IEEE International Conference on Web Delivering of Music (WEDELMUSIC), Florence, Italy, November 2001.

The Recording Studio that Spanned a Continent

Jeremy R. Cooperstock
Centre for Intelligent Machines
McGill University
Montreal, QC
H3A 2A7 Canada
+1 514 398-5992
jer@cim.mcgill.ca

Stephen P. Spackman
Centre for Intelligent Machines
McGill University
Montreal, QC
H3A 2A7 Canada
+1 514 398-7200
stephen@acm.org

ABSTRACT
On September 23, 2000, a jazz group performed in a concert hall at McGill University in Montreal and the recording engineers mixing the 12 channels of uncompressed PCM audio during the performance were not in a booth at the back of the hall, but rather in a theatre at the University of Southern California in Los Angeles. To our knowledge, this is the first time that live audio of this quality has been streamed over the Internet.

This paper describes the hardware configuration and software system used for the demonstration, explaining the motivation for our approach and summarizing some of the important lessons learned during the development process.

Keywords
Networked audio, high-bandwidth data distribution, congestion control, layered coding

1 INTRODUCTION

Real-time transmission of audio data over the Internet has become relatively commonplace, but the quality and number of channels has so far been severely limited by bandwidth constraints. With the ongoing growth of new, high speed networks comes the potential for high-fidelity, multi-channel audio distribution.

Our work in this area began in 1999, in cooperation with the Audio Engineering Society's Technical Committee for Networked Audio Systems (TCNAS) and Dolby Laboraties, when we developed protocols [9] for reliable real-time streaming of AC-3 (encoded Dolby 5.1) audio streams at 448 kbps [1], transported over a 1.5 Mbps AES/EBU carrier. The technology was first demonstrated at the AES 107th Convention in New York, where a concert taking place at McGill University's Redpath Hall was streamed live to New York University's Cantor Hall, without interruption, despite congestion on the network link. While this served as an interesting and compelling demonstration of networked audio far beyond the limitations of bandwidth-starved MP3 files and RealAudio streams, this early demonstration was still, for all intents and purposes, a one-way event, in which the audience in New York had no control over the playback beyond simple adjustment of their local sound system. Our next goal was to demonstrate similar networked technology in a context where the remote end was really in control, hence, the idea for a live, remote mix of high fidelity audio, suitable for recording studio applications.

This culminated in a demonstration, at last fall's AES 109th Convention in Los Angeles, of transcontinental streaming of uncompressed mulitchannel audio between McGill University in Montreal and the University of Southern California, in LA, hence, the "Recording Studio that Spans a Continent." The bandwidth required for his high-quality audio data exceeded 28 megabits per second (Mbps), roughly 500 times the capacity of an analog telephone line! The remainder of this paper describes the technical details at both the audio and computer network level, as well as future directions of the research. First, we begin with a high-level summary of the event.

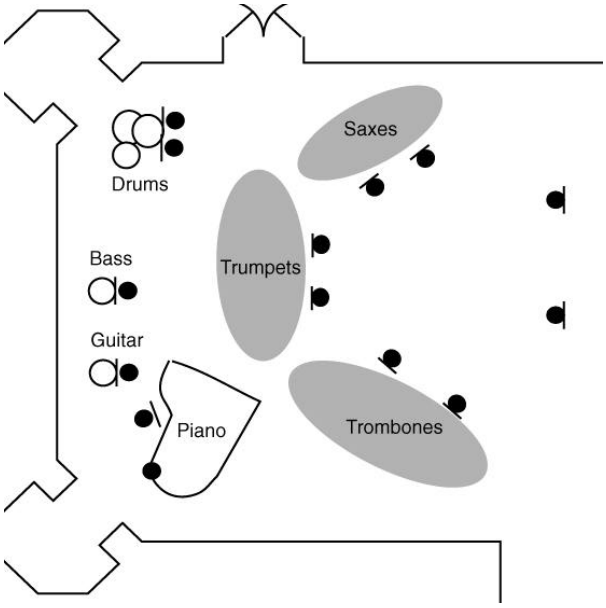

Figure 1. The microphone array used at Redpath Hall, McGill University. Note that some of the 12 channels of audio were fed by a mix of more than one microphone.

