

# **A Virtual Audio Driver for the Internet Speaker**

Ishan Mandrekar  
and  
Vassilis Prevelakis

Technical Report DU-CS-02-05  
Department of Computer Science  
Drexel University  
Philadelphia, PA 19104  
December 2002

# A Virtual Audio Driver for the Internet Speaker<sup>\*</sup>

Ishan Mandrekar and Vassilis Prevelakis

*Computer Science Department  
Drexel University  
{ishan, vp}@drexel.edu*

## Abstract

We have developed the “Internet Speaker” (IS), a network-enabled single board computer embedded into a conventional audio speaker. Audio streams are transmitted in the LAN using multicast packets, and the IS can select any one of them and play it back. A key requirement for the IS is that it must be capable of playing any type of audio stream, independent of the streaming protocol, or the encoding used.

We achieved this by providing a streaming audio server built as a virtual audio device (VAD) in the OpenBSD kernel. The VAD accepts input from any of the existing audio file players, or streaming audio clients. Since all audio applications have to be able to use the system audio driver, our system can accommodate any protocol or file format, provided there exists some compatible player running under OpenBSD.

In this paper we discuss the design and implementation of the server as a VAD, the streaming protocol developed for this application, and the implementation of the client.

**Keywords:** virtual drivers, OpenBSD, audio, multicast.

## 1 Introduction

Audio hardware is becoming increasingly common on desktop computers. With the advent of easy to install and cost-effective networking solutions such as Ethernet and more recently, Wireless Local Area Networks, there has been an increasing interest in allowing audio sources to be shared by multiple devices. One central server can stream audio data to a number of clients all

part of the same network. The clients just need to capture the streamed data and play it on their audio devices (Figure 1). This concept can work at home, as well as a campus or office LAN where it can be used as the basis for wireless surround audio systems, background music, public announcements, etc.

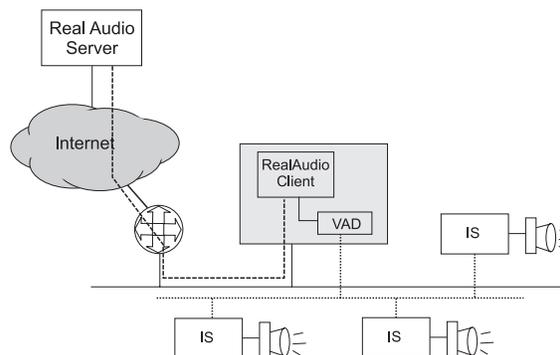


Figure 1: Re-broadcasting the audio from a remote server in the LAN.

However, the popularity of audio services in the Internet has resulted in the deployment of a large number of protocols and data formats producing an on-line Babel of mismatched applications and protocols. Moreover, most of the popular streaming services have adopted incompatible and proprietary formats. It is particularly noteworthy that even though Real Networks released the source code for Helix Client, they did not provide source for the RealAudio G2 and RealAudio 8 decoders.<sup>1</sup> In addition to the streaming audio formats, there are also a large number of encoding schemes for file-based audio players (MP3, wav, and so on).

In order to be able to cope with all these formats, users have to use multiple audio playback applications, each with its own user interface and peculiarities.

<sup>\*</sup>This work was supported by NSF under Contract ANI-0133537.

<sup>1</sup><https://www.helixcommunity.org/2002/intro/client>

Moreover, most commercially available streaming servers cannot stream audio files in their native formats. These files need to be “hinted” with metadata which aids the streamer in generating Real Time Protocol (RTP) [3] packets.<sup>2</sup> This mandates that the audio files undergo a format conversion before they become eligible for streaming.

Our goal was to produce an audio server that would be independent of the format of the audio source. We also endeavored to create a single client that would be able to accept music from any of the above players without the need for specialized plug-ins or updates.

In the next section we discuss the issues surrounding the design of the audio client and the streaming server. We, then, describe our prototype implementation which allows the redirection of the output of off-the-shelf audio player applications using a virtual audio driver. Finally, we discuss how our work compares with related projects and systems, and our plans for future enhancements.

