GRAME - Computer Music Research Lab.
Technical Report - TR-020401

Audio Cards Clock Skew Compensation over a Local Network

Dominique Fober
GRAME Research Laboratory
9 rue du Garet, BP 1185, 69202 LYON Cedex 01, France
fober@grame.fr

Abstract

This paper is the continuation of a previous work done on clock skew compensation over a high latency network. It evaluates the efficiency of the EPTMA clock skew detection algorithm applied to real-time audio streaming over a local network. The presented results include real world apparent deviations of audio card clocks and acuracy of the skew detection. It appears that EPMTA is very suitable to measure clocks deviation in the context of audio transport. Finally, a simple method to compensate for the clock skew is presented, mainly to evaluate a complete solution for audio streaming.

1. Introduction

Real-time transmission of audio buffers over a network raises the problem of the clocks synchronization. Actually, if the sender and receiver clock frequencies differ, the sender will produce either more or less data than the receiver is expected to consume, depending on which clock is faster. We'll refer later to the clock frequencies difference as the "*clock skew*".

The clock skew problem is similar when one wants to transmit discrete time stamped events. In previous works [1] [2], we have proposed a new algorithm named Exponential Peak Tolerant Midpoint Algorithm (EPTMA) to detect and evaluate the clocks deviation in real-time. This algorithm relies only on time stamped packets transmission to operate and offers several advantages:

- it doesn't require any transaction and therefore it may be used independantly of the transport protocol,
- it doesn't rely on a master/slave scheme: each receiver is independantly evaluating its clock skew relatively to the sender.

It has been initialy designed to operate on the Internet i.e. on a high latency network with significant delay jitter, but it is expected to run even better on a local network. The present work has been carried out to confirm the performances of the algorithm in the context of real-time audio streaming on a local network.

A detailled description of EPTMA can be found in [2]. The rest of this paper will mainly focus on experiments and obtained results. It is organised as follow: section 2 presents the different hardware and software components involved in the experiment, section 3 is dedicated to the results, section 4 propose a simple basic method to compensate for the clock skew and section 5 summarizes and concludes.

2. Hardware and software components

2.1. Hardware and operating systems

Experiments have been made on a local 100 Mb Ethernet network. This network was not dedicated to the experiment and was supporting the additionnal traffic corresponding to an Intranet. Five different stations have been used, running 4 different operating systems. Table 1 summarizes the hardware configurations and their associated operating systems.

name	audio card	processor	cpu Mhz	OS
bach	SB Live	AMD Athlon	1000	Linux 2.4.8
berio	SB Live	AMD Duron	700	Linux 2.4.18
marcopolo	IBook	PowerPC G3	500	Mac OSX 10.1
macdom	PowerMac G4	PowerPC G4	350	Mac OS 9.2.1
xp	SB Live	AMD Duron	700	Windows XP

Table 1: hardware and OS.

2.2. Software components

Along all the different platforms, access to the audio hardware has been implemented using PortAudio, a portable open source audio library [3] [4]. Audio setup was making use of the following parameters:

- sample rate: 44 100 kHz
- audio buffers: 256 frames
- num channels: 2

These parameters were natively supported on all the stations involved.

Network transport has been implemented using the Unix socket API except for the Mac OS 9 station which made use of the Apple Open Transport layer.

We used the User Datagram Protocol (UDP) [5] as underlying protocol layer,

2.3. Packets format

The format of the audio transmitted packets is shown on figure 1. The 8 bits *type* field is intended to discriminate different packet types. Two packet types were actually used: *audio packets*, containing audio frames and *end session packets*. The 24 bits *unused* field is only intended for structure alignement. The *serial number* is a unique number incremented at each packet transmission, intended to detect packet losses or duplicates.

```
  0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     type      |                    unused                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         serial number                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                              date                             |
|                             64 bits                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              len              |           sample 1            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            sample 2           |              ...              |
```

Figure 1: audio packets format

The 64 bits *date* field express the sender audio time in frames. The *length* field represents the data chunk length. Finally, audio frames are transmitted as 16 bits samples. The total size of an audio packet is 1074 bytes (including IP and UDP headers).

