OpenBSD audio & MIDI framework for music and desktop applications

Alexandre Ratchov
alex@caoua.org

AsiaBSDCon 2010 — Tokyo, Japan

Abstract

sndio is an enhancement of OpenBSD audio and MIDI subsystems. It features an audio and MIDI server and provides a new library-based API. The framework is designed with certain constraints of music applications in mind: strict latency control and precise synchronization between audio streams. It supports resampling and format conversions on the fly, sharing devices between multiple applications, splitting a device in multiple subdevices and synchronization with non-audio applications.

Simplicity of the architecture, the design and the implementation are of first importance, attempting to obtain a lightweight, fast and reliable tool.

1 Introduction

Desktop and music audio applications require the operating system to expose audio and MIDI hardware. But there's a gap between applications requirements and what the hardware provides. The gap is filled by the *high-level audio subsystem*, a layer between applications and device drivers. It could be used to convert streams from application format to device format; to mix multiple streams allowing concurrent access to the device; to expose the audio stream clock to non-audio applications; to provide a mechanism to synchronize audio streams allowing multiple simple programs to be used to perform a complex task. The sndio framework attempts to address part of these problems on OpenBSD. There are several existing solutions to these problems.

NetBSD has a kernel audio framework [4] that handles format conversions and resampling only; it allows running most applications on most supported hardware, but does not allow running multiple audio applications simultaneously. FreeBSD has a similar kernel framework [5] that can also split hardware devices in multiple logical devices and apply digital effects on the fly; it allows sharing devices between multiple programs, but provides no system level mechanism to synchronize them. OSS is a commercial open sourced kernel framework provided by 4Front technologies with similar features to FreeBSD's one. All these implementations expose APIs based on system calls (as opposed to libraries). Linux uses ALSA: a rich and complex framework running as a user-space library [6]. It's based on plug-ins processing audio streams on the fly. Plug-ins exist for virtually anything, including mixing, effects, conversions, resampling.

To overcome audio subsystem limitations certain user-space libraries and audio servers can be used. For instance GNOME and KDE projects used to use esound and artsd to overcome linux audio subsystem limitations before ALSA became available; now they are replaced by pulse which has yet more features than ALSA plug-ins [7, 8].

Above kernel or user-space frameworks suppose that audio applications are independent, *i.e.*, that synchronization or data exchange between applications are not handled by the audio subsystem. Such frameworks are usable for music production (or other more complex audio tasks) as long as all audio-related processing is done inside a single monolithic program. The jack [9] framework overcomes this limitation: it allows programs to cooperate: they can pass data synchronously to each other; furthermore, jack provides a synchronization mechanism for non-audio applications. On the other hand, jack supports only one sample format and runs at fixed sample rate, which is acceptable for music applications. In theory jack could be used for desktop also, but other frameworks have been privileged by Linux desktop distributions.

The proposed sndio framework, attempts to meet most requirements of desktop applications, but pays special attention to synchronization mechanisms required by music applications. The current implementation supports conversions, resampling, mixing, channel mapping, device splitting, and synchronization of audio streams. Additionally, per-application volume and synchronization are controlled by standard MIDI protocols, allowing interoperability not only with MIDI applications but also with MIDI hardware connected to the machine. These aspects are novel in the Unix world. Simplicity and robustness of the architecture, the design and the implementation are part of the requirements; thus certain complicated or non-essential features are not provided (*e.g.*, effects, monitoring, network transparency).

First, we present the problem sndio attempts to solve and the project goals. Then we discuss all technical choices attempting to show they are a natural consequence of project goals. Then we present the resulting sndio architecture. Finally we show few use-cases to illustrate how sndio is used in real-life.

2 The problem to solve

Need for conversions. New, professional audio interfaces may expose 32-bit formats only, few fixed sample rates and a large number of channels. Such parameters are not supported by most common audio applications, so format conversions are necessary. This is also true, to a lesser extent, for audio interfaces integrated in laptops and desktop computers.

