

# Sicsophone: A low-delay Internet telephony tool

Olof Hagsand<sup>1</sup>, Ian Marsh<sup>2</sup> and Kjell Hanson<sup>3</sup>

<sup>1</sup> LCN Laboratory, IMIT, Royal Institute of Technology, Sweden  
olofh@kth.se

<sup>2</sup> SICS AB, Stockholm, Sweden  
ianm@sics.se

<sup>3</sup> Prosilient Software AB, Stockholm, Sweden  
kjell@prosilient.com

**Abstract.** The end to end delay is a critical factor in the perceived quality of service for Voice over IP applications. Sicsophone is a complete VoIP system that couples the low level features of audio hardware with a standard jitter buffer playout algorithm. Using the sound card directly eliminates intermediate buffering as well as providing fine control over timers needed by a soft real-time application such as VoIP. A statistical based approach for inserting packets into audio buffers is used in conjunction with a scheme for inhibiting unnecessary fluctuations in our system. We also present mouth-to-ear delay measurements for selected VoIP applications and show that several hundreds of milliseconds can be saved by using the techniques described in this paper. A prototype for both UNIX and Windows platforms has been implemented, demonstrating that our system adapts to network conditions whilst maintaining low delays.

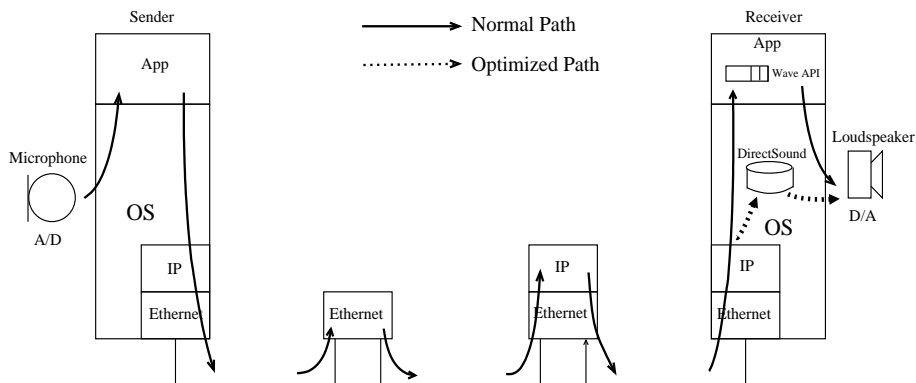
## 1 Introduction

Users of interactive VoIP applications demand low latency conversations. Replaying packetised audio requires that sufficient packets are available to the application in order to avoid gaps or glitches. The digital to analog conversion of sampled voice requires strict, synchronous timing despite the fact that the network and operating system may disrupt the process. The most common method to solve this problem is to introduce a small intermediary buffer between the decoded audio stream and the audio hardware which allows packets to be available for playout. Of course withholding packets instead of immediately playing them increases the total delay for a VoIP user. However, the longer packets can be delayed, the more resilient the receiver is to adverse network conditions. We should already point out that Sicsophone is a working implementation and that the algorithmic complexity is an important factor. We motivate this approach with real delay experiments and results. Hence the goal is not to compare the merits of various playout algorithms, this has been covered by many researchers, rather to give some insight into which issues are important when realising these schemes in real systems.

In this paper we refer to the mouth-to-ear delay as the total one way delay experienced by two speakers including the analog-digital-analog conversion. By jitter we mean the variability in the packet delay. This variability is the reason we need to buffer packets, thus our work focuses on how to detect and compensate for packet jitter in an efficient manner. Our solution is to insert packets directly into the memory of the sound card relieving the need for time consuming data copying operations or context switching. This approach saves precious time, avoids scheduling problems but requires careful buffer management.

Figure 1 illustrates the complete path of audio samples from a microphone at a sender to the loudspeaker at a receiver. Traditionally, a sender writes voice samples to the operating system which are subsequently sent across the network to a receiving host. At the receiver, data is read from the operating system interface where it is the responsibility of the application to adjust the buffer size as required, the traditional data path is shown by the solid lines in the figure.

