Controlling adaptive resampling

Fons ADRIAENSEN,

Casa della Musica,
Pzle. San Francesco 1,
43000 Parma (PR),
Italy,
fons@linuxaudio.org

Abstract

Combining audio components that use incoherent sample clocks requires adaptive resampling - the exact ratio of the sample frequencies is not known a priori and may also drift slowly over time. This situation arises when using two audio cards that don't have a common word clock, or when exchanging audio signals over a network. Controlling the resampling algorithm in software can be difficult as the available information (e.g. timestamps on blocks of audio samples) is usually inexact and very noisy. This paper analyses the problem and presents a possible solution.

Keywords

Jack, ALSA, network audio, resampling

1 Introduction

Adaptive resampling is required when combining audio hardware running at incoherent sample rates into a single system. Incoherent here means that the clocks are not derived from the same source. Although the nominal sample rates are known, their ratio is not exact and may even drift over time. A fixed resampling ratio, e.g. 44100/48000, or even one that takes known errors into account, will sooner or later require samples to be inserted or dropped and this is in general not acceptable.

The problem arises when adding a second sound card to a Jack server, or when two machines, each having their own audio hardware but no common clock, need to exchange audio signals via a network connection.

In hardware this is a relatively simple problem to solve if both sample clocks are available or can be extracted from the data streams. A PLL is used to track the ratio error, and its output controls a variable ratio resampler. Some professional equipment includes such processing on selected digital inputs.

For a sofware solution the main problem is not the variable ratio resampling itself, but how to control it. Audio data is handled in blocks of typically a few milliseconds lenght, and the only information available to control the resampling algorithm are the timestamps for these blocks, and in some cases data provided by e.g. ALSA's *snd_pcm_avail*() and similar functions. All this information has considerable random and systematic errors, and it is by no means evident how to turn it into the required smoothly changing control signal for the resampling algorithm.

None of the currently available solutions such as the *alsa_in* and *alsa_out* clients that come with Jack really gets this right. The purpose of this paper is to analyse the problem and present a solution. The algorithms discussed in the following sections have been implemented in two new applications, *zita_a2j* and *zita_j2a* wich allow to add ALSA soundcards running at arbitrary sample rates as clients to a Jack server. Other implementations, e.g. for networked audio or allowing to link two Jack servers running on the same machine will follow.

2 Requirements

Resampling itself, even with a variable ratio, will not lead to any signficant loss of audio quality if implemented correctly. The main consequences when using fixed-bandwidth resampling (as e.g. in *libsamplerate* and *zita-resampler*) will be a small loss of bandwidth and some additional delay. Both depend on the length of the multiphase filter used by this algorithm, and the tradeoff between the two can be made by the user.

The effect of a non-constant resampling ratio depends on the magnitude of the variation and on its spectrum. Very slow and small changes are equivalent to small movements of a listener w.r.t the speakers, or a performer w.r.t. the microphone. To put this in context, one sample at 48 kHz corresponds to about 7 millimeters in air. So provided such changes are limited to a few samples and very gradual they won't be noticed[1].

Larger variations, even when quite slow, may lead to perceptible delay and pitch changes which, while not really degrading the audio signal quality, may not be acceptable from a musical point of view.

But the really nasty effect of delay modulation occurs when the variations contain higher frequency components, even if these are quite small. The result no longer appears as a modulation of the signal (e.g. vibrato) but as parasitic signals, noise or distortion. Some types of sound are very sensitive to such phase modulation. This is really the equivalent of jitter on the sample clock of an A/D or D/A converter, and sadly, some of the existing implementations of adaptive resampling produce this at level that is some orders of magnitude higher than the worst hardware.

Apart from audio quality considerations there are some other aspects which are important. Both the resampling, and moving audio signals between domains with asynchronous periods will introduce latency. It depends on the application if this is acceptable or not. But at least the delay should be stable and repeatable. Stability means that it must not depend on e.g. whether a Jack client implementing the resampling runs at the start or near the end of a Jack cycle. Again, existing implementations fail to meet this requirement.

Applications implementing adaptive resampling can take up to a few seconds to stabilize their control loops, and need to restart if synchronisation is lost, e.g. when a misbehaving Jack client results in a timeout of the server. This is quite accecptable, but minor incidents such as Jack1 skipping one or a few cycles should not result in a change of the latency nor require a restart.

[1] Unless you are e.g. sending one channel of a stereo pair via the resampler and the other not, but that is really asking for trouble.

