Overcoming Workstation Scheduling Problems in a Real-Time Audio Tool

Isidor Kouvelas Vicky Hardman

Department of Computer Science
University College London
{I.Kouvelas, V.Hardman}@cs.ucl.ac.uk

Abstract

The recent interest in multimedia conferencing is a result of the incorporation of cheap audio and video hardware in today's workstations, and also as a result of the development of a global infrastructure capable of supporting multimedia traffic - the Mbone. Audio quality is impaired by packet loss and variable delay in the network, and by lack of support for real-time applications in today's general purpose workstations. A considerable amount of research effort has focused on solving the network side of the problem by providing packet loss robustness techniques, and network conscious adaptive applications. Effort to solve the operating system induced problems has concentrated on kernel modifications. This paper presents an architecture for a real-time audio media agent that copes with the problems presented by the UNIX operating system at the application level. The mechanism produces a continuous audio signal, despite the variable allocation of processing time a real-time application is given under UNIX. Continuity of audio is ensured during scheduling hiccups by using the buffering capabilities of workstation audio devices drivers. Our solution also tries to restrict the amount of audio stored in the device buffers to a minimum, to reduce the perceived end-to-end delay of the audio signal. A comparison between the method presented here (adaptive cushion algorithm), and that used by all other audio tools shows substantial reductions in both the average end-to-end delay, and the audio sample loss caused by the operating system.

1 Introduction

The ability of current wide area networks, such as the Mbone, to support multimedia conferencing has been recently demonstrated by a series of multicast events. One of the first such events was the Internet Engineering Task Force meeting on March 1992 [1]. The renewed interest in multimedia conferencing is a result of the provision of audio and video hardware in UNIX workstations and the development of multicast for the Mbone [2]. Of the media used in multimedia conferences, audio is the most important [3], as speech is the natural mechanism of human communication, supplemented by video and shared text. Audio is also the only real-time service (video is normally slow-scan), which makes its provision at a reasonable quality of service (QoS) far more difficult.

The Internet's service model offers 'best effort' transmission, and is unable to provide the QoS guarantees needed for real-time traffic. As a result, audio quality is impaired by packet losses and variable transmission delays over the network, but it also suffers from the lack of support for real-time applications in general purpose UNIX operating systems. One approach to solving network problems has been to provide QoS guarantees through resource reservation [4]. Another approach has been to provide network conscious applications that adapt to the problems presented by the Mbone [5]. The latter mechanism has the advantage of being deployable without any network modifications.

Traditional time-sharing operating systems for general purpose workstations do not provide adequate support for real time applications [6]; they operate in an asynchronous fashion, and handle data in blocks [7]. Real-time audio applications are 'soft', in that they have to keep a continuously draining audio device driver fed with blocks of audio samples, but there is no specified instant when audio must be transferred, just a dead-line. However, the lack of a delay bound for timers and external events makes it impossible to guarantee a regular supply of audio samples. Current real-time audio applications ignore the problem, which results in frequently disrupted audio, and increased end-to-end delay.

A solution analogous to that of resource reservations to provide QoS guarantees is to modify the operating system scheduler to provide bounded dispatch latency for applications [7, 8, 9, 10]. Despite experimental proof that this approach can significantly improve the perceptual quality of multimedia applications, it has not been implemented in the majority of desktop workstation operating systems.

There is a very large number of general purpose workstations connected to the Internet, that will soon want to use audio applications with the current scheduler. We present a solution at the application level, that smoothes out scheduling jitter in audio applications using the workstation's device driver buffering capabilities. The solution is analogous to playout adaptation for solving the network jitter, in that it does not require any modifications to workstation hardware or operating system.

This work is part of ongoing research at UCL to develop a robust and flexible audio conferencing component for use on general purpose workstations over the Mbone. Initially began as part of the MICE project [11], and later as parts of projects ReLaTe [12] and MERCI (Multimedia European Research Conferencing Integration, European Union Telematics Applications Programme #1007), the proposed solution has been implemented and evaluated in our Robust-Audio Tool (RAT) [5], which is now funded by an EPSRC project [13]. RAT provides real time audio connectivity between participants in multicast multimedia conferences over the Mbone. RAT is currently supported under SunOS, Solaris, IRIX, HP-UX, FreeBSD and Win32 and ports are under way for Linux PCs and Digital Unix platforms.