## 2 Design

### 2.1 The Server

A typical audio device [7], as shown in Figure 2, consists of two logically independent systems, one for recording, and another for playback. In the recording section, analog input is first passed through a mixer before being fed to an Analog to Digital Converter (ADC), whose output is subsequently stored in a buffer. The audio hardware dictates various configuration parameters such as resolution (8 bits or 16 bits), data formats (linear or  $\mu$ -law), channels (mono/stereo) and the sampling rate.

The playback section makes use of a similar buffer to hold data that is to be passed to the Digital to Analog Converter (DAC). The output of the DAC is channelized through a mixer before being sent to the speakers. The operating system also provides format conversions to supplement the native features supplied by the hardware.

An audio playback application like `mpg321(1)`, works by continuously reading blocks of audio information from the source (file, or network connection), performing some processing and sending the result to the au-

<sup>2</sup><http://developer.apple.com/techpubs/quicktime/qtdev-docs/RM/frameset2.htm>

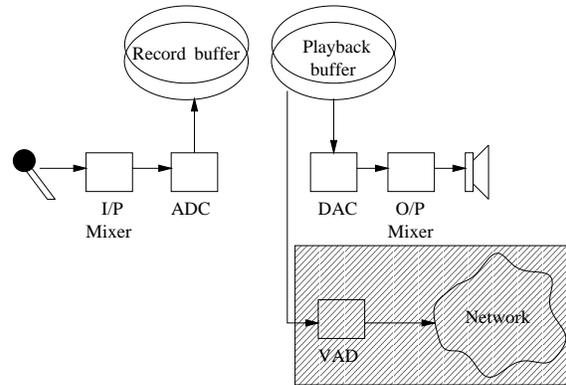


Figure 2: Typical Audio Device with VAD extension

dio device for playback. This is done by writing blocks of data to the audio special device (`/dev/audio`). This data is placed in the playback buffer. The device driver further divides the playback buffer into smaller size blocks before passing the data to the audio hardware. I/O control (`ioctl(2)`) calls are used by applications to ask the device driver about the capabilities of the underlying hardware, or to set various parameters for the audio device (e.g. sampling rate).

The audio device driver in the OpenBSD 3.1 kernel is divided into a high level hardware independent layer, and a low level hardware dependent layer. The high-level driver interacts with user-level applications. It provides a uniform application programming interface to the underlying hardware dependent driver modules.

Low-level drivers provide a set of function calls [2] which allow the hardware to be integrated into the runtime environment of the operating system. A device driver does not need to implement all the calls, but at a minimum a useful driver needs to support opening and closing the device, device control and configuration, and I/O calls.

This observation led us to the design of a *virtual* audio driver that replaces the traditional audio driver and associated hardware, sending the audio information to the network (grey box in Figure 2).

#### 2.1.1 The Virtual Audio Driver

Instead of writing a user level application using streaming protocols (e.g. RTP) and CODECs for complex encodings such as MP3 (MPEG Layer 3), we have added a new virtual audio interface (VAD) to the OpenBSD op-