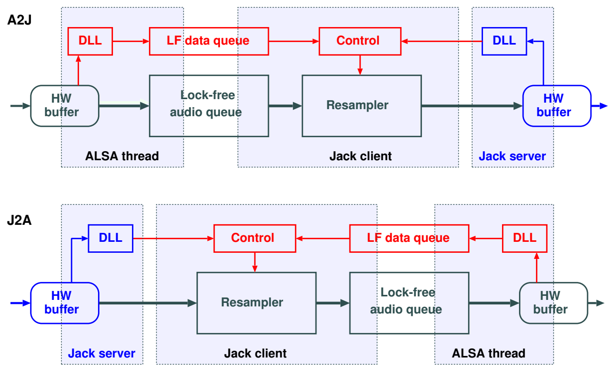

Figure 1: The structure of A2J and J2A

3 Problem analysis

Anyone trying to build an abstract picture of this sort of algorithm and to reason about it will find that it stretches his or her powers of imagination to some degree. It helps to have a particular implementation in mind. In this paper we will use the structure of *zita_a2j* and *zita_j2a* as a framework. However, the analysis is more general and applies in other cases as well.

Figure 1 shows the structure of those applications. In this figure both are shown with the Jack client connected directly to the sound card used by the Jack server — this is the setup we would use to measure the latency (using e.g. *jack_delay*) in a round-trip involving both Jack's sound card and the additional one.

Taking *zita_a2j* as an example, the ALSA thread, as regards audio processing, does little more than transfer audio frames from the ALSA device to a lock-free buffer. It also provides some extra information which will be discussed later. Keeping the ALSA thread as simple as this allows it to run at a higher real-time priority than the Jack server, and with a shorter period time, and this in turn helps to obtain accurate timing information. The actual resampling and the control logic is performed in the process callback of the Jack client. The structure of *zita_j2a* is virually the mirror image of *zita_a2j*.

3.1 Building a model

What needs to do be done is to control the resampling ratio in such a way that on average the same number of samples enter and leave the lock-free buffer. We also want to do this using only very small and smooth changes of the resampling ratio.

Monitoring the *actual* state of the buffer (the number of frames stored in it) won't work for several reasons. The most fundamental one is that we actually can't observe the buffer state in any reliable way from either side, as the other side could be modifying it at the same time. At best we could have an upper or lower bound. Also, this value doesn't change in a smooth way, but jumps each time a period is processed on either side. And these changes occur when the code of the Jack process callback or the ALSA thread actually runs, and this moment is not at all representative of the real timing of the audio samples. For example the process callback of the Jack client can run at any time between the start of the current cycle and the start of the next one, this just depends on the position of the client in the connection graph and on the CPU load of other clients.

The key to creating a working model is to take abstraction of the period based processing, including the random timing errors, and imagine the resampling algorithm as a continuous process.

Suppose we would have two *continuous* functions of time, $W(t)$ and $R(t)$ that provide the number of samples that have been written to resp. read from the buffer at any time t. Then if Δ is the required delay of the whole process, we could evaluate the error $W(t) - R(t) - \Delta$ in each process callback and use this to control the resampling ratio. This, with some refinements and extra functionality, is the basic algorithm used in *zita_a2j* and *zita_j2a*. Remains to create those two functions. Let $J(t)$ be the function on Jack's side and $A(t)$ the one on the ALSA side. Then for *zita_a2j* $J(t) = R(t)$ and $A(t) = W(t)$, and for *zita_j2a* $J(t) = W(t)$ and $A(t) = R(t)$.

On the Jack side, part of the solution is already available in the server. The DLL *(Delay Locked Loop)* that has been part of Jack since many years computes in each cycle a prediction of the start time of the next cycle, while removing most of the jitter due to random wakeup latency. This provides a smooth and continuous mapping between time (as measured by Jack's microsecond timer) and frame counts.

So we can implement $J(t)$ by just reading the timestamp provided by Jack's DLL into t_J, and summing the number of frames read from or written to the lock-free buffer in k_J. We don't actually need the function on Jack's side, since we are only interested in its value at the start of the current period.

