of the hookswitch
	AFQueryPhone	Returns the state of the hookswitch and loop current
I/O Control	AFEnableInput	Enable inputs on an audio device
	AFDisableInput	Disable inputs on an audio device
	AFEnableOutput	Enable outputs on an audio device
	ADisableOutput	Disable outputs on an audio device
	AFEnablePassThrough	Connect local audio to the telephone
	AFDisablePassThrough	Remove the direct local audio/telephone connection
	AFQueryInputGain	Get minimum/maximum input gains for a device
	AFQueryOutputGain	Get minimum/maximum output gains for a device
	AFSetInputGain	Set the input gain of a device
	AFSetOutputGain	Set the output gain of a device
Access Control	AFAddHost	Add a host to the access list
	AFAddHosts	Add a set of hosts to the access list
	AFListHosts	Return the host access list
	AFRemoveHost	Remove a host from the access list
	AFRemoveHosts	Remove a set of hosts from the access list
	AFSetAccessControl	Enable or disable access control checking
	AFEnableAccessControl	Enable access control checking
	AFDisableAccessControl	Disable access control checking
Properties	AFGetProperty	Manipulate properties
	AFListProperties	Get a list of existing properties
	AFChangeProperties	Modify a property
	AFDeleteProperty	Delete a property
	AFInternAtom	Install a new atom name
	AFGetAtomName	Fetch the name of an atom
Error Handling	AFSetErrorHandler	Set the fatal error handler
	AFSetIOErrorHandler	Set the system call error handler
	AFGetErrorText	Translate error code to a string
Miscellaneous	AFNoOp	Don't do anything
	AFFlush	Flush any queued requests to the server
	AFSync	Default synchronization function
	AFAudioConnName	Return the name of the audio server

Table 2: AudioFile client library functions

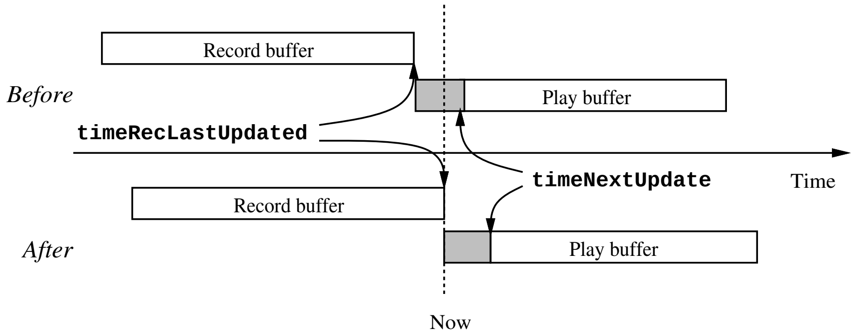

Figure 3: AudioFile periodic updates

Unless a client request falls into the shaded portions of Figure 3, it can be handled entirely out of the server buffers. If a record request falls after recLastUpdate, the server performs an update before handling the request. If a playback request falls before timeNextUpdate, the server writes the data all the way through to the hardware.

7.3 Server Implementation

An AudioFile server is organized like an X server. It includes device independent audio (DIA), device dependent audio (DDA), and operating system (OS) components. The DIA section is responsible for managing client connections, dispatching on client requests, sending replies and events to clients, and executing the main processing loop. The DDA section is responsible for presenting the abstract interface for each supported device and contains all device-specific code. Finally, the OS section includes all the platform or operating system-specific code. Much of the OS and DIA code is based on X11R4.

Instead of using threads, we implemented a simple task mechanism which allows procedures to be scheduled for execution at future times, outside the main flow of control. The task mechanism is used by the server's update mechanism and by the dispatcher to resume execution of partially completed client requests.

At the core of the DIA section is the main control loop, which relies heavily on the select() system call. select() is called with file descriptors for client connections and open devices, as well as a timeout argument for the next task which needs to execute. When select() returns, the server runs any pending tasks and then handles input events and client requests.

Client requests are processed by the dispatcher. The request type is used to index into a table of protocol request handler procedures. All handlers are implemented by the device independent part of the server, but audio specific requests are passed to the device dependent audio server.

Adding DDA support for a new audio device is straightforward. The interface between the DIA and DDA sections of the server is very similar to that in the X server.