2 THE SETUP

Analog audio signals from the microphone array at McGill University in Montreal (see Figure 1) were passed through a pre-amp into A/D converters, which provided our computer with a 12-channel digital data stream at 24bit/96kHz, which was then packetized and transmitted over the Canarie CA*net 3 optical network and high speed Internet2 network. On the receiving end at the Norris Theater of the University of Southern California, the opposite process took place, with data passed through D/A converters and mixed by the recording engineers locally. The end-to-end audio signal path is illustrated below in Figure 2. The sound system was composed of:

- bi-amplified JBL 4648A and a 2446H compression driver with horn for left-center-right channels, amplified with BGW 750 Watt amplifiers
- JBL 4645 (18") subs, amplified with a 750 Watt BGW amplifier
- 12 JBL 3310 left/right surrounds, amplified with 3 BGW 350 Watt amplifiers

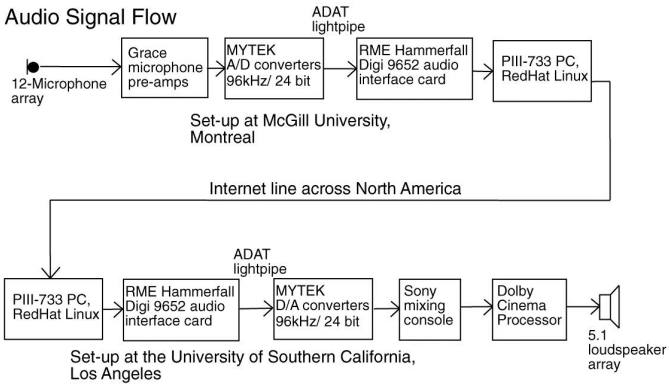

Figure 2. The audio signal path between McGill University and the University of Southern California.

Brant Biles of Mi Casa Multimedia mixed the 12 source channels into the 5.1-channel sound system of the Norris Theater, wrapping the audience with trombones, trumpets and saxes of the McGill Jazz Orchestra and placing the piano and guitar in the front stereo at the left, and the bass and drums in the front stereo at the right. He skillfully used the on-board compressors and equalizers of the Sony DMX-R100 to bring out the presence and blend of the ensemble, achieving a very impressive overall presentation. Most of McGill's room reverberation was captured by the individual group microphones (DPA 4011) with a touch added from the dedicated pair of ambient DPA 4006 microphones. Biles switched off the rear loudspeakers to ensure that everyone in the theatre, including the audience at the rear, would hear a wide horizontal image of the band. The sonic result was an exceptional realism and fullness of musical experience.

The mixing desk for the event was the new Sony DMX-R100 Compact Digital Console running at 96kHz locked into the six pairs of high-speed AES/EBU signals supplied by Mytek 96/24 D/A converters that served only to change the bit stream format from lightpipe ADAT to independent AES/EBU signals. Once mixed digitally into six 96/24 outputs in the Sony console, the signals were converted to analog by an additional Mytek 96/24 D/A converter before being sent to the theatre's 5.1 monitoring system.

For the demonstration, audio data was sampled by Mytek Digital 24bit/96kHz analog-to-digital converters and sent through an ADAT lightpipe (4 channels per pipe) to an RME Hammerfall Digi 9652 audio interface, installed on a PIII-733 PC running Red Hat Linux. Our software managed the reliable transport of the data over the Internet, using special thread scheduling techniques to ensure the precise timing of data transfers to and from the audio devices.

We had also planned to demonstrate transmission of data compressed using the MPEG-2 Advanced Audio Coding library developed by the Franuhofer IIS-A, but time limitations prevented us from completing this component.