In our approach we use the buffering available on sound cards and copy the packet payloads directly into this area. Therefore, we save copying the data to and from the application, plus not performing the de-jittering in the application. We describe our approach in the context of DirectSound [BD98] subsystem on the Windows platforms. It is important to point out that our approach is not confined to this architecture, a ring buffer with pointer support is sufficient to realise the ideas presented in this paper (alternatives for UNIX include [Riz97] or [Ree98]). However we describe the system using DirectSound as it is known to many developers, and was used in our experimental evaluation. It is important to add that we assume the end system is not under heavy load or consider Sicsophone as a hard real-time system.



**Fig. 1.** Sicsophone audio delivery path

If we now look at the steps a receiver must take in order to replay packetised audio from a network in more detail, Table 1 shows four such typical steps. Firstly, de-packetisation, removes the IP and UDP headers and passes the data-

gram together with a Real Time Protocol (RTP) payload to a VoIP application. This step takes a few milliseconds on most systems. This step involves copying the data from the operating system to the application. Step two is to decode the sound samples, this is dependent on the compression scheme as well as the packet size used. Typically this takes from a few milliseconds to tens of milliseconds. Step three is the absorption of network delays through buffering. This is typically under the control of the Internet telephony application. Step four is delivery to the sound application, which usually means copying the data back to the operating system for it to copy the data (again) to the sound system. Our goal was to consolidate some of these steps into a single step, saving the time of intermediary buffering and context switching. We refer to this approach, solely for definition by its software name, Sicsophone.

Step	Process	Overhead	Depends On
1	De-packetisation	10 - 50	Pact. Size
2	Decoding	10 - 50	Coding
3	Buffer Delay	5 - 200	Network
4	Delivery	5 - 120	End System

**Table 1.** Typical receiver incurred delays (ms)

The remainder of this paper is organised in the following manner; Section 2 forms the main body of this work, the low-level adaption of playout buffers using ring buffers. Section 3 presents results of Sicsophone’s performance for mouth-to-ear tests. We also give comparisons of the playout delay of Sicsophone against idealised cases. Section 4 is a description of related efforts with which this paper has commonalities, and we round off the work with some conclusions in Section 5.

## 2 End-system adaption to jitter

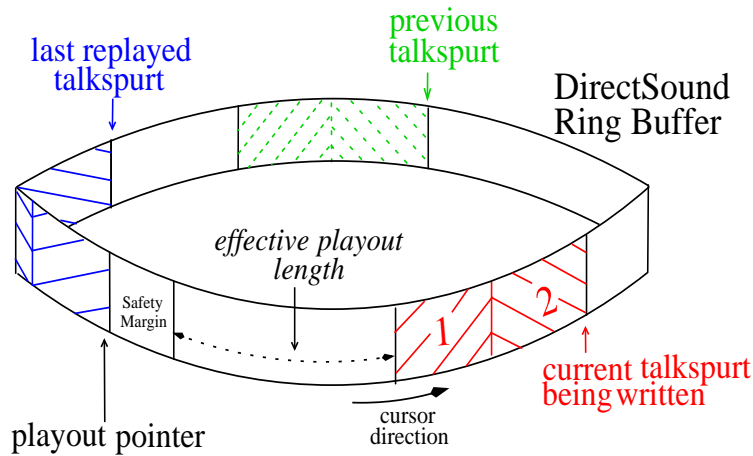
### 2.1 Buffering issues

In this section we outline some issues associated with the data buffering scheme we have chosen. Our goal is to save time by avoiding data copying, setting up direct memory access (DMA) transfers ahead of time, using simple data structures and inserting de-jittered audio packets directly into the memory of the sound card. Direct memory access is used to move data from memory to the sound card and vice versa without intervention of the CPU. Using DMA does not directly provide a time saving, as it can take some time to set up the transfer. However once it is done, the transfer can be done much quicker and more efficiently. This offers significant time savings over posting an interrupt for every packet, particularly in the older (and non-DirectX) versions of the Windows operating systems.