Sharing the device between applications. Most audio interfaces can play or record only one stream at a given time. Basically, this means that only one application may use the device at a given time. To allow concurrent playback or recording, the audio subsystem must be able to mix multiple streams.

Splitting the device in subdevices. Modern audio interfaces may have a lot of channels. In certain circumstances, it is desirable to split a device into multiple independent subdevices; for instance, multichannel hardware could be split into two subdevices: one reserved to telephony (*e.g.*, headphones and microphone) and another one for desktop noises and music (*e.g.*, speakers).

Synchronization. Performing complex tasks using multiple small tools, each performing a simple task is part of the Unix philosophy[1] [1]. In the audio and MIDI domain, this requires simple tools, not only to have access to the audio hardware, but also to be able to work synchronously. The MIDI protocol provides synchronization mechanisms, and the aim of this work is to integrate them at system level, allowing audio applications to cooperate without the need for intrusive code modifications.

Fault-tolerance. If a transient error condition causes the audio subsystem to fail, then it should recover once the error condition is gone. For instance, if a system load burst causes a buffer to underrun (and the sound to stutter), then no application should go out of sync, including non-audio applications synchronized to the audio stream. The user should only observe a transient quality degradation.

Simplicity This is not a technical requirement, neither is it specific to audio. Complicated code leads sooner or later to bugs and fails. Complicated APIs tend to be misused, lead to bugs and fail. Complicated software with too many knobs tend to be misused by the user and fails. The goal of this project is not to support everything, but to handle a small and consistent set of use-cases and to provide a reliable audio framework for audio development.

3 Design considerations

3.1 Performance *vs.* responsiveness

Performance is related to the CPU time consumed to perform a given task, while responsiveness measures how fast an event is processed. There's no direct relation between performance and responsiveness: depending on what a program is designed for, it may have bad performance and good responsiveness or the opposite.

An audio server is a I/O bounded process: after all it's supposed to be only a thin layer between the application and the device driver. Thus, its total CPU consumption is very small and is supposed to be negligible compared to the CPU resources available on the machine. So "wasting" CPU cycles is not a problem as long the total CPU time consumed stays negligible.

Responsiveness, on the other hand, is of first importance: for instance, the audio server must produce data to play as soon as the device requests it, to minimize the chances of buffer underruns which cause the sound to stutter. Similarly, a MIDI "play note" event incoming from a MIDI keyboard must be processed immediately and transmitted to the synthesizer which will produce the actual sound.

3.2 Kernel *vs.* user-space

On Unix-like systems, audio hardware is accessed through a character device driver. Conversion, mixing, and resampling code is an intermediate layer between the device driver and the application. Only the device driver must run in kernel space; conversions and mixing can be performed either in user-space as a regular user process or in kernel mode.

In a user-space implementation, the application produces audio data, transmits it to the audio server, which in turn processes the data and transmits it to the kernel. In a kernel-only implementation, the application produces audio data and transmits it directly to the kernel, which does the conversions and transmits the result to the hardware.

3.2.1 Overhead of data copying

In a kernel-only implementation, communication between a user-space process and the kernel uses `read` and `write` system calls which copy data from one address space to another; obviously, this data copying is useless. In a user-space implementation, things are even worse: the application transmits data to the audio server using sockets which involves *two* extra copies of the data.

Copying data may consume almost as much CPU time as processing it, but generally this is not a problem as long as the total CPU time consumed is negligible. For instance, on a Sharp SL-C3200[2], which we consider as "not very fast", the overhead of the extra copying is around 50% of the CPU consumption, which might seem significant. However, this corresponds to only 0.5% of the CPU time available on the machine, which we'll consider as acceptable[3].

The basic problem of unnecessary data copying is not about kernel *vs.* user-space implementation; indeed a kernel-only implementation is already subject to unnecessary data copying. The basic problem is in not using shared memory for audio communication. This is a general problem; we leave switching OpenBSD audio to zero-copy audio data exchange scheme as a future project, and we don't take it into account in this design.