In parallel with the audio, an MPEG-2 stream provided video, using Cisco System's IP/TV system. However, since our audio software had no way to communicate with the video system, the two streams had to be synchronized manually to the three second delay imposed by IP/TV.

3 SOFTWARE STRUCTURE

Our initial design, used in 1999, was based in part on the Adaptive File Distribution System (AFDP) [2], which provides efficient and reliable delivery of a file to multiple hosts on an internet, using an RTP-like protocol on top of UDP. The fundamental difference between AFDP and streaming audio is that file distribution is generally not bound by real-time constraints. For a streaming audio application, we must trade a small amount of reliability in exchange for real-time performance. While the AFDP model proved sufficient for the modest demands of AC-3 audio, it was readily apparent that the timing demands of uncompressed PCM 24/96 audio required a more aggressive system, both at the network level and in terms of interaction with the audio hardware.

The PCI-based RME Hammerfall audio interface contains a ring buffer divided into a small number of segments of programmable length, thus providing sufficient in-memory buffering of samples between the PC and the ADAT Mytek D/A and A/D converters. Reading from the audio device thus proved relatively simple; the read thread simply blocked on the device waiting for data to become available.

However, on the write side, the risk exists of both buffer overrun and underrun. Fortunately, the ALSA drivers used to communicate with the Hammerfall hardware were designed to assist in maintaining the buffer at a safe level.

For a number of reasons, including minimising latency, keeping the write thread well *centered* in the output audio buffer, controlling software buffer use, and eliminating variations in performance that can result in false detection of late data by the networking software, it was necessary for the read and write threads to activate at regular intervals. However, as discussed in Section 6, the Linux system scheduler runs at only 100 Hz, a very low rate for audio applications. As the various threads in our code have to juggle network I/O, audio hardware I/O, and buffer management within strict time constraints, it was critical to provide very tight bounds on these operations. These considerations and a desire for scalability to very high data rates led us to avoid buffer copying as much as possible.

Thus, we began the design of a new, highly flexible transport protocol, implemented as a widely adaptable high-throughput software system using no specialized or tightly-coupled code structures. The overall architecture is modular but non-stratified; protocols, local I/O processes and codecs run as peer threads. The program supports multiple simultaneous reader modules and can be configured as a local format converter, as a server, as a client, or as a minimal-delay relay.

The top-level program is compiled as a registry of these modules, allowing for simple command-line configuration of the software signal path through the shared core, in what amounts to an "inverted library" architecture.

The central organizing abstraction and the internal communication path, as illustrated in Figure 3, is a circular array of frame slots (a ringbuffer), which modules can address either randomly, subject to windowing constraints, or with self-locking cursors. The circular structure is used not only as a non-copying buffer shuttle but as the main temporal abstraction, with synchronous events being launched from an integrated timing wheel.

The ringbuffer abstraction pprovides synchronization between modules, local data transport, fragmentation and reassembly. The remainder of this section describes each of these services in more detail.

Synchronization

The ringbuffer frames contain readers/writer locks to coordinate access. Most modules access the ringbuffer through a sliding cursor which maintains the appropriate locks automatically. This means that sequential data sources and sinks stay in synchrony without further effort.

Modules like the network receiver and the network resend requester access the ringbuffer asynchronously as if it were an unbounded linear array; each access is directed to the unique ringbuffer slot that would correspond to the desired index. The access fails cleanly if either the requisite lock cannot be acquired or the slot's timestamp does not match the intended access, meaning that the frame requested is not within the window of frames that currently have a mapping into the ringbuffer. In this case the frame is considered lost and failure is passed back to the caller for higher level recovery.