2.4. Client/Server implementation

The PortAudio API is callback based: the rate of the callback invocation is determined by the audio buffer size in frames.

The server task consists in filling an audio packet at each callback and in sending this packet on the network. According to the parameters mentionned above, a packet is sent every 5.8 ms + Δ, where Δ represent the latency of the PortAudio callback activation.

The client task is a little bit more complex: audio input is driven by a real-time thread reading the socket and audio output is driven by the PortAudio callback. As both tasks are running asynchronously, incoming samples are stored in a *ring buffer* to be later consumed by the PortAudio callback. The ring buffer includes 2 indices: one for the read location and one for the write location. The receiver starts playing when the ring buffer is half filled.

The ring buffer size was 28 times the size of the audio packet data chunk. It has always been well enough to compensate for the latency variations, including transport and software layers latencies.

2.5. Clock skew detection

The sender contribution to the clock skew detection consists only in filling the date field of the packet header using the most accurate audio time source. The implementation made use of the PortAudio function `Pa_StreamTime (PortAudioStream *stream)` which returns the current output time in samples for the corresponding stream.

On packet reception, the client evaluates the apparent clocks offset as the difference between packet time stamp and the current local time (also measured using Pa_StreamTime), then compute the latency variation *LV* as the difference between the initial apparent clocks offset and the current one. This latency variation is finally given to the EPTMA algorithm which output variation represents the clock skew evaluation. This technique is detailed in [2].

3. Results

Along all the results below, we made use of the following parameters for the EPTMA algorithm:

• window size:	60
• retained values:	10
• weighting factor:	0.01

Actually, evaluation of the latency variation and check of the clocks deviation has been performed every 20 packet transmissions which represents about 100 ms.

3.1. Latency characterization

The mean round trip time of the network has been evaluated to about 270 µsec using the ping tool. Figure 2 and 3 show the latency variation measured over about 4 minutes of transmission.

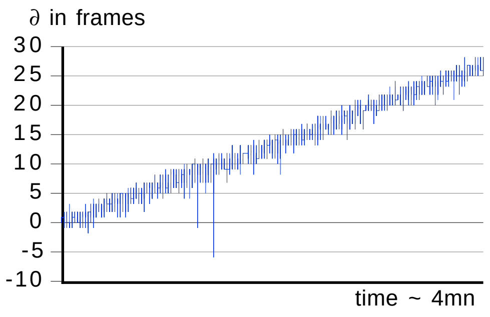

Figure 2: latency variation from linux to linux (bach to berio)

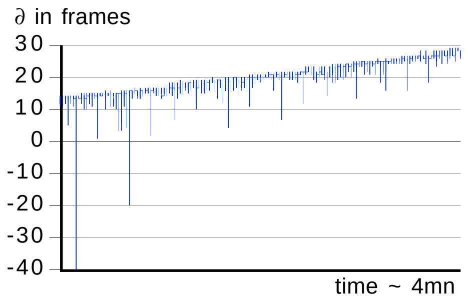

Figure 3: latency variation from linux to Mac OSX (bach to marcopolo)

It appears that the constant range of the latency variation fits in 4 frames with occasionnal peaks up to 40 frames which represents respectively 90 µsec and 900 µsec. Exceptionnaly, the latency reached 100 frames (2,27 ms).

The constant slope represents the clock skew.

Note also that the sender / receiver behavior is not symetric in regard of the latency: figures 4 and 5 shows the latency variation from macdom to berio and in the reverse way.