[1] Write programs that do one thing and do it well. Write programs to work together. — Doug McIlroy

[2] This machine has a Intel PXA27x ARM processor at 416MHz and runs the OpenBSD "`zaurus`" port.

[3] A rough estimation of the percentage of extra CPU time spent in copying data between two address spaces is easily obtained by comparing CPU usages of the following two commands:

```
$ dd if=/dev/zero | dd of=/dev/null
$ aucat -n -i /dev/zero -o - | dd of=/dev/null
```

the second command runs the "`aucat`" audio server in loopback mode; in this mode it uses its standard output and input as its playback and recording devices [11].

3.2.2 No extra latency

The audio latency is the time between the application starts providing samples to the system and the time the user hears them. Samples are buffered and buffers are consumed by the audio hardware at constant rate (the sample rate), thus the latency is simply proportional to the buffer size.

Whether buffers are stored in kernel memory or user-space memory doesn't matter for the latency. Only the end-to-end buffer usage matters for latency.

3.2.3 Stability: underruns and overruns

When the playback buffer underruns, the audio interface has no samples to play anymore, and inserts silence to play, causing the sound to stutter. Underruns may cause the application to loose it's synchronization; while this is annoying for video to audio synchronization, it's catastrophic for music performance.

In an user-space implementation, to avoid buffer underruns, the application must grab the CPU fast enough and produce samples to play; then, the audio server must grab the CPU immediately and submit the samples to the device. If the application or the server are delayed, then the driver will not receive audio data to play at time.

In a kernel implementation, the kernel processes the data and transmits it to the audio device in a single shot, *i.e.* without being preempted. Compared to the user-space implementation, the kernel implementation is not subject to underruns caused by the audio server only.

However that doesn't mean that in real-life kernel implementation gives a noticeable gain of audio stability. Implementing the audio subsystem in kernel space will prevent the server from underruns, but in case the system is busy, the application will underrun anyway, causing stuttering.

One may argue that in the case of an user-space implementation there may be underruns caused only by the server, *i.e* the application is run at time but the server execution is delayed. The author experience is that this never happens: all processes are equally subject to delays. Furthermore the most a process is CPU intensive, the most chances it has to underrun and one of the main requirement for the `sndio` implementation is to be lightweight.

3.3 Server *vs.* library

Only operations involving all streams, like mixing, strictly need to be implemented in the server process. Since conversions, resampling and channel mapping are specific to a single stream, they can be performed by the application itself for instance by wrapping functions to read and write samples. In both implementations, the code is the same and is running in user-space.

The main advantage of the library approach is to not require the audio server if the user plans to run only one stream. The advantage of the server-only approach is to be simpler for both developer and user points of view: the implementation is simpler because all the processing happens in the same place and the user has a single program to configure (the server).

We choose the server-only approach for the current implementation for its simplicity, however switching to the library approach is only a matter of integration effort: pasting server code into the library and adding the necessary configuration file(s).

3.4 Formats and algorithms

3.4.1 16-bit *vs.* 24-bit precision

Samples using 16-bit precision give around 96dB dynamic range[4], which allows distinguishing light leaf rustling (approx. 10dB) superposed to jackhammer sound at 1m (approx. 100dB) [2]. From this standpoint, 16-bit precision seems largely enough for audio processing.

Nevertheless, modern hardware supports 24-bit samples and could theoretically handle 144dB dynamic range, which is much larger than human ear dynamic range (approx. 120dB). However analog parts in audio hardware seldom reach 144dB of dynamic range, and in most cases it's closer to 96dB than to 144dB. In other words, in most cases only 16 most significant bits of 24-bit samples are actually audible[5], least significant bits being hidden by noise and distortion.

On the other hand, 16-bit precision processing requires 32-bit arithmetic and 24-bit precision would require 48-bit arithmetic. The `sndio` audio server is designed to be fast even on slower 32-bit machines, like the `zaurus` port supporting 416MHz Intel PXA27x ARM CPUs. That's why, given the little benefit of 24-bit precision, we use 16-bit precision.

Nevertheless, switching the implementation from 16-bit to 24-bit precision is almost as simple as changing the integer type used to store samples.