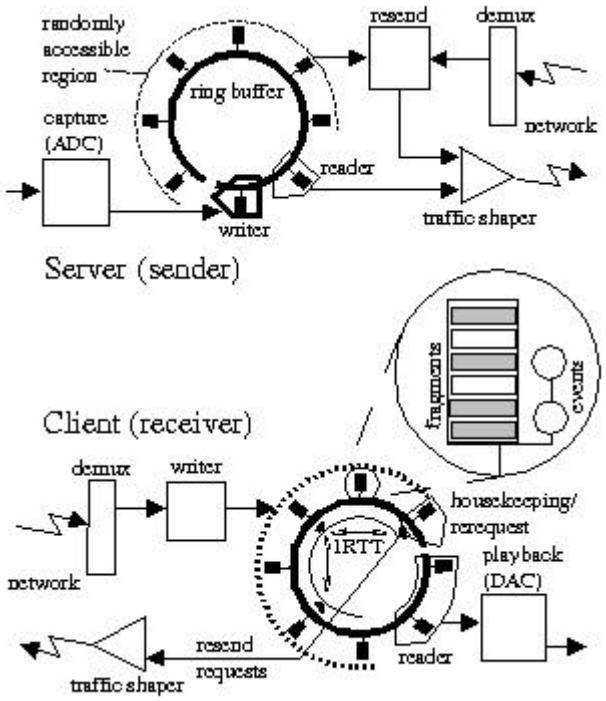

Figure 3. The server and client software structure, constructed around the ringbuffer, which serves as a common abstraction to both entities.

Timing

The labels on the ringbuffer frames are updated, and the window advanced, by a dummy writer process. This ensures that each frame bears a real-time label consistent with the current system state. While this function could have been integrated with a synchronous data source, if any exists, maintaining it as a separate pseudo-module simplified the coding of the network data receiver, which can receive packets badly out of order.

Local Data Transport

Frames are actually stored in the ringbuffer, providing the main data path between the configured modules. This operates without any copy operations: the data source maintains a pool of physical buffers into which incoming data are read, and exchanges newly filled buffers with any stale buffers in the current frame under a writer lock; the

latter are returned to the free pool. This minimizes the duration that the unshareable writer lock must be held. The data sink(s) examine the contents of frames under readers locks, and are designed to do so with the smallest amount of copying feasible, even at the expense of holding the lock longer than would otherwise be necessary. Indeed, the cursors used for reading are allowed to cover multiple adjacent frames at once, to simplify rebuffering, should it be necessary. This is acceptable because the readers' locks can be shared between multiple consumer modules.

Fragmentation

The frame entries in the ringbuffer are stored as arrays of pre-formatted network packet images. This is feasible because a fixed packet size is used throughout the implementation. All modules have to cope with the resultant fragmentation, though the fragment headers themselves are concealed from modules outside the networking code.

Reassembly

Apart from maintaining statistics, the client does not actually distinguish between data initially sent and data resent; received data packets are inserted into the ringbuffer obliviously. Incomplete frames are detected by the resend mechanism simply by reading the frame and noting whether all fragments have been updated.

With the software structure taken care of by the ringbuffer and its associated services, we now turn to a general description of the transport protocol that manages the exchange of data between two hosts.

4 TRANSPORT PROTOCOL

The protocol is implemented with a synchronous sliding window of fixed duration in real-time. Data is initially sent isochronously, as soon as an entire frame becomes available. Explicit timestamps are not used; instead they are inferred from sequence numbers.

For any real-time streaming system, a key concern is how to respond when transmitted data fails to arrive at the client in a timely manner, which may be the result of either network delay or packet loss. In either case, if the client cannot tolerate missing packets, as is the case for an audio stream, some retransmission protocol or recovery technique is required to deal with this situation.

Resend Requests and Retransmission Handling

Resends are driven by automatically generated client requests[1] based on an optimal schedule: requests are issued whenever all packets have not arrived by the frame's anticipated arrival time or at any multiple of a round trip time (RTT) thereafter, the moment at which the response to a presumed previous resend request has become overdue. A resend request specifies the frame numbers that have just become overdue, and a bitmap of the missing packets within each frame. Basic statistics, including current RTT estimates, are transmitted from the client to the server with any resend request.

Redundant resends are suppressed on the server side by combining the received client-measured statistics with the server's own knowledge of its recent actions. A resend is *redundant* if the information requested was sent sufficiently recently that it is predicted to have crossed the resend request in transit; this could happen in the case of synchronization loss, or if the resend traffic is especially high. The server thus maintains no state on behalf of the client; this is intended to facilitate adaptation to a multicast environment.