In this paper we analyse the audio scheduling problem and its implications. The solution and implementation in RAT is described, together with evaluation results that show the success of our method.

2 Background

The audio device can be visualised as two buffers:

- The input buffer continually inputs samples from the analogue audio input (microphone)
- The output buffer is fed with samples from the application and these are synchronously output to the loudspeakers.

The rate of input sample accumulation, and sample output is the same (sampling rate). The operating system buffer in the device driver has a fixed upper limit, and is adjustable under most OS platforms.

The audio application interacts with audio input and output through the two device driver buffers. Samples are read from the input buffer for processing and written out to the output buffer for playback. In contrast with actual audio playback, the transfer of clocks of samples between the audio application and the device driver is adhoc. Blocks are read from and written to the device driver buffers, and all audio processing takes place on this block unit size. The minimum block size is enforced by the workstation's processing power; the larger the block size, the less frequently the application has to read and write audio blocks.

Figure 1 illustrates the basic functionality of our audio tool. The operation is similar to that of other existing conferencing audio tools like VAT [14], NeVoT [15] and IVS [16].

For the purposes of this paper the audio tool can be considered as a process that acts as an interface between the audio device and the network. The process provides a two way communication facility, and samples input from the audio device are processed and transmitted across the network to remote conference participants. Packets are received from the network, processed, and samples played out to the audio device. For a description of the functions involved in processing the audio samples and for more information on the structure of our audio tool see [5].

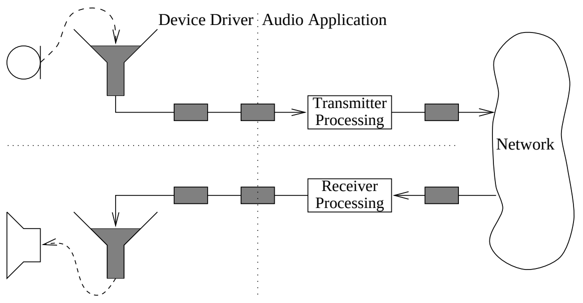

Figure 1: RAT operation

Mbone audio tools use silence detection and suppression to stop the transmission of audio when a conference participant is not active. The current algorithms operate by measuring the average energy in sample blocks [17]. A decision about speech/silence is made by comparing the average energy in a block against a threshold. Blocks that are judged to be above the threshold are labelled as 'speech', and transmitted. Blocks that are judged to be below the threshold are labelled as 'silence' and thrown away. This operation produces talkspurts (periods of continuous speech packets) inter-spaced with periods where no packets are transmitted. The threshold between speech and silence is adjusted during periods of silence.

The effect of sample loss on audio may produce significant degradation in the audio quality, depending on the length of individual gaps, and the frequency of occurrence. Audio gaps indicate a pause in speech to the human brain causing confusion and reducing intelligibility. This problem has been studied by the authors in [3, 5]. The minimisation of sample loss is a primary design goal in audio applications.

In real-time conferencing audio systems interactivity is known to be substantially reduced for round trip delays larger than 600ms. Large round trip delays in conversational situations increase the frequency of confusions and amount of both double talking and mutual silence [18]. The components of this delay in a real-time Internet audio tool are:

- Processing and packetisation delay at transmitter.
- Variable delay experienced by audio packets traversing the network.
- Audio reconstruction buffering at the receiver necessary for smoothing out network delay jitter.
- Delay caused by queued audio samples in the operating system device buffers.

The minimisation of delay is also a primary design goal in audio applications.

3 Audio playback problem

There are two main goals when considering operating system effects for achieving good quality real-time audio communication:

- Continuous audio playback with no unnecessary breakups.
- Low delay in device driver buffers to minimise the end-to-end delay, so that interactivity and normal communication patterns are preserved.

To play-back audio, buffers of samples have to be transferred to the audio device driver. If all the samples in the device driver are played out before more are supplied, then the buffers run dry, and audio playback stops. To avoid the resulting gap of silence in the output audio, the transfer of samples must take place regularly.

