The Network Audio System

Make Your Applications Sing (As Well As Dance)!

Jim Fulton[†]
Greg Renda[‡]

Abstract

Audio input and output is rapidly becoming a standard feature in desktop devices and an expected element of user interfaces. Workstations, X terminals, and personal computers now typically include hardware capabilities ranging from 8 kHz mono "voice quality" up through 48 kHz stereo. Unfortunately, application programming interfaces vary widely across platforms, limiting the types of applications that can be developed in X environments.

The *Network Audio System*, commonly but unimaginatively known as *NetworkAudio*, was developed by NCD for playing, recording, and manipulating audio data over a network. Like the X Window System `[Scheifler92]`, it uses the client/server model to separate applications from the specific drivers that control audio input and output devices. A free, sample implementation of the NetworkAudio server and programming library is available for Sun and SGI platforms; additional product versions are available or have been announced for NCD X terminals, NCD PC-X servers, and SCO workstations. A number of commercial applications have also announced support, and the mailing list for NetworkAudio grows every day.

Introduction

Users of the X Window System are rapidly finding that they want more from their desktop than simply graphics. Driven by the wide-spread support for audio in personal computer applications,

---

[†] *Jim Fulton (**jim@ncd.com**) heads the Standards and Extensions group at NCD, where he focuses on future directions in desktop technologies, Low Bandwidth X, network audio, and user interfaces. He has worked with and participated in the design of the X Window System since its inception at MIT in 1984.*

[‡] *Greg Renda (**greg@ncd.com**) is a Senior Engineer at NCD, where he focuses on 2d graphics and network audio.*

X software developers are demanding audio capabilities from workstations, X terminals, and PC X server vendors.

Although audio hardware is available on many platforms, there is no common programming interface, forcing application developers to write new drivers for every software port. Worse yet, most of the interfaces that are available do not allow audio applications to be run over a network, preventing their use in distributed environments. In contrast, the Network Audio System uses the client/server model to split applications from specific hardware device drivers (see Figure 1):

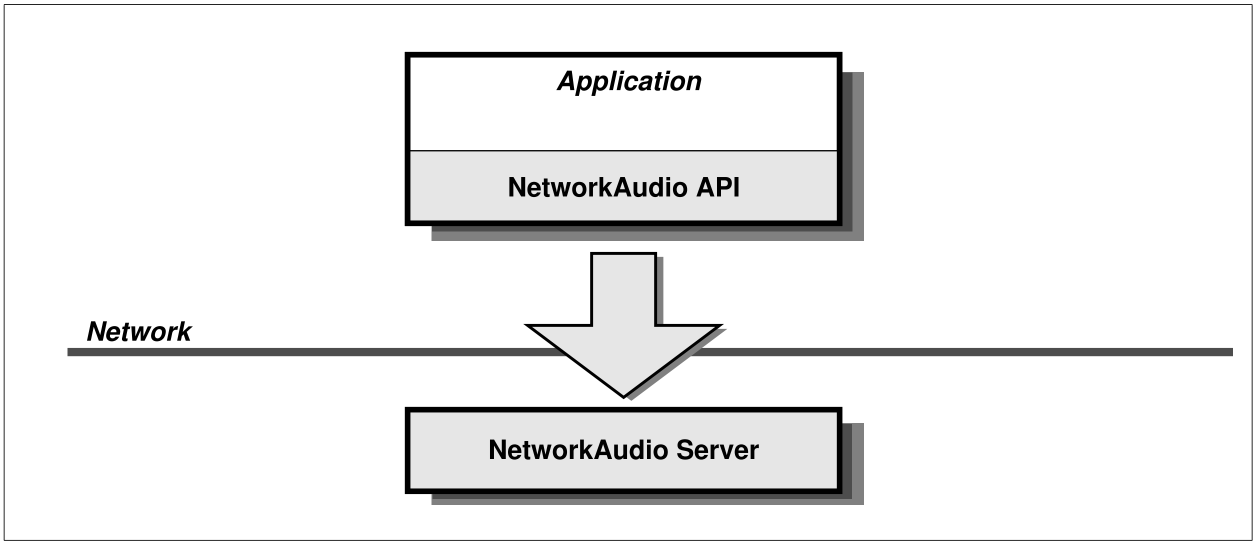

Figure 1: Architecture of the Network Audio System

Previous efforts have typically focused on providing a simple stream to the audio interface for one application at a time (Sun's */dev/audio*), simple storage of sounds `[Billman92]`, or had been proprietary `[Angebranndt91]`. The system most similar to NetworkAudio is *AudioFile* `[Levergood93]`, however it lacks server storage and conversion of data.

