The Ethernet Speaker System *

David Michael Turner and Vassilis Prevelakis

Computer Science Department
Drexel University
{dmt36, vp}@drexel.edu

Abstract

If we wish to distribute audio in a large room, building, or even a campus, we need multiple speakers. These speakers must be jointly managed and synchronized. The Ethernet Speaker (ES) system presented in this paper can be thought of as a distributed audio amplifier and speakers, it does not "play" any particular format, but rather relies on off-the-shelf audio applications (*e.g.,* mpg123 player, Real Audio player) to act as the audio source. The Ethernet Speaker, consists of three elements: (a) a system that converts the audio output of the unmodified audio application to a network stream containing configuration and timing information (rebroadcaster), (b) the devices that generate sound from the audio stream (Ethernet Speakers), and (c) the protocol that ensures that all the speakers in a LAN play the same sounds.

This paper covers all three elements, discussing design considerations, experiences from the prototype implementations, and our plans for extending the system to provide additional features such as automatic volume control, local user interfaces, and security.

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

1 Introduction

Consider a situation where you want to listen to some audio source in various rooms in your house, alternatively you may want to send audio throughout a building. In such situations the traditional approach would be to set up amplifiers, speakers and connect them all up into one large analog audio network (*e.g.,* [4]). If laying wires is not an option, then wireless solutions also exist where the audio signal is sent over radio frequencies and the speakers are essentially radios that listen on a preset frequency.

In this paper we discuss our implementation of a similar architecture using an Ethernet network and small embedded computers as speakers.

Our motivation for undergoing this project was that we believe that (a) using an existing network infrastructure may allow the deployment of large scale public address systems at low cost, (b) having an embedded computer next to each speaker offers numerous opportunities for control of the audio output (*e.g.,* ability to use the built-in microphone to determine the appropriate volume level depending on ambient noise levels). (c) Given that most audio sources nowadays are digital, by reducing the distance that the analog audio signal has to travel may improve the quality of the audio (Figure 1).

Numerous solutions exist for transferring audio over the Internet, but since most local networks are Ethernet-based we wanted to use this experiment to determine whether we can come up with a number of simplifications to the architecture by assuming that all the speak-

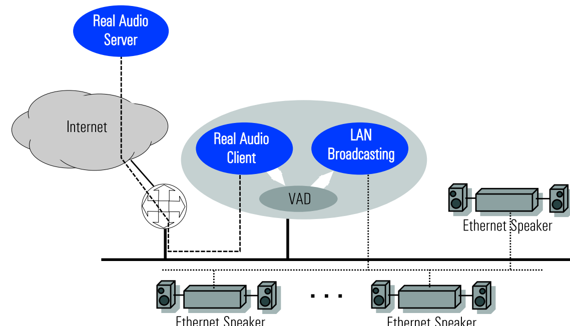

Figure 1: Rebroadcasting WAN Audio into the LAN.

*This work was supported by NSF under Contract ANI-0133537.

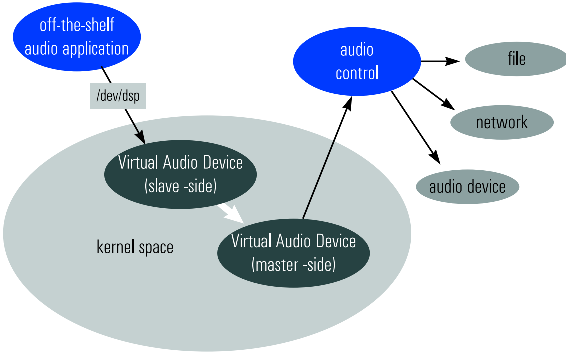

Figure 2: Overview of the Virtual Audio Device.

ers are located on the same Ethernet segment. This ties in very well with the greatly improved range and size of modern Ethernet LANs. In section 2 we discuss the impact of this assumption on the design and implementation of our system.

Another issue that we faced was how to generate the audio streams that would feed these (Ethernet) speakers. We did not want to design yet another audio library or streaming service, or to have to modify existing applications so that they would use our system. The solution we selected was to redirect the audio stream from inside the kernel so that instead of going out the built-in audio card, the stream would be channelled directly to the network or to some other user-level application. This led to the design of the virtual audio device (VAD) that virtualizes the audio hardware in the same way as the Unix system uses pseudo terminals (`pty(4)`).