7.4 Device Dependent Server Examples

This section describes some of the device dependent implementation details of the LoFi, baseboard, and LineServer DDA code.

Alofi — LoFi Server

There are play and record circular buffers for each CODEC device and for each channel of the stereo HiFi device. For HiFi, we implemented a single stereo abstract device, as well as separate left and right devices for those clients not requiring stereo data. The host performs audio I/O by reading and writing these shared memory buffers. The LoFi's DSP runs a simple program that maintains the device time and buffers for each device.

We optimized the update procedure to achieve good performance. The server's periodic updates, which move samples between the server's buffers and the hardware, can consume quite a few CPU cycles, especially at high sample rates. We chose to have the HiFi record update run only if there is an active record client. The first record operation performed under an audio context marks the context as recording. A per-device reference count then controls the

Audio Handling	apass	Record from one AF server and playback on another
	aplay	Playback from files or pipes
	arecord	Record to files or pipes
	abiff	Incoming e-mail notification by audio
	abob	Tk-based multimedia demonstration
	radio	Multicast network audio
	xplay	X-based sound file browser
	abrowse	Tk-based sound file browser
Device Control	ahs	Telephone hook switch control
	aphone	Telephone dialer
	aset	Device control
	aevents	Report input events
	adial	Tk telephone dialer
	axset	Tk version of aset
	afxctl	X-based event display and device control
Signal Processing Utilities	afft	Tk based real-time spectrogram display
	xpow	X display of audio signal power
	autil	Signal generator
Access control and Properties	ahost	AudioFile server access control
	alsatoms	Display defined atoms
	aprop	Display and modify properties

Table 3: AudioFile clients

operation of the record update code. Similarly, the play update runs only when there is outstanding client data. We also initialize the server buffer with silence only when absolutely necessary. In the common case of contiguous playback requests, silence filling is never necessary.

Aaxp and Asparc — Baseboard Servers

The audio servers for the baseboard audio on Alpha AXP workstations and SPARCstations use kernel device drivers with similar interfaces. The server's update procedure uses the device driver read and write interfaces for recording and playing audio data. Because the kernel device drivers do not maintain a time register for the baseboard CODECs, the server maintains an estimated value using the system clock and must occasionally resynchronize with the device driver.

Als — LineServer Server

For the LineServer, an AudioFile server running on a nearby workstation uses a private UDP-based protocol to communicate with the device. The LineServer runs simple firmware that processes incoming packets and moves samples to and from the audio hardware. On the workstation, a periodic update task moves data between the server's buffers and the LineServer's buffers using the private protocol. The server makes every attempt to minimize access to the LineServer, since crossing the network is a relatively expensive operation. Only requests in the update regions require network traffic. For requests that require returning a device time, the server generates an estimate.

8 AudioFile Clients

The AudioFile distribution includes a number of client programs, including applications for recording, playback, and signal processing, as well as telephone, device, and access control. Some clients have graphical user interfaces using X and Tk toolkits [11]. Table 3 summarizes the clients. In the remainder of this section we describe two clients, aplay and apass, in some detail, to illustrate the simplicity of the AudioFile client library programming interface. We also include an answering machine shell script to show how simple AudioFile clients can be combined into larger applications.

8.1 aplay

aplay reads digital audio from a file or standard input and sends it to the server for playback. aplay can serve as the core of a sound-clip browser or voice mail retrieval program or as the final stage in a signal processing pipeline. For example, our software implementation of the DECtalk [3] speech synthesizer uses aplay for output.

aplay handles only "raw" sound files but could be easily extended to handle popular sound file formats. aplay works with any fixed-size encoding format and any number of channels — but the user must know the format of the file and choose an appropriate server device.

