Timing issues in desktop audio playback infrastructure

Alexander Patrakov
SkyDNS LLC
Office 500, House 2, Kulibina Street,
620137 Yekaterinburg,
Russia
patrakov@gmail.com

Abstract

In year 2008, a feature with the name "timer-based scheduling" (also known as "glitch-free") has been introduced into PulseAudio in order to solve the conflicting requirements of low latency for VoIP applications and low amount of CPU time wasted for handling interrupts while playing music. The novel (at that time) idea was to use timer interrupts instead of sound card interrupts in order to overcome the limitation that the ALSA period size cannot be reconfigured dynamically. This idea turned out to hit some corner cases, and workarounds had to be added to PulseAudio. Despite its age, the implementation of the idea is still not 100% correct. This paper explains why it is the case and what can be done to improve the situation.

Keywords

PulseAudio, timing, rewinds

1 Introduction

Traditionally, ALSA playback chain was based on ALSA plugins sitting on top of the hardware device (represented by the "hw" plugin).

Hardware devices have circular buffers in memory, and there are two pointers that point to this buffer: the hardware pointer and the application pointer. The hardware periodically reads a sample pointed to by the hardware pointer and sends it to analog or digital outputs, then increments this pointer. The application writes samples to the memory area pointed to by the application pointer and moves it past the just-written samples.

There are multiple mechanisms provided to the application to write sound data to the sound card: classical unix-style writes via `snd_pcm_write*()` functions, mmap-based access, and the dangerous callback-based API. Still, with any of them, the end result is the same: application pointer tracks the first unwritten position in the soundcard buffer.

If the hardware pointer crosses the application pointer, an underrun happens. To avoid underruns, an application must supply new audio samples in a timely manner.

The hardware notifies the kernel when the hardware pointer crosses some predefined positions (period boundaries) in the circular buffer. There are, again, multiple mechanisms (blocking writes, `poll()`) how these notifications can be passed to the application, so that it doesn't have to busy-wait.

A whole lot of other behavior (format-conversion, resampling, mixing) is provided on top of the raw hardware devices by means of ALSA plugins [1]. The general idea (a circular buffer with hardware and application pointers and per-period wakeups), however, remains. As a result, applications can transparently use a large subset of ALSA API when working with such plugins.

This playback model was popular in the dmix era, and thus applications developed during that time gained dependency on some of the properties of this model. E.g., an assumption is common that wakeups due to the audio device happen regularly (exactly once per period) and can be used as a clock. Another common assumption is that the default buffer and period sizes are suitable for the application's purpose. In fact, there was no way to change them programmatically in the default "plug:dmix" setup.

There is certain latency (influenced by the audio buffer size) between the time when an audio sample is written to the API and when it is actually played back through the speakers. Low latencies are generally expected when an application reacts to user input. E.g., when a user changes equalizer settings in the audio player, they should take effect immediately. This is even more important for games: a gunshot sound should be heard as soon as the shot is made. Voice over IP applications are also sensitive to latency. So, with the traditional playback model, due to the fact that the latency is fixed, low latencies are generally used.

On the other hand, low latencies are *not* optimal for music players. First, low latencies make applications sensitive to process scheduler decisions, increasing the chance of audio dropouts. Second, low latency means high rate of interrupts from the sound card and application wakeups, which is bad for power saving. So, music players and games are under two conflicting requirements related to latency.

2 Timer-based scheduling

The traditional solution was to accept frequent (as required for the worst case) wakeups as a necessary evil, because there is no way to reconfigure buffer and period size on the fly. However, in some cases, a better (albeit more complex) solution exists to this conflict of requirements. The solution ("timer-based scheduling" [2], implemented in PulseAudio [3] and CRAS [4]), involves the use of a dynamically reconfigurable timer instead of sound card interrupts as a source of wakeups.

PulseAudio has a client-server architecture. The server interacts with ALSA devices and performs mixing and routing of sound data received from client applications. Each time the timer fires, the sound card is asked about its current playback position, and, based on this information a decision is made how much data to request from applications in order to maintain their desired latency and to avoid underruns. Note that, if there are low-latency applications playing, the buffer will never be full.