The Network Audio System allows easy use of audio over the network, with emphasis on:

- Sounds can be sent from an application to the server for playing on any attached output devices.
- If a microphone or other audio input device is attached to server, sound data can be recorded and transmitted back to an application.
- Sound data can be stored in the audio server for later use. This allows sounds to be repeatedly played without having to send them over the network every time.
- Multiple sources of sound data can be mixed and manipulated in a variety of ways (*e.g.*, made louder or softer, sampled more frequently or less).
- A variety of sound data formats are supported, with conversion performed automatically within the audio server.
- Simple function calls are provided to read and write many different file formats.

What Is Audio Data?

Physically, sounds are waves of compressed and expanded air. We are able to hear them because the bones of the inner ear vibrate when struck by the waves; these vibrations are interpreted by the brain as sound. Similarly, we speak by reversing the process, causing air to pass over our vocal cords which vibrate back and forth. Audio devices such as microphones and speakers work essentially the same way.

Conveniently, these vibrations occur along a single direction. They can be measured by noting the position of the ***diaphragm*** that is pushing or being pushed by the air (Figure 2). Microphones work by translating the position of such a diaphragm into electronic signals. When these signals are sent to a speaker, they cause another diaphragm to reproduce the original motion of the air, allowing us to hear the sound as it was produced.

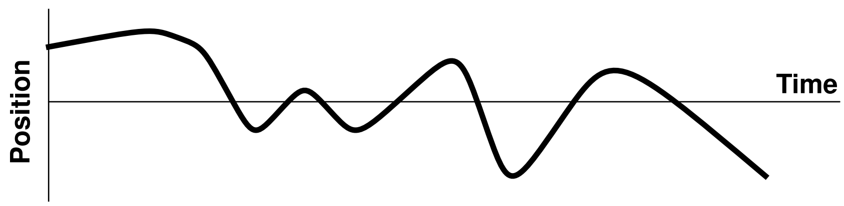

Figure 2: Continuous Graph of Audio Diaphragm Positions

Since the position of the microphone or speaker diaphragm varies smoothly with time, recording it exactly would require an infinite number of measurements. Digital systems such as computers and compact discs instead approximate the motion by taking ***samples*** of the positions (Figure 3). The number of positions recorded per second is called the ***sample rate***; higher rates result in a more accurate representation of the original sound, but take more space to store.

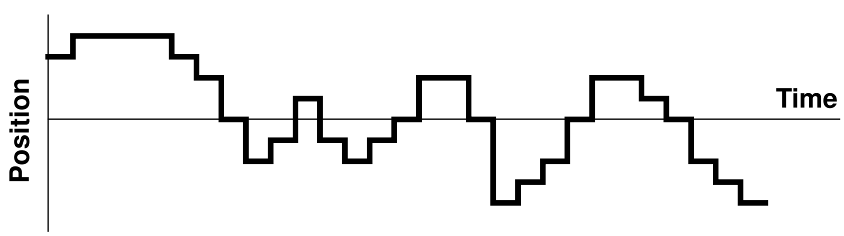

Figure 3: Sampled Graph of Audio Diaphragm Positions

These sequences of position samples are used by NetworkAudio to represent sounds. Playing a sound involves sending a sequence to a speaker; recording takes a sequence from a microphone or other input device and stores it into the server's memory.

Data Formats

The sequence of numbers that are used to encode audio data vary in value from -1.0 to +1.0, representing the two extreme positions that a diaphragm can reach in a given device. The value 0.0 corresponds to the center of the device. However, since floating point numbers occupy a large amount of space and are sometimes difficult to manipulate, most data formats store the sample values by multiplying them by 128 or 32768 and rounding to the closest integer.

The most common data formats are:

- ***Linear*** – Each 8- or 16-bit value contains a scaled sample; the larger version can more accurately represent the original data. Some variants store the signed values directly; others add 128 or 32768 to make them be unsigned. The 16-bit versions also come in most-significant byte first and least-significant byte first varieties.
- µ***LAW*** – Each 8-bit value contains the sign and the *logarithm* of a scaled sample. This enables a larger range of values to be represented in 8 bits by trading accuracy at the extreme positions (which aren't used very often and can tolerate small errors) for precision at the positions nearer to zero (which are used most of the time).

NetworkAudio supports both of these data formats and their variants. In addition, since different devices and applications use different data formats, NetworkAudio automatically converts among the various data formats whenever needed.

Audio Data Inputs and Outputs

The essence of NetworkAudio is that it allows applications to "wire up" one or more inputs to one or more outputs. Servers provide a variety of inputs and outputs, based on the available physical hardware or software emulators:

- ***Physical Devices*** – Most platforms support standard input and output jacks for attaching microphones, CD plays, speakers, tape recorders, etc.
- ***Virtual Devices*** – The audio server itself provides software that emulates hardware tone generators and radio transmitters and receivers (for data that is broadcast over a local area network).
- ***Client Data*** – Applications can send data to and from the server over the network. By splitting the transmissions into pieces, applications can send or receive an unlimited amount of data.
- ***Buckets*** – Audio data can be stored in the server, either as a result of being recorded from another device or having been sent over the network from the application. This allows the data to be played multiple times without having to be retransmitted or rerecorded.