As an example, we show a simplified version of aplay. Readers who wish all the details should read the sources, which are included in the AudioFile distribution.

```
aplay()
{
    aud = AFOpenAudioConn("");        /* open a connection to the server */
    device = FindDefaultDevice(aud);  /* select audio device             */
    /* set up audio context, possibly setting the gain and endian-ness   */
    ac = AFCreateAC(aud, device, (ACPlayGain | ACEndian), &attributes);
```

FindDefaultDevice() locates the lowest numbered audio device that is not connected to the telephone, which is usually the local audio device.

```
    srate = ac->device->playSampleFreq;      /* sample rate              */
    type = ac->device->playBufType;          /* encoding type            */
    ssize = sample_sizes[type] * channels;   /* bytes per sample         */
    buf = malloc(BUFSIZE*ssize);             /* allocate the play buffer */

    nbytes = read(fd, buf, BUFSIZE*ssize);   /* pre-read first buffer    */
```

It is not logically necessary to pre-read the first file block, but doing so avoids putting the latency of the file read between the call to AFGetTime() and the first call to AFPlaySamples().

```
    t = AFGetTime(ac);                      /* obtain initial device time        */
    t = t + (time_offset * srate);          /* schedule initial playback         */

    do {                                    /* loop until done or brown on top   */
        nact = AFPlaySamples(ac, t, nbytes, buf);  /* send samples to the server */
        /*
         * At this point, the buffers in the server hold the samples from time
         * nact to time t.  Next, we figure out how many samples we read
         * from the file, and schedule the next block to start after this one
         */
        t += (nbytes / ssize);
    } while ((nbytes = read(fd, buf, BUFSIZE*ssize)) > 0);
}
```

This code fragment is the inner loop of aplay. It obtains the current device time and schedules the playback of the first block of audio. Thereafter, it schedules each successive block to play directly on the heels of the previous block, so playback will be continuous. After each call to AFPlaySamples(), the time pointer is simply incremented by the number of samples played.

There is no code in aplay for flow control. aplay makes a fundamental, but unwritten, assumption that the file system can supply audio data faster than it is required by the server. The audio data will be buffered in the server until aplay gets about four seconds ahead. At that point, the connection to the server will block occasionally, keeping aplay about four seconds ahead. The file I/O side of aplay can block for as long as four seconds without causing a break in the audio.

8.2 apass

Teleconferencing was one of our major goals in the development of AudioFile. apass is not a complete teleconferencing application; it simply records from one device and after a small delay, plays back on another device. However, apass addresses two of the fundamental problems of network teleconferencing:

- Management of end-to-end delay. In teleconferencing, it is important to have tight control over the end-to-end delay of the audio connection. If the round trip delay exceeds about 300 milliseconds, humans begin to have difficulty with conversational dynamics. apass sets up a strict delay budget, accounting for the various factors involved.

- Multiple clocks. In a system with multiple audio devices, the different devices are usually controlled by different clocks. If the transmitting end runs faster than the receiving end, then excess samples will accumulate in buffers in between, gradually increasing the end-to-end delay. If the transmit clock is slower than the receive clock, the buffers will run dry and the playback will break up. apass tracks the clock rates and resynchronizes as necessary.

apass reads blocks of samples from the transmit server and schedules their playback on the receive server. The delay between input and output is made up of three parts:

- Packetization Delay. The last sample of a block must be recorded before the first sample can be played back. The size of the block sets the minimum end-to-end delay.
- Transport Delay. apass reads samples from the transmit server and sends them to the receive server. The transmission delay, software overhead, and rescheduling delays make up the transport delay.
- Anti-Jitter Delay. apass uses explicit control over playback time to insert extra delay at the receiving server. This absorbs variation in the transport delay, provided that the variation is not larger than the anti-jitter delay.

```
apass()
{
    faud = AFOpenAudioConn(faf);   /* open connections to the from */
    taud = AFOpenAudioConn(taf);   /*  and to audio servers        */

    /* set up audio contexts, find sample size and sample rate    */
    fac = AFCreateAC(faud, fdevice, ACRecordGain, &attributes);
    srate = fac->device->playSampleFreq;
    ssize = sample_sizes[fac->device->playBufType] * fac->device->playNchannels;
    tac = AFCreateAC(taud, tdevice, ACPlayGain, &attributes);

    /* establish a value for the delay from record to playback    */
    delay_in_samples = fsrate * delay_in_seconds;

    ft = AFGetTime(fac);           /* get starting times           */

    /* playback will start delay_in_samples in the future         */
    tt = AFGetTime(tac) + delay_in_samples;

    for (;;) {
        /* record samples and play them back                      */
        factt = AFRecordSamples(fac, ft, samples_bufsize*ssize, buf, ABlock);
        tactt = AFPlaySamples(tac, tt, samples_bufsize*ssize, buf);
```