The advantage of this approach is that the application cannot determine whether it is sending the audio to a physical device or to a virtual device and hence off-the-shelf applications (even proprietary ones) can be used to send audio to our Ethernet Speakers (Figure 2).

In the rest of this paper we describe the basic elements of the Ethernet Speaker (ES) system, including the VAD, the communications protocol used to send audio data and configuration information to the Ethernet Speakers and the platform used to implement the ES. We also describe our prototype and the lessons we learned from its implementation. Finally, we discuss some of our plans for adding some security and remote management features to the system.

2 Design

2.1 The Virtual Audio Device (VAD)

In this section we discuss the modifications we made to the OpenBSD kernel to support the virtual audio device (VAD). Although we have used the OpenBSD kernel, our modifications should be easy to port to any system that uses a similar architecture for the audio driver.

The audio subsystem of most modern Operating Systems provides a path from the application to the audio hardware. While this arrangement works most of the time, it has a number of deficiencies that can be quite frustrating. These include:

- The numerous encoding formats of the audio data require the use of different audio players, each with its own user interface. Some of the players (*e.g.,* real audio player) offer only graphical user interfaces making them unusable on embedded platforms or systems that offer only character-based user interfaces.
- The tight binding between the audio device and the audio hardware means that the sound must be generated close to the computer running the audio application.
- Certain streaming services offer no means of storing the audio stream for later playback (time shifting).

Despite the numerous audio formats in use, the services offered by the audio device driver (`audio(4)`) are well defined and relatively straightforward in terms of formats and capabilities. This implies that by intercepting an audio stream at the kernel interface (system call level), we only need to deal with a small set of formats. In other words, the various audio applications perform the conversion from the external (possibly proprietary) format to one that is standardized. We, therefore, require a mechanism that allows the audio output of a process to be redirected to another process. The redirection occurs inside the kernel and is totally transparent to the process that generates the audio stream. Our system utilizes a virtual audio device (VAD) that plays a role similar to that of the pseudo terminals that may be found on most Unix or Unix-like systems. The VAD consists of two devices (a master and a slave). The slave device looks like a normal audio device with the difference being that there is no actual sound hardware associated with it.

The audio application opens the slave device and uses `ioctl` calls to configure the device and `write` calls to send audio data to the device. Another application can then open the master device and read the data written to the slave part (Figure 2).

2.1.1 The OpenBSD Audio System

The OpenBSD audio driver is an adaptation of the NetBSD audio driver and consists of two parts: the device independent high level driver and the device dependent low level driver. The high level driver deals with general issues associated with audio I/O (*e.g.,* handling the communications with user-level processes, inserting silence if the internal ring-buffer runs out of data, *etc.*), while the low-level drives the audio hardware. User-level applications deal entirely with the high-level audio driver.

In the OpenBSD kernel there is one instance of the high-level audio driver and as many instances of the low-level as types of audio cards connected to the system. The first audio device is associated with the `/dev/audio0` device, the second with `/dev/audio1` and so on.

Applications use `ioctl` calls to set various parameters (such as the encoding used, the bit rate, *etc.*) in the driver and the usual file I/O calls to read and write data to the device.

The interface between the two levels of the audio device driver is well documented (`audio(9)`) so adding a new audio device is fairly straightforward.

In the case of the Virtual Audio Drive, we had to construct a low-level audio device that is recognised by the OpenBSD kernel as a valid device. Using terminology borrowed from the pseudo terminal implementation (`pty(4)`), we note that the VAD driver provides support for a device-pair termed a virtual audio device. A virtual audio device is a pair of audio devices, a master device and a slave device. The slave device provides to a process an interface identical to that described in `audio(4)`. However, whereas all other devices which provide the interface described in `audio(4)` have a hardware device of some sort behind them, the slave device has, instead, another process manipulating it through the master half of the VAD. That is, anything written on the slave device (`vads`) is given to the master device (`vadc`) as input (currently `vads` only supports audio output).