3.4.2 Sample formats to support

There are plenty of sample formats designed for a wide range of use-cases, some of which are far beyond the scope of OpenBSD audio subsystem: providing interface to hardware for music and desktop applications.

Integer (aka fixed point) samples These are the most common formats for audio software and hardware. We support any precision between 1 and 32-bits, big or little endian, signed or biased, padded or packed, LSB or MSB aligned (if padded). These formats cover most hardware available in 2010.

μ-law and a-law samples These formats are used in telephony; they are not linear, which means that programs have to decode/encode samples before processing them. Thus all programs using such formats can decode/encode them and no support is required in the audio subsystem itself.

[4] dynamic range is defined by:

$$20 \log \frac{a}{a_0}$$

where a is the maximum amplitude, and a_0 is the minimum amplitude of the signal. With 16-bit samples, the maximum amplitude is 2^{16}, and the minimum amplitude is 1. The dynamic range is thus:

$$20 \log 2^{16} \simeq 96\text{dB}$$

[5] This statement is false for professional audio interfaces that may support up to 120dB.

IEEE floating point Floating point samples, are actually used as *fixed point* numbers, since only the $[-1:1]$ range is used. Floats in this range can do nothing that integers can't. They are used in software mainly because they reduce development time, especially on `x86` platforms where floating point operations are cheap. Since such formats are not strictly required, and conversions to/from integers is trivial, we don't support them.

Encrypted and compressed streams Audio data must be decoded in order to be processed, we leave this task to audio applications. However, there are audio interfaces with "pass-through" mode allowing applications to send encoded data to an external device. In this case the computer is not involved, so there is no point in supporting such formats in the audio subsystem: after all, computers are to process data.

3.4.3 Choice of resampling algorithm

The following widespread approaches to resampling were considered:

- zero order "hold" interpolation: adds a lot of noise but implementation could be very fast.
- linear interpolation: adds much less noise and the implementation still could be almost as fast as the zero order method
- higher order interpolation: adds less noise but is more CPU intensive and complicated to implement.
- sinc interpolation: gives the exact signal[6]. The computation requires knowing all samples in advance (past and future) and is very CPU intensive [3].

With all of above methods, the original signal's spectrum must be bounded by half of the sampling frequency. To ensure this condition, a low-pass filter must be applied to the signal.

In desktop applications like movie or music playback, most of the resampling requirements are to convert 44.1kHz samples to 48kHz. This involves signals that are already filtered thus no filtering is required. Linear interpolation produces noise, but it's small because source and destination frequencies are not very different. The noise is seldom perceptible with typical music or movie tracks.

Telephony applications use 8kHz sample frequency and require resampling from/to 44.1kHz or 48kHz. Human voice spectrum is naturally bounded around 4kHz, thus no filtering is required either. Linear interpolation slightly degrades the signal quality, but high fidelity is not a requirement for telephony.

Music production requires that the recorded signal is not degraded by computer processing, *i.e.*, processing shall only introduce errors smaller than hardware resolution[7]. This is very hard to achieve in real-time and musicians seems to prefer resampling off-line. Furthermore, in music production resampling can be avoided very easily by using the same sample frequency in all recordings.

We use the linear interpolation method because it covers all use-cases and is fast and simple.

3.4.4 Channel mapping algorithm

The audio subsystem often has to transmit a N-channel stream to a M-channel device with $M \neq N$. For instance stereo streams could be played on 8-channel devices.

The general approach, which is also easy to implement, is to use a *matrix* mixer: each input channel is transmitted to all output channels with user supplied volumes. However this would require to expose to the user $N \times M$ knobs, which is unpractical. Furthermore, in most common use-cases where $M \neq N$, a lot of knobs would be always set to zero. To keep things as simple as possible, `sndio` uses channel *ranges* as follows:

- input and outputs have *start* and *end* channels numbers selected by the user
- channel numbers common to input and output ranges are transmitted with full volume the rest is ignored.