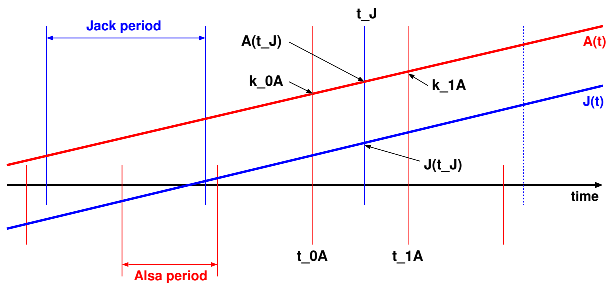

Figure 2: Delay calculation parameters

A similar DLL can be provided at the ALSA side. This requires reading the wakeup time of the ALSA device using Jack's microseconds timer and applying the DLL algorithm which is quite simple. To transfer the data to the Jack side a second lock-free queue is used. For each period the ALSA thread sends a message containing it current status, the computed timestamp for the next period, and the number of frames written to or read from the audio buffer. At the Jack side, during each process callback these messages are read and the frame counts are accumulated into a variable k_{A1}. The most recent data (t_{A1}, k_{A1}) is used, along with the same from the previous period, (t_{A0}, k_{A0}). Since the ALSA thread reports the wakeup time of its *next* cycle, the interval (t_{A0}, t_{A1}) includes the current wakeup time at the Jack side (or in the worst case one of the endpoints will be close), so a simple interpolation is all that is needed to compute $A(t_J)$. Figure 2 shows the relevant parameters for the case *zita_a2j*.

3.2 Resampler delay

The analysis so far has ignored the delay introduced by the resampling process itself. There are two aspects to this. First, this latency can be significant and it must be taken into account when defining the target value. Second, when using k_J, the number of frames transferred between the resampler and the audio queue, as the value of $J(t)$ we are actually making an error. $J(t)$ should increase by exactly the same delta in each period, the Jack period size either multiplied or divided by the resampling ratio, but it doesn't because it

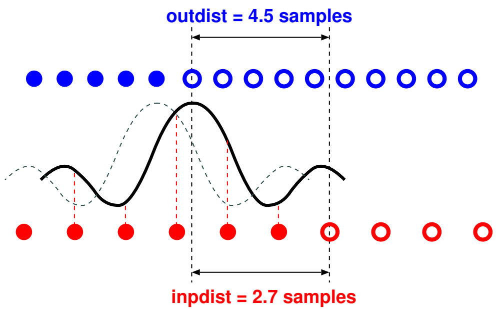

Figure 3: Resampler latency

is constrained to be integer. The sum will be exact on average, there is no long term accumulating error, but we are missing the fractional part.

That fractional part is actually represented by the internal state of the resampler. To see this we will use *zita-resampler* as an example. Constant bandwidth resampling works by evaluating a FIR filter (a near 'brickwall' lowpass wich corresponds to a windowed *sinc*() function in the time domain) for each output sample. The central peak of the impulse response corresponds to the current output sample, and the actual coefficients used depend on the position of the filter w.r.t. the input samples. This is shown in Fig. 3, using a very short filter. Actual resampling filters are much longer.

When *zita_resampler* finishes processing a block of frames it remains in a state ready to compute the next output sample, except that it may have to read one or more input samples before it can proceed. In Fig. 3 the bottom (red) dots represent input samples and the top (blue) ones the output. Solid dots are samples already used or computed. The filter is aligned with the next output sample. In this example one more input sample is required to compute the next output, the first non-solid one in the figure. This will not always be the case, for example if the previous call terminated because the output buffer was full it could be that the next output doesn't require a new input sample. But in any case the distance between the next input sample and the next output one is well-defined and it can be expressed in either the input or output sample rate. This value includes the fractional part that is missing from k_J.

The *Vresampler* class used in *zita_a2j* and *zita_j2a* provides a function *inpdist*() which returns the current value of this delay at the input sample rate, and this is used to correct the value of k_J. One may ask if such a small error actually matters. This depends on the nominal resampling ratio. If this is not the quotient of two small integers then the fractional error is a pseudo-random value and its effect willl be removed by the loop filter that controls the resampling ratio. The worst case results if the two sample rates are nominally the same. The error will be a sawtooth function with a frequency equal to the difference between the two actual sample rates. In this case the loop filter may not completely remove it. The effect was visible in test results of early implementations that did not include the correction in the error calculation.

3.3 Closing the loop

Combining the elements presented above we can now formulate the equations giving the delay error for both cases. Let γ be the resampling ratio, d_{res} be the value returned by the *inpdist*() member of the resampler, Δ the target delay value and $t = t_J$ the start time of the current cycle, then

$$\begin{aligned} E_{A2J} &= W(t) - R(t) + d_{res} - \Delta \\ E_{J2A} &= W(t) - R(t) + d_{res} * \gamma - \Delta \end{aligned}$$

Using the definitions of $W()$ and $R()$, and setting

$$d_A = A(t_J) = [k_{A1} - k_{A0}] \frac{t_j - t_{A0}}{t_{A1} - t_{A0}} \qquad (1)$$

these become

$$\begin{aligned} E_{A2J} &= [k_{A0} - k_J] + d_A + d_{res} - \Delta & (2) \\ E_{J2A} &= [k_J - k_{A0}] - d_A + d_{res} * \gamma - \Delta & (3) \end{aligned}$$

This error is first processed by a second order lowpass filter and then becomes the input to the second order loop filter. This is very similar to the one used in Jack's DLL, see [Adriaensen, 2005]. The first filter is added to further reduce phase modulation by high frequency noise on the error value. Its bandwidth is 20 times that of the loop filter, so it does not affect stability. The resampler code adds another lowpass to smooth ratio changes. Also this one must be dimensioned so it does not affect operation of the loop.