The master (or control) device has its own entry in the `/dev` directory providing an access point for user-level applications.

In addition to the audio data, the slave device passes control information (*e.g.,* values set using the `ioctl(2)` system call) to the control side. Thus the application accessing `vadc` can always decode the audio stream correctly.

2.1.2 Why a Virtual Audio Device?

An audio application does more than just send audio data to the audio device. The control information, mentioned above, is vital to the correct playback of the audio stream. The audio application establishes these settings by configuring the audio device. Moreover the application may check the status of the audio device from time to time. All of the above indicate that we cannot simply replace the audio device (`/dev/audio`) with something like a named pipe; our redirector must behave like a real audio device and be able to communicate the configuration updates to the process that receives the audio stream via the master side of the VAD.

2.2 The Audio Stream Rebroadcaster

The data extracted from the master side of the VAD may be stored in a file, processed in some way and then sent to the physical audio device, or transmitted over the network to the Ethernet Speakers. In this section we discuss the Audio Stream Rebroadcaster which is an application that gateways audio information received from the public Internet to the local area network (see Figure 3).

In addition to providing input to the Ethernet Speakers, the rebroadcaster service may be used as a proxy if the hosts in the LAN do not have a direct connection to the public Internet and thus require the use of a gateway (which has access to both the internal and the external networks) to rebroadcast the audio stream.

Another reason may be that if we have large numbers of internal machines listening to the same broadcast, we may not want to load our WAN link with multiple unicast connections from machines downloading the same data. By contrast, the rebroadcaster can multicast the data received from a single connection on the WAN link.

Moreover, by having a known format for the multicast

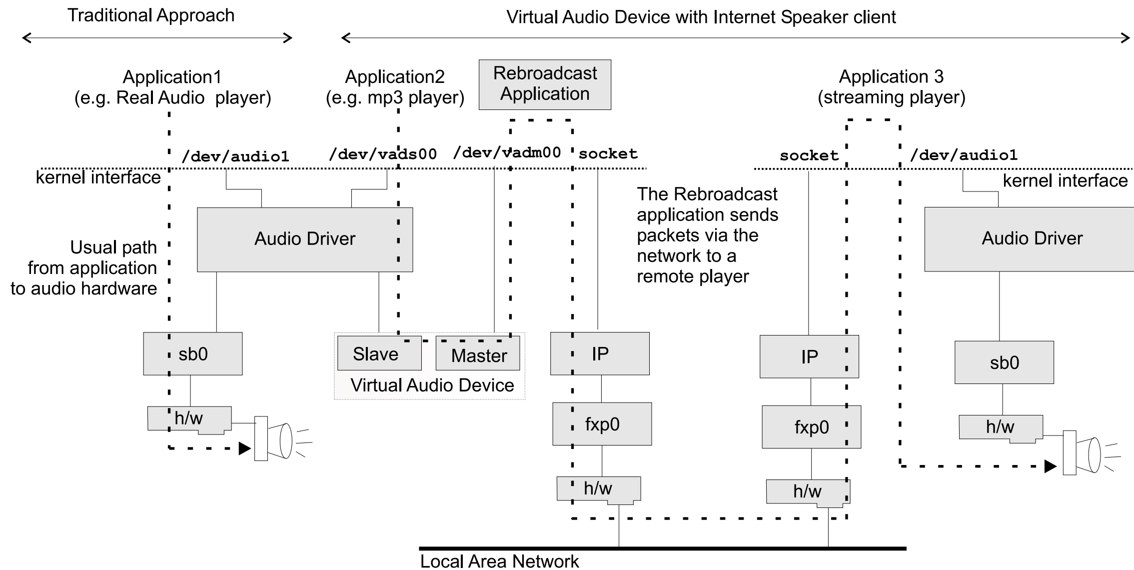

Figure 3: Application 1 plays audio to the normal audio device, while Application 2 plays to the VAD, which sends the audio data to a remote machine via the network.

audio data, we can play back any stream with a single player. These players do not need to be updated when new audio formats are introduced.