3.5 API design considerations

3.5.1 Stream *vs.* shared memory

Audio programs using stream semantics use system calls like `read` and `write` to copy audio data from one address space to another. Programs using shared memory semantics save memory and CPU cycles. Unfortunately, neither kernel drivers nor existing audio applications are easy to switch from stream to shared memory semantics.

Both approaches are useful, depending on the context, but for now we focus on stream semantics; the shared memory approach is planed for future versions of the framework.

3.5.2 Handling of full-duplex streams

In most full-duplex programs there's a dependency between the playback and recording direction, and full-duplex cannot be simply considered as two independent streams. Indeed, most of the time applications require playback and recording direction to be in sync: the n-th written sample is played exactly when the n-th read sample is recorded. The `sndio` approach is to ensure this property at system level, bringing a very simple interface to programs. This simplifies considerably writing or porting audio software using full-duplex.

Applications that need independent playback and recording streams, can simply open two streams instead of using full-duplex.

3.5.3 Asynchronous events

Reading and writing samples doesn't cover all needs of audio applications. There are asynchronous events that programs require: for instance, to synchronize video on an audio stream, the application must have access to the clock ticks of the sound card. Asynchronous events could be exposed through a callback mechanism or through a getter.

[6] Any continuous signal with a bounded spectrum can be sampled and converted back to the original with no loss of information (Shanon theorem). With this standpoint, resampling consists simply in calculating the continuous signal and resampling it to the new frequency.

[7] For instance, when processing signal recorded on hardware with 96dB dynamic range, the noise generated by the processing should be such that the signal over noise ratio stays beyond 96dB.

The getter approach was rejected because there's no easy way to poll for incoming events. Indeed, the poll system call provides events only for when reading and writing is possible (POLLIN and POLLOUT respectively). Furthermore, events can occur during a blocking read or write, during which they couldn't be retrieved[8].

4 Architecture and implementation

4.1 Overview

The sndio framework enhances and completes the kernel audio and MIDI support, following above considerations. The whole audio and MIDI subsystem is thus build around three elements: kernel drivers, the aucat audio server and the library-based API to access hardware devices and software services in a uniform way.

4.1.1 Kernel drivers

Kernel audio and MIDI drivers expose hardware to user-space processes through a character device. Only one process may use a given device at a given time.

4.1.2 Audio server

The aucat program can behave as an audio server [11]. It exposes "logical" subdevices that applications can use as they were actual hardware devices. It acts as a transparent layer between logical devices and hardware devices. It does format conversions and resampling on the fly and allows multiple applications to use the same device concurrently.

Each client has it's own volume setting that can be controlled through MIDI . This allows using a MIDI control surface with motorized faders as a mixer. Volume settings are persistent, meaning that if an application disconnects from the server and connects back to it, it will get the same volume setting. Volumes are reset if the server is restarted.

Multiple subdevices backed by the same hardware device can be defined. For instance, this allows splitting a 4-channel device in two stereo subdevices; this could allow using headphones and speakers as different subdevices. Parameters like the maximum allowed volume are associated to subdevices. Thus, defining subdevices could also allow forcing applications to use a set of parameters.

Besides above features, the server can start multiple streams synchronously. Once started, streams are maintained in sync, even if underruns/overrun occur. For instance, this allows starting an audio player and an audio recorder synchronously; the resulting recorded track will be in sync with the played track. This mechanism is controlled through MIDI and doesn't not require modifying application code. Only the appropriate MIDI software (or hardware) is required.

The server can also expose a clock reference, allowing MIDI software or hardware to be synchronized to an audio stream, again without modifying the code. This mechanism allows audio application running on OpenBSD to cooperate with MIDI applications or external MIDI hardware.

The last two features – control and synchronization through MIDI – are intended to allow multiple small programs to work together in order to accomplish a complex task. This is a small step toward bringing Unix philosophy in the audio domain.

4.1.3 MIDI server

MIDI is a unidirectional point-to-point serial link. A so called *thru box* can be used to allow a single source (*e.g.*, a master keyboard) to send data to multiple destinations (*e.g.*, synthesizers). To a certain extent, thru boxes are for MIDI what hubs are for Ethernet.