Outputs can receive data from more than one input at once; the server automatically merges the data streams so that the sounds are heard together. Operations for mixing sounds, extracting individual tracks of data, and changing the volume are provided.

Playing and Recording

The instructions that a client provides for arranging various inputs and outputs are called ***Flows*** (see Figure 4), similar to the photoflow pipelines of the X Image Extension (see `[Shelley93]`). They are used in much the same way that musicians use audio mixers or patch-panels. They indicate how the components should be connected, how multiple sounds should be mixed, and what changes in volume should be made.

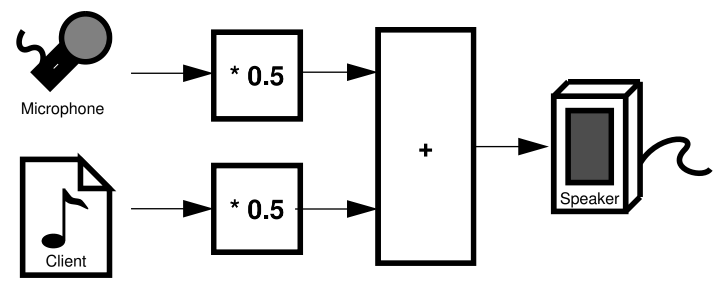

Figure 4: Flows Connect Inputs to Outputs

A flow is made up of a series of ***elements*** which describe sources of input data, operations to perform on data, and places to output the data. A flow is often represented graphically as a tree whose root and leaf nodes are input and output elements and whose interior nodes are mathematical or logical operations.

The input and output elements used in a flow are referred to as the ***components*** of the flow. Data moves along a flow at the highest sample rate used by its components. The server automatically converts data to this sample rate when necessary. At each step, the components in the flow are said to be in one of three ***states***, depending upon whether or not they are producing or consuming any real data:

- ***Stopped*** – An input element in this state has no real data ready and simply emits values of zero. An output element in this state discards any data sent to it.
- ***Paused*** – When an input element temporarily runs out of data or an output element has reached its limit, it becomes Paused. This is similar to being Stopped, except that the component expects that it can resume where it left off when more data or space becomes available.
- ***Started*** – When input and elements are producing or consuming data, respectively, they are said to be started.

When a flow is created, the components are initially placed in the Stopped state, preventing any data from flowing. Once at least one component in the flow has been started, data can begin to move.

Components may change state in several different ways:

- The application can explicitly set the state of each element.
- An element in the flow can set the state of itself or other elements in response to changes in its own state.
- If an input temporarily runs out of data, it becomes Paused.
- In an input permanently runs out of data, it becomes Stopped.

By default, all components in a flow will automatically be stopped if any of the elements in the flow become Stopped or Paused (and have not reset themselves).

Automatic Actions

NetworkAudio provides applications with the ability to force input and output elements to react automatically in response to changes in other elements. This can be used to chain flows together or to stop multiple flows (to prevent pops and hisses that are caused by data temporarily not being ready).

Each component element in a flow may have associated with it a list of ***actions*** to be performed when the element is Paused, Stopped, or Started:

- ***Change Element State*** – The state of any element (including the one triggering the change) can be explicitly set.
- ***Notify Client*** – An event detailing the current state of the element can be sent to the client that created the flow containing the triggering element.

Applications frequently request Stopped Notices to know when a flow is finished and can be destroyed.

Manipulating Audio Data

One of the key features provided by NetworkAudio is the ability to manipulate multiple sources of and destinations for sound data without client interaction. Conveniently, simple mathematical operations on the sample values (such as those shown back in Figure 3) provide useful physical results:

- Adding two sets of samples together is equivalent to playing them at the same time.
- Multiplying a set of samples by a value greater than 1.0 increases the volume of the sound; a value less than 1.0 decreases the volume.

Sound data comes from inputs in the range of -1.0 to +1.0. Within a flow, values may temporarily exceed this range, but are clipped back down when sent to an output. Constants used by elements (such as multiplication, addition, wave form frequencies, etc.) may be modified at any time. The structure of the flow cannot be edited; instead, the whole flow must be redefined.

All inputs and outputs may be shared among the applications. When two or more flows direct sound data to the same output, the server automatically merges the data streams together so that both sounds come out at the same time. If the various components in an flow contain data of differing sample rates, the server internally converts all of the data streams to the highest rate. If this results in more samples for an output than it expected, the server automatically throws away any unneeded values.

Stereo

In addition to supporting simple monaural data (*e.g.*, voice), NetworkAudio also provides the ability to manipulate multitrack data such as stereo recordings. Applications can dynamically extract tracks from sound data as well as mix them together.

