Diplomarbeit im Studiengang Medieninformatik

Audiospace: A universal service for interactive rooms

vorgelegt von	Stefan Werner
an der	Fachhochschule Stuttgart Hochschule der Medien
am	27. Feb. 2004
1. Prüfer:	Prof. Dr. Walter Kriha
1. Prüfer:	Prof. Dr. Jan Borchers

Eidesstattliche Versicherung

Ich versichere an Eides statt durch meine Unterschrift, dass ich die vorstehende Arbeit selbständig und ohne fremde Hilfe angefertigt und alle Stellen, die ich wörtlich oder annähernd wörtlich aus Veröffentlichungen entnommen habe, als solche kenntlich gemacht habe, mich auch keiner anderen als der angegebenen Literatur oder sonstiger Hilfsmittel bedient habe.

Stuttgart, den 27. Februar 2004

Stefan Werner

Abstract

The AudioSpace project provides a universal audio service for interactive rooms. It allows the sharing of multichannel high quality audio over high-speed local wired and wireless IP networks. Other than existing networked audio solutions, the AudioSpace was designed for the low latencies required by interactive applications while being fully compatible with legacy software. This paper describes a Mac OS X-based implementation that integrates with the host platform as an audio device and makes extensive use of the CoreAudio framework. By using system-standard user interfaces and automatic network configuration, the system can be used by unexperienced users. In addition, this thesis introduces a novel approach for clock skew compensation in low-latency applications.

Contents

Chapter 1

Introduction

1.1 Goal of this thesis

The goal of this thesis is to design and implement an audio service for use in the "Media Space" interactive room at RWTH Aachen. The service will allow any application on any computer in the room to access the multi-channel audio interface and speakers that are connected to a central server. The clients will be able to independently choose the speakers they want their sound streams to play through and the overall latency of the system will be low enough to have no or minimal impact on interactive applications. The system will work over Ethernet and allow guest computers to also use the speaker system over wireless Ethernet. In the following, this system will be referred to as the *AudioSpace*.

1.2 Motivation

The network services available on today's operating systems provide transparent sharing of disk space and printers: When a computer is set to share a hard drive partition or a printer on the network, users of other computers in the same network will be able to use the hard drive or the printer as if they were connected directly to their respective computers. This integration is usually provided on system level so that application developers do not need to explicitly equip their applications with these capabilities. Instead, the operating system is providing abstract file access and printing services to the applications and will take care of the networking by itself.

Unfortunately, the development of such transparent network services did not keep up with the rapid development that happened in the computer industry over the last few years: Multichannel audio and high-quality video are not niche applications any more but are widely being used by home users on commodity hardware. Networking bandwidth has also increased, hardly any computer today ships without a Fast Ethernet port for local networks and for connections to broadband Internet services over cable modems or DSL lines. Yet, despite all these developments, only a few specialized applications are able to transfer audio and video streams over networks, often in highly compressed format with in low fidelity and high latencies. Transparent services for sharing audio or video devices, comparable to the printing services described before, are not available.

1.3 Overview

In the following chapter, the detailed requirements of the AudioSpace will be described. In chapter 3, a few basic concepts about networking, audio and Mac OS X will be explained. Chapter 4 takes a look at related previous work, where chapter 5 and 6 document the design and implementation of the AudioSpace software. Chapter 7 analyzes the results from testing the AudioSpace and takes a look at possible future work. Appendix A describes a previously undocumented method for user space audio drivers in Mac OS X. Appendix B describes a new approach for clock skew compensation that was developed for the AudioSpace. Appendix C provides a walk-through of a sample use of the AudioSpace system.

Chapter 2

Requirements

This chapter will outline the specific requirements that the AudioSpace system will have to fulfill.

2.1 Environment

2.1.1 Interactive Rooms

Interactive Rooms like the Stanford iRoom[1] or the Stockholm University's iLounge[2] are being used to explore new possibilities for people to work in environments of ubiquitous computing. Equipped with large touch-screen displays, wireless input devices, cameras and microphones, they provide a testbed for experiments with post-desktop user interfaces. The use of wireless networking allows users to bring personal devices like notebooks or PDAs and use them to interact with the room. In order to adapt to different needs and to test various ideas, they must be easily reconfigurable and must not rely on a static layout. New frameworks provide a platform for developing software, hardware and user interfaces for use in interactive rooms.[1]

2.1.2 The Media Space

The main focus of the Media Space interactive room at the Media Computing group of the RWTH Aachen[3] will be on the research of interaction with non-static time-constrained media like audio and video. The computing back-end will be a number of Apple PowerMac dual-G5 computers running Mac OS X version 10.3. The computers will be connected through a switched GBit Ethernet, and an additional wireless Ethernet will be available in the Media Space room. The user will mostly be interacting with large touch screen displays and multichannel surround sound speakers (8 satellite speakers and one subwoofer). Each of the displays will be connected to its own computer, where the speaker system will be connected to a central audio server. User input will happen through

[1]http://iwork.stanford.edu/
[2]http://www.dsv.su.se/fuse/
[3]http://media.informatik.rwth-aachen.de/

interfaces like gesture or speech recognition or the iStuff[4] framework.[2]

2.2 Application compatibility

The AudioSpace system will be used to play all kinds of audio streams from any computer in the Media Space room through the speakers connected to the audio server. Many of the applications running on these computers will not be specifically written for the AudioSpace but be “off the shelf” software that expects regular audio hardware. Still, these applications should be able to use the AudioSpace without modifications.

Many applications are not prepared for multichannel audio but only process stereo signals. Still, stereo applications should not be restricted to certain speaker pair but be able to use any two out of the nine speakers in the Media Space.

2.3 Usability

Users that are familiar with Mac OS X should be able to use AudioSpace clients with little or no instructions. Therefore, the AudioSpace software should use the system’s usual controls for audio hardware wherever possible, and applications and the operating system should present the AudioSpace to the user as if it were a regular audio device. After setting up a computer to use the AudioSpace for sound output, the applications should behave as if they were using local sound hardware.

The installation process of the AudioSpace software should not require special skills and anyone who has installed other application software on a Mac OS X computer before should be able to install and use the AudioSpace software.

2.4 Latency

In interactive applications, a high latency can ruin the user experience. The system’s feedback on a user’s action must follow within a 0.25 seconds[3] to be recognized as being caused by that action (some sources like [4] mention 0.1 seconds), otherwise they irritate the user. In musical applications, even lower latencies are required: Sometimes response times exceeding 0.01 seconds are already being considered unacceptable[5]. Note that most desktop operating systems are unable to provide such latencies in their standard configurations: Using the built-in system APIs of Windows or Mac OS 9, audio latencies are often in the range of 0.06 to 0.2 seconds[6].

[4]For information about iStuff, see Ballagas, Ringel, Stone, Borchers: *iStuff: A Physical User Interface Toolkit for Ubiquitous Computing Environments*, Proceedings of CHI 2003, p. 537-544.

2.5 Scenarios

To further illustrate the requirements of the AudioSpace, two hypothetical use case scenarios will be described, along with a few requirements that these scenarios imply. These are certainly not meant to be the only possible applications - as a system designed for use in a research facility, the AudioSpace is likely to be used in a variety of experimental scenarios that cannot be foreseen yet.

2.5.1 DVD playback

Scenario: A user wants to watch a DVD with a multichannel audio track on a computer in the Media Space. She wants it to show on the screen connected to that computer, with the front channels of the audio track playing from the speakers to the left and to the right of that screen, and the rear channels through the speakers on the opposite wall. After she realizes that the sun is glaring on that screen, she decides to use a different computer, one on the opposite side of the room. Accordingly, she wants the front and rear speakers to be swapped now, to conform the new situation.

Requirements: The audio server must handle multiple channels. The software must allow the user to route sounds to specific speakers without any cabling and without leaving the computer she's working on. The system must be able to play audio streams fast enough to be in sync with the video. It must be compatible with the DVD playing software.

2.5.2 Musical performance

Scenario: Two musicians are performing in the Media Space. Each of them brings his own computer with a MIDI keyboard connected to it. Both use music software they wrote themselves and perform using two regular channels and a low frequency effect channel. The regular channels play through separate speakers for each computer, the low frequency channels play both simultaneously through the Media Space's subwoofer.

Requirements: Multiple clients must be able to use the AudioSpace simultaneously. The AudioSpace software must be easy to install on guest computers in the Media Space. Speakers must be shareable between clients. The latency must be low enough for musical performances. The AudioSpace must be compatible with arbitrary software.

Chapter 3

Basics

Due to the interdisciplinary nature of the subject, the reader may not be familiar with the fundamentals of every aspect of it. Therefore, the following chapter explains the basics of the areas involved as far as necessary.

3.1 Digital Audio

3.1.1 Sampling

The core concept of handling audio signals in a computer is *sampling*, that is, converting continuous analog signals into discrete time-sampled signals, called *samples*. The Nyquist-Shannon sampling theorem states that when converting analog to digital signals, the sampling frequency must be greater than twice the highest frequency of the input signal in order to be able to reconstruct the original perfectly from the sampled version.[7][8] The sampling frequency is also often referred to as the *sample rate*. Two standard sampling frequencies were established in 1985 by the Audio Engineering Society, 44.1kHz and 48kHz. The frequencies were chosen with respect to the highest frequency audible by the human ear, which is at about 20kHz. Common technologies using that sample rates are the Compact Disc (CD) at 44.1kHz and the Digital Audio Tape (DAT) at 48kHz[9]. Common audio hardware for computers works also at these sample rates. Higher frequencies, up to 192kHz, are being used in recording studios in order to have more headroom when processing signals, but have close to no significance outside recording studios. Playback of a recording at a different sample rate than it was recorded will change both its duration and pitch.

The information stored within a single sample depends on the number of bits that are used to store a sample, the *bit depth*. The standard bit depth for CD, DAT and most computer hardware is 16 Bits, representing values ranging from 0 to $2^{16} - 1$, resulting in a dynamic range (the difference between the highest and the lowest level that can be reproduced) of 96 dB. Equipment in recording studios can digitize at sample rates of up to 24 Bits (144 dB), which is also the bit depth used for DVDs. In software processing of sampled audio data, it is not uncommon to store samples in 32 bit floating point numbers for additional precision.

A continuous series of samples is referred to as an audio stream, in its raw form sometimes called *PCM* (pulse code modulation). When multiple sources are combined in one stream, these are called channels. Stereo signals have two channels, left and right, where surround sound systems in movie or home theaters have up to eight separate channels. The samples of all the channels in a stream at a certain sampling point are called a *frame*.

The *data rate* of an audio stream calculates as $datarate = samplerate * bitdepth * channels$ A compact disc thus has a data rate of $44100Hz * 16Bits * 2 = 176400\frac{Bytes}{second}$, resulting in a total data of over 600 MB for one hour of audio. The hardware components that are used for conversion from analog sound signals to digital samples are called D/A converters, the components that construct analog signals from digital data are A/D converters.

3.1.2 Compression

In order to reduce the memory and bandwidth requirements of digital audio, several compression methods were developed. The process of compressing the signal is called encoding, the reverse is called decoding. Software components that perform the processes of encoding and decoding are called CODECs. Compression algorithms are divided in two categories: Lossless and lossy compression. Losslessly compressed signals restore to the exact same data when decoded, where lossy compressions do not reproduce the exact same data. As a consequence, lossy compressions are able to reduce the amount of data much more than lossless compressions but can have audible artifacts. Most lossy compression algorithms are also much more complex than lossless algorithms and their software implementations require a lot more resources from the computer's CPU.[10]

3.1.3 Audio handling in computers

Most computers systems follow a layered approach: At the bottom, there are hardware drivers that control the audio hardware of the computers, above that is a system framework providing basic mixing services and on top are the applications that talk to the framework. These layers pass their data in buffers, a group of consecutive sample frames. Larger buffer sizes require less CPU, but have a higher latency. Smaller buffer sizes lower the audio latency but have a higher CPU overhead and too small buffers can result in skipping, when the operating system's scheduler is not switching to the audio handling processes in time.

There are two ways of storing multiple channels in one buffer (Figure 3.1): One is concatenating multiple buffers of single channels to one large buffer, the other is interleaving the channels' samples in the buffer.

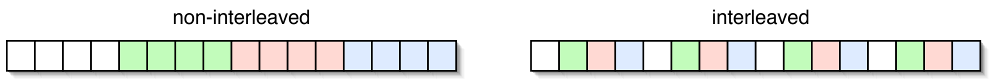


Figure 3.1: Multiple channels in one buffer

Most audio hardware in computers has A/D and D/A converters for stereo signal, with a trend going towards multiple output channels. Often they also include analog amplification that allows the user to connect microphones directly to the computer. The converters in computer hardware have their own clock source for the sample rate, usually a quartz oscillator. Therefore, they can only run at one or several fixed sample rates but not at arbitrary sample rates. In order to play audio that has a different sample rate than the hardware, sample rate conversion or *resampling* is required. Converting from one sample rate to another requires inserting or removing samples to keep the duration of the sound constant. A non-interpolating algorithm that just skips or duplicates single samples is destroying the sample's continuity and leaving audible artifacts. A huge gain in quality can already be had with linear interpolation where a new sample is being created by placing it between two existing samples and where samples are dropped by replacing two samples with a sample between them. Still, linear interpolation has a noticeable quality loss which is why many resampling implementations use higher order polynomial or trigonometric functions for interpolation. The communication between the sound hardware and the system software happens over a shared buffer and an interrupt. The audio driver is writing samples to the buffer, the sound hardware is reading samples from the buffer. The sound hardware is notifying the system via an interrupt every time it needs a new chunk of data. If the audio driver is not able to fill that buffer in time, dropouts in the audio signal occur.