The Audio Steam Rebroadcaster consists of two applications, the producer that runs on the same machine as the virtual audio device (and hence the application that receives the audio stream over the public Internet) and one or more consumers running on other hosts in the LAN. The producer sends the audio stream as multicast packets to reduce the load on the network.

Early versions of our design [10], sent onto the network the raw data as it was extracted from the VAD. However this created significant network overhead (around 1.3Mbps for CD-quality audio). On a fast Ethernet this was not a problem, but on legacy 10Mbps or wireless links, the overhead was unacceptable. We, therefore, decided to compress the audio stream.

Fortunately, there are a wide variety of audio compression technologies that more than satisfy our needs. We decided on the use of Ogg Vorbis (w.xiph.org) for multiple reasons. It is completely patent free and it's implementation is distributed under the General Public License. Additionally, being a lossy psycho-acoustic algorithm, it provides excellent compression, comparable to proprietary solutions such as MP3.

The introduction of Ogg Vorbis into the Ethernet speaker architecture brings about several interesting considerations. It is commonly known that the combination of multiple lossy codecs onto the same set of data can lead to greater quality loss than necessary. This is because the algorithms may choose to eliminate very different segments of data in order to achieve the same goal. The best one can hope for would be that the audio quality would not get any worse. If a user were to take their favorite MP3 file and play it over the Ogg Vorbis equipped Ethernet Speaker it would pass through two very different lossy audio compression algorithms. In order to try and compensate for this loss of quality we simply set the Ogg Vorbis quality index to its maximum. This causes the algorithm to throw away as little data as possible while still providing adequate compression. Luckily, our experience so far has not revealed any audible defects to the stream.

2.3 The Communication Protocol

Early in the design of the Ethernet Speaker System, we decided that the communication will be restricted to a single Ethernet LAN. The reason for this decision was that a LAN is a much friendlier environment to communicate providing low error rates, ample bandwidth, and most importantly, well behaved packet arrival. So far at Drexel, even though the main campus network often sees large peaks in traffic, we have not experienced packet loss or transient network disruptions that allowed the input buffer of the ESs to empty and thus affect the

audio signal.

Another advantage of using a LAN is that we get multicast support by default. Once packets have to cross routers, then multicasting becomes an option few network administrators are likely to support. Nevertheless, by using multicast packets, we avoid the need to have the Ethernet Speakers contact the Rebroadcaster in order to receive the audio stream. As noted earlier, the VAD is sending configuration information as well as audio data. The configuration information must be passed on to the Ethernet Speakers so that they can decode the audio stream. We do this by having the Rebroadcaster send control packets at regular intervals with the configuration of the audio driver. The Ethernet Speaker has to wait till it receives a control packet before it can start playing the audio stream.

The advantage of this approach is that (a) the Rebroadcaster does not need to maintain any state for the Ethernet Speakers that listen in, and (b) clients do not need to contact the Rebroadcaster to retrieve the audio configuration block.

In this way our Ethernet Speakers function like radios, *i.e.,* receive-only devices. This design requirement simplifies the architecture of the audio producer enormously: the Rebroadcaster is just a single-threaded process that collects audio from the master-side VAD and delivers it to the LAN.

2.4 The Ethernet Speaker

The task of refining the design of the ES runtime environment was made much easier by the fact that we had previous experience with such systems from work in embedded systems such as VPN gateways [11] and network monitoring stations [12]

We were, thus, aware that the ES has to be essentially maintenance-free so that once deployed, the administrators will not have to deal with it. This requirement in turn leads to two possible configurations: one that boots off the network or one that boots off a non-volatile RAM chip (Flash memory).

In either case the machine will have to receive its configuration from the network. Network setup may be done via DHCP, but we also need additional data such as the multicast addresses used for the audio channels, channel selection, etc.

We established that the machine runtime should be based on a ramdisk configuration. The rationale behind this decision is that if we use a Flash boot medium, we would not be able to have it mounted read-write because a power (or any other) failure may create a non-bootable machine. In the case of the network-based system, having machines mount their root and swap filesystems over the network would lead to scalability problems.

The requirement that we should be able to update the software on these machines without having to visit each machine separately made the network boot option more appealing.