Some types of multitrack output devices (especially stereo speakers) can be represented as collections of single-track devices that applications may also wish to access separately. Figure 5 shows an example of stereo data sent from two separate clients actually being played on two mono speakers (one of which is receiving data directly from the third client).

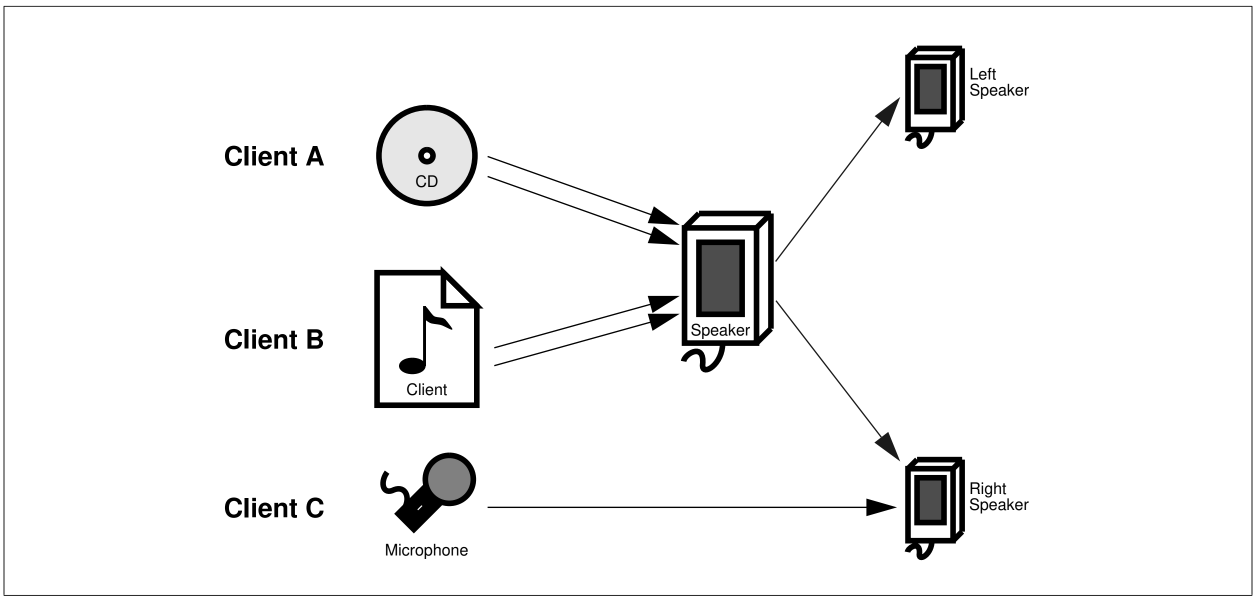

Figure 5: Stereo and Mono Outputs

In this situation, the two sets of stereo data are mixed as if the stereo speaker were a physical device. The individual tracks are then sent to the mono speakers, where the data for the right track is mixed with the data coming from the microphone.

By providing access to the separate mono speakers as well as the joint stereo speaker, NetworkAudio gives applications fine-grained control over how sound is produced. For convenience, however, most implementations provide at least one mono and one stereo speaker, even if they have to be simulated in software when appropriate hardware is not available.

Sending Audio Data Over The Network

Although NetworkAudio provides a number of input devices and built-in sound generators, its primary purpose is to allow applications to play and record audio data over the network.

Clients send data to a special type of input element in a flow. Data from this element is used in the flow just as if it were coming from any other input device. If a sound is to be used more than once, the application can store it in the server by creating a ***bucket*** object and directing the data to it instead of to an output device.

This process can be reversed to retrieve audio data from the server. Large amounts of data can be passed back to the application directly from the various input devices without requiring temporary storage for all of it in the server. Although the most common use of this mechanism is to record data from microphone, it can be used with all supported input devices (*e.g.*, buckets, tone generators, or radios).

Figure 6 shows several examples of very simple flows that transfer data to and from the server. Putting a multiplier immediately before a speaker output allows the volume of the data to be adjusted dynamically.