3.2 Networking

The standard method for describing networks is the using a layer model. It is describing networks in a stack of layers, where each layer is building on defined services provided by the layer below it and providing defined services to the layer above it (see figure 3.2). The services a layer is providing to the layer above it are specified in an interface. Layer n on node A communicates with layer n on node B using a defined protocol. By defining communication between layers in a standard interface, the actual implementations of these layers are exchangeable without affecting the other layers - for example, changing from a dial-up modem to a DSL connection does not require any changes to the web browser.

Usually, a network protocol is not sending infinitely large blocks of data but is dividing it in *packets*, consisting of a protocol-specific *header* part that contains metadata like size, source and destination of the packet and a *payload* part that contains the actual data the packet is transporting.

The standard networking technology for Mac OS X is TCP/IP over Ethernet. Figure 3.2 shows a schematic overview of the layers in such a network. The TCP/IP reference model which will be used here has using four layers: On the bottom is the host-to-network layer, above of it is the network layer, then the transport layer and on top is the application layer. The host-to-network layer connects TCP/IP to the physical network, the network layer is mainly providing routing services between hosts and the transport layer is providing interfaces to the applications running on these hosts. The standard for the network layer is IP, the transport layer can be either TCP or UDP.

The TCP/IP model is a practical simplification of the more detailed OSI reference model which is using seven layers.

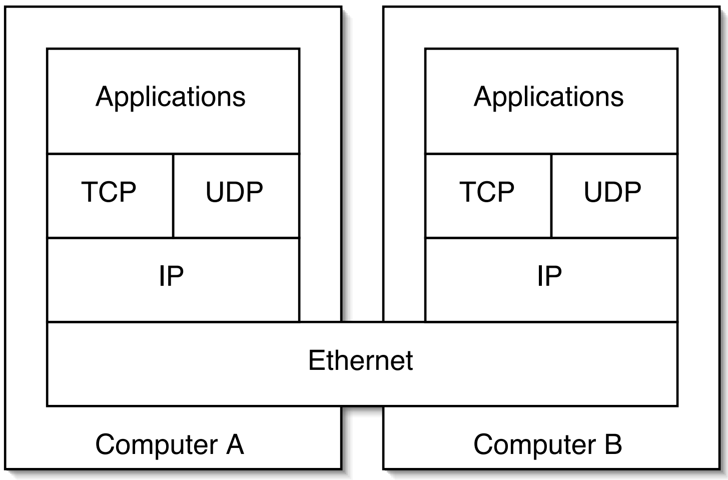


Figure 3.2: A typical Ethernet-based LAN

3.2.1 IEEE 802.3 Ethernet

Ethernet is the most widespread hardware standard for local networks. Ethernet adapters and cables are cheap, and hardly any computer today ships without an integrated Ethernet interface.

Ethernet is based on a shared medium that all stations are directly connected to. The stations have unique MAC (media access control) addresses that usually are stored in the interface's ROM. Characteristic for Ethernet is the CSMA/CD (Carrier Sense Multiple Access/Collision Detection) algorithm by which the stations decide who is allowed to send when: A station that intends to send begins by listening on the media if any other station is currently sending. If the media is free, the station starts sending while still listening at the same time. If stations start sending at the same time, their signals will overlap, resulting in unidentifiable garbage - a *collision*. When a station detects a collision, it starts sending a special collision signal that causes all sending stations to stop sending, wait for a random period and then to try to resend. If a station is unable to send after 16 tries, the packet is being dropped.

The CSMA/CD algorithm is a reason why shared media Ethernet networks can't reach their nominal maximum throughput: The number of collisions is increasing with the number of stations and the amount of traffic, and a lot of bandwidth gets wasted by colliding signals. A solution for that problem was provided with the invention of the switch device which doesn't connect all the stations to one wire but establishes direct connections between two stations when data is being sent between them, virtually eliminating collisions. While the nominal speed of an Ethernet network does not increase when the central hub is replaced with a switch, every station has the full bandwidth available and does not have to share it with all the other stations.

Ethernet is specified for a variety of speeds and media. The ones used in currently shipping Macintosh computers are Fast Ethernet and Gigabit Ethernet for Cat5 twisted pair wire cables.[11]

Fast Ethernet

Fast Ethernet is the extension of CSMA/CD and the Ethernet packet format to 100MBit/s, of which, due to protocol overhead and design limitations, only up to 60MBit/s are usable as payload[11]. In terms of audio streaming, this bandwidth translates to 70 channels at 24Bit/44.1kHz.[17]

Gigabit Ethernet

Gigabit Ethernet was designed to extend the existing Fast Ethernet to a nominal speed of 1000MBit using the same Cat5 cables. Again, the protocol overhead makes not the full bandwidth available to the stations.

Wireless Ethernet

Wireless Ethernet as defined in IEEE 802.11 is a technology for local networks on radio waves. In contrast to wired Ethernet, a wireless interface is not able to listen for collisions while it's sending and therefore cannot use the CSMA/CD protocol. A different procedure, CSMA/CA (CA = Collision Avoidance) is being used. Collision avoidance stands for the practice that a station that detected existing signals on the media will wait for a random time until it tries to send. Since a station is unable to detect collisions while sending, the sending station is sending an announcement prior to the actual transfer and waits for an acknowledge message from the receiver after the transfer. This adds to additional overhead of the protocol.

The most common standards for wireless Ethernet are 802.11b (11MBit/s) and 802.11g (54MBit/s) that both operate at a carrier frequency of 2.4GHz. The 802.11a standard that transmits 54MBit/s at 5GHz was available before 802.11g appeared, but was not as successful as it was not directly compatible to the already widespread 802.11b.

Since radio waves go through walls, wireless Ethernets have no physical access restrictions like wired networks (if you can't get to the cable, you can't get to the network). Most wireless adapters understand the WEP encryption scheme which does not live up to its promises (WEP stands for "wired equivalent privacy") as a few design flaws make it very vulnerable to attacks. Apple is selling interfaces on the 802.11b and 802.11g standards under the brand names AirPort and AirPort Extreme.

3.2.2 IP networking

The technologies for local networking explained in the previous section are usually taken care of on operating system level. Hardly ever does an application programmer need to implement these protocols herself. In practice, network applications do not touch Ethernet at all but only deal with IP addressing and one of the protocols building on top of it.

IP

IP, short for “Internet Protocol”, marks todays standard for the majority of computer to computer networks. While originally invented, as the name suggests, for the Internet, it quickly gained popularity in local networks, replacing proprietary protocols such as IPX or NetBEUI.

IP implements the network layer. Nodes in an IP network have a logical IP address that unlike a MAC address is not tied to a physical device or a certain host. IP is a best-effort protocol that makes no delivery guarantees. IP packets may not reach their destination, may arrive incomplete, damaged, out of order or twice without any notification to the receiver or the sender.

IP addresses can be assigned statically by the user or dynamically, usually via a DHCP server. IP version 4, as used by today’s Internet, is using 32 bit to support $4 * 10^9$ addresses, which is not sufficient to have worldwide unique addresses. The newer IP version 6 supports, amongst other improvements, $3.4 * 10^{39}$ addresses stored in 128 bit address fields. IP addresses can be assigned manually by the user, or a central DHCP server in the network can assign IP addresses automatically. A special address in IPv4 is 127.0.0.1, also known as *localhost*: Each computer is seeing itself under the address 127.0.0.1, and applications can use this address to send packets to other applications on the same computer. The services on an IP host are identified by a *port* number that is associated to them.

IP is using a minimum header size of 20 Bytes.[11]

UDP

UDP (User Datagram Protocol) is a protocol on the transport layer that is designed to run on IP. The services it provides to the application layer are data checksumming and application multiplexing. UDP has a very low data overhead the size of the UDP header size is 8 Bytes. UDP, like IP, is a best-effort protocol and does not detect or compensate packet loss or packets arriving out of order.[14]

TCP

TCP (Transmission Control Protocol) is a connection-oriented transport protocol. Before sending data to a remote host, the sender must first establish a connection. Once the receiver accepts the connection, datagrams can be transmitted in both directions. The TCP protocol ensures integrity, order, uniqueness and provides features such as flow control and congestion avoidance. These additional services come at a price, though: TCP is slower than UDP, which is why most multimedia applications like video streaming or real-time applications like games prefer UDP for their communication. The TCP header is 24 Bytes in size, resulting in a larger overhead than UDP.[13]

RTP/RTSP

RTP (Real Time Protocol) and RTSP (Real Time Streaming Protocol) are protocols optimized for streaming audio and video data over the Internet. They are used in conjunction, with RTP transporting the actual data and RTSP taking care of controlling the stream. They don't fit entirely in either the transport or the application layer but sit between the chairs. Usually, they're used on top of UDP/IP but specifications exist to run it over other protocols too.

RTP and RTSP provide services that are useful for broadcast or video conferencing, like timing information, loss detection, security, synchronization, source identification and quality of service feedback. Despite their naming, they cannot guarantee real-time delivery as they have no influence on the timing behavior of the network layers under them.[10]

Zeroconf/Rendezvous

Zeroconf[1] is a system for automatic discovery of devices, services and computers in a local IP network without the need for central servers. The Apple developer pages list the following features[19]:

- allocate IP addresses without a DHCP server
- translate between names and IP addresses without a DNS server
- locate or advertise services without using a directory server

Apple has included Zeroconf in the OS X operating system since version 10.2 and is advertising it under the brand name *Rendezvous*. Apple's software is making extensive use of Zeroconf to provide for example file-sharing or instant messaging services between computers that are connected through a network without requiring any configuration from the user.

Zeroconf is an IETF standard with freely available specifications and reference implementations.

3.2.3 FireWire

FireWire or IEEE1394 is not a networking technology in the usual sense. It's main purpose is not to connect computers to computers but to connect high-bandwidth periphery like cameras or hard drives to a computer. FireWire was developed by Apple in the late 1980s and soon was adopted as an IEEE standard. It was specified as IEEE1394a for transfer speeds of 100, 200 and 400MBit/s over four wires. Optionally, a six-wire FireWire cable can also provide power for external devices. FireWire allows up to 63 devices per bus, cable lengths of up to 4.5 meters and allows flexible topologies. Recently, a faster "FireWire 800"/IEEE1394b was introduced, with current implementations delivering 800MBit/s over nine wires while being backwards compatible to IEEE1394a. The road map is prepared for future speeds of 1600MBit/s and 3200MBit/s.

The reason why FireWire is interesting for networks is that many operating systems provide an implementation of the IP protocol (see the following section) over FireWire, allowing users to

[1]http://www.zeroconf.org

connect multiple computers in a network using FireWire cables. While FireWire is not suitable as a multi-purpose networking technology that could replace Ethernet, it provides a good alternative for high-speed transfer of data between two computers.

3.3 Mac OS X

Mac OS X is an operating system by Apple Computer, Inc that runs on their current line of Macintosh computers. While the first version of Mac OS X was released in March 2001, large parts of the system are a lot older than that: Most of the core is derived from NextStep, the operating system that the "NeXT" computers ran in the early 1990's. This heritage is still showing through in the naming of some of the keywords in the system (the names of many framework classes start with "NS") and the copyright notes of header files. The Mac OS X kernel is a derivate of the Mach microkernel, with many changes towards a more speed-optimized monolithic design. On top of Mach is a BSD layer, a set of libraries, APIs, services and tools from the University of Berkeley's Unix distribution. Most of it is taken from the FreeBSD variant, but there are also parts from OpenBSD, NetBSD and BSD386. The core parts of Mac OS X are also available as an open source operating system under the name of Darwin[2].[22]

The higher frameworks of Mac OS X are proprietary and not available as open source: Some of these frameworks have been developed by Apple for their previous "Classic" Mac OS versions, like Quicktime (a media framework) and Carbon (a general application framework for the C language). Other frameworks are again taken from NextStep, like the Cocoa, an object oriented application framework for Objective-C and Java, or are Apple's implementation of cross-platform standard APIs like OpenGL. A number of APIs in Mac OS X are new developments by Apple, for example CoreAudio for audio and MIDI applications.[22]

3.3.1 Mach and BSD

The core of Mac OS X is derived from the Mach kernel and the BSD operating system. The BSD libraries give Mac OS X many attributes found in traditional Unix systems, like the UFS file system, the BSD network sockets API or pthreads for multithreading. The Mach kernel, which is responsible for scheduling, has an important attribute for multimedia applications: It can mark threads as real-time threads, in which case the kernel will try to assign a certain amount of CPU time to the the thread in regular intervals. However, Mac OS X cannot guarantee that scheduling and does therefore not fully qualify as a real-time operating system.