The midicat program creates software thru boxes allowing applications to send data to other applications [12]. Hardware MIDI ports can also be connected to thru boxes as they were programs. This, in turn, allows a single device to be shared by multiple programs.

Thru boxes and hardware MIDI ports are accessed by programs in an uniform way. In other words the midicat program emulates a hardware MIDI port with a physical thru box connected to it.

4.1.4 Audio and MIDI access library

A library-based API, the libsndio, is provided to access hardware devices and software services in an uniform way.

The API is very simple. It's designed to help avoiding programing mistakes. Besides its simplicity, it doesn't require managing by hand synchronization of playback and recording directions of full-duplex streams.

4.2 Kernel drivers and dependencies

Audio and MIDI hardware is exposed to user-space processes through the corresponding kernel drivers. Only the following subset of features offered by the audio driver are used by the sndio framework:

- read and write system calls to receive recorded samples and send samples to play. The audio driver splits inputs and outputs in blocks of equal size, but this feature is not required and is used only as an optimization.
- ioctl to get and set audio parameters, like the encoding, the sample rate, the number of channels and the block size. For full-duplex, we use always the same parameters for playback and recording.
- ioctl to start and stop playback and/or recording. For playback, first, the device is stopped, then few blocks of samples are written, and finally, once there are enough samples to play, the device is started.
- poll is used to check whether the device is readable or writable. Certain audio APIs – including the OpenBSD one – tend to use slightly modified poll semantics: POLLIN and POLLOUT are set only if at least one block can be read or written. Since we don't depend on input or output being block-based, any poll implementation is usable.
- ioctl to fetch the number of samples processed by the device. These counters are used for synchronization

[8] But probably programs using blocking I/O seldom need asynchronous events.

purposes, and their value must be correct. We don't use their absolute value; we only use the difference between values returned by successive calls. Thus they are allowed to overlap.

- `ioctl` to fetch the number of errors, *i.e.*, the number of samples inserted by underruns or dropped by overruns. In the OpenBSD implementation, the counter of processed samples is not incremented during underruns and overruns, thus we use the number of errors as a correction to the total number of samples. The number of errors is allowed to overlap.

A very strong requirement is the number of processed samples to be correct (underrun and overrun corrections included). The requirement of correct counters may seem surprising, since if an API exposes such counters, then one expects them to be correct. Nevertheless, the author noticed they were not accurate in the initial OpenBSD implementation.

MIDI hardware is used as a simple serial port, and no `ioctl`'s are used. We don't use "cooked" interfaces like the OSS-style `sequencer` device.

4.3 Server architecture

4.3.1 Data processing chain

The `aucat` audio server is build as a network of small processing units running inside a single Unix process. Each unit, called `aproc` structure, performs a simple task (like resampling or mixing), which consists in processing audio data on its input(s) and storing the result on it's output(s) for processing by its neighbor unit. Processing units are interconnected by FIFOs, called `abuf` structures. Terminal `aproc` structures are sockets, files or devices, and have either only inputs (file writers) or only outputs (file readers). Fig. 1, illustrates the data processing chain corresponding to a typical configuration of the audio server.

To some extent, `aproc` structures can be considered as execution threads, but they are much simpler than actual machine threads: all of them use the same code structure and none of them requires a stack. This allows optimizing them a lot, avoiding system threads or alike. Any processing unit sleeps while there's no data to process and is waked up when data is available on one of its inputs or when data can be sent to one of its outputs. There's no scheduler because interactions are very simple; each processing unit calls its neighbor when data is pushed to an output or pulled from an input.

Buffers have two ends (reader and writer end), writing on the buffer triggers its reader, and reading from the buffer triggers its writer. This creates a cyclic dependency not only between neighbor processing units, but across all the network. At first glance such cycles are not triggered; furthermore, each iteration involves data processing, and once there's no data to process anymore, the cycle will break by itself. However the caller unit state is stored on the machine's stack, and cyclic calls could make the process stack grow indefinitely. Rather than dealing with cycles, which would complicate the implementation, we preferred avoiding the cyclic dependency. This is easy because the network of processing units itself has no cycles, and thus all we have to do is preventing units from calling their caller.