A new stream can appear at any time, or an application can request volume change of an existing stream. The server is expected to deal with such requests quickly, i.e. without waiting for the already-buffered sounds to play out. Indeed, waiting for these sounds could take more than a second, which is too much. Thus, the server has to discard already-mixed samples from the sound card buffer and replace them with a new version, which takes the new stream or the volume change into account. Such operation is called a "rewind". It is an essential ingredient in an implementation of a dynamic-latency sound server, unless a tight limit is placed on the amount of buffered audio data. PulseAudio rewinds. CRAS doesn't, but Chrome/Chromium never requests latency high enough to cause a problem.

As the timer-based approach is more complex than the traditional approach, there are more questions to be answered by the implementation (such as PulseAudio) and parameters to be decided upon.

- Total sound card buffer size.
- The amount of time to sleep after writing sound data.
- The amount of old data to leave in the buffer "just in case" when rewinding.
- What latency limits to export to clients.
- How much data to ask from a client at a time.

3 Buffer and timing constraints

PulseAudio uses a large buffer (up to 2 seconds, if the hardware allows) by default. This is good for the purpose of providing high latencies for music players and thus for reducing the rate of CPU wakeups. The default can be overridden with the `tsched_buffer_size` parameter that is accepted by `module_udev_detect`, `module_alsa_card`, `module_alsa_sink` and `module_alsa_source`. The unit of the `tsched_buffer_size` parameter is microseconds.

This default, however, poses a problem[1] if any part of the audio processing pipeline inside the PulseAudio process turns out to be CPU-intensive. Examples of CPU-intensive steps include conversion to a compressed format such as DTS. When PulseAudio is running under Valgrind, or on a weak embedded CPU, even resampling becomes a problem.

The problem is related to the fact that PulseAudio only has a finite budget of time it can run with real-time priority without making blocking system calls. `rtkit` contains a hard-coded limit that doesn't allow expanding this budget past 200 ms. This limit exists for safety reasons, because a misbehaving real-time application can otherwise wedge the whole system. Thus, in the worst case (which always happens at the start of a high-latency stream) PulseAudio has to finish its processing of two seconds of audio in 200 ms, or it gets killed.

The situation is further aggravated by the fact that the `cpufreq` subsystem considers "low" (i.e. less than 80%) load as an excuse to keep the CPU frequency at the lowest possible value.

A solution that a user affected by the problem can apply is to set the buffer size to a lower value, such as 200 ms.

4 Wakeup timings

In the traditional timing model, the application usually is woken up once per period. The period size comes from application settings or from the defaults. There is not much that can be done beyond that (e.g. in response to underruns), because buffer and period sizes are not dynamically reconfigurable.

With timer-based scheduling, better reaction to underruns is possible, and PulseAudio implements that. It looks at the sink's latency (which is just the amount of time until it underruns unless supplied with new data), subtracts the scheduling watermark, and sleeps for that time. The default watermark is 20 ms. It is increased if an underrun or a near-underrun happens, and decreased if sufficient time has passed without such bad events[2].

This logic is further complicated by the fact that the requested latency is specified in the sound card's clock domain, while sleeping is done using the system clock domain. If the sample rate reported by the card is not precise, then these two values can differ. PulseAudio contains a "smoother"[3] that takes timestamps in both clock domains, estimates the actual sample rate, and then converts the intervals as needed.

[1] `https://plus.google.com/+ColinGuthrie/posts/EG7nT9TXTpd`

[2] See `http://cgit.freedesktop.org/pulseaudio/pulseaudio/tree/src/modules/alsa/alsa-sink.c`, functions `check_left_to_play()`, `decrease_watermark()` and `increase_watermark()`

[3] `http://cgit.freedesktop.org/pulseaudio/pulseaudio/tree/src/pulsecore/time-smoother.c`

The traditional solution for mapping between soundcard and system clock domains would be using a delay-locked loop with a filter containing some integrators in it [5]. This solution works for JACK, but cannot be employed in PulseAudio, because the timestamp reports are assumed to be regular in time, which is valid only if the traditional period-based timing scheme is used. Therefore, the smoother in PulseAudio uses a 10-second window and builds a least-squares linear approximation between the sample count and the wall-clock timestamp based on data within that window.

A special rule (that cuts the sleeping time in half) is applied until one buffer worth of sound data is played. This is needed because some sound cards contain a hardware FIFO queue that consumes the initial portion of data much faster than one would expect according to the size of that portion and the sample rate.