The values k_J and k_{A1} are obtained by accumulating differences in each cycle. To make the equations above represent the actual round trip delay error we need to initialise them with the correct values. Since k_{A0} is just k_{A1} from the previous cycle it needs no initialisation. A bit of arithmetic (which is left as an exercise for the reader and wich may tickle his or her powers of imagination a bit) will show that the correct initial values are

$$\begin{aligned} k_J &= -P_J/\gamma \\ k_{A1} &= P_A \end{aligned}$$

if the ALSA device is the input, and

$$\begin{aligned} k_J &= P_J * \gamma \\ k_{A1} &= -P_A \end{aligned}$$

in the other case, with P_J and P_A being the Jack and ALSA period sizes, and γ the resampling ratio.

The key to understand this is that at any time the delay must be the sum of the number of frames in both hardware buffers, the lock-free queue, and the resampler, while taking the different sample rates into consideration.

A related matter is to determine the target round-trip delay value. Some more (simple) arithmetic will show that

$$\begin{aligned} \Delta_{min,A2J} &= T_{res} + 2T_J + (1+c)T_A \\ \Delta_{min,J2A} &= T_{res} + 2T_J + 2T_A \end{aligned}$$

where T_{res} is the delay of the resampler, T_J and T_A the Jack and ALSA side period times, and c is the maximum expected wakeup latency of the ALSA thread, e.g. 1/2 when this thread is allowed to run half a cycle late. These values allow for worst case conditions, e.g. a Jack client running near the end of the cycle.

3.4 Improving the settling time

The system discussed so far will provide a constant processing delay, but with the loop running at is normal bandwidth (around 0.05 Hz in the current implementation) it could take a long time to reach the target value Δ. We can do two things to speed up convergence of the loop. One is to run the loop filter at a higher bandwidth initially. The other is to use the first delay error measurement to force a situation close to the required one, and then let the loop take care of the remaining error. This requires modifying the state of the lock-free queue. Note that once the system is running we can't set the number of frames in the queue to any specific value in a safe way (as the other side may be accessing the queue at the same time). The only thing we can do is force a *relative* change by either reading or writing a number of frames, and that is fact all we need.

If N is the delay error rounded to the nearest integer, then for the A2J case we set $k_J = k_J + N$ and read N frames from the queue, and for the J2A case we set $k_J = k_J - N$ and write $-N$ frames to the queue. Adjusting k_J removes most of the error from the model, and applying the corresponding change to the queue ensures that the model remains in sync with the actual situation.

Both example applications do remove the initial error in this way, then run the loop at higher bandwidth for the first 4 seconds.

4 Implementation details

4.1 Delay error calculation

The k_J, k_{A0} and k_{A1} values used in (1,2,3) are integers that are incremented by frame counts in each iteration, so they will overflow at some time. Since we only ever take *differences* of those, the overflow doesn't matter. But this part of the calculation *must* be done as a subtraction of integers without conversion to a floating point value, as suggested by the square brackets in the equations. The remaining parts can be done safely in floating point since these terms are recomputed each time and there will be no accumulating roundoff error.

Jack represent its microseconds timer as a 64-bit integer type. These are difficult to use in calculations so they are converted to double. To avoid loss of precision, the actual value of t_J, t_{A0} and t_{A1} is the microseconds time masked to 28 bits, and divided by 10^6 to obtain seconds. This representation will wrap around every 268.4 seconds or so. Again since we are always taking differences this is easy to detect and correct.

4.2 Error recovery and reporting

In case of a fatal error in the ALSA device or a timeout of the Jack server there is no alternative but to restart the loop initialisation. The two applications discussed here contain some state management code to allow this without having to restart the actual processes. This uses a third lock-free queue (not shown in Fig. 1) to send commands from the Jack side to the ALSA thread.

When restarting the loop will settle quite fast, as the previous resample rate correction can be re-used.