We thus decided to use a ramdisk-based kernel that is loaded over the network. The ramdisk is part of the kernel, so that when an ES loads its kernel, it gets the root filesystem and a set of utilities which include the rebroadcast software. The ramdisk contains only programs and data that are common to all ESs. Each machine's network-related configuration is acquired via DHCP, the rest are in a tar file that is `scp`'d from a boot server (note that the boot server's ssh public keys are stored in the ramdisk).

The procedure for constructing the ramdisk kernel is similar to the creation of the installation floppy or CDROM that is part of the OpenBSD release.

The configuration tar file is expanded over the skeleton `/etc` directory, thus the machine-specific information overwrites the any common configuration.

The network boot was possible because the machines we use as Ethernet Speakers use the PXE network boot protocol. We found that the network boot procedures for the i386 platform were not sufficiently explained in the OpenBSD documentation, so we produced a netboot "How To" document that is included in our distribution. This document has been submitted to the OpenBSD project for possible inclusion in the project's documentation.

3 Implementation

The virtual audio driver currently runs under OpenBSD 3.4, but we believe that since the audio subsystems between the various BSD-derived Operating Systems are quite similar, porting the VAD to NetBSD and perhaps FreeBSD will not be too difficult.

In this section we discuss some of the technical issues we had to address in developing the VAD and the Audio Steam Rebroadcaster application.

3.1 Rate Limiter, or why does a 5 minute song take 5 minutes?

In a conventional system involving actual audio hardware, the producer-consumer relationship established between the driver and the hardware itself is inherently rate limited. If a five second audio clip is sent to the sound device than it will take five seconds for the actual hardware to play that sound clip. The audio driver cannot send audio data down any faster than this. If an application tries to write to the audio device at rate faster than the hardware can play, it will eventually fill up the ring buffer and the call will begin to block until space is freed.

However, with the VAD there is no underlying hardware to impose a data rate limit. Depending upon the users needs this can be either useful or troublesome. For the purposes of the Ethernet Speaker this creates a serious problem. Without any rate limiting the rebroadcaster will send data that it receives from the VAD as fast as it is written. Assuming that the rebroadcaster is receiving an audio stream from a an MP3 player, then the only speed constraint would be the speed of the I/O and the processor. The producer will essentially send the entire file at wire speed causing the buffers on the Ethernet Speakers to fill up, and the extra data will be discarded, resulting in noticeable audio quality loss. In the above example of the MP3 player you will only hear the first few seconds of the song.

The solution is to instruct the rebroadcaster to sleep for the exact duration of time that it would take to actually play the data, we will effectively limit the rate enough to ensure that it cannot be sent faster than it can be played. The actual duration of this sleep is calculated using the various encoding parameters such as the sample rate and precision.

While we could have integrated this rate limiting into the driver itself we decided against that and developed it separately into the Audio Rebroadcast application. We did not want to limit the functionality of the VAD by slowing it down unnecessarily. Other uses for the VAD might require different needs; we did not wish to make these decisions for the user.

3.2 Synchronization

With the original goal being the placement of any number of Ethernet Speakers on a LAN, issues of synchronizing the playback of a particular audio stream arise. As an Ethernet Speaker accepts data from a stream it needs to buffer the data in order to handle the occasional network hiccup or extraneous packet. It is this buffering that causes problems in synchronizing the playback of the same stream on two different Ethernet Speakers. In earlier versions of the system this problem was most severe when ESs were started at different times in the middle of the stream.

The solution that we implemented is fairly straightforward. Inside, each periodic stream control packet we place a timestamp that serves as a wall clock for the ESs. In addition to this "producer time," we send a timestamp within each audio data packet that instructs the ES when it should play the data. The wall clock and the audio data packet's timestamp are relative to each other, thus allowing the ES to know whether it is playing the stream too quickly or slowly. With this information we can adjust accordingly by either sleeping until it is time to play or throwing away data up until the current wall time. It is important to note however that it is necessary to provide an epsilon value that provides the ES with some leeway. If this is not done than data will be unnecessarily thrown out and skipping in playback will be noticeable.