Modern workstation audio device drivers provide a selection of audio sampling frequencies. Conferencing applications usually aim to transmit telephone quality speech, and consequently would like to use a sampling frequency of 8KHz. However, audio sampling frequency crystals usually do not have a nominal frequency of exactly 8kHz, and can vary considerably from one workstation to the next [19]. Timing events using the workstation clock without compensating for the drift in the audio crystal will lead to one conference participant's audio buffers becoming full, while a remote workstation's buffers may run dry. To simplify operations the sampling clock of the audio device is used instead. This is achieved by using the number of samples read from the audio device as an indication of the amount of time that has elapsed. Since one crystal is used for audio input and playback, the number of samples read gives a very accurate count of the number of samples played out. With the accurate knowledge of the number of samples that have been consumed by the audio device, a receiver can calculate the drift from a transmitter and compensate by adjusting the duration of silence periods.

4 Implications for a Real-Time Audio Tool

Networked audio tools have a strict loop of operation in order to meet real-time timing restrictions. In each cycle of this loop the following basic operations take place:

- A block of samples is read in from the device driver input buffer.
- An equivalent number of received and processed samples are written out to the audio device.
- Other processing takes place which may involve packets being transmitted onto or received from the network.

Using this order of operations attempts to ensure that as many samples as needed are fed to the output buffers of the audio device. Feeding more would create a backlog of samples in the device driver to play out and would increase the delay. Providing less would result in gaps of silence in playback because of the buffers running dry.

On a non-multi-tasking machine, keeping up with this loop is not hard to achieve. Under a time-sharing operating system - like UNIX - this may not always be possible. The UNIX scheduler decides when a process gets control of the CPU. Processes are serviced according to their priority, and there is no useful upper bound on the amount of time a process may be deprived execution [7, 8].

As a result of other processes being serviced, a relatively large amount of time may elapse in between two instances of the audio tool being scheduled. If the time between schedules is larger than the length of audio data that was written last time, then an audible gap of silence will result in the output audio signal. The persistent effect of the gap in the audio output is to restrict the timing of the output, with respect to the input; extra delay is accumulated in the device driver, since each block of samples is played out later than it should have been. Measurements have shown that the accumulated interruptions caused by an intensive external event on a loaded workstation, can create a delay of several hundreds of milliseconds over the period of a cycle of operation of our program. Since the audio system is timed from the read operations of the audio device, the time that elapses when the audio tool process does not have control of the CPU will be evident; there is lots of audio in the input buffer of the audio device waiting to be read. In response to the waiting input audio, the process will execute the loop several times in succession, until it reads the blocks in. For each of the blocks read, a block of equivalent size will be written out to the output buffers. (Figure 2) shows the delay increase diagrammatically.

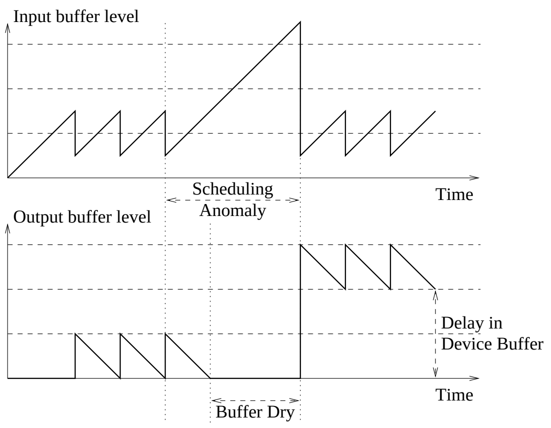

Figure 2: Accumulation of delay in output buffer

The extra buffered audio samples in the device driver prevent additional gaps, as there is now extra audio to play out while the audio tool process is not executing. Current Internet conferencing audio tools like VAT [14], NeVoT [15] and IVS [16] rely on this effect. During a talkspurt, the first CPU starvation causes a gap in the audio. Successive starvations will only cause a gap if they are longer in duration than the longest starvation so far. Consequently, the delay caused by the buffered samples in the audio device increases up to the length of the maximum interruption. At the end of the talkspurt the accumulated delay is zeroed, as transfer of samples to the audio device stops, and the audio device output buffer drains. If silence suppression is not enabled in the transmitting tool, then the increase in delay will persist throughout the duration of the session.