AFRecordSamples() and AFPlaySamples() accept the parameters ft (from-time) and tt (to-time) respectively, and return factt (from-actual-time) and tactt (to-actual-time). factt, the current transmit server time, will be approximately equal to ft+samples_bufsize, because the pacing flow control is provided by the transmit server.

```
        est_delay = average_recent(tt - tactt); /* average recent delay estimates */

        /* if the delay has drifted outside of the allowable
           region, then resynchronize the connection                      */
        if ((est_delay < delay_lower_limit) || (slip >= delay_upper_limit))
            tt = tactt + delay_in_samples;

        /* finally, update the start times of the next block              */
        ft += samples_bufsize;  tt += samples_bufsize;
    }
}
```

apass uses a very simple algorithm for synchronizing clocks. tt – tactt is an estimate of the current end-to-end delay minus the packetization and average transport delays. This difference should be about delay_in_samples, but will vary from this nominal value. apass averages several recent estimates to determine if the end-to-end delay is within the range specified by delay_in_samples plus or minus the anti-jitter specification. If the receive clock is too fast, the average delay will eventually drift below the lower end of the range. If the receive clock is too slow, then the delay will drift above the upper end of the range. In either case, tt is reset to nominal delay, resynchronizing the connection.

Note that time values from the two audio servers cannot be directly compared because they have different initial values and slightly different rates. apass avoids this problem by tracking the buffering available at the receiving server.

A robust teleconferencing application would likely choose a more sophisticated algorithm for managing multiple clocks, such as resynchronizing whenever the audio is quiet, or resampling at the receive sample rate.

8.3 A Trivial Answering Machine

Separate simple clients can be used to construct interesting applications. For example, we can implement a simple answering machine as a shell script.

```
#!/bin/sh
#
    while true; do
#
    aevents -ringcount 3.0             # wait for the phone to ring three times
    ahs off                            # answer the phone (take off-hook)
    aplay -f -d 0 outgoing_message.snd # play outgoing message
    aplay -f -d 0 beep.snd             # play a beep
#
# record up to 30 seconds, or until the caller stops talking
#
    arecord -silentlevel -35.0 -d 0 -silenttime 4.0 -l 30.0 -t -1.0 >>messages.snd
    aplay -f -d 0 thanks.snd           # play a thank-you message
    ahs on                             # hang up the phone
#
# add a date stamp to the message file using a text to speech
#   synthesizer (not part of AudioFile ...)
#
    date | tts >>messages.snd
    done                               # Go back and get the next message
```

8.4 Other AudioFile Applications

There are already several applications which use AudioFile but are not distributed with it. We mention some interesting examples here.

DECtalk We built a software-only version of the DECtalk text-to-speech synthesizer, called tts. The synthesizer generates waveform samples on the standard output, which we can pipe to aplay for output.

DECspin DECspin is a Digital product for network audio and video teleconferencing. DECspin uses AudioFile to provide its voice capability.

VAT A team at the University of California, led by Van Jacobson, has built a network teleconferencing application using IP multicast protocols [4]. VAT can use AudioFile for its audio I/O.

Sphinx Sphinx II is a continuous speech recognition system [6] developed at Carnegie Mellon University. We use AudioFile's high fidelity input capability to supply audio to Sphinx.

9 Performance Results

In this section, we present some performance results for our implementation of the AudioFile System. First, we measure the time to complete client library operations. Next, we measure the CPU load for recording and playback. Finally, we briefly discuss our experience using TCP as the transport protocol.

9.1 Server and Client Performance

We measured latencies and performance of our AudioFile implementation by timing various client library functions. We tested with two types of systems (MIPS and Alpha AXP) under six local and networked configurations:

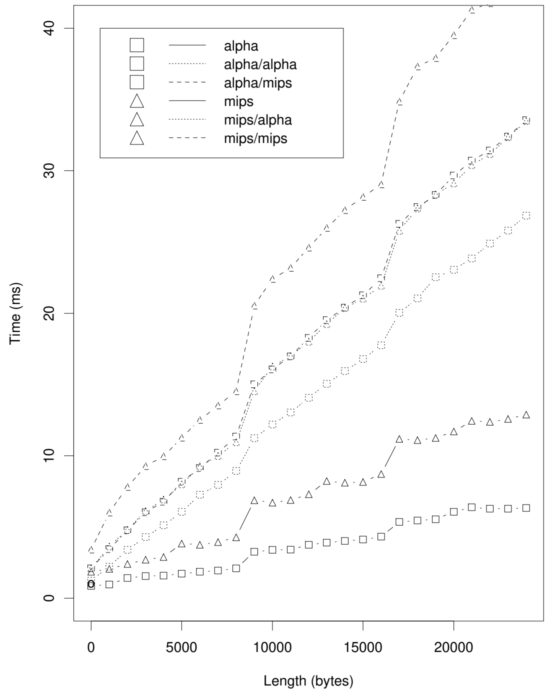

Figure 4: AFRecordSamples() timings

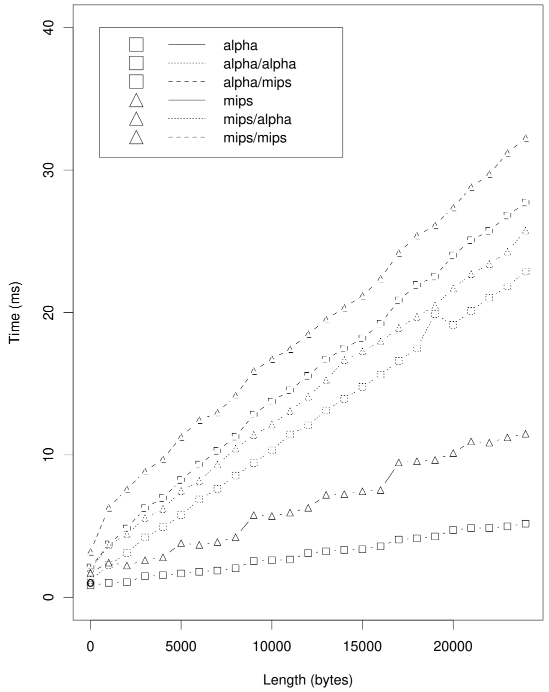

Figure 5: AFPlaySamples() timings

alpha	Alpha local client & server
alpha/mips	Alpha client, MIPS server
alpha/alpha	Alpha networked client & server
mips	MIPS local client & server
mips/mips	MIPS networked client & server
mips/alpha	MIPS client, Alpha server

We did all testing with the LoFi server, Alofi, using the 8 KHz CODEC device. All MIPS systems were DECstation 5000/200s running ULTRIX 4.3, and all Alpha AXP systems were DEC 3000/400s running DEC OSF/1 for Alpha AXP V1.2. All network testing took place on a lightly loaded 10 Mbit/sec Ethernet.

Play and Record

The AudioFile library functions that move data have latencies that depend on the length of the data. Figures 4 and 5 show the elapsed time for various length AFRecordSamples() and AFPlaySamples() requests on the different system configurations. The requests were scheduled to hit entirely in the server's buffers (and therefore not block). These times were computed by computing the average time for 1000 operations.

The timings for short requests represent the basic overhead for a server/client exchange. In the record case, the jumps at approximately 8K bytes are due to "chunking" performed in the client library. Record requests larger than 8K bytes (not samples) are broken into 8K byte chunks to better control interactions with transport protocol heuristics and to simplify server implementation. Each record request completes synchronously — the client library waits for the reply before sending the next chunk. A 16K byte request therefore takes the same time as two independent 8K byte requests.

Play requests are chunked in a similar fashion, but the client does not wait for each chunk to complete. The client sends all chunks to the server and waits only for the reply to the last one. The resulting play timing is a nearly linear function of the play request size.

Open Loop Record/Play

The timings of various AudioFile operations have implications for applications that process audio in real time. Simple applications, such as playing a file, do not really care how long the operations take to complete, as long as the throughput exceeds the audio data rate. However, other applications, such as teleconferencing, depend on minimizing the time needed to handle samples.