All of the above assumes that the sound card can accurately report its hardware pointer at arbitrary point in time. However, this assumption is false on cards that do double-buffering of audio data transfers. Such cards can be distinguished using the `snd_pcm_hw_params_is_batch()` ALSA API function. Typically, such batch cards provide position reports that are accurate to only one period, and timer-based scheduling makes a period as large as possible to avoid useless CPU wakeups from the interrupts originating from the sound card. Since position reports are totally inaccurate, one just cannot obtain an estimation of time-to-sleep accurate up to 20 ms.

Currently, PulseAudio disables timer-based scheduling on batch cards[4], because it cannot save the CPU from unneeded wakeups.

CRAS does not have this watermark-based logic and does not use the mapping between soundcard and system clock domains for the purpose of wakeup timing. Indeed, CRAS doesn't have to do so, because it respects the client's idea how many frames should remain in the soundcard buffer when asking for more data, instead of asking as late as possible, and thus stays far from any edge cases.

5 Rewinds

As already explained, rewinds are needed in order to provide low-latency reaction to unpredictable events such as new streams and volume changes, while keeping the average latency high in order to save power. Rewind handling is an especially problematic area, with a lot of code written but never properly tested.

5.1 Rewind-related APIs

Both ALSA and PulseAudio offer APIs that let applications rewind their audio streams.

As already mentioned, ALSA's view on playback devices is based on the notion of a circular buffer in memory, with the hardware pointer and the application pointer associated with it. Here are the rewind-related API functions offered by ALSA: `snd_pcm_rewindable()`, `snd_pcm_rewind()`.

The `snd_pcm_rewind()` function tries to move the application pointer backwards by the specified number of samples, and returns the (possibly lower) number of samples that the pointer has actually been moved by. Of course, attempting to request rewinding into the already-played portion of the buffer does not make sense. The `snd_pcm_rewindable()` function returns the maximum safe amount of rewindable samples[5].

PulseAudio does not base its playback model on a mmap-able circular buffer. Instead, it has one stream-oriented function that clients use to submit samples: `pa_stream_write()`.

Seeking is done at the same time as writing new samples, using the last two arguments. Unlike ALSA, which supports only rewinds relative to the application pointer ("write index", as PulseAudio calls it), PulseAudio API can also be used to rewind to an absolute position in time, or relatively to the "read index" (the sample that is currently being played). The raw read index and write index can be obtained in the `pa_timing_info` structure via the `pa_stream_update_timing_info()` and `pa_stream_get_timing_info()` pair of functions.

OSS does not support rewinds in ways other than the (deprecated) mmap interface, which only works on top of raw hardware devices. I.e. no resampling, no channel remixing, only exclusive access to the sound card.

JACK, SDL, libao and the `waveOut` family of Windows APIs do not support rewinds at all. Android's AudioTrack API and CRAS don't support them, either.

5.2 Testing rewinds

Rewind operations can be used by software only if they actually work as described. E.g., a perfect implementation of rewinds needs to ensure that, after rewinding over some samples and writing exactly the same samples back, the audible result is exactly the same as if the rewind didn't happen at all.

Currently, for ALSA, the most common application that does a lot of rewinds is PulseAudio, and it does that only in response to dynamic events such as new stream appearing or volume changing, where a user already more-or-less expects a glitch and thus may not realize that something is wrong. So, in order to really ensure that rewinds work, a more systematic testing methodology is needed.

A simple ALSA-based program[6] has been thus written that exercises the `snd_pcm_rewind()` func-

[4] `http://cgit.freedesktop.org/pulseaudio/pulseaudio/commit/?id=826c8f69d34ef49e86fe0ab6c93c1ffba8916131`

[5] There are disagreements on the intended meaning of the word "safe".

[6] `http://permalink.gmane.org/gmane.linux.alsa.devel/122179`

tion in such a way that is impossible to confuse the correct operation and a glitch. The program uses a buffer with four periods. After the initial filling of the buffer with silence, the application uses blocking writes, as follows. Each time it gets a chance to write a period worth of samples, it rewinds one period, writes one period of silence and one period of square waves. Therefore, if rewinds are implemented correctly, the hardware pointer only sees silence, and nothing should be heard from this application. Any non-silent output (without the application complaining that the rewind did not yield the expected result, and without near-underruns) is an indication of a bug somewhere.

Hardware ALSA devices pass this simple test. Many other devices currently don't.