3.3.2 Kernel space and user space

The memory in OS X is divided in two regions, the kernel space and the user space. The kernel space is reserved for only the very essential core parts of the system like the scheduler or memory management and hardware drivers that require direct hardware access, where the user space is being used for other system services, libraries and user applications.

[2]http://developer.apple.com/darwin/projects/darwin/

Since the kernel is the central part of the operating system, the kernel and the user space are strictly separated and applications in the user space cannot access the kernel space memory directly. Even more, applications in Mac OS X are separated from each other and cannot directly access other applications' memory spaces, ensuring that an error in one application leaves other applications and the system unaffected. In the opposite direction, the kernel itself is not able to access user space libraries.

This has significant implications for the application developer: As user space applications have no impact on system stability, they can be interrupted and examined at any point. This functionality is widely used in software development tools ("debuggers") and makes finding errors in applications a lot easier. The kernel on the other hand cannot be interrupted as easily, and therefore tools for developing and debugging kernel code are much less comfortable. Where programming mistakes in a user space crash only that application, errors in a kernel module often result in a system crash that requires a system reboot.

Apple recommends that software stays outside the kernel space whenever possible.

3.3.3 CoreAudio

The audio parts of Mac OS X do not inherit from previous systems. Where in Mac OS 9 applications had to go through third party APIs such as ASIO, VST[3], or OMS[4] in order to get access to low-latency multichannel hardware, to use audio plug-ins or to communicate over MIDI, Mac OS X comes with all of this included in the CoreAudio framework.[23] In practical situations, CoreAudio is able to deliver latencies of less than 4 ms even on consumer grade hardware under heavy CPU loads[6], which makes Mac OS X a good platform for interactive audio applications.

CoreAudio is using a "pull" model for its audio streams: Each time a node requires data, it is calling the previous node in the chain. This makes the end node the node that determines the timing. Usually, the audio output driver is the end node in the signal chain.

HAL

CoreAudio's hardware abstraction layer (HAL) sits between the applications and the kernel. Its purpose is to make the applications independent of the kernel drivers and vice versa. It allows multiple applications to access the audio hardware with arbitrary stream formats. It provides functions to record and play audio streams, to list the audio hardware installed in the system and to configure these devices through properties. In addition, it provides time-stamping mechanisms for high precision timing.[23]

The default stream format for the HAL is interleaved float32 samples.

[3]http://www.steinberg.net/
[4]http://www.opcode.com/products/oms/

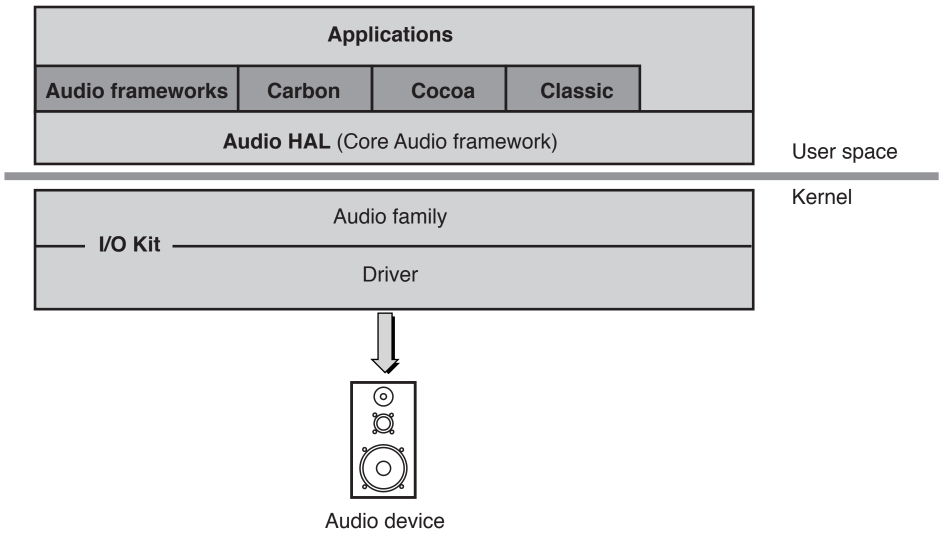


Figure 3.3: CoreAudio's HAL in OS X

Audio Drivers

The vast majority of audio drivers in Mac OS X is implemented as kernel extensions. As the HAL is already taking care of the communication with the applications, sample rate conversion and mixing, the audio drivers can be kept very simple and mainly need to deal with the audio hardware and pass sample buffers between the hardware and the HAL. The interfaces for writing kernel audio drivers are well documented and a number of examples are provided.

In addition to kernel space audio drivers, Mac OS X provides a mechanisms for user space audio drivers. The API for user space drivers however is almost completely undocumented. In order to determine whether or not this API is suitable for use in the AudioSpace project, the author examined this API closer, with the details being documented in Appendix A. In short, user space drivers are possible in OS X, but they do not take advantage of most of the HAL's services and have to communicate with the applications directly.

AudioUnits

Mac OS X introduced its own standard for audio components, AudioUnits. These units can generate, process or receive audio streams. Applications can instantiate and use these components through a standard interface. Often, software synthesizers or effects are implemented as AudioUnits. Mac OS X ships with a number of ready to use units like mixers, reverb, delays and a HALOutputUnit that wraps hardware outputs in an AudioUnit, and a number of software companies that produce audio software make their effects plug-ins available as AudioUnits too. Other standards for audio plug-ins supported by Mac OS X are Steinberg's VST and plug-ins for Digidesign's ProTools.

Chapter 4

Previous Work

In the following, related work to the AudioSpace will be described. First, there will be a look at existing applications to transport audio over networks, then at existing approaches for tapping the audio streams of Mac OS X applications.

4.1 Audio over network

4.1.1 Audio over Ethernet

A number of systems exist to replace the multi-core cables in audio studios with the much cheaper Cat5 Ethernet cables. These systems provide very low latencies, multiple channels and a high audio quality, but they rely on hardware end devices and cannot be used from computers, which disqualifies them for use in the AudioSpace.

Gibson MaGIC

MaGIC (Media-accelerated Global Information Carrier) is a technology by Gibson Guitar Corp, that was first introduced in 1999. Gibson, being primarily a maker of fretted string instruments, is selling MaGIC equipped guitars and licensing MaGIC for use in mixing desks, amplifiers and effects, promoting the technology to be used as a fully digital system for stage and studio use. MaGIC allows to transmit up to 32 channels of 32bit information with up to 192kHz sample rate. The reference implementation is based on Fast Ethernet, using Cat-5 cabling and the IEEE 802.3 protocol, but the MaGIC network layer can also run on other media (e.g. GBit Ethernet, optical). The MaGIC network layer is not compatible with IP.[20]

EtherSound

EtherSound is also an Ethernet-based transport protocol for digital audio established by DigiGram. EtherSound is advertised as a system used for audio distribution in large buildings, intercom, broadcast and live sound. Information about the protocol is rather sparse, but DigiGram claims a latency of 125μsec at quality ranges up to 24bit/48kHz. DigiGram does not provide products to connect personal computers to an EtherSound network. The EtherSound licensing fees depend on the number of products sold by the licensee and the number of channels that the product supports.[21]

CobraNet

CobraNet by Peak Audio is also based on 802.3 Ethernet. Just like EtherSound, it is primarily advertised as a system for audio distribution in large buildings. The standard is a latency of 256 samples (5.33ms) through buffering on both sides, some more recent implementations allow latencies of 2.67ms and 1.33ms. CobraNet can work in existing switched Ethernet LANs. According to Peak Audio, 802.11b Wireless Ethernet is not suitable for use with CobraNet. The CobraNet network layer is not compatible with IP.[28]

4.1.2 Audio over IP

The availability of large bandwidths in the Internet enabled the transfer of compressed audio and video streams over the Internet in real-time. With the adoption of IP for local networks, IP is also being used to send uncompressed audio over local networks.

Internet streaming

What comes quickly to mind when thinking about networked audio are streaming technologies such as Quicktime[1], Windows Media[2] or RealMedia[3]. These, however, aim at completely different goals: These streaming services are designed to stream mostly prerecorded streams from one central provider over the Internet (and thus taking complex paths over a number of routers) to multiple recipients. They do not pursue low latencies but can well live with latencies in the range of seconds: These streaming services are not aimed at interactive use at all but only for presentation and broadcast purposes. Such streaming technologies operate with latencies in the range of up to 10 seconds and rely lossy compression algorithms, severely reducing the audio quality in favor of bandwidth savings.

Internet streaming services should be able to reach all kinds of target platforms and cannot rely on services specific to a certain hard- or software platform, therefore they are not very tightly integrated in the host system and only available in a few applications that were specifically designed to use them.

[1]http://www.apple.com/quicktime/
[2]http://www.microsoft.com/windows/windowsmedia/
[3]http://www.real.com/

Voice over IP

Another popular application of networked audio is voice over IP (VoIP). The aim of this technology is a bi-directional transport of voice and sometimes video data over IP networks for communication purposes, replacing traditional telephony solutions that require a direct connection.

Most VoIP implementations follow the H.323 standard. H.323 is aimed at being a universal and flexible communication service and therefore comprises specifications for dialing, routing, negotiating and controlling. The audio quality of data transmitted over H.323 is very low, targeting mainly voice data and maintaining compatibility with low-bandwidth access technologies like analog modems. While VoIP is used for real-time communication, latencies are still large enough to be unsuitable for interactive systems.

Esound (esd)

The “enlightened sound daemon”, short Esound or ESD[4] is an audio service written for the GNU/Linux operating system[5]. It provides mixing services allowing multiple applications to access the audio hardware simultaneously. Connections to the Esound process are being made over local or remote sockets, allowing clients that access the service from remote computers over the TCP/IP protocol. Esound is not real-time capable and crackling or stuttering may occur. This may be one of the reasons why Esound development has virtually stopped in favor of the real-time capable Jack audio server (see section 4.2.3).

The Esound server and a number of client applications supporting the Esound protocol have been ported to Mac OS X. Especially interesting is the Esound kernel driver that has been developed for Mac OS X[6]: While it doesn’t provide the desired low latency, it is a fine example of a universal audio service that works with all kinds of applications.

Both the Esound driver and the server for OS X do not provide a GUI. The driver even requires a recompilation to change the address of the destination server.

aRts

A probably more popular Linux sound daemon exists with aRts[7], which is the sound server of choice of the KDE project. It does also provide optional network transparency, and like with Esound it appears that development has stalled (the project web site hasn’t been updated in almost a year) and the latency is reportedly not very low.

[4]http://developer.gnome.org/doc/whitepapers/esd/

[5]On earlier versions of the GNU/Linux operating system, audio devices could be used by only one application at a time. This has led to the development of several “audio servers” which provide an abstract audio device that can be accessed by multiple applications simultaneously and mix the incoming streams. More recent developments around the audio drivers and the kernel, mainly the ALSA project, removed the necessity for such servers.

[6]http://homepage.mac.com/samoconnor/Esound/

[7]http://www.arts-project.org/

FX Teleport

FX Teleport[8] is a commercial VST plug-in for Windows that enables the user to run other VST plug-ins on a remote computer. A version for Mac OS X is promised but has not been released yet (Jan 10th 2004). Latencies are stated to be as low as 6ms.

Wormhole

Wormhole[9] is a brand-new commercial audio over TCP/IP solution for OS X that implemented as an AudioUnit. Claimed are low latencies, but gives no quantitative figures. Wormhole is a commercial project that costs $25 per license. Wormhole is using a static IP address configuration and works point to point.

Since Wormhole was released on Jan 10th 2004, it was too late to evaluate it for the AudioSpace project, but it is mentioned here for completeness.

4.1.3 Audio over FireWire

mLAN

mLAN is a technology pioneered by Yamaha. Intended to replace MIDI and analog audio cables, it transmits multiple channels of digital audio and MIDI data over a single FireWire (IEEE 1394) cable. mLAN is mainly being used with instruments, but Mac OS X also includes drivers to talk to mLAN devices that are connected to the computer's FireWire port. However, one cannot use mLAN to connect two or more PowerMacs directly and applications running Mac OS X cannot access mLAN hardware directly.

The fact that mLAN is a closed system that cannot be used from custom hardware disqualifies it for the use in the AudioSpace.

4.2 Tapping Audio

As the AudioSpace will run with legacy software, a way must be found to get access to the audio streams of these applications. Mac OS X itself does not offer methods for sending audio from one application to another. However, some third party implementations for sending audio between applications on OS X are available.

[8] http://www.fxteleport.com/
[9] http://www.apulsoft.ch.vu/

4.2.1 ReWire

ReWire[10] is a standard developed by Propellerheads, a software company producing software instruments and sequencers. ReWire is an API that allows to route audio between applications on the same computer, like plugging the output of a CD player to the input of a tape deck. In order to take advantage of that, an application must explicitly support the ReWire protocol which requires an SDK license from Propellerheads. Such a license is free, but only available to registered companies. Propellerheads does explicitly not issue ReWire licenses to private persons and schools. ReWire is widely used in many professional and semiprofessional audio applications.

4.2.2 Audio Hijack Pro

Audio Hijack Pro[11] is an application that puts itself between Mac OS X' HAL and an audio application and intercepts function calls. Audio Hijack Pro allows the user then to record the audio data the "hijacked" application is playing or to play it back with effects added to it.