One potential problem of using the sound card memory as a buffer is it could be overrun by packets arriving too quickly. Modern sound cards however are equipped with megabytes of RAM to store down-loadable sound samples, DirectSound can allocate buffers up to this physical size when a hardware buffer is initialised. We do however keep the buffer from being overrun by mechanisms explained in Section 2.3. Another potential cause of overrun or underrun is misaligned or drifting clocks, there is no mechanism in Sicsophone to combat this.

Another important but often overlooked issue is mixing. For an application like VoIP where the voice channel is stopped and started continuously, we would like to minimise this setup time. Valuable time can be lost by setting up mixers for software and hardware buffers where we normally do not want to mix audio from different sources. Therefore we allocate a DirectSound primary buffer to give better delay characteristics, as it does not need to be mixed before output to D/A conversion.

The coding scheme used is another issue. Packets have to be decoded before insertion into the buffer if they are not in PCM format. However using PCM allows us to DMA the payload into the sound card memory without any audio format conversion. This is the *fastest* path from packet reception to playout. It is however possible to support other audio formats, however they require extra CPU cycles for decoding the audio, and a small buffer to hold the data before and after decoding. Using buffers in this manner makes the assumption a certain number of bytes in the buffer corresponds to a well defined playout time. This is the case for coding such as PCM but not for highly compressed audio formats.



**Fig. 2.** DirectSound buffer structure

Figure 2 shows the interface offered by DirectSound. Data is written at the write pointer and replayed by the trailing playout pointer. The read and write pointers are updated by the system, and continuously encircle the buffer. Read-

ing and writing the pointers requires that the system almost instantaneously updates their current positions. Some of the older operating systems used in our tests did not give sufficiently fine granularity over the positions of the timers. In Sicsophone they are used as **both** timers and pointers. They function as a timer by indicating if a packet is too late. If the read pointer has already passed the point where a packet should be, and it has not been written, then we know that the next packet is late. Insertion is simply a modulo operation and a buffer copy. In order not to replay old data written into the buffer when no packets are being sent, we must write ‘silence’ samples into the ring. This is because some background noise is needed so that the aware the communication path is still open. Pure silence can be quite disconcerting.

Used as pointers, the read and write pointers give memory locations where data is read from or written to depending on the operation to be performed. Given these pointers it is simple to adjust the effective buffer length. It is the position of the read pointer relative to the write pointer. The closer the read pointer is to the write pointer the shorter the effective buffer length will be. Note there is a small margin of 15ms in front of the read pointer to allow data that has been written to be “ready” for playback. Use of this safety margin is recommended by Microsoft.

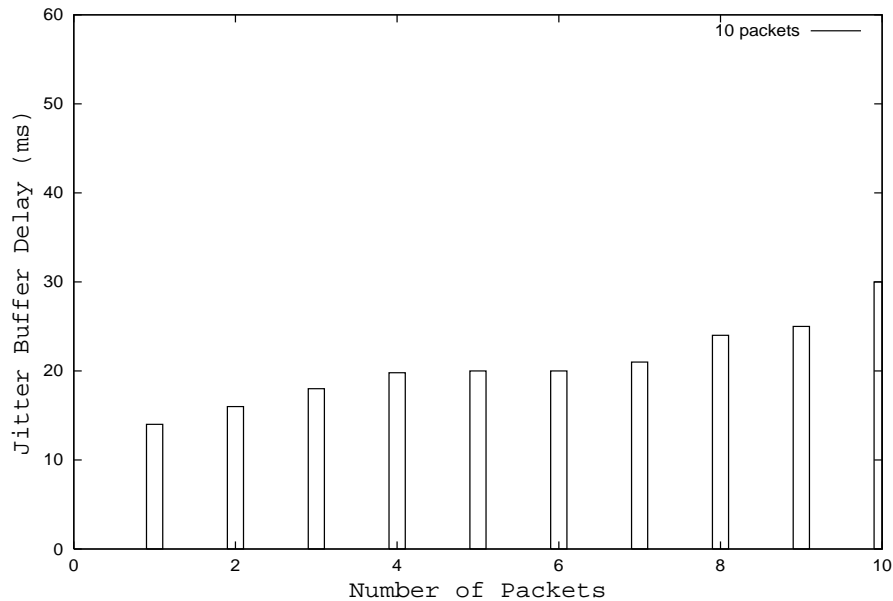
To give a concrete example, using an estimate of the network delay and its variation, described previously, we insert packets at a specified “distance” ahead of the read pointer. Therefore a translation from milliseconds to bytes is needed;  $bytes = (samples/sec \cdot bits/sample \cdot P_i) / 8000$ . For example, if one substitutes 8000 for samples/sec, 8 bits per sample and 200ms for the playout point this equals 1600 bytes. This means that the write pointer can simply be set 1600 bytes in front of the read pointer. The safety margin should also be added to this value, which corresponds to an additional 125 bytes. To re-iterate once the playout point has been calculated, it is trivial to insert packets into the buffer, no complex data operations are needed.