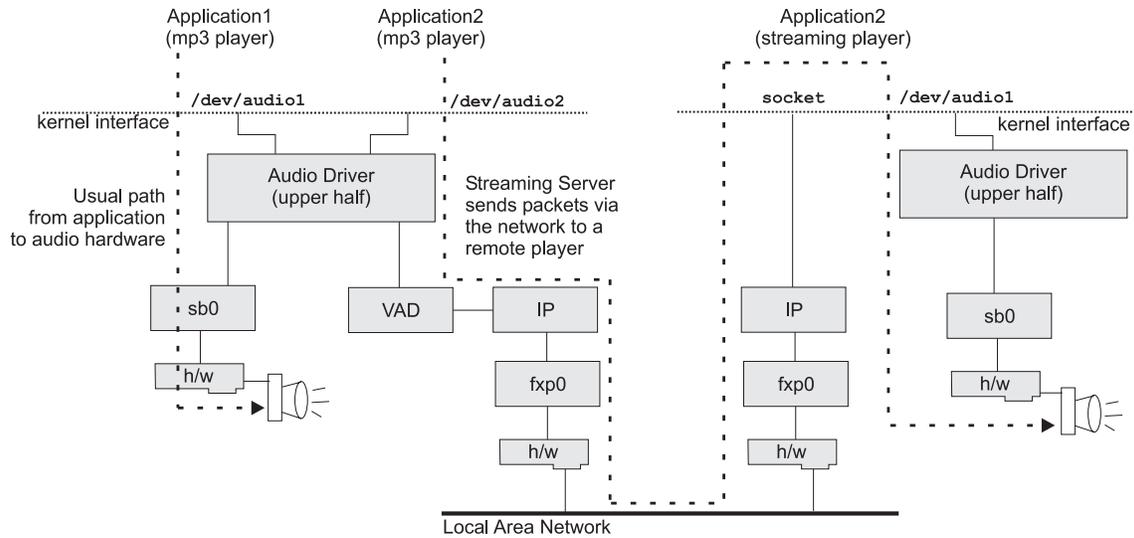


Figure 3: Application 1 plays audio to the normal audio device, while Application 2 plays to the VAD, which sends the audio data to a remote machine via the network.

erating system. The VAD packetizes and multicasts the audio play buffers onto the network.

The design of the Virtual Audio Driver (VAD) was heavily influenced by the design of the pseudo terminals (*ptys*) that first appeared in BSD 4.2.

The reason for the wide adoption of the *ptys* is that, as far as the application program is concerned, they are indistinguishable from a real terminal. They implement every nuance of the terminal interface even including settings such as the communication speed that are meaningless to a virtual device. This allowed applications (shells, editors, games, etc) to be used on *ptys* without any modification to the programs or to the way they were used.

In a similar manner, we endeavored to produce a virtual audio driver indistinguishable from a real audio device to the application program. The veracity of the emulation is due to the fact that, to the operating system, the VAD looks like the driver of a real audio device (Figure 3). The VAD responds to the configuration request (probe) at boot time and is thus assigned a minor device number so that application programs can address it.

By using the virtual driver technique, we provide an audio streaming facility that does not depend on the audio application, or the encoding format. Moreover, the music file does not need to be in a special “*hinted*” format unlike the Darwin Streaming Server.<sup>3</sup> This enables the

VAD server to accommodate new audio encoding formats as long as a player has been ported to OpenBSD.

Another benefit of this approach is that the machine that produces the audio stream (server) does not need to have an audio card and can support multiple instances of the VAD allowing multiple clients to generate separate audio streams.

Each VAD device is bound to a specific multicast address (and port). Once the program opens the VAD device and starts writing to it, the VAD will begin sending packets to the network.

## 2.2 The Client

Our client, which we call “The Internet Speaker” (IS) is a rather simple device, comprising a single board computer (SBC) running an embedded version of OpenBSD, a network connection (preferably wireless), an audio card, and a pair of speakers. The IS receives music from the network and plays it on its speakers. Since Drexel has a campus-wide network, the IS can receive music anywhere within the our campus.

Being an embedded device, the IS has limited resources, so it was considered important to standardize the format of the incoming streams. Another constraint is synchronization; if more than one ISs are playing the same channel within earshot, any delay between the two out-

<sup>3</sup><http://developer.apple.com/darwin/projects/streaming>

puts creates cacophony. Finally, since we expect a large number of ISs to be deployed in our campus, we had to minimize the impact of the ISs on the network resources.

These three constraints made the use of an off the shelf player infeasible. We briefly experimented with the Real Time Audio Tool [6] but it did not allow us to convert, for example, Real Audio streams into something the IS could play.

In the end we decided to use multicast packets for the streaming audio with one address per channel. This achieves synchronization between playing ISs and reduces the load on the network. We then “pipe” the output of existing player applications into the network. In this way we do not need to have the players in the ISs but in one or more audio servers. By removing the players we can also standardize the user interface of the ISs. Finally, we redirect the packets received at the ISs to their corresponding audio devices without any post-processing. This reduces the processing power requirements at the ISs and achieves format standardization of the incoming streams.