Audio Hijack Pro is using the "Application Enhancer" (APE) framework from Unsanity[12], a framework that allows writing code that will be executed in any other application's memory space, primarily to replace system calls with custom functions. Since that is relying on assumptions about what function calls the respective host application is using and how it is using them, applications using APE tend to be not fully compatible and often require updates with newer releases of the operating system.

4.2.3 JackOSX

The Jack audio server for Linux has been ported to Mac OS X a while ago. Jack is not only providing a low latency mixer but is also an audio routing system, comparable to ReWire. The user can create arbitrary connections between applications that use Jack and use the sound output of one application as input for another. Since hardly any applications for Mac OS X are using the Jack API, a group of developers started to write wrappers that act as VST or AU plug-ins. With these plug-ins, Jack can be used to create connections between applications that can use VST or AU plug-ins.

The next step for Jack on Mac OS X started in summer 2003 when the author of this thesis was taking a first look at the mostly undocumented AudioHardwarePlugin API in Mac OS X (see Appendix A): Having read about these investigations in user space audio drivers on Mac OS X, members of the Jack on Mac OS X project contacted the author and received a copy of a simple prototype of such a user space driver. Based on that prototype, the Jack developers eventually developed a user space audio driver that interfaces with Jack, allowing almost every application on Mac OS X to be used with Jack. The full package was called JackOSX[13] and was first released to the public on January 7th 2004, mentioning the author's contribution in the documentation[14].

[10]http://www.propellerheads.se/technologies/rewire/
[11]http://www.rogueamoeba.com/audiohijackpro/
[12]http://www.unsanity.com
[13]http://www.jackosx.com
[14]http://www.jackosx.com/Documentation.pdf

4.3 Comparison to the AudioSpace requirements

With the exception of the Esound driver for Mac OS X, all the audio over network solutions described above are tied to a certain application range: CobraNet, EtherSound and MaGIC are focused on using Ethernet cables as a cheap replacement for analog multi-core audio cables. Quicktime Streaming, Windows Media or RealMedia are all designed for broadcasting compressed media from one sender to many receivers. FX Teleport and Wormhole are specialized on effects in audio applications and are building on the plug-in APIs of these applications. They all have in common that a program that wants to use their services has to meet certain requirements, be it supporting the streaming protocol directly or being a host for a certain kind of plug-in.

In contrast, the AudioSpace system needs to work with any application and therefore cannot be tied to a certain plug-in API. The operating system should treat the AudioSpace like a regular audio device and present it as such to the applications and the user.

On the networking side, the restrictions are not always as strict: VoIP or broadcast streaming work with any IP connection, with no difference whether it's an intercontinental satellite connection or a 10m cross-cable between two workstations. The AudioSpace works in a different environment: It is safe to make certain assumptions about the network infrastructure and the computers in that network, where at the same time it should be compatible with any application that can run on these computers.

Most of the systems that for tapping into an OS X application's audio stream are not universal enough either: ReWire works only with applications that are explicitly written to use it and Audio Hijack Pro is not working with every application. It looks like using an audio driver is the best, like done in JackOSX or the Esound driver for Mac OS X.

Overall, Esound's shared audio is going in the same direction as the AudioSpace, but is very lacking in usability and latency.

Chapter 5

Design and Implementation

5.1 Approach

5.1.1 Guidelines

The first step of designing the AudioSpace software was to outline the general approach to the problem. In addition to the requirements that were given in chapter 2, a few guidelines for the development process were laid out:

low client overhead The AudioSpace software should have as little impact on the client's resources as possible. Chances are the client is needing its resources for tasks such as software synthesis, decoding and playback, and the AudioSpace software should not interfere with that. If the client's driver software requires too much system resources, the results can be skipping, clicking or other undesirable effects. A low CPU and RAM overhead on the client computer's resources will result in a much better user experience and make it much less noticeable for the user that he is using remote audio hardware.

well integrated in the system The AudioSpace should play to the host system's rules and conform to its development guidelines. Hacks and patches are much more likely to break with system updates and can potentially collide with other software installed in the system. Playing by the system's rules will also guarantee a maximum of compatibility with existing and upcoming software.

straightforward This may sound obvious, but experience shows that in the process of developing an application there's always the danger of getting caught in details, often leading to an unnecessary complexity of the final product. High complexity can lead to programs that are hard to understand, hard to debug and instable.

don't reinvent the wheel The operating system is providing a number of services for software running on it. Usually, system software is much better taking advantage of the system's capabilities than 3rd party software. A 3rd party reimplementation of such services is likely to be not as good as the software built by the same company that built the system itself. Since system software is being used by many 3rd party developers, it receives much better stress-testing than newly written software can get. System software has been tested on all possible hardware and software configurations where a new implementation can only be tested in environments the developer has access to.

multithreaded Since the server hardware in the Media Space is a PowerMac G5 with two CPUs, the server should be able to split the load across the two processors as good as possible.

practical No matter how well-designed a system looks on the paper, no matter how beautiful it is from a scientific viewpoint: At the end of the day, what counts is the answer to the question "but does it work"? Since the AudioSpace system is going to be used on a regular basis, the "it works"-category is a very important aspect and must not be neglected just for design ideals.

5.1.2 Design

Based on these guidelines and the analysis in chapter 3, the following design was derived (figure 5.1):

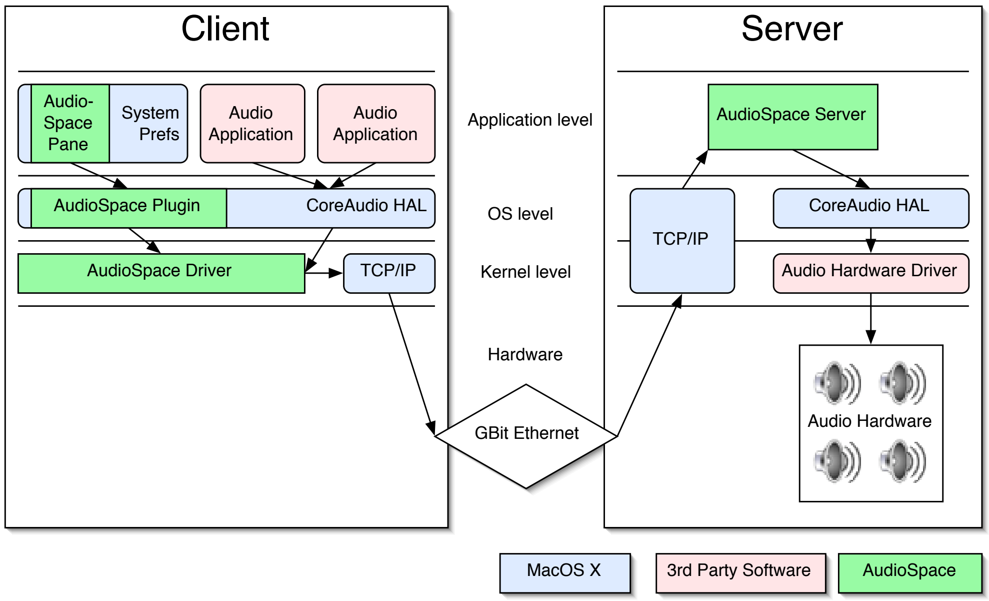


Figure 5.1: How the AudioSpace integrates in the host system

Audio driver for the clients

The ideal integration in the system can be achieved with a regular audio driver. That way, the AudioSpace will work with any application that supports sound output in an OS X compliant way and not be restricted to applications that support a certain plug-in format.

Helper applications on the client side

Since the audio applications depend on the drivers, faulty or complex driver software can cause the applications that are using them to become unstable. Since the AudioSpace requires more user intervention than a regular audio driver, like selecting the destination server or assigning local channels to remote speakers, the code could potentially become complex and with it prone to errors. By putting these features in to separate applications where possible, the more critical code could be kept out of the driver and with it, out of the scope of applications using the driver.

It was decided that the configuration tool should be a plug-in for OS X system control panel, a so-called *PreferencePane* and that a separate startup item would load the saved defaults when the system boots, providing persistent preferences without the need for file access from the driver.

UDP/IP networking

While the IP protocol adds a slight additional CPU and bandwidth overhead over raw Ethernet packets, it was chosen as the base for the network transfer. Since IP is very popular, the system routines for it are usually highly optimized and well-documented, especially in the case of the BSD IP stack which is used in Mac OS X. Sending raw Ethernet packets is possible in Mac OS X, but it has restricted access and documentation and sample code is rather rare.

As a bonus, IP networking provides independence of wired Ethernet, allowing connections over FireWire cables or wireless Ethernet.

At first sight, TCP may look like a good base protocol: It maintains the packet order, is built for continuous streams of data and offers connection handling. However, a few tests revealed that TCP has latencies ranging from 1ms up to 30ms when sending packets from a user space application to a user space application on the same machine (not involving any physical network). UDP instead showed to have an average latency of 0.5ms with spikes up to 4ms. Considering the latency requirements outlined in section 2.4, this makes TCP unacceptable for musical applications.[10] While UDP does not protect from packet loss or out of order arrival, this has close to no relevance when used in local Ethernet networks[15], so the decision was made in favor of UDP.

Rendezvous

Using IP enables the use Rendezvous as well. The use of Rendezvous significantly enhances the user experience as it will completely shield the user from the technical details of IP networking.

AudioUnits

Since Mac OS does already come with a number of useful audio services included, it is only natural to use them for signal processing where possible. The big advantage over custom routines with equal functionality is that the components that come with the operating system are usually well-integrated in the system, well-tested and should be better optimized (for example using the AltiVec SIMD unit of the G4 and G5 processors) for Apple's current and future hardware. Since the AudioUnits are also being updated with newer releases of Mac OS, the AudioSpace application will be able to profit from these enhancements automatically.

Cocoa server

For the server application, the combination of the Cocoa framework and the ObjectiveC programming language was chosen over alternatives like Java, Carbon/C or Qt/C++. Cocoa and Objective-C are well supported by the development tools included in Mac OS X and with the Interface Builder tool it is very easy to create graphical user interfaces for applications in a short amount of time. The Cocoa framework supports the use of the BSD networking API and provides classes for dealing with multithreading.

Chapter 6

Implementation

6.1 Client driver

6.1.1 AudioHardwarePlugin

The CoreAudio documentation describes only how to implement kernel drivers for audio devices, no hints are given about any user space drivers. Despite that, there is an API for user space audio drivers, found in the AudioHardwarePlugin.h header file. The author spend some time evaluating this undocumented API, the interested reader can find the detailed results in Appendix A.

The conclusion was that this API is unsuitable for the purposes of the AudioSpace: Drivers that built on that API are loaded as an plug-in in the application's space, therefore instanced once for every audio program loaded on that computer. An audio driver sending network packets would therefore send a separate stream of packets on the network for every application using it, using more network and system resources than necessary. In addition, it turned out to be hard to make the driver compatible with all possible audio applications. Therefore, it was decided to use a kernel space driver instead.

6.1.2 Kernel Space Driver

The Mac OS X developer documentation recommends implementing audio drivers in the kernel. While in other cases it is recommended to stay in user space wherever possible, all the audio drivers Mac OS X ships with - except the iSight driver which was added in October 2003 with in 10.3 - are kernel extensions. In contrast to the AudioHardwarePlugin API, writing audio drivers as kernel extensions is well documented and a few examples are provided. Since the HAL is providing a lot of services for the kernel driver, a kernel driver can be kept very simple and doesn't have to answer applications' questions like an AudioHardwarePlugin has to. With the addition of the available source code for the Esound driver (see section 4.1.2) it was surprisingly simple to implement kernel extension that transmits audio packages over the network. First tests using a modified Esound driver showed that the kernel driver had a lower latency than a AudioHardwarePlugin, despite the extra abstraction layer of the HAL between the between it and the application. Furthermore, the kernel driver was also much more reliable when the system was under load, where a AudioHardwarePlugin was much more likely to stutter. Since a kernel driver also has the benefit of getting the buffers readily mixed from the HAL, the decision was made in favor of the kernel driver, despite the harder debugging and user space communication.

Since the kernel driver has only very limited access to user space APIs and is therefore unable to use Rendezvous or property list files, it needs a way to communicate with user space helper applications. The kernel APIs offer a communication method for IOKit drivers that could have been used here, but there is another API available, especially for audio drivers: AudioDriverPlugin.

Unfortunately, just like the AudioHardwarePlugin API, no documentation for the AudioDriverPlugin API is available except for a few comments in the header file. Luckily, a third party developer[1] published some sample code implementing an AudioHardwarePlugin that was used as a reference.

The AudioDriverPlugin allows to implement custom properties for a driver that can be accessed by any user space application through the HAL. The HAL then calls the respective functions of the kernel driver to pass the information it got from the User space applications. For the AudioSpace driver, two properties were implemented: One for the IP address of the server, one for the UDP port.

6.2 Networking

6.2.1 Protocol

The AudioSpace protocol was kept as simple as possible: Before sending a stream, the client announces it with a header packet, containing information about the sample rate, the number of channels, the channel to speaker association and the size of the audio buffers it's about to send. After that, the client starts sending packets of 16bit interleaved PCM samples.

To allow an interactive association of the channels the client is sending to the speakers of the server, the initial packet can be resent at any time with new values for the channel/speaker association.