It would also be nice, for similar reasons, to test the correct operation of the `snd_pcm_rewindable()` function. However, no valid test can be devised at this time, because there are disagreements about the semantics of the return value. As this function only recently stopped crashing on some plugins[7], PulseAudio does not use it, and uses an approximation based on `snd_pcm_avail_delay()` instead.

5.3 Causes of incomplete rewindability in hardware

Currently, for the "hw" plugin the `.rewindable` callback is implemented, effectively, as a difference between the application pointer and the hardware pointer. That is, "you can rewind up to the hardware pointer". However, the hardware pointer information is only updated either on period boundaries, or on explicit request (via `snd_pcm_avail()`) from the application. Failure to perform such request would lead to alsa-lib basing its calculations on an outdated value of the hardware pointer, and, thus, to the overestimated results for `snd_pcm_rewindable()`.

There is a disagreement whether the `snd_pcm_rewindable()` function should indeed return the difference between the application pointer and the hardware pointer. PulseAudio contains a safeguard that does not allow the rewound application pointer to come too close to the hardware pointer, because[8]

> *some DMA controllers go nuts (such as breaking the stream, causing interrupt storms, or something else seriously buggy) when trying to write to data that the DMA controller is just about to transfer.*

The default safeguard is the largest of 256 bytes or 1.33 ms.

On some cards, the hardware pointer position is not known exactly. For example, `ymfpci` updates its hardware pointer using a timer that fires every 5 ms. Therefore, the hardware pointer position reported to userspace may lag behind the real one by up to 5 ms, and the number of rewindable samples may be also overestimated by the same amount.

Also, the hardware itself may report the hardware pointer position imprecisely. E.g., on common Intel HD Audio controllers, the granularity of the reported pointer position (as measured by calling `snd_pcm_avail()` and `snd_pcm_rewindable()` repeatedly) is 32 or 64 bytes. This is probably related to the DMA block size.

An idea was expressed that alsa-lib should take the above sources of uncertainty over the hardware pointer position or DMA engine weirdness into account when returning the number of rewindable samples.

An opposite viewpoint is expressed by Clemens Ladisch[9]:

> *It would make sense to report the pointer update granularity, but not to adjust the return value of* `snd_pcm_avail/rewindable()`*.*

However, on many cards, the pointer update granularity is simply unknown. A set of patches has been posted by Pierre-Louis Bossart[10] (and later merged) that are expected to help assessing the granularity of pointer updates at runtime. Still, nobody so far has tried to use the information exposed by these patches in PulseAudio.

5.4 Rewindability of self-contained ALSA plugins

User-grade programs (i.e. everything except sound servers) usually don't talk to hw devices. Instead, they use ALSA PCM plugins for functionality like mixing, channel remapping, sample rate conversion and software-based volume control. There are also more exotic plugins for tasks like software AC3 or DTS encoding, or spectrum equalization. Finally, there are plugins that allow ALSA programs to talk to sound servers like PulseAudio or JACK.

The implementation of rewinds is the simplest in plugins where each sample sent to the slave is determined only by the corresponding input sample. That is, the plugin never looks at non-current sample and doesn't keep any state. In this case, the implementation of the rewind operation should just rewind the slave by the same amount of samples. Also, to answer the question "how many samples can be rewound safely", the plugin should just ask the slave and forward the answer. Here are the plugins where this logic or a simple variation of it applies: `alaw`, `asym`, `copy`, `empty`, `hooks`, `lfloat`, `linear`, `mmap_emul`, `mulaw`, `multi`, `route`. These plugins are indeed rewindable. The `softvol` plugin is rewindable for the same reasons as long as nobody changes the volume.

[7]Most of the fixes are in alsa-lib 1.0.28 and one is in 1.0.29

[8]`http://permalink.gmane.org/gmane.linux.alsa.devel/127256`

[9]`http://permalink.gmane.org/gmane.linux.alsa.devel/127290`

[10]`http://permalink.gmane.org/gmane.linux.alsa.devel/133961`

The iec958 plugin is used by some old cards (such as ATIIXP and CMI8338) to convert raw PCM to IEC958 frames and back. Each IEC958 subframe corresponds to one audio sample and one channel. Besides the PCM sample itself, the subframe contains a preamble, and one bit for each of Validity (for DAC), User data, Channel status and Parity. Different subframes use different types of preamble. This is needed to distinguish between left and right channels, as well as to mark the beginning of user data and channel status. The whole audio block contains 384 subframes.