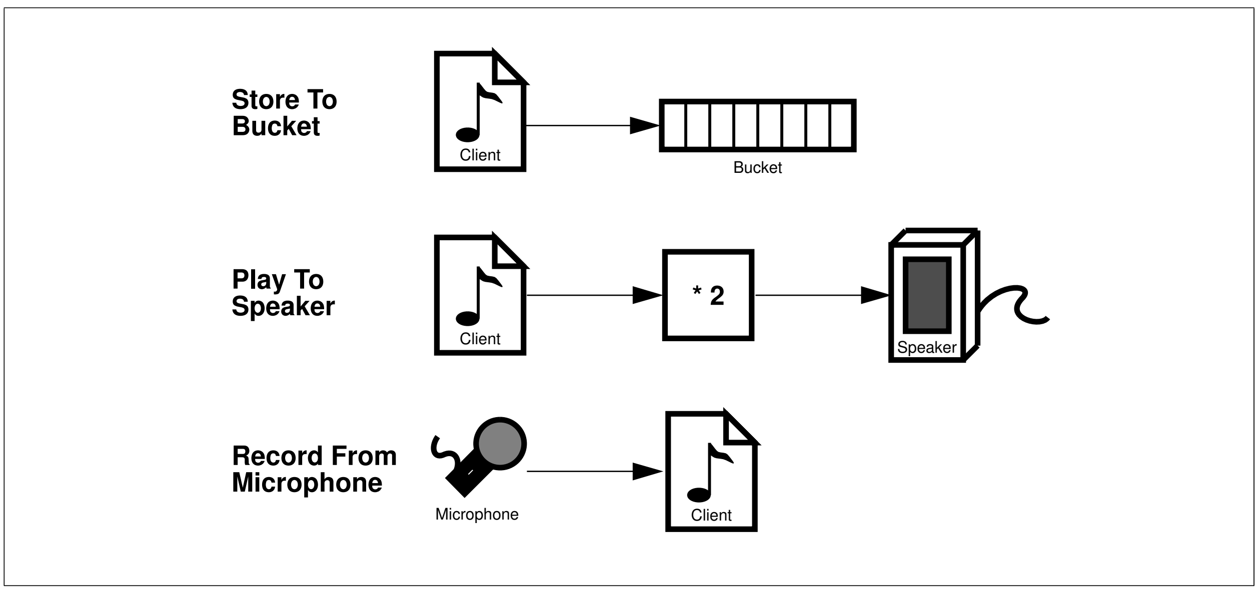

Figure 6: Common Uses For Client Data

In a client/server architecture, network transfer delays can sometimes make the arrival of data less predictable than if it were coming from a physical device. This can result in underruns (data not arriving in time) or overruns (more data arriving than there is room for) if the delays are sufficiently large. If an underrun or overrun occurs, the affected element is "Paused" until more data or space becomes available. To avoid pauses, applications can control the amount of data that is kept for each input and output element and can request notices whenever an input begins to run out of data or an output has to buffer up too much data. These notices (known as ***watermarks***) allow the application to balance server memory use with frequency of data transfer.

Storing Data In The Server

Applications that use certain sounds more than once can save time and network bandwidth by storing the data in the server in objects called ***buckets***. Such data can then be used repeatedly as input to other flows without have to transfer it across the network again. This technique is commonly used in applications that use lots of audible cues to draw the user's attention.

Buckets can be shared among all applications. This allows a control program, such as a desktop manager, to store commonly-used sounds for cooperating applications to all use. Some server implementations even build in prerecorded sounds to save applications start-up time.

Virtual Input Devices

As a convenience to the application, several types of simulated input devices are provided by NetworkAudio. A variety of built-in ***wave forms*** (see Figure 7) can used to generate simple tones (whose frequencies can be varied dynamically). These have been used to implement telephone dialers as well as fairly sophisticated sound effects.