The protocol does not include a disconnect event: Instead, the server is monitoring the time between incoming packets and considers clients from which it hasn't received packets for more than 500 ms as disconnected.

Note that the client is not waiting for any confirmation from the server: As Apple recommends to keep networking code out of the kernel whenever possible, it was decided that sending packets is bending this rule far enough already. Trying to receive network packets in the kernel using almost undocumented functions was considered not worth it.

[1]http://acretet.free.fr

6.2.2 Rendezvous

Implementing Rendezvous in was straightforward and very easy using the examples from Apple's developer tools as reference. As Rendezvous is not available in the kernel, the client is using Rendezvous in the AudioSpace PreferencePane and is passing the IP address and the UDP of the selected server to the kernel driver through its custom device properties.

6.3 Server

6.3.1 Application

The server side application was developed in the Objective-C language using the Cocoa framework, which allowed using Apple's InterfaceBuilder tool for rapid UI development. The network programming was done using the BSD sockets API as that is better documented than the Cocoa network classes. The server is split in several classes (figure 6.1):

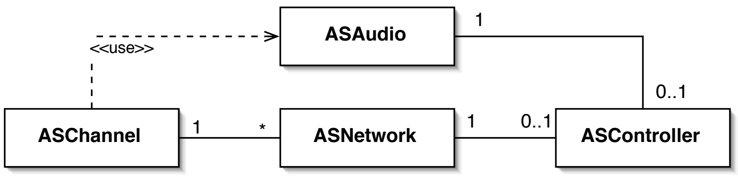


Figure 6.1: The AudioServer

ASController is responsible for the UI and maintaining the persistence of the user's preferences.

ASNetwork opens the UDP sockets, receives incoming packets and delegates them to the respective channels. In addition, ASNetwork is publishing a Rendezvous announcement making the server known in the local network. ASNetwork is keeping track of all the clients connected to the server and is automatically creating and deleting ASChannel objects when required.

ASChannel is being instantiated for every client connecting to the server. ASChannel holds some information about the channel (IP address, stream format), creates a ringbuffer for the packets, implements an audio callback to play sound from the ringbuffer and compensates the client's jitter and clock deviation.

ASAudio is the central audio class that creates and manages the AudioUnits required for mixing and playing audio on the hardware. It also provides a list of the available hardware devices in the system that ASController can query to allow the user to select the output device.

6.4 Audio

6.4.1 Audio mixing and output

The AudioSpace server is using CoreAudio's AudioUnits for the audio related tasks. They provide solid implementations of common services and are most likely well optimized for the G5 and Mac OS X. Also, the AudioSpace server could then profit from improvements and optimizations of these AudioUnits in future releases of Mac OS X.

CoreAudio comes with three mixing units, StereoMixer, 3DMixer and MatrixMixer. StereoMixer works, as the name suggests, only with 2 channels while the AudioSpace requires at least nine channels. The 3DMixer has support for more than two channels and a number of other interesting features like three-dimensional positioning of stereo sounds with a simulation of all real effects involved, like phase shifting, delays and the Doppler effect. Unfortunately, the 3DMixer does not treat all output channels equally but is restricted to a number of predefined speaker configurations like quadrophony or a 5.1 set with two front speakers, two rear speakers, a center speaker and a subwoofer. The MatrixMixer is the most flexible of all these mixers, allowing any number of input streams to be routed to any number of outputs. The MatrixMixer requires that the input streams have the same number of channels as the output stream of the MatrixMixer. Also, the MatrixMixer works only with a static number of inputs that cannot be changed when streams are playing. Since one can disable unused inputs and disabled inputs use hardly any system resources, this is not too much of a restriction when one sets the MatrixMixer to a sufficient number of channels before initializing and then enables and disables the required channels on the fly.

The HAL output is present as two AudioUnits, the HALOutputUnit and the DefaultOutputUnit. These Units are the same except for one detail: The HALOutputUnit allows the program to choose the audio device it wants to use where the DefaultOutputUnit uses whatever device was set in the system's preferences.

For the AudioSpace server, the HALOutputUnit and the MatrixMixer unit proved to be flexible enough. The restrictions of the MatrixMixer could be worked around: The user is allowed to set a maximum number of clients that are allowed to connect to the server and the MatrixMixer unit is initialized with that as the number of inputs. The server keeps track of the occupied inputs in a one-dimensional array with one entry for each input, containing an ID of the client connected to it or a value indicating that the channel is not occupied. When a client connects, the server goes through the list and searches for a free input. If it finds one, it writes the ID of the client to the array, connects the client to the input and enables it. If it doesn't find a free channel, no connection is made. When a client disconnects, the server disconnects it from the mixer, disables the input and clears the ID in the array.

6.4.2 Jitter

The travel time of network packets is not constant: On the sending computer, the application has to share the CPU with other applications and may have to wait, other network traffic on the computer may delay the time the packet takes until it is being sent on the physical network. On the network cable, collisions may occur, networking devices like switches and routers can impose another nonconstant delay, and the receiver's networking software's response time is again restricted by the operating system's scheduler. As a result, when a sender is sending packets at regular intervals $\triangle t$ with $t_n = t_{n-1} + \triangle t$ over a network with a latency of l the receiver will receive these at irregular intervals $\triangle t' = \triangle t + l \pm j$. This variation j is referred to as *jitter*. Usually, the jitter increases with more devices between the network peers.

Jitter has no implications for non-time constrained data, but audio and video streaming applications have problems with jitter: Since they require data at constant rates in order to play a non-interrupted stream, jitter can cause drop-outs. In order to compensate jitter, the common practice is to insert a FIFO (first in, first out) buffer between the network and the playback application and to collect a few incoming packets before starting the playback. This emphprebuffering introduces additional latency, but makes the system more resistant to jitter. Note that it is impossible to make the system completely resistant to jitter, it will always be a compromise between maximum tolerable jitter and buffer size. The additional latency is at least twice the size of the maximum tolerated jitter, as it needs to be able to cover the gap of $2 * j$ between packet n arriving at $\triangle t' = \triangle t + l - j$ and packet $n + 1$ arriving at $\triangle t' = \triangle t + l + j$.

To compensate the jitter in the AudioServer, each ASClient class has its own ringbuffer in which it stores the incoming samples and from which the audio thread reads from. To deal with different network and computer environments, the user can adjust the buffer size to the requirements of the situation.

6.4.3 Clock skew

A major problem in distributed systems is that the nodes usually have no common clock source. In the case of AudioSpace, each of the computers is calculating its sample rate from its internal clock. These clocks are not infinitely precise, usual clock speed deviations are quoted to be between 10^{-5} and 10^{-7} percent. While this is not a problem for an isolated system it is one for networked audio: Assuming a client sends at a sample rate of 48,000Hz to a server that runs 0.01% faster, playing the stream at 48,004.8Hz, the server receives 288 samples per minute less than it requires. When playing the soundtrack of a movie over the network, this will over time lead to a loss of synchronization between audio and video. When the difference becomes larger than the receiver's buffer size, the receiver will drop or repeat buffers, resulting in audible artifacts like skipping, stutter and cracks. Increasing the buffer size will not eliminate the problem, it'll only increase the time before dropouts happen, at the expense of latency.

How do existing systems deal with this problem? It turns out that hardware-based solutions like MaGICdeclare a master device that is sending a master clock over the network cable and all the other devices synchronize their clocks to that.[20] This works well for these dedicated hardware devices, but is not feasible for the AudioSpace: Since the server's timing depends on its audio hardware which usually has a fixed sample rate, it cannot synchronize to an incoming stream, even less to multiple streams. Remote effects systems like FX Teleport don't have to worry about this either: While there is no master clock in the network, the sound stream ends on the same computer it started. The remote node doesn't have to worry about timing at all, it is simply processing the sound data as quickly as possible and sends it back to its origin.

After an evaluation of existing solutions to clock skew compensation, a new algorithm was designed for the AudioSpace system: It provides a robust compensation of clock skew up to 0.1 % without any audible artifacts and is suitable for low latency applications. A detailed description of the algorithm is given in Appendix B.

AudioSpace's implementation is using CoreAudio's Varispeed AudioUnit. It is a high-quality sample-rate converter that is able to switch between different sample rates on the fly. The Varispeed AU is expecting float32 non-interleaved streams and requires that the number of input channels equals the number of output channels. As the incoming streams from the client are in 16 Bit integer format with interleaved samples and a number of channels that does not need to match the number of channels of the MatrixMixer, additional AUConverters are put before and after the Varispeed unit.

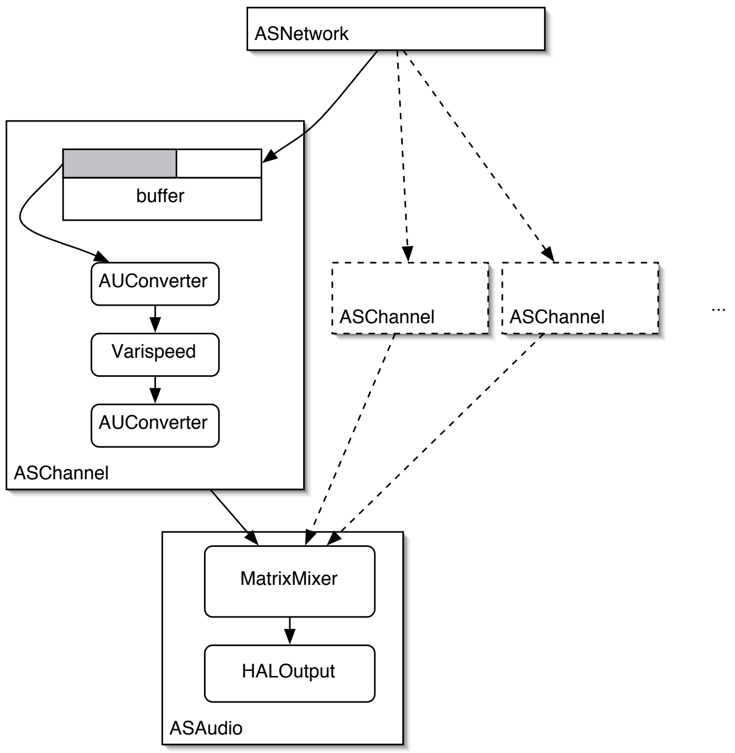


Figure 6.2: The use of AudioUnits

6.4.4 Performance optimization

One source of latency is the scheduler. In a timesharing multitasking operating system, a process that is waiting for an event (like a mouse click or a key press) is not necessarily getting called right after the event happens but will have to wait until the operating system's scheduler calls it. The time between the event and the scheduler calling the responsible thread can take several milliseconds and in common non-real-time desktop systems, there is no guarantee whatsoever about what the maximum delay is.

In the case of the AudioSpace, it pretty quickly became apparent that the AudioServer was frequently getting buffer-underruns because the AudioServer application was not called quickly enough after a network packet for it arrived. This could be fixed by setting the priority of the network receiving thread to real-time priority.

6.5 User experience

The AudioSpace was designed to deliver a user experience that fits into the behavior that users expect from Mac OS X applications, following the Apple guidelines for the user experience[2].

An illustrated walk-through of a typical use case is provided in Appendix C.

6.5.1 Server

The server application is a self-contained bundle that can be installed simply via drag and drop and launched by double-clicking it's icon. The server's settings dialog is available through the application menu, the settings are saved in a *property list*[3] file in the users' home directory. The preferences dialog itself is using help tags to give the user additional information about the controls.

6.5.2 Client

Since the client software consists of several files, it comes as an installer package. Launched by a double-click, it will install the AudioSpace driver, a PrefernecePane and a Startup Item. The installer recommends a reboot to load the kernel driver, but does not force the user to reboot the computer immediately.

For the configuration of the driver, the regular Mac OS X tools can be used: The AudioMIDISetup tool lists the AudioSpace device like any other audio device and lets the user set the desired sample rate and number of channels.

The configuration options that exceed the capabilities of regular audio devices, a separate application had to be used: It is not possible to add custom controls to the AudioMIDISetup application. Apple is recommending to use a plug-in for the system's preference application, a so-called *PreferencePane*[24]. In the AudioSpace's PreferencePane, the user can locate and select AudioSpace servers and assign output channels on the client side to speakers on the server side.

[2]http://developer.apple.com/ue/

[3]Property Lists are XML files using a predefined DTD and are the Mac OS X standard for storing preference settings.

Chapter 7

Results

7.1 Performance of the System

The performance of the implementation was tested on various computer configurations running Mac OS X 10.3.2 with all the system updates available on Jan 19th 2004.

7.1.1 Localhost

The first test was performed on an iBook with a 800MHz G3 processor using the built-in sound device. The iBook was both client and server at the same time, sending 2 channels of audio at 48kHz sample rate over the local loopback IP device. The buffer size of the kernel driver had to be at least at 128 samples, buffer sizes lower than that resulted in dropouts. The receiver had to prebuffer at least 512 samples, using less samples caused buffer over- and underruns to happen. Also, the server application had to be the front-most application or be set at a higher priority than the other applications. If an other application got the system's focus, over- and underruns occurred. Also, the server was very likely to drop buffers when other processes were using a lot of CPU. The average CPU load of the server application was around 25%.