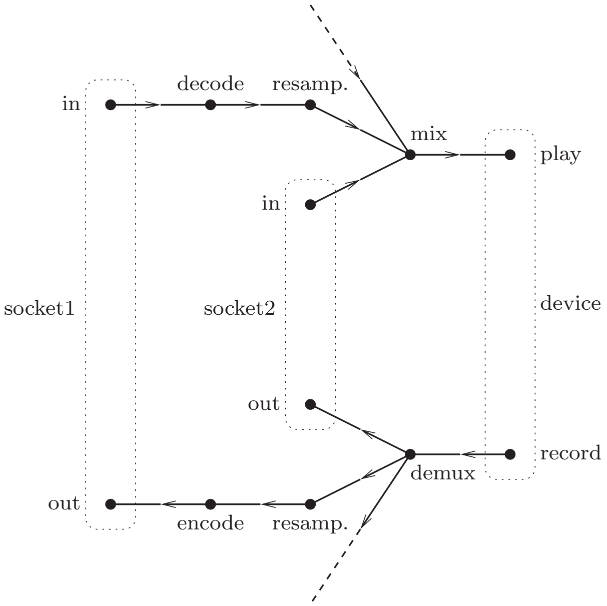

Figure 1: Network of processing units of a server instance: processing units are represented as bullets, and FIFOs are represented as lines. Two full-duplex sockets are connected to the mixer and demultiplexer. Socket "1" uses format conversions and resampling on the fly while socket "2" uses the server native parameters.

To each file descriptor is associated a "reader" terminal `aproc` responsible for reading it and/or a writer terminal `aproc`. The server polls on its file descriptors in a infinite loop. Every time a descriptor becomes readable (writable), its corresponding reader (writer) terminal `aproc` is invoked. When there are no file descriptors to poll anymore, the server exits.

Note that this framework for non-blocking processing is not specific to audio. It can be used for anything else and we'll use it for MIDI . That's why, the `midicat` binary is simply a link to the `aucat` binary.

4.3.2 Server and non-server modes

Processing units are elementary bricks, and depending on how they are connected, the `aucat` program could perform different tasks. There are three main tasks we're interested in:

Server mode. At server startup, the audio device is opened, and it's corresponding writer and reader `aproc` structures are created. They are attached to a mixer `aproc` (for playback) and to a demultiplexer `aproc` (for recording) respectively. The server starts listening for incoming connections on its Unix domain socket, and the main loop can start.

Every time a connection is accepted, reader and writer terminal `aproc` structures are created for the corresponding socket, and it's inserted to the list of files to poll. Reader and writer `aproc` structures are connected to the mixer and/or the demultiplexer `aproc` structures. If con-

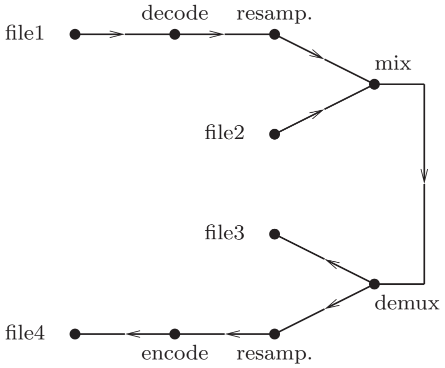

Figure 2: Network of processing units of an `aucat` instance performing off-line conversions. Two files (1 and 2) are mixed and the result is recored on two other files (3 and 4) using different parameters.

versions and/or resampling are required, then the corresponding `aproc` structures are created and inserted in the processing chain, as shown on fig. 1.

Player & recorder mode. One may notice that playing data from a socket or from a regular file would be almost the same. Only the reader and the writer implementation changes. So, attaching regular files at startup costs nothing. The `aucat` program has command line options to attach a regular file to play or to record, and thus can be used as a general purpose player/recorder.