To illustrate some of the implementation limits, we coded a loopback test that reads samples from a device and writes them back as quickly as possible. The test uses a non-blocking record function that returns only what samples are available. The algorithm is shown in this code fragment:

```
for(;;) {
    now = AFRecordSamples(ac, next, 8000, buffer, ANoBlock);
    length = now - next;
    AFPlaySamples(ac, next+4000, length, buf);
    next = now;
}
```

The rate at which this loop iterates is governed entirely by the AudioFile overhead and represents a limit for handling real-time audio. The average times to complete one iteration are shown in Table 4.

Configuration (client/server)	Time (ms)
alpha	0.87
alpha/alpha	1.27
alpha/mips	2.17
mips	1.93
mips/alpha	2.15
mips/mips	3.45

Table 4: Loopback timing

AudioFile's overhead establishes a minimum latency for real-time applications. However, we feel that AudioFile will be adequate for all but the most demanding real-time requirements. In some networked configurations, however, AudioFile's overhead will be small compared to the network delays. For example, a link across North America has a minimum 15 millisecond propagation time, not including transmission and routing time.

9.2 CPU Usage

We also investigated the CPU usage for playback and record operations. The tests consisted of playing and recording 30 seconds of audio at two sample rates and data types: 8 KHz μ-law and 44.1 KHz CD-quality stereo. Tests were done in a local configuration with the server and client running on the same machine, using UNIX domain sockets. Table 5 summarizes the server and client CPU usage for the two cases. Both user and system times (in seconds) are given, and the percentage column indicates the system load, computed by dividing total CPU time by the duration. These results include the costs of transferring data to LoFi using programmed I/O. CPU usage would be reduced by use of shared memory transport or by a DMA I/O device.

9.3 Data Transport

AudioFile can be used over almost any transport protocol, though the details of the protocol may affect real-time audio performance. This section discusses our experience using TCP as the transport layer.

Although applications such as apass may exercise tight control over timing, most do not have strong real-time requirements. TCP is usually sufficient for these applications because the delay caused by retransmission of lost packets is small compared to the buffering of unplayed samples. On the other hand, applications like teleconferencing do require timely delivery of the audio data.

We found that a naively implemented teleconferencing application displayed serious problems when used over a transcontinental TCP link. We observed frequent and lengthy dropouts in the audio stream, which were especially

System	Test case	Server User	Server Sys	Client User	Client Sys	Total	% Load
Alpha	8 KHz playback	0.2	0.1	0.0	0.1	0.4	1.2
DEC 3000/400	8 KHz record	0.1	0.1	0.1	0.1	0.5	1.5
	44.1 KHz playback	2.9	2.0	0.0	0.5	5.4	18.0
	44.1 KHz record	3.7	1.1	0.7	1.4	6.9	22.9
MIPS	8 KHz playback	0.2	0.2	0.0	0.1	0.5	1.7
DEC 5000/200	8 KHz record	0.2	0.1	0.2	0.1	0.6	2.0
	44.1 KHz playback	1.9	1.9	0.0	2.6	6.4	21.3
	44.1 KHz record	4.4	4.0	1.1	3.4	12.9	43.0

Table 5: CPU usage for playback and record operations

likely with bidirectional data streams. These stem from packet losses caused by a phenomenon known as "ACK-compression" [10, 20], a subtle consequence of the use of window-based flow control. The duration of each dropout is exacerbated by TCP's slow-start algorithm [5], which comes into play when packets are dropped by the network.

ACK-compression occurs when the spacing between acknowledgments is changed by delays in the routers. This can cause cause TCP to send large bursts of packets, which overrun the buffers in a router, causing packets to be dropped. Unfortunately, the TCP slow-start algorithm converts these losses into lengthy recovery periods during which data flows more slowly. On a connection such as a long-haul T1 circuit, it can take several seconds to restore full throughput.

TCP is arguably the wrong transport protocol for applications such as teleconferencing, since it tries to guarantee ordered packet delivery without any concern for packet delay. Many applications instead need guarantees on bandwidth and latency, but they may be prepared to accept some lost data. Networks and protocols that provide such guarantees are active areas of research. To manage these issues, all of the teleconferencing applications mentioned in Section 8 are split among sites, using special protocols over long-haul paths, and only communicate locally with AudioFile servers.

10 Summary

The AudioFile System provides device-independent, network-transparent audio services. With AudioFile, multiple audio applications can run simultaneously, sharing access to the actual audio hardware. Network transparency means that application programs can run on machines scattered throughout the network. Because AudioFile permits applications to be device-independent, applications need not be rewritten to work with new audio hardware.