To avoid burdening the server, we also require that the IS selects which channel to play to, without having to make arrangements with the server.

## 2.3 Protocol Design

The communication protocol between the audio streaming server and the ISs has to be lightweight and easy to implement. On the server side all the protocol is running in kernel space, so a simple design decreases debugging (and kernel reboots). Also, a simple protocol imposes a lesser burden on the resources on the client side, allowing lightweight clients to be used.

Given the real-time nature of the transmissions, and the continuous nature of the audio stream, there is no need for retransmissions or error correction. Incorrect packets are simply discarded. This is acceptable in our target environment which, being a LAN, exhibits low packet loss rates.

Clients may join a transmission at any time and are able to recover from lost packets or transient network problems. This allows the server to be oblivious to the number or type of the clients operating in the network.

Finally, we use multicast packets to reduce network load and synchronize the clients.

## 3 Implementation

### 3.1 The Streaming Server

When a client application accesses the audio device (via the `open(2)` call), the data flow to the network *via* the VAD. This is done by having the VAD open a kernel UDP socket. The destination address in the socket’s address structure is set to a predefined multicast address and destination port.

Every time data gets written into the play buffer by a user-level application, it is packetized and sent to the network in the following manner: a special control packet containing information to configure the audio device at the client side precedes the transmission of the actual data. The contents of the play buffer are then sent as one or more 1024 byte packets.

The socket is finally closed when the user process terminates the connection with the audio driver.

As we mentioned earlier, the approach we adopted is similar to the use for the Unix pseudo terminals (*ptys*). As in the case of *ptys*, we provide a new low-level driver that looks like a driver for a real audio card. The difference with the *pty* implementation is that pseudo terminals allow the I/O to the *pty* to be directed to another user-level process. Although this technique is more flexible and allows better handling of the audio stream, we decided not to adopt it as it is particularly wasteful in terms of resources. Since the main application of the VAD is the redirection of audio streams to the network, directing the stream to a user-level process which then transmits it to the network would involve superfluous data copying and context switching.

We, therefore, decided to inject the audio byte stream directly into the network as a series of multicast packets. This allows the VAD to construct the packets and forward them to the networking code entirely within the kernel. This minimizes copying and sends the packets without another context switch.

In our current prototype, the multicast address is defined during kernel generation, but in future releases we will allow this to be defined via an `ioctl(2)` or a `sysctl(8)` call. We envisage an interface similar to the `ifconfig(8)` call that allows the address and various parameters to be defined as well as enabling, or disabling the device.

Synchronising the audio output of two devices is not as simple as ensuring that we send the audio data to both devices at the same time. Variations in the routing topology (e.g. different number of hops between the server and the various receivers), Operating System actions (e.g. interrupt processing, scheduling the client process, etc.) may result in loss of synchronization. The human ear is sensitive to the resulting phase differences, so we were careful to create an operational environment where such conditions are unlikely to occur.

We have standardized the configuration of the clients (hardware, operating system, applications), and our network topology follows a logical design where machines in the same area are normally connected to the same Ethernet segment. The only case where this is violated, is where we have wireless clients next to a client connected to the wired LAN.

### 3.2 The Client-Server Protocol

The client must select a multicast stream and then start playing the audio. The design of the protocol is such that the client never contacts the server. We achieve this by interspersing control packets between the normal data packets. The control packets (Figure 4) contain sufficient information (sampling rate, bit precision, encoding, channels etc.) for the client to configure its own audio hardware so that it can process the data sent by the server. The data in the ordinary packets is sent directly to the audio device via the standard interface (`/dev/audio`).