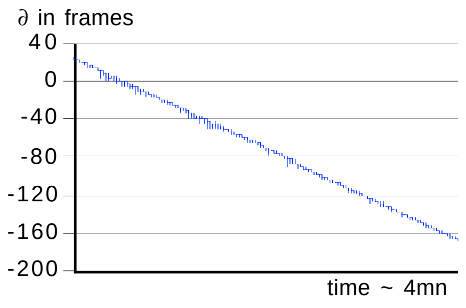

Figure 4: latency variation from macdom to berio

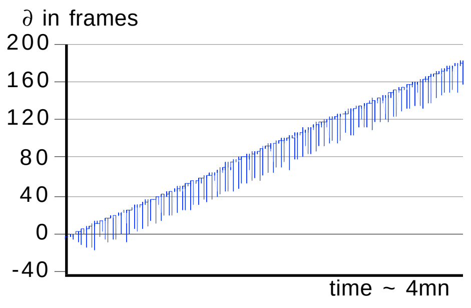

Figure 5: latency variation from berio to macdom

It appears clearly that the software layers latency predominates the network latency itself.

3.2 Clock skew characterization

Theoritically and according to manufacturer's specifications [6], cristal oscillators of sound cards are stable to ± 100 parts per million and better. Therefore we expected to measure up to two frames deviation for 10 000 consumed frames. In fact and except for the XP station, first evaluation of the clock skew showed better results. Table 2 summarizes this evaluation for the bach station. *∂ frame* represents in seconds, the time necessary for 1 frame drift at receiver side and *skew* express the receiver deviation per 10 000 consumed frames.

from -> to	∂ frame (sec)	skew
bach to berio	9,55	0,024
bach to macdom	1,09	0,208
bach to marcopolo	8,58	0,026
bach to xp	0,12	1,922

Table 2: clock skew characterization

It appears that audio time generally presents a good acuracy except for Windows XP which deviates clearly from other systems. Actually, audio time depends on audio card clocks acuracy but also on the low level functions to access this time. One of the surprising result is the difference of the measures between *bach-to-berio* and *bach-to-xp* because in fact, berio and xp represent the same station running Linux or Windows XP using the same audio card. This result is unexplained and comes probably from low level components behavior. However, as we are only interested in high level behavior, it didn't change anything for our experiments, but the question remains unresolved.

3.3 EPTMA performances

Performance of the clock skew detection algorithm is characterized as described in [2]. For the record, the system behaves like follows: at initialization stage (time 0), the receiver and sender clock offset has a given value which will continue to move up to the stable stage.

At the stable stage time, the additionnal clock deviation becomes equal to:

$$K = k \pm \mathcal{E}(eptma)$$

where k is the stable constant deviation and $\mathcal{E}(eptma)$ represents the error due to the skew profile evaluation.

The algorithm performances are characterized with both k and $\mathcal{E}(eptma)$ values: the k value is important because it is to be added to the provisions made for the latency compensation and the error $\mathcal{E}(eptma)$ denotes the stability of the system.

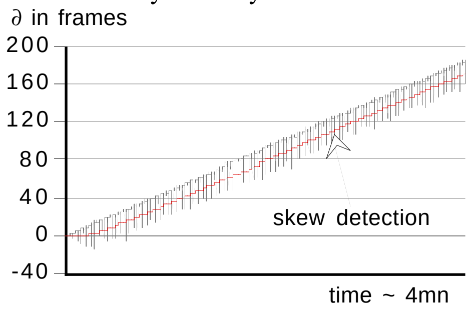

Figure 6: clock skew detection

The results presented below come from several real audio transmissions between the stations described in table 1.

The figure 6 gives an example of the detection using the latency variation shown by figure 5. The clock skew is clearly detected.

Table 3 gives the efficiency of EPTMA measured for the *bach* station. The *t* column represents the time necessary to obtain the acuracy mentionned in the last column. All the values are expressed in frames.

	from -> to	t	k	± ε
1	bach to berio	4960	0,4	0,4
2	bach to macdom	8060	11,1	0,35
3	bach to marcopolo	4960	0,9	0,45
4	bach to xp	9720	129	0,65

Table 3: EPTMA results for bach.