Outgoing packets are labeled with a priority, to be used in bandwidth control by a traffic shaper. Data being sent for the first time are in general given a higher priority than resends, and, ideally, packets containing higher-significance data, such as control information and higher order sound sample bits, are given priority over less important data. However, no throttling was used for the AES demonstration; this is certainly not good network citizenship. As a result, sufficiently severe network contention could potentially have caused our network utilization to *increase* to the limit of our 100Mbps local connection.

What happens in this situation can be described as a positive feedback loop: a competing data stream causes congestion, which leads to packet loss in our application. Retransmission is then requested to recover the lost data. If the competing stream persists during the retransmission, congestion becomes even more severe, leading to a further increase of packet loss, and so forth, eventually leading to congestion collapse.

While this potentially disastrous situation did not occur in the event, our approach should not be recommended as general practice! Instead, congestion avoidance and prevention mechanisms need to be introduced. For real-time streaming applications, these must be less drastic than TCP, which simply reduces transmission rate, yet still effective in reducing network load. Section 5 describes a promising approach in this direction that we are presently exploring.

RTP and TCP vs. UDP

Although it would have been straightforward to implement this particular strategy using the physical formatting conventions of RTP [6] for data packets, we elected to maintain the flexibility of our software as a protocol research vehicle, using semi-automatic generation of packet

[1] This approach is commonly referred to as *automatic repeat request*, or ARQ.

layouts according to the requirements of a protocol variant instead of fixed conventions. This has cost little and benefitted much, given that RTP implementations, unlike TCP, are typically very tightly bound to particular applications rather than appearing in reusable toolsets or kernel modules.

As to control packets, we have shifted from the approach of our earlier protocol, which made limited use of TCP [4] for the handling of resend requests and retransmissions, to using raw UDP [5] for everything.

Upon reflection, the use of TCP for retransmission handling was rather naïve. Packet loss is usually a sign of network congestion, a condition that causes TCP to activate its congestion avoidance mechanism [3]. Under conditions of mild congestion, this means a higher than usual delay for the delivery of data, as TCP attempts to be *well behaved* for the common good. However, as congestion increases, TCP backs off further, leading to delays that could well be in excess of those tolerable by the application. Worse yet, TCP's guarantee of *in order* data delivery means that any difficulty in delivering retransmission requests will cause subsequent requests to pile up behind it, until the situation is resolved. When normal delivery is eventually resumed, one or more round trip times later, this produces a burst of delayed resend requests, whose servicing may in turn actively contribute to an ongoing congestion problem.

Thus, although an overly aggressive UDP transmission scheme could well exacerbate transient network congestion, the alternative of relying solely on TCP to deal with this situation is unacceptable when robustness is a critical concern. With UDP, we are free to implement whatever retransmission scheme we deem appropriate, without being constrained by the conservative nature of TCP. Furthermore, by treating failures as independent, unordered events, we can reduce the tendency of the protocol to become increasingly bursty under load.

5 CONGESTION CONTROL

Layered coding congestion control is a common method used in real time multimedia streaming [7][8]. Rather than reducing transmission rate, layered coding reduces the data resolution by chopping the least significant bits of each sample, and hence reduces bandwidth demands during times of congestion.

In practice, it is useful to transmit all the bits of a signal, broken into separate *logical channels* (e.g. bits 0-3 on channel 1, bits 4-7 on channel 2, etc.) and allow clients to *subscribe* only to those channels necessary to reconstruct the audio samples to the desired level of resolution. For example, in the case of a multicast transmission to a heteregenous client base, where only a subset of the clients are experiencing congestion, this scheme permits uncongested clients to receive the full resolution of the data.