CTRL	SAMPLING RATE	ENCODING	PRECISION	CHANNELS
------	---------------	----------	-----------	----------

Figure 4: Control Packet

Multimedia streaming servers make use of the RTP control protocol (RTCP) to obtain feedback on the quality of data distribution. The protocol consists of periodically transmitting control packets (sender and receiver reports) to all participants in the session, using the same distribution mechanism as the data packets. This feedback may be directly useful to the server for control of adaptive encoding schemes, or alternatively may be used to diagnose faults in the distribution. Sending reception feedback reports to all participants helps in analyzing whether problems are local or global.

The VAD server transmits the audio data in its raw (uncompressed) form over a relatively benign LAN envi-

ronment, where network resources are plentiful. This does not justify incurring the extra overhead that feedback entails, given that the probability of congestion, or other network problems is low in a LAN. Also, the feedback traffic imposes a burden on the server.

Clients can join or leave the multicast groups anytime without notifying the server. The server remains oblivious of the number of clients “tuned in”, and its performance does not degrade as the number of the client increases. This makes the application very scalable and it can be used across campus or office networks.

### 3.3 The Audio Client

We have constructed a sample client that receives packets from the network and plays them back via the local audio device. The client is a simple program that is designed to run on a variety of platforms (our current implementation runs under OpenBSD and RedHat Linux 7.3). By far, what makes this client portable is the fact that it only has to deal with a simple protocol and a single audio encoding scheme. We expect to be able to support most popular platforms within the next couple of months.

We make use of the multicast API of the `pnet6` [1] library to obtain membership of the multicast group that the server is transmitting to. The client program takes two command line arguments, namely the IP address of the multicast group and the destination port number that the server is transmitting to. It then creates a UDP socket and executes a `JOIN` command. If the join is successful, the client becomes a member of the multicast group and can now start receiving all the traffic destined to the group.

An important prerequisite for this to work is that the network infrastructure supports the IGMP protocol for multicast group management. [8] Ethernet networks support this protocol by default, however, when the network includes routers, they, need to be appropriately configured. We believe that within a private network, multicasting has more chances of establishing itself, than in the Internet at large.

The client executes a call-back routine every time it receives a packet. If the received packet is a control packet, the audio parameters are configured else the packet is written to the audio device. To differentiate between audio streams with varying parameters, the client compares the configuration parameters in every control packet to

the parameters which have been previously used to setup the audio device. If there is a mismatch, it invokes the configuration setup routine before sending data to the device.

The client can be terminated at any time, by sending an interrupt signal to the running process. The interrupt handler closes all open file and socket descriptors and terminates the process.

Since the client receives raw data using the native encoding of the audio driver from the network, there is no decompression or decoding overhead incurred, thereby minimizing the load on the client. This also makes the client code extremely simple and concise since all it has to do now is to send the received data to the audio device. Thus the client can be easily ported to a variety of UNIX platforms which have a similar audio device interface. The low computing requirements make future ports of the client to hand-held devices a viable option.

The client application runs in the Internet Speaker, an embedded platform for audio playback over the LAN. The IS is a single board computer with a Pentium class CPU and 64Mb of RAM. The computer also includes a 4Mb flash memory as a boot medium, a PCMCIA slot for the network interface, and an audio card. The device boots a kernel that contains a RAM disk with the root partition and downloads its configuration from the network.

The design of the IS borrows a lot from development carried out as part of the embedded VPN gateway [9] project which used a similar embedded design for a combined Firewall and Virtual Private Network gateway.

Channel selection is currently handled by a number of pushbuttons on the front panel of the machine. The interface attempts to mimic the pre-set buttons on car radios. The IS monitors a number of “well known” multicast addresses for audio information and identifies the push-buttons that are associated with active audio streams. The user presses one of these “active” buttons and the IS begins to playback the appropriate audio stream.

This interface is rather limited and we plan to augment it by advertising available programs on a separate multicast channel. This information can include the name of the song being played and the multicast address of the channel. The user will then be able to select the desired audio track by touching the name of the channel on a touch-sensitive screen.

## 4 Evaluation

The format used by the server for the audio data is essentially the raw data that are sent to the audio device. This is usually an uncompressed stream in one of the encodings supported by the virtual audio device. This creates a rather sizable stream.