10.1 Areas for Further Work

It is remarkably difficult to get something as big as AudioFile completely right. We are very pleased with our basic design decisions, but we do have a list of items which, if implemented or fixed, would make AudioFile still more useful.

- Audio devices should support multiple sample rates, and the library should support sample rate changing.
- Audio devices should have an ordered list of supported data formats, so that the device can express a preference for one format over another.
- The protocol and library should offer improved support for synchronization and conversion between clocks, including clock prediction routines and the simultaneous reporting of all device clocks.
- Clients should be able to refer symbolically to "the local loudspeaker" or "the telephone".
- Clients which deal with files should know about various popular sound file formats.
- The server should support compressed data types, which would make AudioFile more useful in low-bandwidth environments.

10.2 Conclusions

We believe that AudioFile has done well in meeting our design objectives: network transparency, device independence, simultaneous clients, simplicity, and ease of implementation. We also believe our experience has validated our principles:

- Client control of time. AudioFile permits both real-time and non-real-time audio applications using the same primitives.
- No rocket science. Our decisions to build on top of standard communications protocols and not to use threads have improved the portability of the system. It is also arguable that AudioFile performs so well precisely because of its minimalist underpinnings.
- Simplicity. Simple play and record clients require very little code. Indeed, many applications can be constructed using independent AudioFile clients organized by shell scripts.
- Computers are fast. We did not let fear of per-sample processing get in our way. Our slowest implementation uses less than 2% of the CPU for telephone-quality playback.

10.3 How to Get AudioFile

AudioFile can be copied by anonymous FTP from crl.dec.com (192.58.206.2) in /pub/DEC/AF/AF2R2.tar.Z. Some stereo sound bites are in AF2R2-other.tar.Z. A more detailed description of the design and implementation of AudioFile is also available as a technical report [8].

We apologize for the code being cluttered with left-justified chicken scratches. [8] We wanted AudioFile to be portable, but at least one major vendor does not support function prototypes with their stock C compiler.

We have created an Internet mailing list af@crl.dec.com for discussions of AudioFile. Send a message to af-request@crl.dec.com to be added to this list.

10.4 Acknowledgments

Many people have contributed to AudioFile. We would like to thank Ricky Palmer and Larry Palmer for the SPARCstation DDA code. They, along with Lance Berc and Dave Wecker, persevered as early users of AudioFile. Lance Berc's experiments with long-distance teleconferencing taught us much about the network issues. Jeff Mogul offered valuable assistance on understanding these issues. Guido van Rossum contributed the DDA code for the Silicon Graphics Indigo only two weeks after we released the first public distribution. Dick Beane provided useful comments on drafts of this paper. We would also like to thank Victor Vyssotsky and Mark R. Brown for putting up with us.

References

[1] Susan Angebranndt, Richard L. Hyde, Daphne Huetu Luong, Nagendra Siravara, and Chris Schmandt. Integrating audio and telephony in a distributed workstation environment. In *Proceedings of the USENIX Summer Conference*. USENIX, June 1991.

[2] B. Arons, C. Binding, K. Lantz, and C. Schmandt. The VOX audio server. In *Multimedia '89: 2nd IEEE COMSOC International Multimedia Communications Workshop*, 1989.

[3] Edward Bruckert, Martin Minow, and Walter Tetschner. Three-tiered software and VLSI aid developmental system to read text aloud. *Electronics*, Apr. 21, 1983.

[4] Steve Deering and Steve Casner. The first IETF audiocast. *ACM Communications Review*, 22(3), July 1992.

[5] Van Jacobson. Congestion avoidance and control. In *Proc. SIGCOMM '88 Symposium on Communications Architectures and Protocols*, pages 314–329, Stanford, CA, August 1988.

[6] Kai-Fu Lee. *Automatic Speech Recognition: The Development of the SPHINX System*. Kluwer Academic Publishers, Norwell, MA, 1989.

[8] You really should read Thompson's paper on the Plan 9 C compiler [18].

[7] Thomas M. Levergood. LoFi: A TURBOchannel audio module. CRL Technical Report 93/9, Digital Equipment Corporation, Cambridge Research Lab, 1993.