7.1.2 Small wired setups

Transmitting six channels of audio were the maximum the iBook was able to handle: The server application occupied 60% of the computer's CPU and the slightest use of other applications on that systems caused dropouts in the sound. The high CPU load made the system also very unresponsive.

Testing the system with two channels of 48kHz audio on two PowerMac computers with a 1.6GHz G5 CPU each over a switched 100MBit Ethernet, the sender's buffers could also not be set lower than 128 samples. This may indicate that the limit for the lowest possible driver buffer size does not depend on the CPU of the host but is rather a limit of the system's scheduler or the precision of the IOKit timers.

The receivers buffer size could be set lower than with the iBook - prebuffering 256 samples was sufficient to not cause any over- or underruns when the server was the front-most application. However, some system events like the start of the screen-saver or the power manager turning off the screen caused over/underruns, and with it dropouts. Network transactions on the receiving computer like using a web browser or an email application would also cause dropouts. The sender on the other hand had a much higher resistance, and it was well possible to use other applications on it, even those requiring lots of CPU. Rendering 3d images or playing DVD movies did not interrupt the sound stream.

No interruptions could be attributed to clock skew: A continuous sound stream could be played through the system for over an hour without any dropouts.

7.1.3 Large wired setup

A stress-test was then performed on a couple of dual 2GHz G5 computers at the RWTH Aachen. Each of these computers was running 10.3.2 with all the system updates available on Feb 2nd 2004. The computers were connected to each other over a 100MBit Ethernet hub, and one of the computers had a MOTU 828mkII FireWire audio interface connected to it.

The computer with the MOTU card acted as the server, sharing 6 channels out of the 14 output channels the MOTU offers. Then clients connected to it, playing audio streams with 6 channels each. The sampling rate was set to 48kHz on all the computers.

An upper limit was determined at 4 computers sending 6 channels each. At this setup, buffer over- or underruns would happen on the server side, although they were hardly audible. With only 3 clients with 6 channels, no dropouts could be detected. It is not clear where exactly the bottleneck was: The load of the application was not evenly spread over the two CPUs of the server but kept one of the CPUs busy at 80%, while the other CPU was idle. It is quite possible that spikes in the CPU load caused dropouts here. On the other hand, the data rate of 4*6 channels at 48kHz excluding UDP and IP headers is 2.2MB/s. While this is far from the theoretical bandwidth limits of Fast Ethernet, it could still be that this was a result of occasional collisions on the non-switched Ethernet. If this was a network related bottleneck, it could be solved with the use of a switch instead of a hub.

To clearly determine the bottleneck, this test should be repeated with a switched GBit Ethernet. If the same limit showed up then, it's clearly a CPU limitation, otherwise the network was the limiting factor. Unfortunately, no such test environment was available in time.

In contrast to the single G5, the dual G5 server proved to have much less problems when other applications were used while the AudioServer was running. However, the screen saver or power management turning off the display also caused dropouts in the sound stream.

7.1.4 Wireless setups

Two tests were done in wireless setups: In both tests, the computers were PowerMac G5s with two 2.0GHz CPUs each, connected over a 802.11g wireless Ehternet. The tests showed that the transfer over wireless networks is highly sensible to the presence of other traffic on the same network. Where an otherwise unused wireless network showed no dropouts when using the AudioSpace, a very busy wireless network caused many dropouts, resulting in unacceptable sound quality.

7.1.5 Latency

To determine the latency, the sound played by the server was routed back into the client computer and recorded there. The time between an event in the original signal and the recorded signal then equals the total time an audio signal takes through the signal chain. To isolate the latency of the AudioSpace itself, a second experiment was made using a similar setup without the AudioSpace software, playing the sound through a local audio device instead.

In both tests, the sending and recording computer was PowerMac G5 1.6GHz. The recording interface was the built-in audio device of the G5, and the playback interface was an AudioTrak Maya EX[1] USB interface. The output of the Maya EX was plugged in the audio input of the G5. For the first test, an iBook G3 800MHz was used as the server, connected directly to the G5 with a Cat5 Fast Ethernet cable. The software used for creating the test signal, playing and recording was Audacity[2]. The client and the server were set to use 48kHz as sample rate, which is also the native sample rate of the Maya EX. The server was set to the smallest buffer size, prebuffering 384 samples. With each setup, the test was run four times and the results were averaged.

The results attributed the AudioSpace system a maximum latency of 12.5ms. As the server buffer (384 samples) and the clients' driver buffer (128 samples) cause a latency of 9.4ms (512 samples at 48kHz), the remaining 3.1ms latency must come from the network, the UDP/IP stack and the mixing/resampling AudioUnit chain of the server.

7.2 Compatibility

The AudioSpace client driver was tested with a variety of software titles. No compatibility issues were noticed.

7.3 Limitations and problems

Some limits are present in the specifications and do not need to be tested: The AudioSpace software runs only on computer running Mac OS X version 10.3, leaving computers running Windows or previous versions of Mac OS behind.

The tests revealed further limitations: The constant resampling of multiple audio channels in high-quality formats puts high stress on the server's CPU. This turned out to be a potential performance bottleneck when handling a larger number of clients sending audio streams simultaneously.

[1]http://www.audiotrak.de/eng/maya51u.html
[2]http://audacity.sourceforge.net/

The fact that system services like the screen saver and power management or competing networking operations could interrupt the stream on the server side puts some limitations to the system: In order to run the AudioSpace reliable with low latencies, it is necessary to set up the server in a way that the server application does not get interrupted. Measurements would include disabling unused system services like file sharing, web serving, remote login or screen savers. Furthermore, no other user applications should run on the server at the same time, especially not applications that use networking.

Overall, the tests revealed that for the current implementation, the network bandwidth is not the limit. Rather, the major limitations seem to be the CPU power of the server and the scheduling on the server side.

7.4 Future Work

One future enhancement would be to implement the driver for platforms other than Mac OS X. That would allow users with Windows or Linux notebooks to connect to the AudioSpace. Since the protocol itself does not depend on any Mac OS X exclusive APIs, this should be an easy task for anyone familiar with driver programming on the respective host OS.

The high CPU usage on the server side is also offering room for improvements: The Varispeed resampling AU could be replaced with a less complex resampling algorithm. This could result in a lower quality and it is probably necessary to compare different algorithms to find an optimal compromise between performance and quality.

The fact that CoreAudio is executing the whole graph of AudioUnits in one single thread does not take full advantage of a dual-CPU computer like the G5s used in the Media Space. If one improved the AudioSpace by splitting the graph in subgraphs that run in separate threads, the load would be better divided across the CPUs and allow for more simultaneous channels. A problem that needs to be solved then is the synchronization, as the CoreAudio documentation explicitly forbids blocking in an AudioUnit's callback, which makes it difficult to use the usual thread synchronization mechanisms like semaphores.

The use of lossy compression would allow connections over lower-bandwidth media like Bluetooth or IrDA. One algorithm worth looking into would be ogg vorbis[3] as it has no licensing fees. Also one would have to take a look at whether there is the possibility to use CoreAudio or Quicktime to handle the de/encoding. This would however increase CPU load on both server and client and add latency and would result in quality loss.

Another idea is to support the transfer of AC-3 encoded multichannel audio. This format is commonly used for the audio tracks of DVD movies. The current AudioSpace requires that the client decodes the AC-3 sound to discrete channels and sends them over the network. If the protocol were extended to support non-PCM formats, the client could simply send the unencoded AC-3 stream over the network, saving CPU and bandwidth, and let the server handle the decoding. The AudioSpace server would then act like a digital amplifier for home cinemas.

[3]http://www.vorbis.com/

A larger project would be extending the applications and the protocol to a two-way communication, allowing to share not only audio outputs but also audio inputs. Such a system could be used to build distributed audio applications with each node in the network acting as an autonomous signal processing unit. That way, complex signal processing that exceeds the power of a single computer could be realized. Modular programs like MAX/MSP[4], pd[5] and jMax[6] or a synthesis language like SuperCollider[7]could be a good base for such applications.

7.5 Conclusion

Where previous systems for networked audio relied on specialized software and were unable to provide low latencies and high audio quality, the AudioSpace software is able to deliver latencies low enough for demanding interactive applications and is compatible with legacy software through a tight integration in the host operating system. The use of system-standard configuration tools that are familiar to the user and self-configuring networking does not require the user to learn new concepts.

The use of system-provided audio components and a new skew compensation algorithm ensure high sound quality, low latency and efficiency. By building on standard networking protocols, the system is independent of the physical components of the network and is ready to be used with upcoming network technologies.

CoreAudio has turned out to be a very comfortable and powerful environment for developing real-time audio applications, supported by the good performance of the underlying operating system. The Cocoa framework with the Interface Builder and XCode development tools as well as the rich AudioUnit library in the CoreAudio framework allowed a rapid development process. While the current AudioSpace software certainly has room for improvements, the performance is very pleasing and encourages future work in that direction.

[4]http://www.cycling74.com/products/maxmsp.html
[5]http://www.pure-data.org/
[6]http://www.ircam.fr/produits/logiciels/jmax-e.html
[7]http://www.audiosynth.com/

Appendix A

User space drivers in Mac OS X

The CoreAudio SDK contains a header file called “AudioHardwarePlugin.h” which describes itself as an “API for the CFPlugIn that implements an audio driver for the HAL from user space”. Outside the comments in this file, there is no documentation available and Apple did not provide any sample code. As a user space driver could have been very useful for the AudioSpace, the undocumented AudioHardwarePlugin API was examined closer. In order to provide some guidance for readers that are interested in implementing an AudioHardwarePlugin, the results of that investigations are printed here.

The AudioHardwarePlugin contains an interface specification for a CFPlugin, OS X’ API for application plug-ins. The HAL will load plug-in bundles that implement the AudioHardware interface when they are placed in the /Library/Plug-ins/Audio/HAL directory. Mac OS X 10.3 by default installs a driver for the iSight camera’s sound hardware in that folder, which provides a good reference for how an AudioHardwarePlugin’s Info.plist file should look like.

The interface contains a number of calls of which most are also present in the HAL’s AudioHardware.h header file. In fact, an AudioHardwarePlugin looks like the HAL to applications and applications will be using it as if it were the HAL. The HAL documentation serves as a good reference for how exactly the AudioHardwarePlugin will be accessed by applications.

In contrast to kernel space drivers, AudioHardwarePlugins cannot take advantage of the HAL’s conversion or mixing services. They will need to implement these services by themselves, which makes them more complex than kernel space drivers.

AudioHardwarePlugins are instanced for every application using the HAL and are being run in the respective application’s address space. Improper AudioHardwarePlugins therefore can easily irritate applications. Also, many applications respond with crashes to incomplete implementations of the AudioHardwarePlugin interface. It appears that many applications make assumptions about how the HAL will behave and were not programmed to cope with other implementations of the AudioHardware API than the HAL.

Running as user space applications, AudioHardwarePlugins don’t have nearly as precise timing as real hardware or kernel drivers have. Especially under heavy CPU load, timers inside the AudioHardwarePlugin can become very unreliable.

It turns out that the AudioHardwarePlugin API is not well-suited for the AudioSpace: The timing problems were a serious issue when using more complex applications on less powerful computers. Comparable kernel audio drivers were much more reliable in these situations. Also, the lack of the HAL's mixing services would either require extra efforts to implement one's own mixing routines which then would need to communicate with all the instances of the AudioHardwarePlugin or each AudioHardwarePlugin would send a separate audio stream to the network, resulting in high traffic and high load on the server side. An additional problem would be the application compatibility: Since each application would then talk to the AudioHardwarePlugin directly, the AudioHardware-Plugin would need to be tested with a lot of software titles and even after passing all these tests full compatibility with future applications could not be guaranteed.

Due to this shortcomings, it was decided that the AudioSpace software will not use the AudioHardwarePlugin API but instead use the traditional API for kernel space audio drivers.

Appendix B

Clock skew compensation

Many suggested solutions for the problem of clock skew compensation, like [25] and [26], rely on timestamped network packets. Each audio packet gets a sending timestamp s_i in the sender's local time and the receiver takes timestamps a_i on every packet's arrival. A comparison of both local and remote timestamps to the corresponding previous timestamps s_{i-1} and a_{i-1} is being used to calculate an estimate of the clock deviation [26][1]:

$$e_i = \frac{a_i - a_{i-1}}{s_i - s_{i-1}} \tag{B.1}$$

Since network latencies and operating system schedulers add jitter, the estimate is being smoothed by taking the mean average $\hat{e}$[26]:

$$\hat{e}_i = \hat{e}_{i-1} + \frac{e_i + \hat{e}_{i-1}}{a} \tag{B.2}$$

with a being the smoothing factor. The smoothing factor directly impacts the speed in which the average reacts to fluctuations: Low values reduce the filtering effect, letting jitter influence the averaged value where large values stabilize the average but make it less reactive. The smoothing factor depends on jitter and the buffer size and needs to be determined by experiments[2]. The resulting smoothed average gives an estimation of the clock skew, with a value < 1 indicating that the sender's clock runs faster than the receiver's clock, a value > 1 indicating that it's slower and a value equal to 1 suggesting equal clock speed. Based on this estimated clock skew, the receiver has to insert or remove frames to avoid a buffer under- or overflow. The number of frames that needs to be inserted or removed per second depends on the sample rate:

$$n = samplerate * (1 - \hat{e}_i) \tag{B.3}$$

The simplest way of doing that is dropping or repeating random packets with undesirable consequences: Repeating or dropping buffers in a continuous audio signal will be audible as stutter, the loss of continuity in the modified signal creates audible cracks. According to [25], inserting/removing single frames at regular or irregular intervals creates audible artifacts attributable to phase discontinuity. The alternate approach suggested in [25] is scanning the buffer for similar

[1]The formulas are derived from the C source code in the original source.