## 2.2 Fast startup adaption

In an adaptive VoIP application we normally consider changing the buffer size during a silence period so as not to introduce audible glitches in the analog audio stream. Since the goal of this work is to produce a low-delay VoIP tool we would like to keep the buffer length as small as possible. However in the startup phase we have little knowledge of the network condition and therefore have to use default values for the network delay<sup>1</sup>. We therefore adjust the buffer length after monitoring only a few packets to find a fast estimate quickly.

Figures 3 and 4 show packet delay in the jitter buffer during the start up phase of Sicsophone as an example. The y-axis shows the waiting time in the buffer (in ms) and the x-axis shows the number of packets received, sorted by the time spent in the buffer, note this is **not** the sequence number. It only indicates the number of packets and their respective waiting times.

<sup>1</sup> Sicsophone’s defaults are initial values of a 20ms minimum and an initial maximum of 60ms.



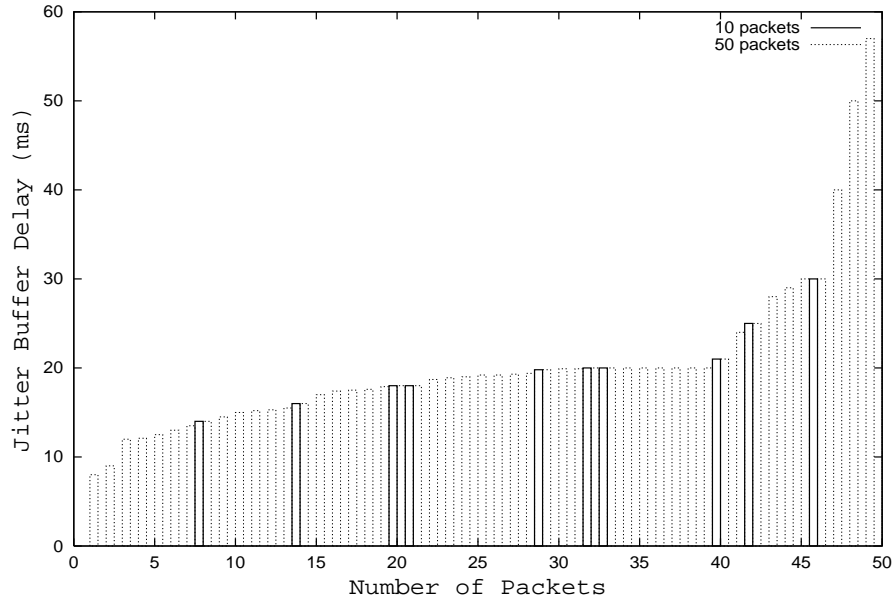
**Fig. 3.** Jitter for the first ten packets

Figure 3 shows the buffer state after ten packets have been received and Figure 4 after an additional 40 packets have arrived, the original ten are shown with somewhat bolder lines. After ten packets were stored, the time spent in the buffer varied between 14 and 30 ms whereas after 50 packets have been received the median delay incurred is approximately 20 ms.