Of course, with this implementation we do not achieve a perfect synchronization. There are several deficiencies. Firstly, we completely ignore any latency in the transmission of our packets. For example, consider a scenario in which one ES receives the wall clock synchronization slightly after another. Both ESs will believe that they have the correct time and will thus play their data, resulting in one being behind the other. Also, there is the issue of slight phase differences that could develop when two ESs are on different architectures. One could imagine that in combination these two problems to combine to create a serious synchronization problem. However, in our initial testing it appears that any phase difference attributed to network delay or otherwise is negligible in the face of our current solution. However, considering the bandwidth capacity on the LAN the tests were performed, network delay could prove to be a problem on slower setup.

3.3 VAD implementation

The preliminary design of the VAD device involved a single driver that would attach to the hardware independent audio driver (dev/audio.c). The VAD would then be able to intercept and process any audio data written to the audio device file. With full access to this raw audio data the driver would then send it directly out onto the LAN from within the kernel. The benefits of such a setup were most likely simplicity and efficiency. If all the functionality of the VAD was left in the kernel then the design would is significantly more compact and straightforward. The performance would most likely be better as well because we could avoid the overhead of multiple context switches from sending the data back out of the kernel and in again.

However, in the course of implementation we encountered problems with this design. Since we are developing in kernel space the code must remain relatively simple. If we decided to add complexity to the code in the form of off the shelf compression or security, we would run into problems. Consequently, it became apparent to us as we began developing the VAD that it might be a good idea to separate the streaming functionality from the actual driver itself to make it more modular. Essentially the VAD is supposed to provide a way for us to access the uncompressed audio data written to it so that we can be unconcerned with the prior format it was in. It is our goal to stream this audio data out over a LAN to multiple clients. However, if we separate the streaming functionality from the driver than we allow it to be used for any purpose. With a VAD device configured on a system any application can now have access to uncompressed audio, whether the source of that audio was in some proprietary format or not. By putting only the data access functionality into the driver we have made a addition to the kernel that gives any user space application this ability, for whatever reason they need it.

More importantly, there was an interesting aspect of the OpenBSD audio driver architecture that made implementing this design difficult as well. When data is written to the device it enters the hardware independent audio layer and is stored in a ring buffer. For the first block of data ready to be played, the independent audio driver invokes the hardware specific driver attached to it. Then, in the typical example, it is the job of the hardware specific driver to initiate a direct producer-consumer relationship between the hardware and the independent audio driver. Usually this is done by triggering a DMA transfer of the data to the actual hardware. The hardware specific driver passes an interrupt function, provided by the independent audio driver, to the DMA routines that is called every time a transfer is completed, usually by the audio hardware interrupt service routine. This interrupt function notifies the hardware independent driver that a block of data has been transferred and can be discarded. With this relationship intact, the hardware specific driver is essentially out of the picture, cutting out the middleman so to speak. Therefore it is only invoked once, when the first block of data is ready to be played.

The problem arises when there is no actual hardware to create such a relationship with. Our VAD driver takes the place of the hardware specific driver however there is no actual sound hardware. So when the first block of data is written the independent audio driver assumes that the VAD driver will set up the DMA and pass along the interrupt function. In our case, the independent audio driver assumes the VAD is just the middleman, expecting it's interrupt routine to be called by the hardware, it never invokes the VAD entry points again. A strong example of the fact that the OpenBSD audio architecture was not meant to support pseudo devices. Our solutions to this problem were inelegant and involved either modifying the independent audio driver or creating a separate kernel thread to periodically call the interrupt routine.

3.4 Platform

The machines we used for the Ethernet speaker reflect our belief that a mass produced low-cost Ethernet Speaker platform will be relatively resource poor. We have, therefore, used Neoware EON 4000 machines that have a National Semiconductor Geode processor running at 233MHz and 64Mb RAM, non-volatile memory (Flash) and built-in audio and Ethernet interfaces (Figure 4).

While weak compared to the capabilities of current workstations, the hardware is perfectly adequate for the intended application. Moreover, the use of such low-cost hardware will allow the Ethernet Speaker to have a cost of less the $50.

Our test environment also included a SUN Ultra 10 workstation in order to make sure that our programs and communication protocol worked across platforms.