Although there is no theoretical barrier to varying the information rate of an encoded signal arbitrarily, the current generation of audio compression algorithms (e.g. AC-3, AAC, MP3) based on frequency domain transformations do not support the dynamic variation of stream quality through bit decimation, as their encodings are highly non-linear. Furthermore, varying the encoding resolution dynamically on such signals typically introduces audible artifacts as lookup tables and analysis window sizes change. The prominence of these artifacts varies between algorithms, and is reported to be not too severe for AAC, though it suffers from relatively large granularity in the available quality steps. Fortunately, a new generation of codecs is starting to appear, including the QDX perceptual codec from QDesign, which are designed with continuous adaptation of the bit-rate specifically in mind [Beaton, personal communication].

For the streaming of uncompressed PCM data, however, the bit decimation approach offers a straightforward mechanism for congestion control [9]. Applying the layered coding technique to the uncompressed audio stream, congested clients can subscribe to just enough channels to provide the *n* most significant bits of each sample, where *n* decreases monotonically with observed congestion. Unfortunately, the simplicity of this scheme is also its weakness, as the bit decimation makes no distinction between potentially redundant audio channels, nor does it allow the client to choose between signal degradation and loss of channel separation.

In order to obtain these capabilities, it is first necessary to encode the data into sum[2] (Σ) and difference (Δ) signals through a reversible transformation. For the simple case of two-channel (A and B) stereo, the corresponding differential encoding is found by:

$$\Sigma = (A + B)/2$$

$$\Delta = (A - B)/2$$

Assuming that the Σ and Δ values were each transmitted as multiple *logical channels*, the client would then be free to opt for a high quality monophonic signal in lieu of an incrementally degraded stereo signal during periods of congestion. Obviously, for practical application in this case, it would make sense for the client to make a smooth, rather than abrupt, transition from stereophonic to monophonic playback.

Although it is beyond the scope of this paper to describe in

[2] When normalized by the number of channels, this value represents the monophonic version of the signal.

detail the extensions to this framework related to multichannel audio, the basic idea is that as one adds more physical signal sources, clients should simply be able to subscribe to additional logical channels, while bandwidth remains available. As congestion sets in, clients would then be able to choose appropriate trade-offs between signal resolution and channel separation, as desired by each, independently.

6 IMPLEMENTATION ISSUES

When dealing with audio data at rates approaching 28 Mbps, practical considerations play a significant role. This section describes some of the issues we faced in moving from a carefully considered design to a full implementation suitable for demonstration purposes in front of a discerning audience.

Timing

Striking the delicate balance between the timing constraints imposed by the operating system and those of the audio devices can pose serious problems for the coder, and often ones that can only be tackled empirically.

Because of system timing effects, we were eventually forced to accept a 48Hz frame rate, which is extremely low for this kind of application, and results in very large and highly fragmented frames. Higher frequencies caused various strange problems, which we attribute to the fact that we were using out-of-the-box Red Hat Linux, with a 100Hz system scheduling clock. While our code was designed to operate with smaller, more frequent frames, the 48Hz frame rate guarantees us a valuable two full iterations of the scheduling mechanism per frame.

Synchronization

Another issue we faced was how to deal with the clock differences between two audio devices connected by a computer network. Would the drift between client and server hardware result in a timing problem, especially if the demonstration was run for a full hour? If so, then the complexity of our buffer management would have to increase considerably, since we could no longer guarantee that "one byte produced equals one byte consumed by the hardware."

Fortunately, as it turned out, the quality of time bases has improved considerably since Internet protocols were invented. While it was once unthinkable to let the end points take care of their own clocks and assume they would remain synchronized, we found that source-destination clock synchronization now works with audio devices, even over one hour and several thousand kilometers of intervening network infrastructure.

Drivers

One of the most serious challenges we faced was interfacing the Mytek Digital 96/24 A/D and D/A converters through the RME Hammerfall Digi 9652 PCI card under Red Hat Linux. As we painfully discovered, the ALSA drivers we were attempting to use had never been tested on the Hammerfall at data rates of 24bit/96kHz. In fact, handling the unexpected format and channel rearrangement that the Hammerfall provides under these settings triggered a significant rewrite of the driver code. This problem was finally solved less than one week before the demonstration, leaving us only a few days to complete the rest of the coding and debugging.