Therefore, the plugin needs to keep a simple internal state: the number of subframes sent since the last subframe with the Z-type preamble (which is used to mark the left-channel subframe which also contains the beginning of the first channel status word). Before alsa-lib 1.0.28, rewinds didn't affect the state. Therefore, right after a rewind, a wrong type of preamble was used, and wrong bits (not continuing what was sent before) of channel status were sent down the link. This could cause a momentary resynchronization glitch on some receivers. As of alsa-lib 1.0.28, this is fixed by updating the state after each rewind, and thus the iec958 plugin fully supports rewinds now.

The adpcm plugin converts between linear PCM and IMA ADPCM, which is only useful for ancient ISA cards. Again, the conversion is not stateless: the per-channel state includes the predicted sample value and the step size index, and is updated at each new sample according to simple table-based rules. As of alsa-lib 1.0.29, rewinds don't change the state. It is a bug. To solve it, one has to make this state per-sample per-channel, organized in a circular buffer similarly to the sound samples. This is not done yet.

The dmix and dshare plugins currently fail the rewind-correctness test for unknown reason. On them, the snd_pcm_rewind() function returns exactly the same number of samples as requested, however, the test program produces non-silent output.

As dsnoop is a capture-only plugin, it is not reviewed here. The share plugin could not be tested due to unrelated bugs, but, according to the source code, its .rewindable callback always returns 0.

5.5 External plugins

ALSA comes with two SDKs for building third-party plugins. The ioplug framework is for building plugins that output sound to some external systems, and extplug is for building filters. Also there is a ladspa plugin that wraps, well, third-party LADSPA plugins. There are two big problems in this area.

First, the plugin is not notified about rewinds at all. There is simply no such callback in the snd_pcm_ioplug_callback and snd_pcm_extplug_callback structures. The common code (wrongly) pretends that ioplug-based plugins are fully rewindable, but the rewind operation merely moves the application pointer back by the specified number of frames, and returns that number. Extplug-based plugins, as well as the ladspa plugin, simply forward rewind-related requests to the slave.

An important special case is that rewinds do not work (i.e. do nothing, "successfully") in the pulse ALSA plugin, even though native PulseAudio API does support rewinds.

But maybe it is possible to detect rewinds even without the corresponding callbacks?

For ioplug-based plugins, it may be possible to figure out from within the .transfer callback if there was any rewind operation between the previous call and the current call, by looking at the application pointer in the snd_pcm_ioplug structure. This may be sufficient to implement rewinds in the pulse plugin, but, as this approach does not allow to figure out the real amount of rewindable samples, it is a bad idea.

Extplug-based plugins don't have any access to their own application pointer, because it is hidden behind a private snd_pcm_extplug_priv structure. So they just don't have any chance to handle rewinds properly.

To solve the problem mentioned above, it would be necessary to add new callbacks. But this would cause the second issue, which is much worse. Imagine that someone has to implement these callbacks. Many ioplug/extplug-based plugins wrap external libraries. In order to implement rewinds, a plugin would have to tell the library to restore its old state. Mission impossible: these third-party libraries (as well as LADSPA API), in the vast majority of cases, don't have API functions that save and restore the state. I.e. this is the same problem as above, but one layer deeper and thus beyond our control. Besides, there is physically no way to e.g. undo sending of Bluetooth packets. As a result, rewinds just cannot be implemented correctly in the majority of ioplug/extplug-based plugins, and it was, as it seems, a mistake to offer them.

An interesting exception to the above non-rewindability rule is the jack plugin, especially since JACK itself is non-rewindable. The trick is that the plugin creates a real-time thread, and the JACK callback is invoked in the context of this thread, exchanging the samples with the JACK server. This looks very much like a real sound card, which periodically reads samples from the memory buffer. The .rewindable callback still yields a questionable result, though, by not taking into account the hardware pointer position uncertainty, which is one JACK period in this case.

It may be theoretically possible to extend this "low-latency worker thread" idea to other ioplug/extplug/ladspa plugin types – i.e. to create a thread just for the purpose of calling the .transfer callback instead of calling it when the client writes data. A natural period for calling this callback would be one slave period, but then the resulting minimum latency would be three slave periods (if the slave allows using two periods), which is one period more than without this thread.