Fast adaption is worthwhile during the start up phase of a VoIP session. The alternative approach is to be conservative in the start up phase and have long playout buffers until a value for the playout point can be calculated or a RTCP report received. In the presence of delay spikes [RKTS94] we can re-estimate the jitter value quickly. Since the goal of this work is to produce a low delay VoIP tool we chose quick adaption. Furthermore, usually there are sufficient silence periods during the startup phase of a conversation to perform fast adaption. In the case where there are none (such as call waiting, i.e. music playing) we adjust the buffer when a packet is excessively delayed or if there is a loss. Failing these possibilities we adjust the buffer length and possibly induce an audio glitch.

### 2.3 Bounding the estimated network delay

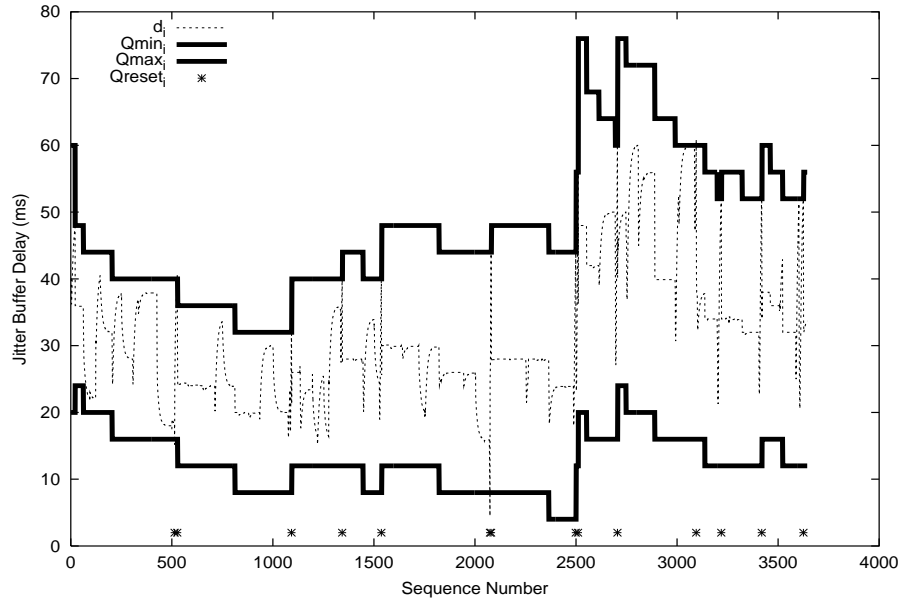
Sudden increases in the network delay can cause problems for a VoIP application. They typically result in a sequence of dropped packets whilst the receiver buffer is estimated and resized. A spike is referred to a sudden and rapid increase in the network delay which is typically short lived, often less than one round trip time. One solution is to track the increase in the delay and adjust the buffer



**Fig. 4.** Arrival of 10 and 50 packets

length accordingly. The alternative is to include the values of the jitter estimate but not to adapt the buffer size directly. It is because of this temporal property that has led us to be more conservative to network conditions after setup and not to adapt the DirectSound buffer length to sudden and transient changes in the network delay.

We have implemented a system which bounds the delay jitter estimate. As stated, the spikes are not completely ignored but we do not react immediately to their presence. The estimated jitter value should vary between an upper  $Q_{max_i}$  and a lower  $Q_{min_i}$  (see Figure 5) bound in a range, where  $Q_{min_i} < d_i < Q_{max_i}$ . If the running estimate breaks either of the boundaries we re-calculate the new buffer length, taking into account the value of the spike, but reset the mean estimate to the middle value of this new range. The values of  $Q_{min}$  and  $Q_{max}$  are calculated using a simple running average method. Figure 5 shows an example of a receiver jitter buffer during a call between two machines on a local network. The y-axis shows the jitter buffer length and the x-axis the sequence number. The system starts with  $Q_{min}$  and  $Q_{max}$  set to the default values of 20ms and 60ms for the minimum and maximum bounds respectively. At the bottom of the figure we show the range breaks as stars to highlight them. We have found this scheme to work relatively well, in the given trace there were only 14 breaches of the range from over 3600 packets (less than 0.5%). This example shows only a LAN example, however WAN facets are included in the next section. More importantly we did not make unnecessary changes to the DirectSound primary buffer.