7 CONCLUSIONS AND FUTURE WORK

Our transmission of 12 channels of 24bit/96kHz PCM audio consumed a minimum, with no retransmissions, of 28 Mbps while the video stream required an additional 3 Mbps. With moderate competing traffic on the network, our protocol continued to deliver the audio data reliably. During the demonstration, we observed a highly reduced frame rate on the video signal through the IP/TV MPEG-2 display. Although we initially speculated that this was due to bandwidth congestion, we eventually concluded that it was more likely caused by a misconfigured ethernet switch setting on the receive side.

We measured the total network link capacity between McGill and USC at approximately 70 Mbps just prior to the demonstration. In the brief testing period that preceded the event, we ran the protocol smoothly up to 48 Mbps by simulating additional channels of artificial audio.

Experimentation indicated that a 4k byte fragment provided the best performance. While this goes against conventional expectations, which hold that a 1500 byte fragment, as the maximum guaranteed transfer unit for an IP packet, would yield superior performance, it is consistent with our earlier observations [2][9]. This is likely the result of a tradeoff against system call overhead.

In the coming months we plan to explore techniques for dynamic bandwidth control in a multicast environment, using the same codebase. We are also exploring automated synchronization of various audio formats with raw video and DV streams, for real-time streaming and editing applications.

ACKNOWLEDGEMENTS

The authors would like to express their appreciation to the members of the team who gave so many hours of their time to help make this demonstration a success, notably, John Roston, Quan Nguyen, Jason Corey, Andrew Brouse, and Michael Fleming, from McGill, Chris Cain, Kevin Perry, Chris Kyriakakis, and Christos Papadopoulos of USC, and Brant Biles and Bob Margoulef of Mi Casa Multimedia. Many thanks are due to the companies and organizations who supported this research, in particular, Canarie Inc., Cisco Systems, the Audio Engineering Society, Mytek Digital, RME Audio, and especially Paul Barton-Davis

who worked tirelessly with us to get the ALSA drivers speaking properly to the Hammerfall cards. Finally, our deepest gratitude goes to Zack Settel and Wieslaw Woszczyk, who initially challenged us with this exciting research opportunity.

REFERENCES

1. Advanced Television Systems Committee. ATSC A/52. Digital Audio Compression (AC-3) Standard, Dec. 20, 1995. http://www.atsc.org/Standards/A52.

2. Cooperstock, J.R. and Kotospoulos, S. Why use a fishing line when you have a net? An adaptive multicast data distribution protocol. In *Proc. USENIX '96* (San Diego, January 1996).

3. Jacobson, V. Congestion avoidance and control. In *Proc. SIGCOMM '88* (Palo Alto, August 1988).

4. Postel, J. Transmission control protocol. RFC 793, USC/Information Sciences Institute, Sep. 1981.

5. Postel, J. User datagram protocol. RFC 768, USC/Information Sciences Institute, Aug. 1980.

6. Schulzrinne, H. Casner, S., Frederick, R., and Jacobson, V. RTP: A transport protocol for real-time applications. RFC 1889, USC/Information Sciences Institute, Aug. 1996.

7. Shacham, N. Multipoint communication by hierarchically encoded data. *In Proc. IEEE. INFOCOM*, Vol.1, (Florence, Italy, May, 1992).

8. Vicisano, L., Crowcroft, J., Rizzo, L. TCP-like congestion control for layered multicast data transfer, In *Proc. IEEE INFOCOM*, (San Francisco, CA, March, 1998).

9. Xu, A., Woszczyk, W., Settel, Z., Pennycook, B., Rowe, R., Galanter, P., Bary, J., Martin, G., Corey, J., and Cooperstock, J.R. (July-August, 2000) "Real-Time Streaming of Multichannel Audio Data over Internet." Journal of the Audio Engineering Society.