To calculate a lower bound on the stream size (assuming no compression and current audio encoding formats) let us consider a CD quality audio source. This generates 44100 16 bit samples per second in two channels. We, therefore, have:

$$44100 \times 2 \times 16bits = 1.4Mbps$$

This is just for the audio data, without counting the protocol overhead or the control packets. To calculate the actual bit-rate we played an MP3 audio file on the server that was 4582086 bytes long.

On the client we received a total of 70846 packets, of which 14231 were control packets. Most of the data packets were of the maximum size allowed by the protocol. The rest were partial, since playback buffers vary in size. The total amount of audio data received by the client was 50769054 bytes and the playback lasted 286 second.

The control packets were 71 bytes long while data packets had a protocol and framing overhead of 46 bytes. This gives:

$$\frac{((56615 \times 46) + 50769054 + (14231 \times 71)) \times 8}{286} = 1.52Mbps$$

Notice that if we send the compressed file, we will achieve about one order of magnitude reduction in the bit-rate.

Due to the use of multicasting, the network load is the same regardless of the number of clients listening in on the transmissions. Similarly, if no one is listening on a transmission, the bandwidth is wasted.

In a Fast Ethernet network such a stream consumes about 1.5% of the available bandwidth, so a small number of such streams (less than 10) would not adversely effect the operation of the network.

However, by utilizing even a trivial audio compression algorithm the bit-rate can be drastically reduced without any appreciable loss in quality. The compression is done by the server on a per-stream basis, so the incremental overhead of the compression is not significant.

On the client the costs are proportionately small, since audio processing imposes a very small computational burden. The client is assumed to have a less capable processor, and it only needs to decompress a single stream (the one that its playing), so we do not anticipate problems in that area either. We hope to have compression implemented in the next release of our streaming server and client.

Adding a simple compression scheme probably exhausts the limits of the current kernel-based design. Additional enhancements will have to be performed outside the kernel, necessitating the redirection of the audio stream to another user-level process. This is likely to increase the load to the server because of the additional context switch overhead and the copying of all this data between user and kernel space. We hope to be able to provide this functionality in a future release so that we can measure the performance hit and evaluate it against the expected benefits.

Another area of concern is that GUI-based players cannot be run in unattended mode. We currently do not have a fully satisfactory response to this problem, but we believe that as software manufacturers modify their user interfaces to accommodate persons with disabilities they will provide the hooks to automate control of their applications.

## 5 Related Work

SHOUTcast<sup>4</sup> is a MPEG Layer 3 based streaming server technology. It permits anyone to broadcast audio content from their PC to listeners across the Internet or any other IP based network. It is capable of streaming live audio as well as on-demand archived broadcasts.

Listeners tune in to SHOUTcast broadcasts by using a player capable of streaming MP3 audio e.g. Winamp for Windows, XMMS for Linux etc. Broadcasters use Winamp along with a special plug-in called SHOUTcast source to redirect Winamp's output to the SHOUTcast server. The streaming is done by the SHOUTcast Distributed Network Audio Server (DNAS). All MP3 files

inside the content folder are streamable. The server can maintain a web interface for users to selectively play its streamable content.

Since the broadcasters need to use Winamp with the SHOUTcast source plug-in, they are limited to the formats supported by Winamp. Also it is not clear whether the system supports multiple concurrent Winamp server sessions on the same machine. Moreover, the server is designed to support outbound audio transitions (from the LAN to the outside) and thus does not support multicasting. Finally, the server is tied to the Windows platform.

The Helix Universal Server from RealNetworks is a universal platform server with support for live and on-demand delivery of all major file formats including Real Media, Windows Media, QuickTime, MPEG4, MP3 and more. It is both scalable and bandwidth conserving as it comes integrated with a content networking system, specifically designed to provision live and on-demand content. It also includes server fail-over capabilities which route client requests to backup servers in the event of failure or unexpected outages.

A similar application to the VAD is the Multicast File Transfer Protocol [4] (MFTP) from StarBurst Communications. MFTP is designed to provide efficient and reliable file delivery from a single sender to multiple receivers.