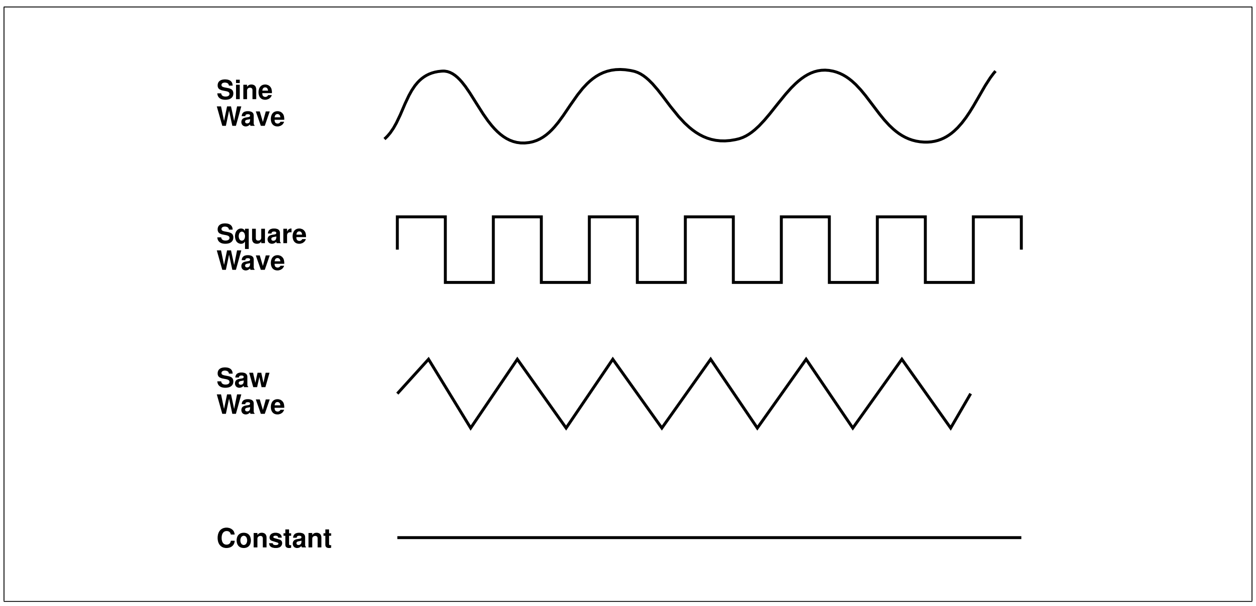

Figure 7: Sample Built-in Wave Forms

Future implementations will also provide ***radio*** devices that read from or write to the network directly. When used with a datagram protocol, these will allow true, efficient broadcasting of audio to multiple desktops from a single source. Additionally, they could provide direct server-to-server transmission for teleconferencing that does not require transmitting the data through the application first.

Implementation

Like X, the Network Audio System consists of a server which drives the hardware and an programming interface library against which client applications are linked (see Figure 8).

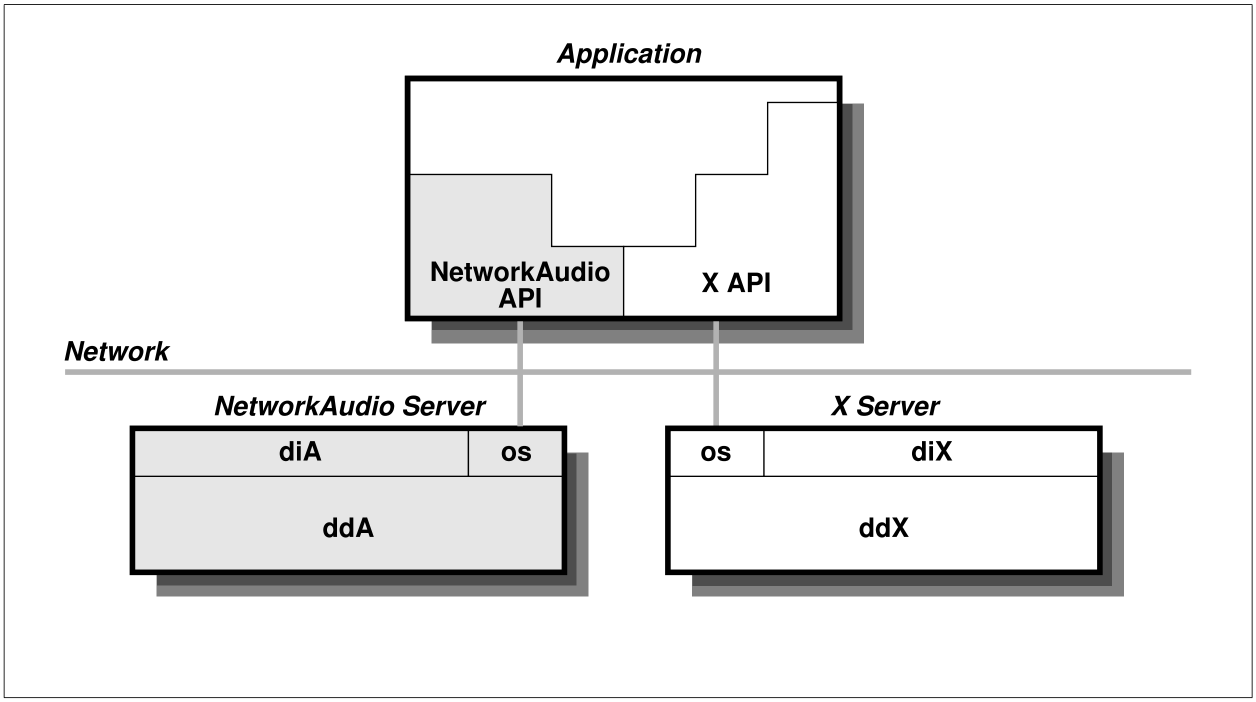

Figure 8: NetworkAudio Architecture Is Similar To X

Server

The NetworkAudio server consists of a high-level, device-independent portion (*diA* and *os*) and a low-level, device-dependent portion (*ddA*). The portable *diA* code handles parsing of the wire protocol, allocation and management of resources, conversion of data formats and sample rates, generation of tones from virtual devices implemented in software, and compilation of audio data Flows.

The *ddA* code is made up of drivers for the particular devices to be controlled by the server. It is actually quite small, consisting of interfaces for performing the following operations:

- Initializing the various devices
- Setting device attributes such as gain, line mode, and sample rate.
- Enabling and disabling hardware interrupts.
- Reading from and writing to hardware.