The Jack1 server will occasionally skip some cycles while creating or removing clients or when making port connections. This can be detected and handled quite easily and without introducing any errors in the loop. The particular implementation of the lock-free audio queue allows it to recover from overflow or underflow conditions by just reading or writing the number of frames that were missed before. When doing this, the same adjustment is made to k_J to remove the error from the delay calculation.

A fourth lock-free queue is used to convey status and optional monitoring information for output in the main thread.

4.3 The lock-free queue

When recovering from skipped cycles the lock-free audio queue may be in an overflow or underflow condition. The same is true when removing the initial delay error as discussed in 3.4. Also, the value of N used there can be positive or negative.

The lock-free queue must implemented in a way that maintains correct read and write counters and the corresponding data pointers at all times. It must also allow logical read or write operations of any number of items, including negative ones.

Such an implementation need not be more complicated than some existing ones, and in fact it can be much simpler. The C++ class used in *zita_a2j* and *zita_j2a* uses a 2^N buffer size as would most. It maintains read and write counters as 32-bit integers. These are just incremented by the number of items the user claims to have read or written, without enforcing any limits. The read and write indices are the respective counters modulo the buffer size. Since the buffer size divides the 2^{32} range of the counters, these can be allowed to overflow without any consequence. It is the user's responsability to avoid buffer overflow or underflow if that matters, and the class provides the necessary information to make this easy, so no functionality is lost by this particular implementation.

5 Results

Figure 4 shows the result of a 1 kHz sinewave signal processed by *zita_j2a*, with the sample rates being 48 kHz at the Jack side and 44.1 kHz for the ALSA device. The Y axis is the phase of the output signal (w.r.t. to the generator) in degrees, the X-axis is in centiseconds. The loop stabilises in about 15 seconds. After that time, variations are less than 0.5 degrees peak-to-peak. One degree at 1 kHz corresponds to 2.78 microseconds. The small bump at around 145 seconds is the result of switching the desktop workspace. This measurement was done one a single CPU machine, running a standard (unpatched) kernel and having no HW video acceleration.

Figure 5 is the result of the same test but with both ends running at 48.0 kHz. There is a periodic dip every 83 seconds, which corresponds

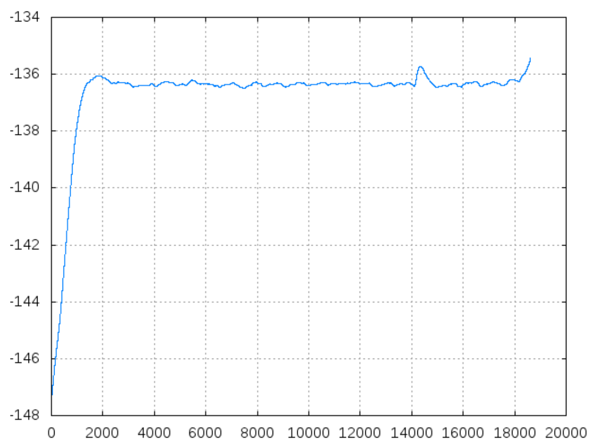

Figure 4: zita-j2a, 48.0 to 44.1 kHz

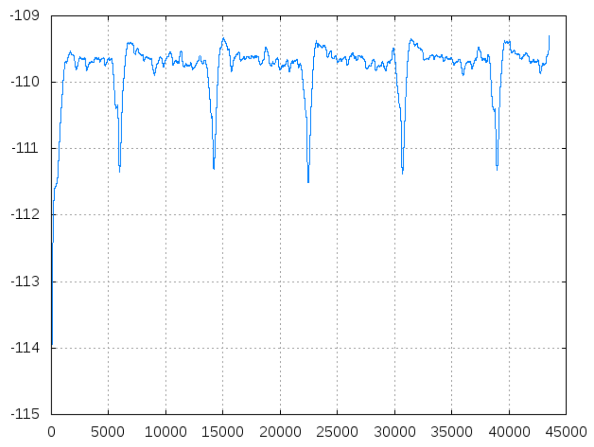

Figure 5: zita-j2a, 48.0 to 48.0 kHz

to a frequency of around 0.012 Hz. This is the difference between the period frequencies of both soundcards. Every 83 seconds the interrupts of the two cards occur at almost the same time and compete for the CPU. This effect would probably not be visible on an SMP machine. It's harmless in practice as the delay changes are very small and smooth.

6 Acknowledgements

The results reported in this paper build on the work done by the Jack and ALSA developers. A particular thanks to the members of the respective mailing lists for the prompt answers to all my questions.

References

Fons Adriaensen. 2005. *Using a DLL to filter time.* Available at `http://kokkinizita.linuxaudio.org/papers/index.html`.