5 Adaptive buffer solution

In order to eliminate gaps in the audio signal, and minimise delay in the device driver buffer, adaptive control of the buffers is sought. The adaption algorithm will trade off the two variables. The adaption algorithm controls the amount of audio in the buffers, and ensures that there is always a minimum amount of audio to reduce the gaps caused by scheduling anomalies. Monitoring a history of the size of scheduling anomalies means that excessive audio is not buffered.

Intuitively, a situation where the amount of samples in the device driver buffer (called the cushion!) can cover for small frequent interruptions is ideal. This principle will not introduce excessive delay, but large infrequent interruptions will still cause a gap in the audio signal. The size of the cushion must be determined by the current performance of the workstation, which can be estimated by analysing a history of scheduling anomalies. As new processes start, or external events happen, the overall perceived load and behaviour of the scheduler will change. In order to maintain the desired sound quality and minimise the delay, the cushion size must be adaptive, and should attempt to reflect the state of the workstation.

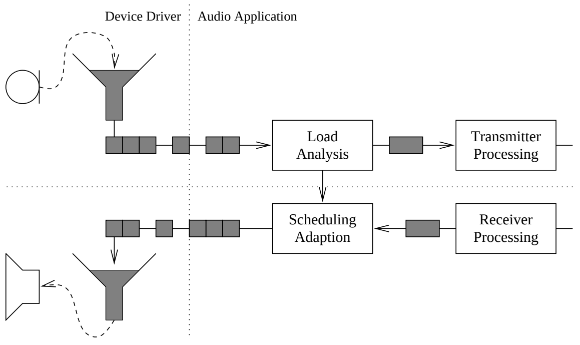

Figure 3: Adaptive cushion algorithm

5.1 Measuring workstation state

An estimate of the state of the workstation can be achieved by a simple modification to the structure of the audio application.

In the audio tool application loop, instead of reading a block of fixed size from the audio device driver, a non-blocking read is made, and all the stored audio is retrieved. The amount of audio gives an exact measure of the amount of time that has elapsed since the last time the call was made.

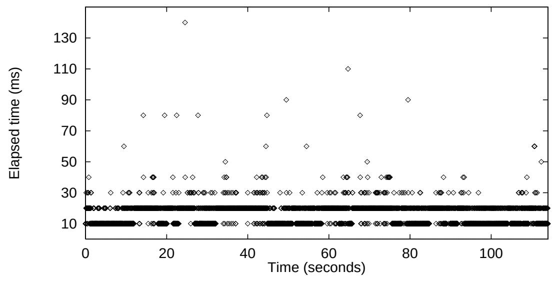

Figure 4: Read length variation

A history of the time between successive calls reflects the loading of the workstation. Figure 4 shows measurements from a lightly-loaded SUN Sparc 10 workstation over a period of two minutes. There are readings as often as every 10ms, which gives enough load samples to be able to monitor load changes as they happen.

There is an exception event to cater for in the implementation; when the amount of audio that is returned by the read call is equal to the total size of the device driver buffer. It is likely that an overflow has occurred, and some input samples have been lost, and therefore the track of time. The audio tool process resynchronises using external mechanisms like the workstation clock. This situation does not happen very often, since the total length of the device driver buffer is configurable, and is usually set large enough to cater for a few seconds of continuous audio input.

5.2 Cushion size estimation

Based on the workstation load information, the target fill level for the device driver buffer can be estimated.

Figure 5 shows the distribution of the measurements presented in figure 4. The X axis represents the elapsed time in milliseconds. Y axis is logarithmic and shows the number of times each different value occurred. It can be seen that the vast majority of measurements are smaller or equal to 40 milliseconds (320 samples). However the largest measurements are 130 milliseconds long.

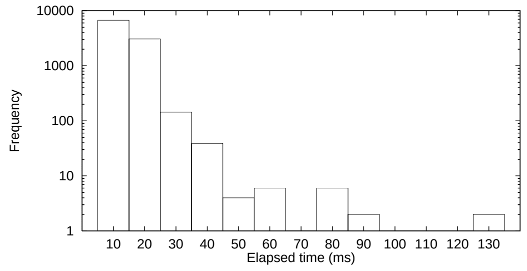

Figure 5: Load measurements distribution

The estimation algorithm should maintain a cushion level that will cater for the majority of scheduling anomalies, without resulting in gaps in the playback. This might be trivially achieved by maintaining output buffer levels at the maximum measured read length, but this will result in excessive delay being introduced. The adaptation algorithm introduced here allows the user to specify how much audio breakup should be traded for a reduction in delay.