Several *ddA* layers have been written, ranging in complexity from direct hardware interfaces (*e.g.*, simple FIFO-driven devices) to software-only versions using vendor-supplied interface libraries and device drivers. New implementations seem to take between two days and two weeks to write.

Flows

One of the more distinguishing features of NetworkAudio is the way in which applications specify the movement of data from inputs, through operators, to outputs using tree-structured set of "wiring instructions" known as ***flows***. The application transmits this information to the server in the form of a list of elements, each representing one node in the tree. Each node lists the other elements from which it should receive data. This "backwards chaining" is used by the server to reconstruct the original tree.

For example, Figure 9 shows a flow that could be used to dub in voice on top of a CD. The voice data is converted from mono to stereo and then is merged with the CD data. The result is then sent to a stereo speaker for playing.

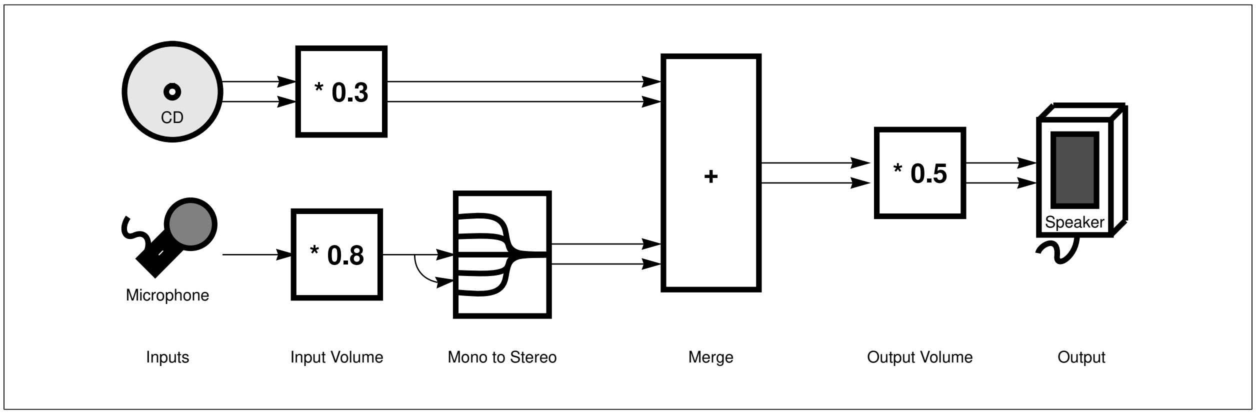

Figure 9: Dubbing Voice on Top of Stereo CD

For each output in the flow, the server works backwards through the mathematical operations in the flow to compute the amount that each input value will contribute to that output. The result is a multiplier and offset constant for each track of the contributing inputs. In the example shown above, each track of the speaker output can be represented by the formula:

`((0.3 * CD) + (0.8 * MIC)) * 0.5`

which, when simplified, yields:

`(0.15 * CD) + (0.4 * MIC)`

A nice property of this approach is that it is works for complex flows as well as simple ones, allowing sophisticated applications to run as efficiently as simple ones.

Master Flow

A key design goal of NetworkAudio was to allow multiple applications to operate simultaneously. It was unacceptable to require that only one flow be given access to each device at any one moment. Instead, all active flows are merged together into what is called the "master flow." If an output is used by more than one flow, all of the values from the individual flows are added (or, optionally, averaged to prevent clipping) together.

The actual generation of audio data is done by an interrupt handler that is called from either a hardware interrupt or an interval timer, depending upon the implementation. Multiple values are computed at a time to reduce the frequency of interrupts and increase the performance of the system. Figure 10 summarizes the stages through which data moves:

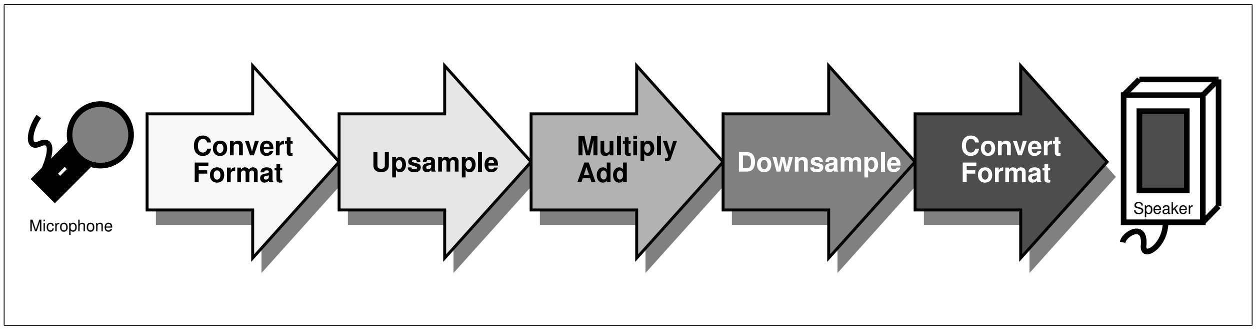

Figure 10: Data Within A Flow

When data is read from an input, it is automatically converted into the server's internal data format and up-sampled to match the highest sample rate used by any of the active flows. The computed output values are then down-sampled and converted to match the sample rates and data formats requested by the output.