The current understanding is that, instead of adding an extra level of buffering in ALSA for plugin rewindability, it may be a better idea to teach PulseAudio to identify non-rewindable ALSA devices as such, and deal with them as appropriate. Indeed, this low-latency worker thread creates frequent wakeups and thus nullifies the primary motivation behind timer-based scheduling anyway.

5.6 Rate plugin

The rate plugin converts the sample rate of the audio data. The process is based on the idea to find a digital representation of the same analog signal that is represented by the sequence of input samples. Due to Shannon's sampling theorem, a perfect resampler should reject frequencies higher than half of the lower sample rate, and pass all lower frequencies through. Therefore, its time response can be described by the appropriately scaled sinc function [6]. The sinc function, however, has infinite support and thus has to be windowed or approximated by some other function with a finite support in order to become useful. Such approximations introduce distortions in the resampled sound: components with frequencies below the ideal cut-off frequency get attenuated, and also "aliased" content (with frequencies not present in the input signal) appears in the output. There are several libraries that implement audio resampling, using different approximations, and thus having different quality and speed.

It follows from the description above that each output sample is influenced by several input samples, and that each input sample affects several output samples. So the process of sample rate conversion is stateful.

The rate plugin delegates the process of sample rate conversion to a pluggable external converter. Alsa-lib itself contains a very simple (and low-quality) converter based on linear interpolation. Alsa-plugins contain converters based on the Speex resampler, the ffmpeg resampler, and libsamplerate.

This architecture suffers from the same limitations as discussed above for extplug. Namely:

- there are no rewind-related callbacks in the `snd_pcm_rate_ops` structure;
- none of the underlying libraries supports rewinds explicitly, or allows to save and restore (or otherwise alter) its state programmatically.

Therefore, in the current architecture, the rate plugin cannot be rewindable. And indeed, it isn't rewindable, as of alsa-lib 1.0.28.

The same objections apply to resamplers used by PulseAudio, and there is already a bug[11] reported by a user who noticed imperfect stitching of resampled audio before and after the volume change of an unrelated stream.

This situation is far from ideal, especially since sample rate conversion is a very common part of the audio processing pipeline. An important difference here from the `ioplug/extplug/ladspa` case is that there is, in fact, no task to wrap arbitrary third-party libraries, especially since none of the existing resampler libraries are actually suitable. The process of sample rate conversion is well-defined mathematically, the set of input samples affecting a given output sample is known, so it is possible to write a rewindable windowed-sinc resampler implementation from scratch. But nobody did it so far.

5.7 PulseAudio virtual sinks

Some PulseAudio virtual sinks (e.g., `module-ladspa-sink` and `module-echo-cancel`) perform non-trivial audio processing and keep state. PulseAudio sink API includes the rewind operation, and plugins generally supply it. However, the implementation either only moves pointers, or resets the filter completely (because the backend library is not rewindable), which is wrong. To fix this, one needs to remove the "reset the filter" recommendation in the `module-virtual-sink` template module, and explain how to express the fact that the virtual sink is not rewindable.

To be fair, the recently-submitted LFE filter patchset by David Henningsson[12] takes rewinds into account.

5.8 Dealing with non-rewindable devices

PulseAudio currently contains some logic[13] that disables timer-based scheduling and rewinds on `ioplug` plugins such as `a52`. However, the code does not match the (extplug-based) `dca` plugin as non-rewindable, and needs to be updated.

Initially, the idea was to fix the `snd_pcm_rewindable()` ALSA API function so that it always returns 0 for non-rewindable plugins (which is a good idea anyway) and use it. However, there are two reasons why this solution cannot work.

First, `snd_pcm_rewindable()` works only when the buffer size is already set. As already explained, rewinds are needed only in order to compensate the unacceptably-high latency implied by a large buffer. If rewinds are impossible, the buffer must be small. So, in order to see whether we need a small buffer, we would need to set the buffer size already.

Second, without sending a test sound, it does not actually help to distinguish a rewindable device from a non-rewindable one. Indeed, a rewindable-in-principle device with an empty buffer (and the

[11] `https://bugs.freedesktop.org/show_bug.cgi?id=50113`

[12] `http://lists.freedesktop.org/archives/pulseaudio-discuss/2015-January/023042.html`

[13] `http://cgit.freedesktop.org/pulseaudio/pulseaudio/commit/?id=cb55b00ccd25d965b1222e74375aee05427a449b`

buffer is initially empty) cannot be rewound.