The algorithm operates by maintaining a history of past load measurements. After a new measurement, the desired cushion size is estimated based on the recent history. The load measurements are stored in a circular buffer, and to avoid the processing overhead of examining all the logged data, a histogram of load measurements is maintained. The histogram is incrementally built, and each time a new load measurement is made, the oldest one in the circular buffer is removed from both the buffer and the histogram, and replaced with the new one. An estimate is made by examining the histogram, and calculating a cushion value that will cover a given percentage of measurements.

Let b_i be the number of load samples of length i and h the length of the history. Then we have:

$$h = \sum_{i=1}^{\infty} b_i$$

If t is the threshold of read lengths which is being catered for, then the cushion c is given by:

$$c = \min\ x : \sum_{i=1}^{x} b_i > t$$

The cushion adaptation algorithm is configurable by adjusting the history length h and by varying the threshold t of load measurements that are going to be catered for.

The desirable behaviour would be to follow the trend in scheduling load, and avoid rapid jumps in cushion size due to very short-lived bursts. This can be achieved by increasing the history length, which in effect low pass filters the load information. The increase in the history length, however, is at the expense of fast adaption since the cushion estimate now depends on older data.

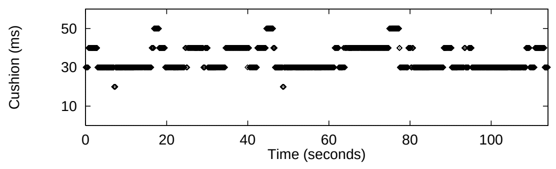

Figure 6: Fast adapting cushion

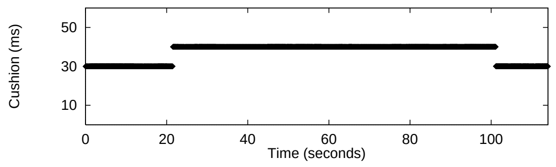

Figure 7: Stable cushion

Figure 6 shows the estimated cushion for the data presented in figure 4. A relatively small averaging period was used (200 measurements), and therefore the cushion varies continuously in an attempt to match the changing workstation performance. Figure 7 shows the estimated cushion using a larger averaging period (2000 measurements). The estimate is a lot more stable and represents the trend in workstation load. By increasing the averaging period, a reduction in average delay and a small increase in audio gap has resulted.

It is important to note that the algorithm design aims to smooth out short-lived scheduling anomalies, and cannot provide a solution if the workstation cannot (on average) keep up with the requirements of the real-time process. This can be detected if there is a constant trend in cushion increase. In this case, the audio tool should reduce the amount of processing performed (load adaption). This work is planned as part of the EPSRC RAT project [13].

5.3 Buffer adjustment

Maintaining a given amount of audio in the device driver buffer can easily be accomplished. The output buffer is initialised with a given number of samples (cushion size). On each cycle of the program, we calculate how many samples remain in the device driver buffer using the elapsed time given by the read length:

$$\text{samples left in cushion} = \text{cushion size} - \text{read length}$$

The cushion is refilled by writing out the required number of samples. An outline of the code that achieves this is given in figure 8.

```
/* Initialise cushion */
write(fd, mixed_audio_buffer, cushion_size);
for (;;)
{
    /* Read all there is (non blocking) */
    elapsed_time = read(fd, input_buffer, MAX_SIZE);
    /* Refill cushion */
    write(fd, mixed_audio_buffer, elapsed_time);

    /* Do all other processing */
}
```

Figure 8: Maintaining the cushion size

When the elapsed time exceeds the cushion because of a scheduling anomaly, then we shouldn't write back as much as we read, because this will effectively increase the cushion size. The cushion should be re-initialised by writing out its full size in this case. To maintain synchronisation between the input and playback operations the extra samples that were not written out are discarded. As these samples correspond to the silence period that resulted from the cushion overrun, no additional disturbance results from us discarding them and the timing relationship in the played out signal is maintained.