The slow speed of the processor revealed a problem that was not observed during our testing on faster machines; namely the need to keep the pipeline full. If we use very large buffers, the decompression on the ES has to wait for the entire buffer to be delivered, then the decompres-

Figure 4: The Ethernet Speaker is based on the Neoware EON 4000.

sion takes place and finally the data are fed to the audio device at a rate dictated by the audio sampling rate. If the buffers are large, then time delays add up, resulting in skipped audio. By reducing the buffer size, each of the stages on the ES finishes faster and the audio stream is processed without problems.

4 Related Work

4.1 Audio Streaming Servers

SHOUTcast [1] 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.

The Helix Universal Server from RealNetworks [2] 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.

[1] http://www.shoutcast.com/support/docs/

[2] https://www.helixcommunity.org/2002/intro/client

4.2 Audio Systems

There are also a number of products that may be described as “internet radios”. The most well known is Apple's AirTunes for its Airport Express base station. Such devices accept audio streams generated by servers either in the LAN or in the Internet and play it back. Their feature sets and capabilities are very similar to those of the ES, with the exception that (at least the ones we have tested) they do not provide synchronization between nearby stations. As such they may not be able to be used in an ES context where we have multiple synchronized audio sources within a given room.

4.3 Network Audio Redirectors

The Network Audio System [8] (NAS) developed by NCD uses the client/server model for transferring audio data between applications and desktop X terminals. It aims to separate applications from specific drivers that control audio input and output devices. NAS supports a variety of audio file and data formats and allows mixing and manipulating of audio data.

Similar to NAS are audio servers such as Gstreamer [14] and aRts [5] that allow multiple audio sources to use the workstation's audio hardware.

The most sophisticated of the lot is JACK [3], which is a low latency audio server.

All the above are constrained by the fact that they need to either have new applications created for them, or at least to recompile existing applications with their own libraries. Obviously this works only with source distributions.

EsoundD [2] utilizes the fact that most applications use dynamically-linked libraries and uses the LD_LIBRARY_PATH variable to insert its own libraries

before the system ones. The applications (apparently the Real Audio Player is included) use the EsoundD libraries and thus benefit from the audio redirection and multiplex features without the need for any recompilation. Of course, if the application is statically linked (not very popular lately), this approach fails.

The closest system to the VAD is the maudio [1] virtual audio device for Linux. The maudio provides functionality that is very close to that of the VAD, and we have used it to port our audio rebroadcaster service to Linux.

A similar application to the VAD is the Multicast File Transfer Protocol [6] (MFTP) from StarBurst Communications. MFTP is designed to provide efficient and reliable file delivery from a single sender to multiple receivers.

MFTP 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.

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 [7]. 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. Unfortunately, we have to wait until the MSNIP appears in the software distributions running on our campus routers.

5 Future Plans

5.1 Security

For the Audio Steam Rebroadcaster, security is important. The ESs want to know that the audio streams they see advertised on the LAN are the real ones, and not fake advertisements from impostors.

Moreover, we want to prevent malicious hosts from injecting packets into an audio stream. We do this by allowing the ES to perform integrity checks on the incoming packets.

In the current version some security can be maintained by operating the Ethernet Speakers in their own VLAN. Note, however, that there exist ways for injecting packets into VLANs, so this must be considered as an interim measure at best.

Our basic security requirements are that (a) the ES should not play audio from an unauthorized source, and (b) the machine should be resistant to denial of service attacks.

Having a machine that receives all its boot state from the network creates an inherently unsafe platform. Any kind of authentication that is sent over the network may be modified by a malicious entity, thus creating the environment for the installation of backdoors that may activated at any moment.

We are considering taking advantage of the non-volatile RAM on each machine to store a Certification Authority key that may be used for the verification of the audio stream.

For the audio authentication digitally signing every audio packet [15] is not feasible as it allows an attacker to overwhelm an ES by simply feeding it garbage. We are, therefore, examining techniques for fast signing and verification such as those proposed by Reyzin et al [13], or Karlof et al [9]. We are also looking into whether we can take advantage of the services offered by the IEEE 802.1AE MAC-layer security standard.