[8] Thomas M. Levergood, Andrew C. Payne, James Gettys, G. Winfield Treese, and Lawrence C. Stewart. Audiofile: A network-transparent system for distributed audio applications. CRL Technical Report 93/8, Digital Equipment Corporation, Cambridge Research Lab, 1993.

[9] D. L. Mills. Network time protocol (NTP). Internet RFC 958, Network Information Center, September 1985.

[10] Jeffrey C. Mogul. Observing TCP dynamics in real networks. In *Proc. SIGCOMM '92 Symposium on Communications Architectures and Protocols*, Baltimore, MD, August 1992.

[11] John K. Ousterhout. An X11 toolkit based on the Tcl language. In *Proceedings of the USENIX Winter Conference*, January 1991.

[12] Steven J. Rohall. Sonix: A network-transparent sound server. In *Proceedings of the Xhibition 92 Conference*, June 1992.

[13] Robert W. Scheifler and James Gettys. *X Window System*. Digital Press, Bedford, MA, 3rd edition, 1991.

[14] Henry Spencer. How to steal code -or- inventing the wheel only once. In *Proceedings of the USENIX Winter Conference*, pages 335–346. USENIX, February 1988.

[15] D. C. Swinehart, L. C. Stewart, and S. M. Ornstein. Adding voice to an office computer network. In *Proceedings of GlobeCom 1983*, November 1983.

[16] Robert Terek and Joseph Pasquale. Experiences with audio conferencing using the X window system, UNIX, and TCP/IP. In *Proceedings of the USENIX Summer Conference*. USENIX, June 1991.

[17] Charles P. Thacker, Lawrence C. Stewart, and Edwin H. Satterthwaite Jr. Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, 37(8):909–920, Aug. 1988.

[18] Ken Thompson. A new C compiler. In *Proceedings of the Summer 1990 UKUUG Conf.*, pages 41–51, London, July 1990.

[19] Stephen A. Uhler. PhoneStation, moving the telephone onto the virtual desktop. In *Proceedings of the USENIX Winter Conference*. USENIX, January 1993.

[20] Lixia Zhang, Scott Shenker, and David D. Clark. Observations on the dynamics of a congestion control algorithm: The effects of two-way traffic. In *Proc. SIGCOMM '91 Symposium on Communications Architectures and Protocols*, pages 133–147, Zurich, September 1991.

Author Information

Lawrence C. Stewart joined the Cambridge Research Lab (CRL) in 1989 after 5 years at Digital's Systems Research Center. His interests include speech, audio, and multiprocessors. He was one of the designers of the first Alpha AXP computer system. Before joining Digital, Larry was at Xerox PARC. He received an S.B. from MIT in 1976, and M.S. and Ph.D. degrees from Stanford in 1977 and 1981, all in Electrical Engineering.

G. Winfield Treese joined CRL in 1988 after working at MIT on Project Athena. Win's interests are in networks and distributed systems. He received an S.B. in Mathematics from MIT in 1986 and an S.M. in Computer Science from Harvard University in 1992. He is now pursuing a Ph.D. at MIT in the area of computer networks.

James Gettys joined CRL in 1989. Jim's focus at CRL is multimedia audio and video systems. Before joining CRL, Jim spent two years at the Systems Research Center. Before that, he was a Digital engineer and visiting scientist at MIT working on Project Athena. He is one of the two principal designers and developers of the X Window System. Jim received an S.B. from MIT in 1978.

Andrew C. Payne joined CRL in 1992 after receiving a B.S. in Electrical Engineering from Cornell University. As a co-op at Digital in 1990, he helped build the first Alpha AXP chip. Andy's interests include signal processing, speech, and user interfaces.

Thomas M. Levergood joined CRL in 1990. Tom's focus is on speech and audio-related research. He is also involved in Alpha AXP system and software projects, including an experimental evaluation of split user/supervisor cache memories. He received his B.S. in Electrical Engineering in 1984 and M.S. in Electrical Engineering in 1993, both from Worcester Polytechnic Institute.

All of the authors can be reached at: Digital Equipment Corporation, Cambridge Research Lab, One Kendall Square, Bldg. 650, Cambridge, MA 02139, or by e-mail as {stewart, treese, jg, payne, tml}@crl.dec.com.