5.3.1 Varying the cushion size

The methods described here vary the cushion size in line with the desired value. The cushion size reflects the current state of the workstation, which may change in response to other processing on the workstation.

Changing the cushion size during silence periods has minimal effect on the perception of audio [20]. It is thus preferable to make any adjustments necessary during such periods.

However, it is possible to alter the cushion size during a talkspurt. A decrease in size will result in a reduction of the end-to-end delay that will be perceived at the end of the talkspurt. This can be achieved by simply writing out fewer samples than those needed to refill the current cushion. At this stage we have to decide what to do with the remaining samples that we did not transfer to the audio device. The simplest solution is to discard them. If the change in cushion size is small enough, as is usually the case, then the missing samples will not be noticed [20].

Increasing the cushion size is not as easy as decreasing it, since extra samples are needed. However, there may be extra audio available if the decision to increase the cushion follows a scheduling anomaly (see section 5.3). If there isn't extra audio available to fill the gap, then it has to be artificially created. Techniques for artificially creating extra audio required during periods of sample loss have been extensively studied by the authors [3, 5].

6 Discussion of results

The adaptive cushion algorithm has been implemented and tested in RAT. It has produced a perceivable reduction in the end-to-end delay of the system.

A simulator was built to evaluate the performance of the adaptive cushion algorithm. The simulator inputs load measurements and talkspurt information, and uses these to calculate the resulting delay and audio gaps for different adaptation strategies. In our experiments, we used real load measurements collected from RAT, and modelled talkspurt and pause durations after statistical data given in [21]. The data presented below uses load information collected on a medium loaded Sun Sparc 10 workstation running Solaris 2.4 over a period of 20 minutes. Results were collected during a multicast multimedia conference, while transmitting and receiving audio with RAT and moving video with vic [22].

Adaptation	Read %	Avg del	Del σ	Max del
No cushion		43	36	240
195/200	97.5	31	4	70
970/1000	97	30	2	40
1800/2000	90	30	0	30
1970/2000	98.5	32	4	40

Table 1: Audio delay results (in ms)

Table 1 shows resulting end of talkspurt delay values for different adaptation conditions. The first row represents operation of the audio tool without the use of the cushion mechanism - as is used in all other existing audio tools. The remaining rows represent results from the use of different parameters in the adaptive cushion algorithm. The two values in the column describing the adaptation parameters show the number of load measurements that are required to be covered by the cushion and the length of the history that is kept. The column labeled ”Read %” gives the ratio of these two values indicating the percentage of measurements that are expected to be covered by the cushion. For each adaptation method the average, standard deviation and maximum of the delay are given in milliseconds.

Table 2 shows resulting audio gaps in talkspurts for the same adaptation conditions.

Adaptation	Read %	Avg gap	Gap σ	Total gap
No cushion		34	43	14340
195/200	97.5	21	47	8760
970/1000	97	21	48	9010
1800/2000	90	22	48	9090
1970/2000	98.5	20	46	8520

Table 2: Audio gap results (in ms)

It can be seen that the adaptive cushion algorithm results in improvement to both gap size and delay compared to the standard mechanism used in other audio tools.

7 Conclusion

The recent interest in multimedia conferencing is a result of the incorporation of cheap audio and video hardware in today’s workstations, and also as a result of the development of a global infrastructure capable of supporting multimedia traffic - the Mbone. Audio quality is impaired by packet loss and variable delay in the network, and by lack of support for real-time applications in today’s general purpose workstations.

This paper has presented an adaptive cushion algorithm that copes with the problems presented to real-time audio conferencing applications by a general purpose operating system. The continuity of the audio signal is ensured during scheduling anomalies by using the buffering capabilities of the audio device driver. The algorithm also restricts the amount of audio in the output audio device buffer to minimise the end-to-end delay. Negligible overhead in processing power is incurred with the adaptive cushion algorithm since a history of workstation load is built up incrementally over time.

The results presented in this paper show that there is a significant improvement in both minimisation of delay and audio gap size to be obtained from using the adaptive cushion algorithm.

8 Further work

Work is continuing in this area to tune the adaptive cushion algorithm. In particular we hope to identify a mapping between different user preferences and suitable history lengths for multi-way interactive multimedia conferences.