Library

The NetworkAudio library contains two pieces. At the lowest level is a collection of "protocol wrapper" routines which are used to communicate with the server. This layer is equivalent to Xlib in that it tends only be used by higher level "toolkits" or sophisticated applications. One difference between Xlib and the NetworkAudio library is that the low-level routines in NetworkAudio all have an argument which indicates whether to run asynchronously (which is faster, but which requires error handlers to deal with problems) or synchronously (in which case any error is returned to the caller immediately). This can greatly simplify error checking within an application.

In addition to the low level routines, a set of high-level utilities are provided to make common NetworkAudio tasks quick and easy. In particular, simple functions are provided for reading from and writing to audio files, storing sound buckets in the server, playing stored sounds, managing audio events, and interacting with X programs (NetworkAudio can hook right into an Xt/Motif main loop).

One of the especially nice features of the NetworkAudio library is its ability to read and write audio data in a variety of file formats transparently to the application. This allows programs to mix and match sounds created from most of the other major audio platforms:

- ***.snd***, ***.au*** – Sun and Next hosts.
- ***.voc*** – SoundBlaster cards on IBM PCs and compatibles.
- ***.wav*** – MS-DOS and Microsoft Windows.
- ***.aiff*** – Apple and SGI computers.

Example Programs

The sample implementation of NetworkAudio contains a variety of example programs, including a sound file "chooser," an audio data editor, a translator for converting among the various file and data formats, a telephone dial-tone generator and recognizer, and a simple ***audiotool***-like utility for use with OpenWindows Deskset applications. A growing number of games and commercial software are being shipped with NetworkAudio support built in.

Future Directions

Although NetworkAudio is rapidly gaining wide-spread use for simple-to-moderate audio applications, it still needs additional support for tight synchronization with graphics for high-end video conferencing. Some mixture of the model used by AudioFile and the primitives provided by the proposed X Synchronization extension [Glauert93] will probably be added to NetworkAudio shortly.

The Network Audio System was designed to be extensible at several levels. New data formats and device types can be added to the server without requiring changes to applications; similarly, new file formats can be added to the library transparently. As audio becomes more pervasive, both of these elements will be tested by the addition of support for compressed data streams and MPEG. Finally, virtual devices implemented in software, such as the radios mentioned before, should provide interesting avenues of exploration.

Summary

NetworkAudio provides a powerful, yet simple, mechanism for transferring audio data between applications and desktop X terminals, PCs, and workstations. Applications specify how various inputs and outputs should be hooked together; the server automatically routes data among the components, converting sample rates and data formats as needed.

Sounds may be stored in the server and reused multiple times or can be sent directly to attached output devices such as speakers. Applications may dynamically adjust the volume at which the sounds are heard. Input devices such as microphones can be used to record audio data. Applications can read the data back over the network, store the results in the server for later use, or even redirect it to an output device.

With the Network Audio System, X developers can combine the audio support needed for today's applications with the power and flexibility of networked computing needed for tomorrow's.

Where To Get Source

Source code for the sample implementation of the Network Audio System is available for anonymous ftp from *ftp.x.org* in the directory */contrib/audio/nas/*It uses the standard unrestrictive copyright used by the X Consortium; vendors and application developers are welcome and encouraged to incorporate it into products.

References

[Angebranndt91] Susan Angebranndt et al., "Integrating Audio and Telephony in a Distributed Workstation Environment," *USENIX Proceedings*, Summer 1991.

[Billman92] Richard Billman et al., "Workstation Audio in the X Environment," *The X Resource*, Issue 4, O'Reilly and Associates, 1992.

[Glauert93] Tim Glauert et al., *X Synchronization Extension*, X Consortium draft standard, 1993.

[Levergood93] Thomas M. Levergood et al., *AudioFile: A Network-Transparent System for Distributed Audio Applications*, Digital Equipment Corporation Cambridge Research Laboratory, 1993.

[Scheifler92] Robert W. Scheifler and James Gettys, *X Window System*, Digital Press, 1992.

[Shelley93] Robert N. C. Shelley et al., *X Image Extension Protocol Reference Manual, Version 4.12*, X Consortium Draft Standard, Special Issue C, O'Reilly & Associates, Inc., January 1993.