[2]is using a smoothing factor of 16 for a packet rate of 26ms-1 and additionally applies a clamping function to restrict the change from one iteration to the next to a maximum of 10%.

passages which then are duplicated or dropped without strong discontinuities. This algorithm requires buffers that are large enough to ensure that such stationary fragments can be found in the buffer. The authors used a buffer of 200ms for their implementation that according to their paper worked well for compensating large clock skews in streams of voice recordings or pop music, but had audible artifacts with classical music. I chose not to use this algorithm for the AudioSpace as the goal of achieving latencies < 20ms while retaining high quality on complex signals is not possible.

In [26], sampling rate conversion is being used to compensate clock skew: The clock skew is estimated using formula B.1 and formula B.2 and used as the conversion rate for a sample rate converter that gets its input from the stream buffer and sends its output to the audio hardware. Given the case that calculated clock skew equals the real clock skew, the sample rate converter will prevent buffer over- and underflows and the number of frames the sample rate converter reads from the buffer will be equal to the number of frames that are received over the network. As long as the sample rate variations are low enough, the change in pitch produced by the sample rate conversion will not be audible, and with a sample rate converter that is using a good enough algorithm, no audible aliasing will be present. Unlike the previous algorithm, this will not depend on the buffer length or the signal complexity, making it much more universally applicable.

Unfortunately, there are two points where I see flaws in this approach: First, there is the assumption that the estimated clock skew will eventually converge precisely to the real clock skew. Since both the timestamps that go into the equations nor the the math unit of the computers processor are limited in their precision, the estimated clock skew may be slightly off the real clock skew, causing the sample rate converter to be slightly off by a few frames. The magnitude of the imprecision will be insignificant in most situations, but since the difference between estimated and real clock skew adds up over time one cannot rule out that a buffer under- or overflow may eventually happen over long periods. The second flaw I see in that approach is the lack of feedback: The estimated clock skew is the only input variable of the process, and any variations or errors of that will directly affect the result. The proposed method does its best to compensate the cause of buffer over- or underflow, but is unable to detect if it does actually prevent the symptoms.

The general idea of the algorithm that was finally developed for the AudioSpace originates in a different view on the whole problem: While the clock skew is the initial source of the problem, it is not directly the (usually inaudible) timing differences that bothers us but the consequence, the eventual buffer over- and underflows. If once can keep the buffer queue length at a constant level without audible artifacts, the problem is solved, without knowing the exact clock skew.

Instead of trying to quantify the timing differences, it was chosen to simply monitor the buffer queue length and change the playback speed in a way that'd drive the queue length towards a predefined length. If the queue is too short, the playback must slow down to remove less samples from the buffer than are received during the same time. If the queue is too long, the playback must accelerate to take more samples from the buffer than are received (figure B.1). This can be achieved with a resampling component between the buffer and the audio hardware, similar to [26], resulting in minimal artifacts regardless of the buffer size or the complexity of the signal. In experiments, it was found that simply switching back and forth between two playback speeds, one below the actual sample rate and one above the actual sample rate, was sufficient to keep the buffer queue length in a safe range. While momentarily, the playback speed would always be wrong, in the long run the average speed would match the actual clock skew. Crucial is the difference between the two speed changes: A too small difference would not compensate for much clock skew, where a too large one results in audible pitch shifting.

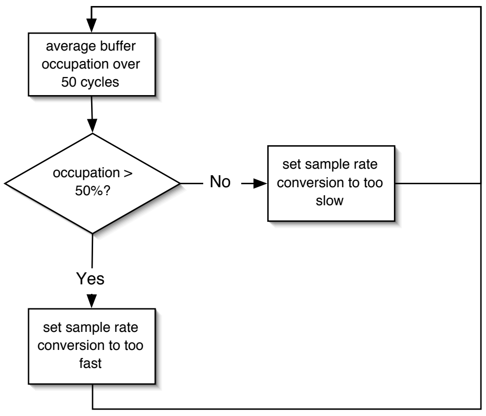

average buffer occupation over 50 cycles
occupation > 50%?
No
set sample rate conversion to too slow
Yes
set sample rate conversion to too fast

Figure B.1: skew compensation algorithm

In psychoacoustics exists the notion of the *just noticeable difference* or JND that describes how large a pitch change can be until the human ear notices it.[9] The precise value depends on the volume and frequency of the signal, but a rule of thumb says that changes of up to 0.25% are unnoticeable[27][3] Experiments with the AudioSpace showed that an abrupt change of 0.3% could still be noticed on some signals like a sine tone. Reducing the difference to 0.1% made that effect disappear.

The algorithm was implemented in the AudioSpace server and tested with various iBook, PowerBook and PowerMac computers. The algorithm was tested by monitoring the the buffer queue length of the server while playing sound from a remote computer. Figure B.2 shows a transcript of two tests over the duration of one minute. One test was conducted without any skew compensation, the other was done with the compensation algorithm described above. The diagram shows how the uncompensated buffer queue is steadily growing due to the sender sending faster than the receiver is processing the signal. As a result, two buffer overflows occurred during the test period. In comparison, the compensated buffer stays at a constant buffer length over the whole duration. Since the skew is completely compensated, the implementation is invulnerable to long term deviations.

[3]Usually, the JND is quoted as 4-5 cents[27]. Cents are a unit used in music and psychoacoustics where 100 cents equal one semitone or $\frac{f2}{f1} = 2^{\frac{cents}{1200}}$.

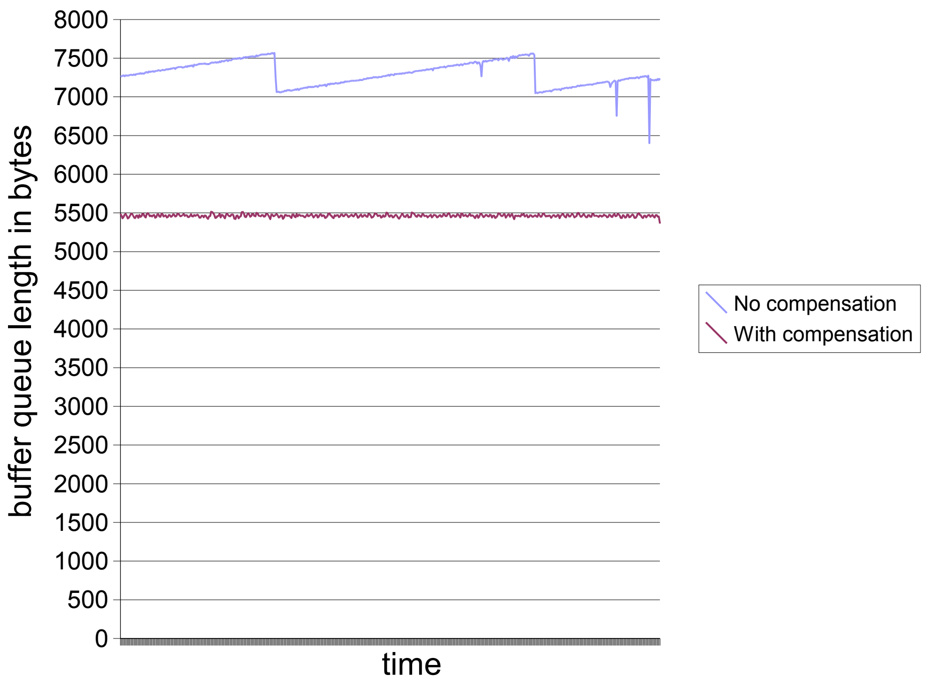
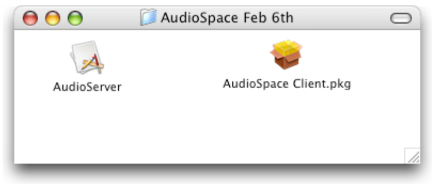


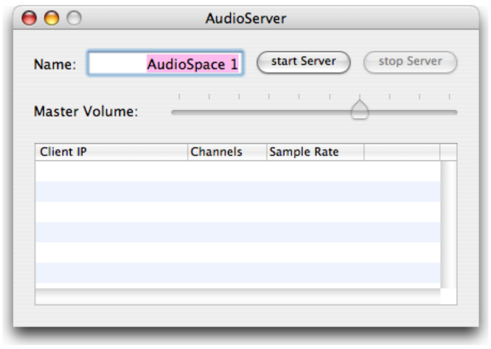
Figure B.2: buffer queue length

Appendix C

An illustrated walk-through

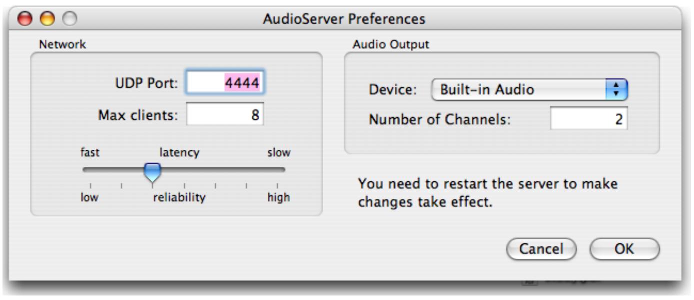


The AudioSpace distribution contains two files: One is the server application, the other is the installer package for the client(s). The server application can be copied anywhere on the server's hard drive, where the client is installed by double-clicking the package.



Name: AudioSpace 1

After launching the server application, the main window appears. In its default state it offers a text field to enter a name for the server and a button to start the server.



UDP Port: 4444

Max clients: 8

Device: Built-in Audio

Number of Channels: 2

You need to restart the server to make changes take effect.

The user can access the AudioSever's preferences through the application menu. The preference dialog offers controls to set network and audio options. The network options are:

UDP Port This is the port on which the AudioServer will listen for incoming connections. Unless there are other network services running on that computer that use the default port, this does not need to be changed.

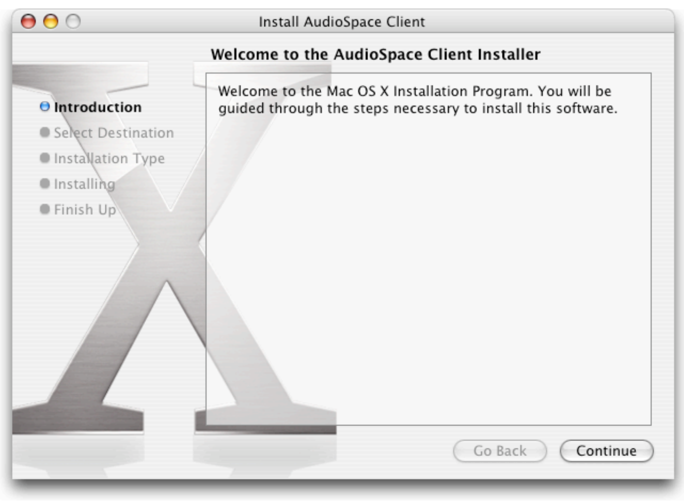
Max clients As the CPU load on the server increases with every client that connects, the user has the ability to set a limit on the number of connections that the server accepts.

latency/reliability slider This slider controls the size of the network buffer. As one cannot expect that the user knows about the technical details behind the buffer size, the slider is labeled with the actual consequences of the buffer size. A smaller buffer size leads to a shorter latency but a higher probability of drop-outs on busy networks, where a higher buffer makes the stream more tolerant for other network traffic, but increases latency.

The audio settings are:

Device This pop-up menu offers a list of the audio output devices that were detected in the system. The user can choose here which she wants to use for output.

Number Of Channels This value sets the number of channels that the server is sharing over the network. This is useful if not all of the output channels of the selected device are connected to speakers. The maximum number of channels that can be shared is 16.

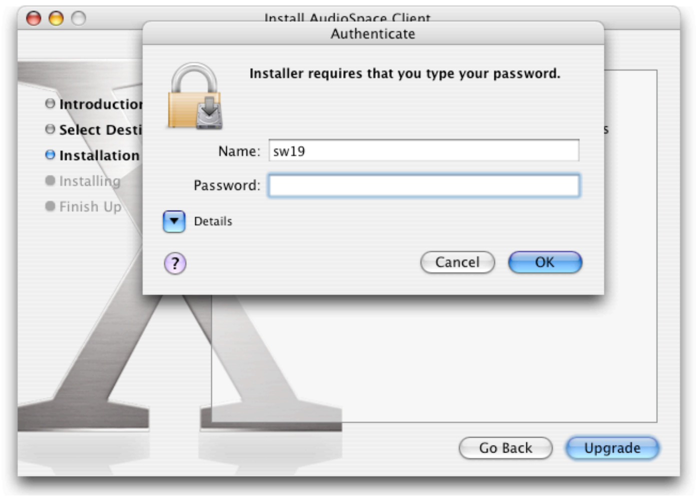


Install AudioSpace Client

Welcome to the AudioSpace Client Installer

Welcome to the Mac OS X Installation Program. You will be guided through the steps necessary to install this software.