Even for exchanges with the *xp* station, results provided by EPTMA are pretty good: the time necessary to reach the stable state is lower than 10 000 frames and the system stability is under 1 frame deviation.

Concerning the *xp* station, results are presented by the table 4. The skew is more important and therefore the stabilization time is greater while the system stability remains under 2 frames deviation.

	from -> to	t	k	± ε
5	xp to bach	9840	132,9	0,65
6	xp to macdom	10100	-121,1	0,95
7	xp to marcopolo	9480	-132,9	0,85

Table 4: EPTMA results for xp.

The clocks deviation *k* is consequently more important. A graphical example of the system behavior in case of transmission with the *xp* station is shown in figure 7.

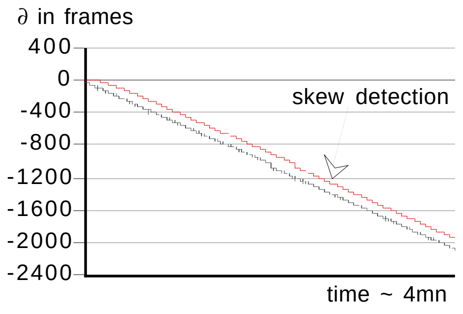

Figure7: xp to marcopolo skew detection.

4. Clock skew compensation

In order to evaluate a complete system, including detection and compensation, we have implemented a mechanism to correct the clocks deviations.

First, it has appeared that any implementation should take a great care of the available audio time source acuracy which is critical from results point of view. For the PortAudio MacOS9 implementation for example, in the lack of system support, the audio time was incremented by the size of the audio buffer (in frames) inside the audio callback, which globally means that the minimum time resolution was 512, because the MacOS9 Sound Manager don't allow smaller buffer sizes. An improvement to the PortAudio implementation has been proposed (and accepted) to get a better resolution.

Considering that modifying the audio card clock frequency is out of the possibilities of a user level application, we have decided to compensate the clock skew simply by modifying the read index inside the receiver ring buffer: depending on which clock is faster, the receiver will periodically skip one frame or read one frame twice in order to compensate the clocks deviation.

As seen above (table 2), the expected rate of the corrections is generally greater than 1 second and may reach several tens of seconds. Only exchanges with *xp* require several adjustements per second.

We have transmitted a sinus signal between the different stations in order to better detect the signal distortion. As a result, even with the *xp* station, no audible distortion has ever been noticed.

Of course, more sophisticated correction methods may be implemented on top of the EPTMA results (such as interpolation for example) but it is out of scope of the present work.

5. Conclusion

We expected to show the suitability of the EPTMA algorithm in the context of audio transport on a local network. Results of our experiments confirm that it's a really satisfying solution to detect audio time deviations. Although with less performances, the algorithm should operate on a high latency network such as the Internet. This hypothesis is to be confirmed with future works.

References

[1] D. Fober, Y. Orlarey, S. Letz. Real Time Musical Events Streaming over Internet. *Proceedings of the International Conference on WEB Delivering of Music*, 2001, pages 147-154

[2] D. Fober, Y. Orlarey, S. Letz. Clock Skew Compensation over a High Latency Network. To be published in *Proceedings of the ICMC*, 2002, ICMA San Francisco

[3] Phil Burk, Ross Bencina. PortAudio - An Open Source Cross Platform Audio API. *Proceedings of the ICMC*, 2001, ICMA San Francisco

[4] PortAudio. A Portable Audio Library. http://www.portaudio.com

[5] J. Postel. User Datagram Protocol. IETF, RFC 768, 1980

[6] E. Brandt R.B. Dannenberg. Time in Distributed Real-Time Systems. *Proceedings of the ICMC*, 1999 ICMA San Francisco, pp.523-526