Thus, a new API is needed in order to test whether an ALSA PCM is rewindable at all. Such API (`snd_pcm_hw_params_can_rewind()`) has been added in April 2008, but removed[14] 10 days later, because it was thought (wrongly) that the `snd_pcm_rewindable()` API function is more useful.

6 Client-side timing

Some legacy applications (e.g. many ALSA-based media players) rely on the audio subsystem as a source of timing. In particular, they expect the wakeups to come in a regular fashion, in strict accordance to the period size. To satisfy such legacy clients, PulseAudio has a special `PA_STREAM_EARLY_REQUESTS` flag that can be specified when creating a stream. Without this flag, requests will be made as late as possible. The `pulse` ALSA plugin always sets this flag.

On a system where timer-based scheduling works, and the CPU scheduler behaves reasonably, this flag usually works as expected. Indeed, due to the ability to program the timer for an arbitrary interval, PulseAudio can emulate any period size to the client.

Problems begin[15] when PulseAudio decides not to use timer-based scheduling (e.g., due to a batch card). In this case, PulseAudio uses the period size that is supported by the sound card and is close to the one specified in the `daemon.conf` file. Now suppose that the stream is moved to a different sound card that does not support this period size. As PulseAudio only wakes up and requests data from the client only on interrupts from the sound card, it no longer can wake up the client precisely when needed. In theory, clients are notified when buffer metrics change, and can adapt, but, in practice, no client handles this seriously. Worse, wrapper libraries such as alsa-lib and SDL cannot handle this easily, as they don't have the notion of dynamic changes of buffer metrics in their client API.

A separate question is what to do with clients like Wine or QEMU that (for various legitimate reasons) request very low latencies that are impossible to satisfy with the default buffer and period sizes. To add insult to the injury, the `pulse` ALSA plugin accepts almost any period size and, due to lazy creation of the PulseAudio stream, has no way to tell the client that the requested buffer and period sizes were actually not used.

The arguments listed above highlight the fact that PulseAudio, in non-tsched mode, does not perform adequate isolation of clients from the actual sound hardware, in terms of the supported and advertised period sizes. The bug can be fixed by asking sound data from a client using a separate timer, not based on soundcard interrupts, and possibly lying to the client about the total latency where regularity of requests matters more than exact latency estimation.

[14]`http://git.alsa-project.org/?p=alsa-lib.git;a=commitdiff;h=c88672d86fe713e8f049df895fc3b64c472fbf5d`

[15]`https://bugs.freedesktop.org/show_bug.cgi?id=66962`, wrongly closed as fixed at the time of this writing

7 Conclusion

Timer-based scheduling does solve the real problem that it is intended to solve: it achieves dynamic latency, which should be good for power saving. If no resampling or other stateful audio processing is used, it "just works" on simple devices that DMA one sample at a time and report their DMA position precisely. On devices with more complex buffering models, it runs into corner cases described in this paper. But none of the listed problems look unsolvable – after all, there is always a possibility to fall back to the traditional period-based playback model. And there is indeed development work ongoing to provide more detailed timing information, to implement rewinds correctly in new PulseAudio effects, and to make other improvements – which is a good thing.

8 Acknowledgements

The author would like to thank Lennart Poettering for writing PulseAudio, and the current developers for continuing with the project.

References

[1] ALSA project – the C library reference. PCM (digital audio) plugins. `http://www.alsa-project.org/alsa-doc/alsa-lib/pcm_plugins.html`.

[2] Lennart Poettering. 2008. What's cooking in PulseAudio's glitch-free branch. `http://0pointer.de/blog/projects/pulse-glitch-free.html`.

[3] www.freedesktop.org. PulseAudio. `http://www.freedesktop.org/wiki/Software/PulseAudio/`.

[4] www.chromium.org. CRAS: Chromium OS Audio Server. `http://www.chromium.org/chromium-os/chromiumos-design-docs/cras-chromeos-audio-server`.

[5] Fons Adriaensen. 2005. Using a DLL to filter time. `http://kokkinizita.linuxaudio.org/papers/usingdll.pdf`.

[6] Julius O. Smith, 2015. Digital Audio Resampling Home Page, "Theory of Operation" section. `http://www-ccrma.stanford.edu/~jos/resample/Theory_Operation.html`.