On the client, double-clicking the package launches the Mac OS X installer which will install the necessary files.



Install AudioSpace Client

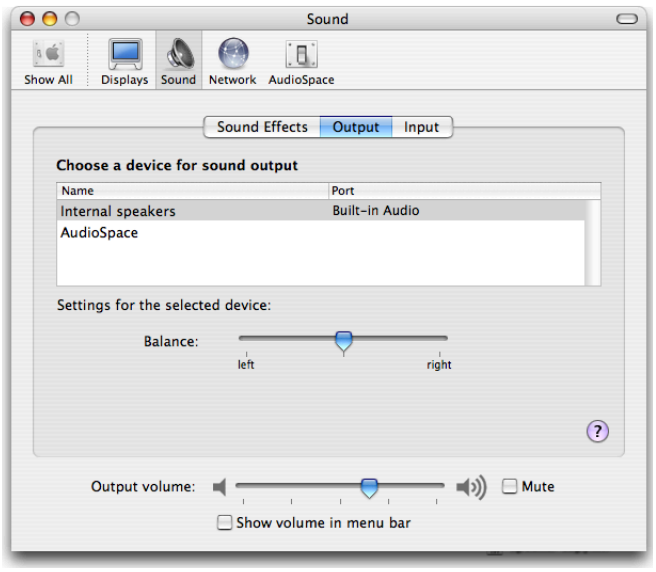
Authenticate

Installer requires that you type your password.

Name: sw19

Password:

To installer requires an administrator's password in order to install the kernel drivers. After the installation, a reboot is required.



Sound

Choose a device for sound output

Name	Port
Internal speakers	Built-in Audio
AudioSpace	

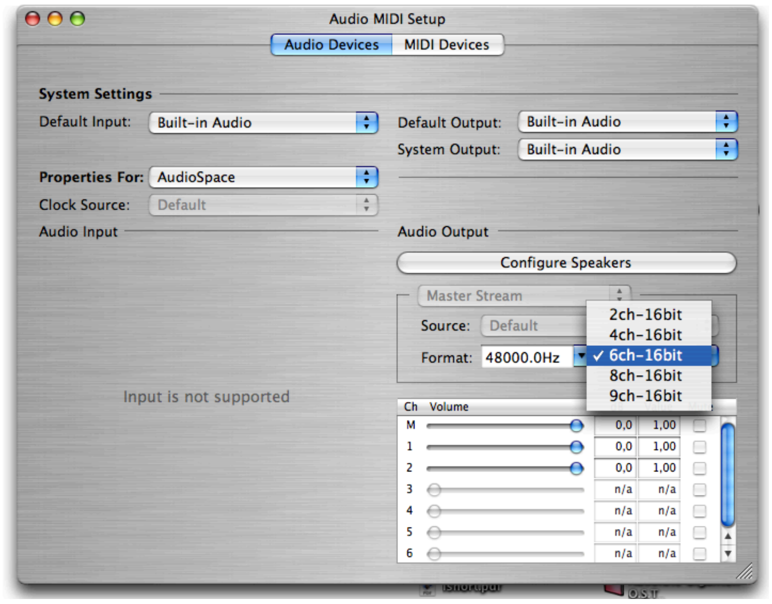
Settings for the selected device:

Balance: left right

Output volume: Mute

Show volume in menu bar

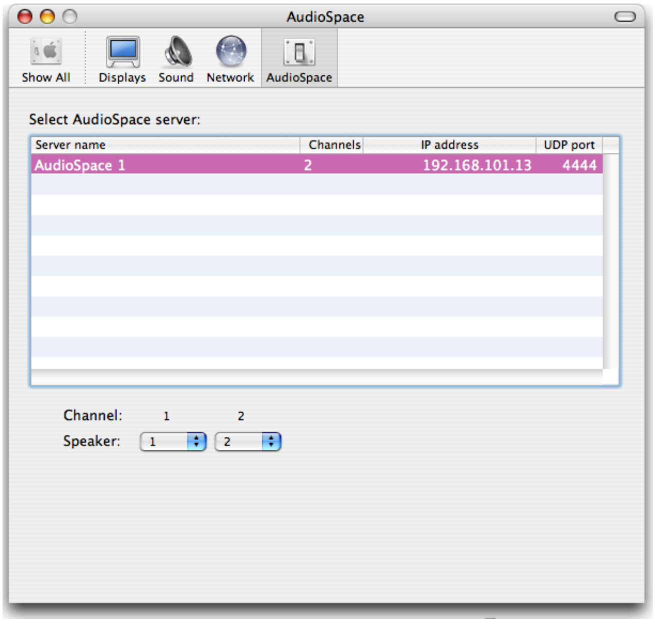
After the reboot, an additional AudioSpace device is available in the system's audio preferences.



System Settings
Default Input: Built–in Audio
Default Output: Built–in Audio
System Output: Built–in Audio
Properties For: AudioSpace
Clock Source: Default
Audio Input
Input is not supported
Audio Output
Master Stream
Source: Default
Format: 48000.0Hz 6ch–16bit

Ch	Volume			
M		0,0	1,00	
1		0,0	1,00	
2		0,0	1,00	
3		n/a	n/a	
4		n/a	n/a	
5		n/a	n/a	
6		n/a	n/a	

Using the AudioMIDISetup, the user can set the number of channels that she wants to stream to an AudioSpace server and the sample rate of the AudioSpace device. Higher values cause higher CPU load on both the client and the server as well as a higher network traffic. Therefore, it's recommended to set this to conservative values.

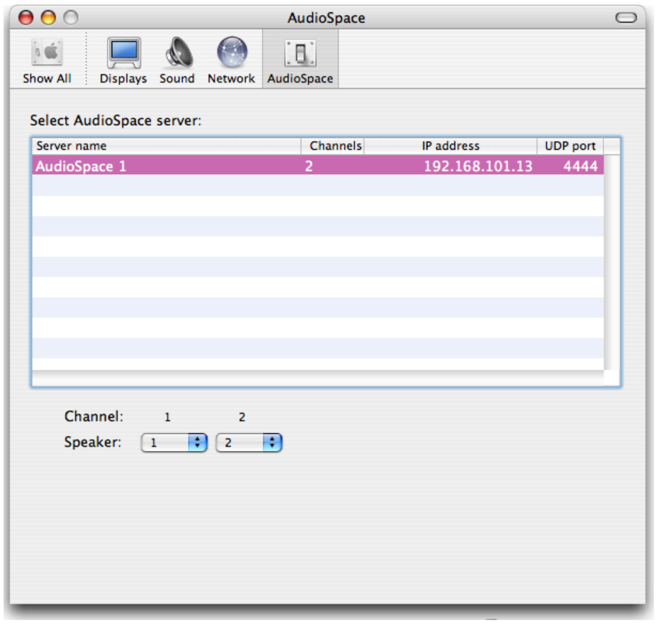


Select AudioSpace server:

Server name	Channels	IP address	UDP port
AudioSpace 1	2	192.168.101.13	4444

Channel:	1	2
Speaker:	1	2

The system preferences now include an extra icon for the AudioSpace driver configuration.

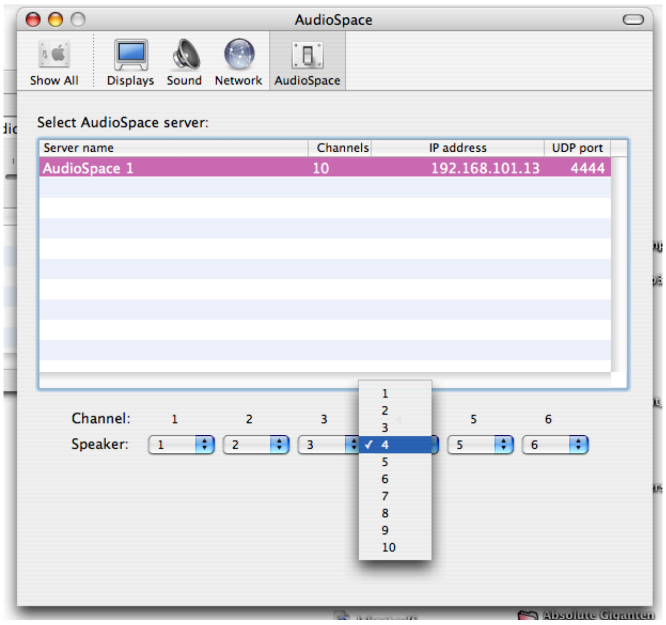


Select AudioSpace server:

Server name	Channels	IP address	UDP port
AudioSpace 1	2	192.168.101.13	4444

Channel: 1 2
Speaker: 1 2

After clicking it, the AudioSpace preferences panel launches. The panel lists all the AudioSpace servers it could find on the network and allow the user to select one.

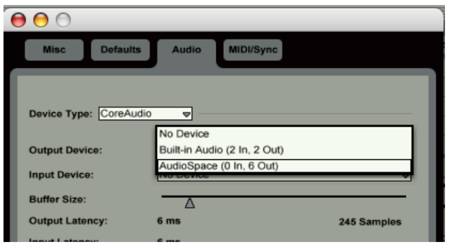


Select AudioSpace server:

Server name	Channels	IP address	UDP port
AudioSpace 1	10	192.168.101.13	4444

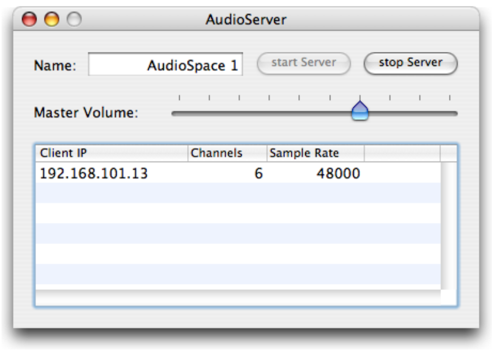
Channel: 1 2 3 5 6
Speaker: 1 2 3 4 5 6

After selecting a server, the user can assign local channels to remote speakers.



Device Type: CoreAudio
Output Device:
Input Device:
Buffer Size:
Output Latency: 6 ms 245 Samples

Any audio application on the client will now be able to use the AudioSpace.



Name: AudioSpace 1

Client IP	Channels	Sample Rate
192.168.101.13	6	48000

The server will list all the connected streams with their IP address, sample rate and number of channels. Once running, the server now offers controls to set the global volume and to stop the server. Note that it's not possible to change the name of a running server.

Bibliography

[1] Brad Johanson, Armando Fox, Terry Winograd: *The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms* (2002)

[2] Jan Borchers: *The Aachen Media Space* (2003)

[3] Jef Raskin: *Das intelligente Interface* (2001)

[4] Jeff Johnson: *GUI Bloopers*

[5] Wessel, D. and Wright, M.: *Problems and Prospects for Intimate Musical Control of Computers.* proceedings of CHI 2001

[6] K. MacMillan, M. Droettboom, I. Fujinaga: *Audio Latency Measurements of Desktop Operating Systems* (2001)

[7] Harold Nyquist: *Certain topics in telegraph transmission theory* (1928)

[8] Claude Shannon: *Communications in the presence of noise* (1949)

[9] Curtis Roads: *The Computer Music Tutorial* (1996)

[10] Horst M. Eidenberger: *Medienverarbeitung in Java* (2003)

[11] Dr. Franz-Joachim Kauffels: *Lokale Netze*, 11th edition (1999)

[12] *RFC-791: Internet Protocol* (1981)

[13] *RFC-793: Transmission Control Protocol* (1981)

[14] *RFC-768: User Datagram Protocol* (1980)

[15] *RFC-1180: A TCP/IP Tutorial*, (1991)

[16] Jörg Rech: *Datenschalter - Die Technik von LAN-Switches*, c't magazine 18/2002 p.208-213

[17] Roman Beilharz: *Im Netz der Klänge*, c't magazine 21/2003, p.188-191

[18] *RFC-1122: Requirements for Internet Hosts – Communication Layers* (1989)

[19] Apple Computer, Inc: *http://developer.apple.com/macosx/rendezvous/* (2004)

[20] H. Juszkiewicz, N. Yeakel, S. Arora, A. Beliaev, R. Frantz and J. Flaks: *Media-accelerated Global Information Carrier Engineering Specification* (2003)

[21] Digigram: *Digigram EtherSound - Audio Distribution over Standard Ethernet* (2002)

[22] Apple Computer, Inc: *Inside Mac OS X: System Overview* (2002)

[23] Apple Computer, Inc: *Audio and MIDI on Mac OS X* (2001)

[24] Apple Computer, Inc: *http://developer.apple.com/documentation/UserExperience/Conceptual/Preference*
(2004)

[25] Orion Hodson, Colin Perkins, and Vicky Hardman: *Skew detection and compensation for internet audio applications* (2000)

[26] R.Akester, S.Hailes: *A new audio skew detection and correction algorithm* (2002)

[27] J. O. Pickles: *An Introduction to the Physiology of Hearing.* (1982)

[28] Peak Audio, Inc: *Audio Networks An Overview* (2001)

Index