**Fig. 5.** Bounded jitter buffer playout delays

### 3 Evaluation and results

We divide this part into two sections, the first gives the total delay of popular VoIP tools compared to Sicsophone in a laboratory environment, with basically no, or little, network delay. Secondly, we show the performance of the playout algorithm using data taken from Internet measurements. This allows us to account for Sicsophone’s WAN delay by comparing it with the best possible playout delay. This is done by post-processing measurement data. We chose to give the results in this manner to estimate the total mouth-to-ear delay by including both the WAN and LAN delays. In other words, the first set incorporates the delay due to coping with the operating system and the second with the network conditions.

#### 3.1 Mouth-to-ear measurements

The delay reduction by Sicsophone is the main result of this paper. We performed one way mouth-to-ear measurements with a range of VoIP tools with the results summarised in Table 2. It’s important to state that no parameter tweaking of these tools was done, we used their default installation values. The experimental setup used was as shown in Figure 1. We used a signal generator which generated a 1Hz square signal. The square wave serves as a trigger, the signal is packetised and sent over the IP network and played back through the loudspeaker at the

destination. The square wave is detected by an oscilloscope and the difference in time between the waves was measured.

Audio Tool	Latency (ms)
Sicsophone prototype	25-100
Ericsson Lanphone	300
Vocal Internet Phone 4.5 (SB)	450-550
Vocal Internet Phone 4.5 (PJ)	580-620
NetMeeting 2.1 (SB)	620
NetMeeting 2.1 (PJ)	750
VAT 3.4 (Solaris)	1200
RAT 3 (Solaris)	1500

**Table 2.** Mouth-to-ear latency measurements (SB=SoundBlaster and PJ=PhoneJack)

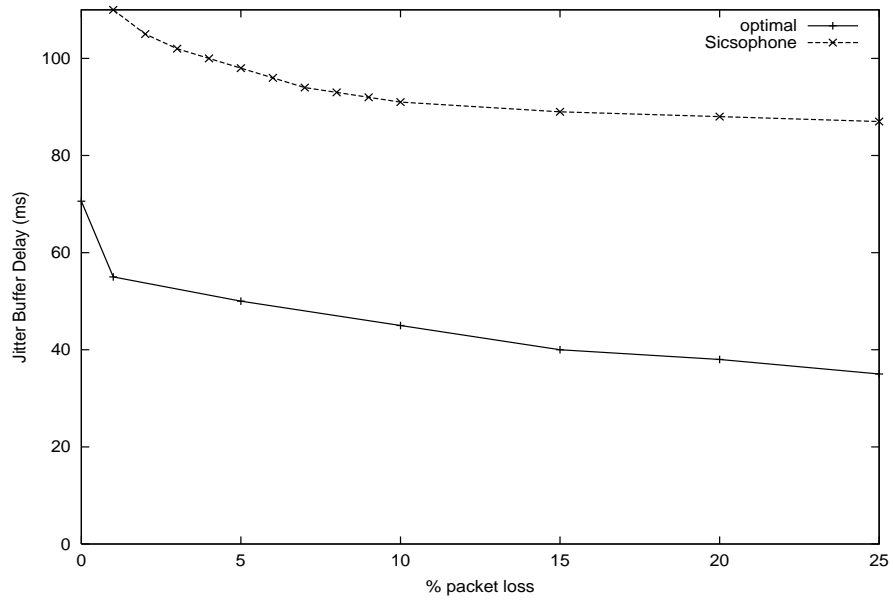
The measurements were done using a signal generator feeding a sender and an oscilloscope to measure the time difference between the sender and receiver. In retrospect it would be better to add measurements that stressed the playout buffer, by using real speech rather than just a pulse. This was chosen so as to simply trigger (and calculate) the end-to-end delay. We can see that there are large variations between the various applications. One important result of this paper is to highlight the design of end systems for VoIP applications. Considerable time savings, 10's to 100's of milliseconds, can be saved by using an approach similar to the one described.