5.2 Automation

Another area where we are working on is the ability for the device to perform actions automatically. One example will be to set the volume level automatically depending on the ambient noise level and the type of audio stream. So for background music the ES would lower the volume if the area is quiet while ensuring that audio segments recorded at different volume levels produce the same sound levels.

Alternatively, if an announcement is being made, then

the volume should be increased if there is a lot of background noise so that announcements are likely to be heard.

We plan to implement these features by taking advantage of the microphone input available on our machines. This input allows the ES to compare its own output against the ambient levels.

5.3 Management

Our intention is to have multiple streams active at the same time and ESs being able to switch from one channel to another. This implies the ability to receive input from the user (*e.g.,* some remote control device). Alternatively all ESs within an administrative domain may need to be controlled centrally (*e.g.,* movies shown on TV sets on airplane seats can be overridden by crew announcements). We want to investigate the entire range of management actions that may be carried out on ESs and create an SNMP MIB to allow any NMS console to manage ESs.

6 Conclusions

Existing audio players adopt complex protocols and elaborate mechanisms in order to deal with network problems associated with transmission over WAN links. These players are also largely incompatible with each other while the use of unicast connections to remote servers precludes the synchronization of multiple players within the same locality. Moreover, these multiple connections increase the load both on the remote server and 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 Audio Rebroadcast Application that allows clients to play audio streams received from the network. The communications protocol also allows any client to "tune" -in or -out of a transmission, without requiring the knowledge or cooperation of the server.

The software for the VAD and Audio Rebroadcasting application is available as Open Source Software and can be downloaded at `http://www.cs.drexel.edu/~vp/EthernetSpeaker`.

References

[1] A simple audio mirroring device . `http://freshmeat.net/projects/maudio`.

[2] EsounD The Enlightened Sound Daemon . `http://www.tux.org/~ricdude/dbdocs/book1.html`.

[3] JACK: A Low Latency Audio Server. `http://jackit.sourceforge.net`.

[4] QSControl CM16a Amplifier Network Monitor Product Information Sheet. `http://www.qscaudio.com/pdfs/qsconspc.pdf`.

[5] The Analog Real-Time Synthesizer. `http://www.arts-project.org/doc/manual`.

[6] K. Miller et all. StarBurst multicast file transfer protocol (MFTP) specification. Internet Draft, Internet Engineering Task Force, April 1998.

[7] Bill Fenner, Brian Haberman, Hugh Holbrook, and Isidor Kouvelas. Multicast Source Notification of Interest Protocol (MSNIP). draft-ietf-magma-msnip-01.txt, November 2002.

[8] Jim Fulton and Greg Renda. The network audio system: Make your applications sing (as well as dance)! *The X Resource*, 9(1):181–194, 1994.

[9] C. Karlof, N. Sastry, Y. Li, A. Perrig, and J. Tygar. Distillation codes and applications to dos resistant multicast authentication, 2004.

[10] Ishan Mandrekar and Vassilis Prevelakis. An Audio Stream Redirector for the Ethernet Speaker. In *International Network Conference*, Plymouth, UK, July 2004.

[11] V. Prevelakis and A. D. Keromytis. Drop-in Security for Distributed and Portable Computing Elements. *Internet Research: Electronic Networking, Applications and Policy*, 13(2), 2003.

[12] Vassilis Prevelakis. A Secure Station for Network Monitoring and Control. In *Proceedings of the 8th USENIX Security Symposium*, August 1999.

[13] Leonid Reyzin and Natan Reyzin. Better than BiBa: Short one-time signatures with fast signing and verifying. In *Seventh Australasian Conference on Information Security and Privacy (ACIP 2002)*, July 2002.

[14] Christian F.K. Schaller and Scott Wheeler. Introduction to GStreamer & KDE. `http://conference2004.kde.org/slides/schaller.wheeler-introduction_to_g%streamer_and_kde.pdf`.

[15] Wong and Lam. Digital signatures for flows and multicasts. *IEEETNWKG: IEEE/ACM Transactions on NetworkingIEEE Communications Society, IEEE Computer Society and the ACM with its Special Interest Group on Data Communication (SIGCOMM), ACM Press*, 7, 1999.