The concern that messages sent by clients participating in a multicast, can flood the server is mentioned in the Multicast FTP draft RFC.[4] Under the MFTP protocol, after a file is multicast, clients contact the server to get missing or corrupted blocks of the file. MFTP aggregates these requests (NAKs) from each recipient, so that one NAK can represent multiple bad or dropped packets. The VAD design acknowledging that a small number of discarded packets is acceptable for audio transmissions, allows the clients to ignore bad, or lost packets.

MFTP also uses a separate multicast group to announce the availability of data sets on other multicast groups. This gives the clients a chance to chose whether to participate in an MFTP transfer. This is a very interesting idea in that the client does not need to listen-in on channels that are of no interest to it. We plan to adopt this approach in the next release of our streaming audio server, for the announcement of information about the audio streams that are being transmitted via the network. In this way the user can see which programs are being multicast, rather than having to switch channels to monitor the audio transmissions.

<sup>4</sup><http://www.shoutcast.com/support/docs/>

Another benefit from the use of this out-of-band catalog, is that it enables the server to suspend transmission of a particular channel, if it notices that there are no listeners. This notification may be handled through the use of the proposed MSNIP standard [5]. MSNIP allows the audio server to contact the first hop routers asking whether there are listeners on the other side. This allows the server to receive overall status information without running the risk of suffering the “NAK implosion” problem mentioned earlier. Unfortunately, we have to wait until the MSNIP appears in the software distributions running on our campus routers.

## 6 Conclusions

Our motivation for doing this work was that we felt that existing audio players presented an inferior compromise between interoperability and reliability. In a LAN environment, there is no need for elaborate mechanisms for adapting to network problems. On the other hand the need for multiple connections to remote audio servers increases the load on the external connection points of the network and the work that has to be performed by firewalls, routers etc. Finally, the large number of special purpose audio players (each compatible with a different subset of available formats), alienates users and creates support and administrative headaches.

By implementing the audio streaming server as a virtual device on the system running the decoding applications, we have bypassed the compatibility issues that haunt any general-purpose audio player. Our system confines the special-purpose audio players to a few servers that multicast the audio data always using the same common format.

The existence of a single internal protocol without special cases or the need for additional development to support new formats, allowed the creation of the “Internet Speaker” an embedded device that plays audio streams received from the network. The communications pro-

ocol also allows any client to “tune” -in or -out of a transmission, without requiring the knowledge or cooperation of the server. The protocol also provides synchronization between clients broadcasting the same audio stream.

## References

- [1] Libpnet6, An Advanced Networking Library. <http://pnet6.sourceforge.net/>.
- [2] Writing a pseudo device, NetBSD Documentation. <http://www.netbsd.org/Documentation/kernel/pseudo>.
- [3] H. Schulzrinne et al. RTP: A Transport Protocol for Real-Time Applications. RFC: 1889.
- [4] K. Miller et al. StarBurst multicast file transfer protocol (MFTP) specification. Internet Draft, Internet Engineering Task Force, April 1998.
- [5] Bill Fenner, Brian Haberman, Hugh Holbrook, and Isidor Kouvelas. Multicast Source Notification of Interest Protocol (MSNIP). draft-ietf-magma-msnip-01.txt, November 2002.
- [6] I. Kouvelas and V. Hardman. Overcoming Workstation Problems in a Real-Time Audio Tool. In *USENIX Annual Technical Conference*, January 1997.
- [7] James S. Lowe and Luigi Rizzio. Multimedia Driver Support in the FreeBSD Operating System 5. In *USENIX Annual Technical Conference, FREENIX track*, June 1998.
- [8] C. Kenneth Miller. *Multicast Networking and Applications*. Addison-Wesley Longman, Inc., 1999.
- [9] Vassilis Prevelakis and Angelos Keromytis. Designing an Embedded Firewall/VPN Gateway. In *Proceedings of the Third International Network Conference*, Plymouth, UK, July 2002.