### 3.2 Comparison with ideal playout conditions

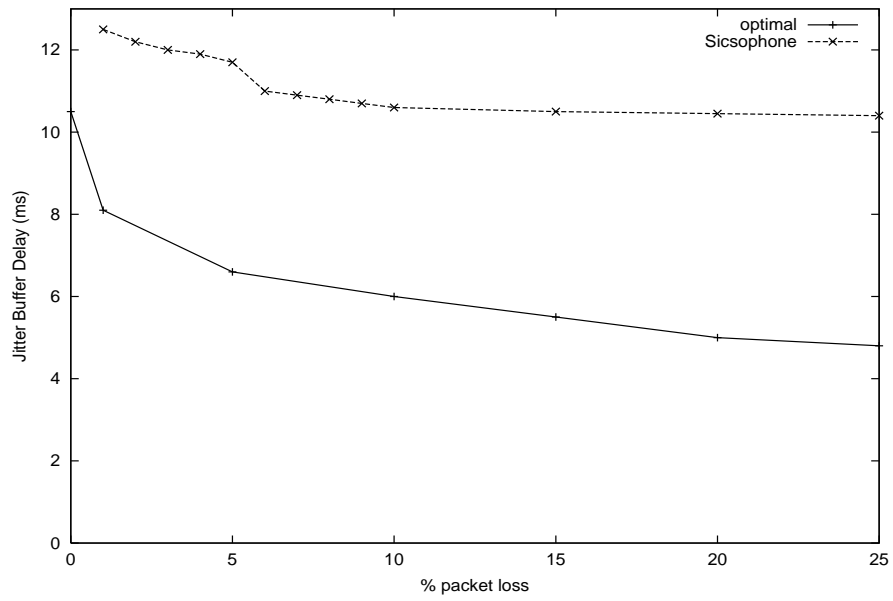
In the introduction we have eluded to a jitter buffer playout algorithm essentially has to tradeoff either delay or loss depending on the arrival process. Low delay implies a short playout buffer, incurring higher packet loss due to late arrivals. Whilst longer buffers reduce loss, they introduce delay into the system. When comparing the performance of algorithms it makes sense, therefore, to consider *both* loss and delay.

Figures 6 and 7 show the results for two Internet trace files. To calculate the optimal playout point we order all the packets and remove the 1% with the highest delay. We then calculate the delay needed to play the remaining 99% of the packets resulting in the delay for 1% packet loss, this process is repeated up to 25% packet loss (although in practice more than 10% would be deemed unacceptable for PCM). Figures 6 and 7 show Sicsophone's playout delay performance in comparison with an optimum.

In Figure 6 Sicsophone is about 50ms from the ideal playout point and remains more or less constant as the packet loss increases. For a given loss rate, e.g. 5%, Pinto and Christensen [PC99] quote a slightly lower delay than Sicsophone, 72ms compared to 98ms. The result in this case is due to the large variation



**Fig. 6.** Playout delays for a trace from UCI, California to INRIA, France



**Fig. 7.** Playout delays for a trace from Amherst, Mass to GMD, Berlin

of jitter ( $\pm 20\text{ms}$ ), which makes it hard to settle to a constant value for the minimum buffer length, which can be verified by looking at the absolute jitter. A second test is shown in Figure 7 which shows a trace from the University of Massachusetts in Amherst to GMD in Berlin. In this case the jitter is lower and the difference between the optimal and Sicsophone is less than 5ms. We have shown worst and best cases from the measurement data available at that time.