9 Acknowledgements

We would like to thank the multimedia group at UCL and in particular Jon Crowcroft, Mark Handley and Angela Sasse for their valuable input.

References

[1] Stephen Casner and Stephen Deering. First IETF Internet audiocast. *ACM Computer Communication Review*, 22(3):92–97, July 1992.

[2] S. Deering. Host extensions for IP multicasting. Request for comments RFC 1112, Internet Engineering Task Force, August 1989. Obsoletes RFC 0988 1054.

[3] Vicky Hardman, Martina Angela Sasse, Mark Handley, and Anna Watson. Reliable audio for use over the Internet. In *International Networking Conference (INET)*, 1995.

[4] Lixia Zhang, Stephen Deering, Deborah Estrin, Scott Shenker, and Daniel Zappala. RSVP: a new resource ReSerVation protocol. *IEEE Network*, 7(5):8–18, September 1993.

[5] V. Hardman, Kouvelas I., M.A. Sasse, and A. Watson. A packet loss Robust-Audio tool for use over the Mbone. Research Note RN/96/8, Dept. of Computer Science, University College London, England, 1996.

[6] Newton Faller. Measuring the latency time of realtime Unix-like operating systems. Technical Report TR-92-037, International Computer Science Institute, Berkeley, California, June 1992.

[7] Olof Hagsand and Peter Sjodin. Workstation support for real-time multimedia communication. In *Usenix Winter Technical Conference*, San Francisco, California, January 1994.

[8] Sandeep Khanna, Michael Sebrée, and John Zolnowsky. Realtime scheduling in sunos 5.0. In *Usenix Winter Technical Conference*, 1992.

[9] Sape J. Mullender, Ian M. Leslie, and Derek McAuley. Operating-system support for distributed multimedia. In *Usenix Summer Technical Conference*, Boston, Massachusetts, June 1994.

[10] Tom Fisher. Real-time scheduling support in Ultrix-4.2 for multimedia communications. In *Third International Workshop on network and operating system support for digital audio and video*, pages 282–288, San Diego, California, November 1992. IEEE Communications Society.

[11] M. Handley, P. Kirstein, and M. A. Sasse. Multimedia integrated conferencing for european researchers (MICE): piloting activities and the conference management and multiplexing centre. *Computer Networks and ISDN Systems*, 26(3):275–290, 1993.

[12] J. Buckett, I. Cambell, T. J. Watson, M. A. Sasse, V. J. Hardman, and A. Watson. ReLaTe: Remote language teaching over superJANET. In *UKERNA 95 Networkshop*, University of Leicester, March 1995.

[13] A.S. Sasse and Vicky Hardman. Multi-way multicast speech for multimedia conferencing over heterogeneous shared packet networks (rat - Robust-Audio tool). EPSRC project #GR/K72780, February 1996.

[14] Van Jacobson and Steve McCanne. The LBL audio tool vat. Manual page, Available from ftp://ftp.ee.lbl.gov/conferencing/vat/, July 1992.

[15] Henning Schulzrinne. Voice communication across the Internet: A network voice terminal. Technical Report TR 92-50, Dept. of Computer Science, University of Massachusetts, Amherst, Massachusetts, July 1992.

[16] T. Turletti. The inria videoconferencing system (ivs). *ConneXions - The Interoperability Report*, 8(10):20–24, October 1994.

[17] L. R. Rabiner and Schafer R. W. *Digital Processing of Speech Signals*. Prentice Hall, 1978.

[18] Paul T. Brady. Effects of transmission delay on conversational behavior on echo-free telephone circuits. *Bell System Technical Journal*, 50:115–134, January 1971.

[19] Van Jacobson. Multimedia conferencing on the Internet, August 1994. SIGCOMM '94 Tutorial.

[20] John G. Gruber and Leo Strawczynski. Subjective effects of variable delay and speech clipping in dynamically managed voice systems. *IEEE Transactions on Communications*, 33(8):801–808, August 1985.

[21] Paul T. Brady. A statistical analysis of on-off patterns in 16 conversations. *Bell System Technical Journal*, 47:73–91, January 1968.

[22] Steve McCanne and Van Jacobson. vic: A flexible framework for packet video. In *Proc. of ACM Multimedia '95*, November 1995.