## 4 Related work

The early 90's produced a surge in packet audio playout research. One of the first efforts to implement a voice application on an IP network with an adaptive buffer playout strategy was NeVoT [Sch92]. The playout algorithm implemented in Sicsophone is almost identical to NeVoT [Sch92]. They used a variation estimate similar to the one given earlier, however they make a slight distinction for the first packet in a talkspurt and subsequent ones. The playout for the first packet is delayed longer due to lack of information on the network state after the silence period. Our work shares theirs in the choice of a ring buffer for buffering packets, only we perform the copying by using DMA transfers directly rather than copying the data from the application to the operating system. VAT (Visual Audio Tool) [JM92] was a well known VoIP tool that implements a playout buffer similar to the one described, including a circular buffer to hold the packets before playout. We use an additional scheme to prevent the jitter estimates from varying too rapidly plus focus on the efficient insertion of packets into the playout buffer. Moon *et al.* [RKTS94] present four different playout algorithms for packet audio. They calculate an estimate of the network delay and jitter as an average from all the packets received. The authors highlight the jitter spikes we mentioned and also do not adapt the buffer size to these spikes. Pinto and Christensen [PC99] describe an algorithm for jitter compensation based on the target packet loss rate. Their "gap based" approach compares the current playout time with the arrival time and calculate a gap for both early and late packets. They compare the current playout delay, for any particular talkspurt in progress, with an optimal playout delay. This optimal theoretical delay is defined as minimum amount of delay to be added to the creation time of each packet which would result in a playout of a talkspurt at the given loss rate. Our calculation of the optimal playout is similar to the one described in this paper. Luigi Rizzo describes a generic sound card driver for FreeBSD [Riz97]. Aspects of this work resembles ours, in particular, handling of timers, DMA transfer and buffer size allocation. They include hooks to use the driver for VoIP applications, one such example is a `select()` call which can be scheduled to return only when a certain amount of data is ready for consumption. Rosenberg *et. al* in [RQS00] looked at combining target-based playout algorithms in conjunction with FEC schemes, and propose a number of new playout algorithms based on this coupling. Kouvelas and Hardman in [KH97] keep the flow of audio constant during high operating system load by using buffering in the audio hardware. They also look at reducing the amount of buffering in the application by keeping the buffers in the application

as small as possible. In our case we try and totally eliminate it dramatically by only using the sound card's storage possibilities.

## 5 Conclusions

In this paper we have shown how careful buffer management combined with a simple statistical playout scheme can reduce mouth-to-ear delay for VoIP applications. As stated at the start of this paper, the delay is one of the most important factors in the perceived QoS and hence has been the focus of this work. The results are encouraging as the mouth-to-ear delay of Sicsophone on a LAN is around 50ms on a Windows NT system with DirectX 8.0. We also include an estimate of the delay induced by network conditions using a standard playout algorithm. We have proposed a system which tries to reduce the perceived mouth-to-ear delay of real-time packet audio communication.

## References

- [BD98] Bradley Bargaen and Peter Donnelly. *Inside DirectX*. Microsoft Press, 1998.
- [JM92] Van Jacobson and Steve McCanne. VAT - LBNL Audio Conferencing Tool, July 1992. Available at <http://www-nrg.ee.lbl.gov/vat/>.
- [KH97] Isidor Kouvelas and Vicky Hardman. Overcoming workstation scheduling problems in a real-time audio tool. In *Proc. of Usenix Winter Conference*, Anaheim, California, January 1997.
- [PC99] J Pinto and K Christensen. An algorithm for playout of packet voice based on adaptive adjustment of talkspurt silence periods. In *Proceedings of the IEEE 24th Conference on Local Computer Networks*, pages 224–231. ACM, October 1999.
- [Ree98] Dicken Reed. A new audio device driver abstraction. In *Proc. International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, 1998.
- [Riz97] L. Rizzo. The FreeBSD audio driver. *Lecture Notes in Computer Science*, 1356, 1997.
- [RKTS94] Ramachandran Ramjee, Jim Kurose, Don Towsley, and Henning Schulzrinne. Adaptive playout mechanisms for packetized audio applications in wide-area networks. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, pages 680–688, Toronto, Canada, June 1994. IEEE Computer Society Press, Los Alamitos, California.
- [RQS00] Jonathan Rosenberg, Lili Qiu, and Henning Schulzrinne. Integrating packet FEC into adaptive voice playout buffer algorithms on the internet. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, Tel Aviv, Israel, March 2000.
- [Sch92] 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.