Off-line conversion mode. Instead of opening and creating the device `aproc` structures, one can connect the mixer output to the demultiplexer input, as shown on fig. 2. In this mode `aucat` plays (and mixes) input files and records the result on output files. It can be used as a general purpose conversion, resampling, mixing and channel extraction utility.

4.4 Latency control

One of the main objections against audio servers is the extra latency they add. This an implementation problem and the extra latency they add can be controlled and/or avoided, as follows.

The latency is the time between a sample is produced and the time it is played by the audio hardware. Audio samples are buffered, and buffers are consumed at constant rate: the sample rate. Thus the latency is:

$$\texttt{latency} = \frac{\texttt{buffered}}{\texttt{rate}}$$

So controlling the latency is equivalent to controlling the number of buffered samples which is easily achieved through flow control.

The number of buffered samples includes any buffers involved, including “hidden” buffers like the kernel buffers of the Unix-domain connection and device driver buffers. It's very easy to calculate:

$$\texttt{buffered} = \texttt{written} - \texttt{played}$$

The number of samples played is exposed by the kernel driver, while the number of written samples can be counted easily. Thus, the maximum latency can be fixed by preventing the application from filling buffers beyond a fixed limit. This mechanism is implemented inside the equivalent of the `write` system call of the `sndio` library. The maximum latency is set by the program at initialization, and the above mechanism ensures that the program never goes past it.

The server mixes multiple streams, and the result of the mixing is stored into a buffer for output to the device. This buffer contributes to the total latency. But since it's common to all clients, its size cannot be controlled by a single client; it's set at server startup. The size of this buffer is thus the minimum buffering a program can request[9]. The `aucat` program provides an option to set it.

4.5 Handling overruns and underruns

Underruns – causing sound stuttering – happen when the client doesn't provide data fast enough or when the server doesn't process it fast enough. This is mostly caused by system load[10], which cannot be completely avoided on a non-realtime system.

When a client buffer underruns, the server inserts silence to play in the stream. The effect of this is to shift the clock exposed to the application by the amount of silence inserted. This is not acceptable because the program may use the clock to synchronize non-audio events to the audio stream. There are two options:

- Drop the same amount of samples as the amount of silence inserted. From the program point of view, there's a short transient sound quality degradation. For instance, this behavior is suitable for a audio and MIDI sequencer, where loosing the tempo transiently is not acceptable. However if the underrun duration is long, the program will not recover quickly; for instance a program cannot be suspended and resumed with this approach.

- Pause the clock by the amount of silence inserted. The underrun can be possibly very long. However this approach is not suitable for music applications. Furthermore, multiple synchronous streams don't underrun the same way, their clock won't pause the same way and in turn they will go out of sync.

There's no “best approach”, it depends on the program goal, that's why we let programs choose their underrun policy.

Besides the clock reference, the server must keep in sync playback and recording directions of full-duplex streams. If the playback direction underruns, then since silence is inserted; to keep the recording direction in sync, the same amount of silence must be inserted in it. Similarly, if the recording direction overruns, then since samples are dropped from it; to keep the playback direction in sync, the same amount of samples to play are dropped.

The server process itself may also underrun or overrun, if the machine is busy. This is less likely to happen since it is I/O bounded and the OpenBSD scheduler prioritizes such

[9] Actually, in the current implementation, kernel device buffers also count in the minimum latency, but that will be fixed later.

[10] Or poor implementation.

processes. But the server uses the `sndio` interface to access the actual audio device. In other words, `aucat` is "the client of the device", and thus its buffer underruns and overruns will be handled by the lower layer, namely the device.

4.6 Sharing the dynamic range

Mixing streams means adding their samples. Obviously if multiple 16-bit streams are added, the result cannot fit into 16-bit integers. A simple solution would be to take the average of inputs instead of their sum, as follows:

$$y(t) = \frac{1}{N}\sum_{i=0}^{N-1} x_i(t)$$

where $x_0, \ldots, x_{N-1}$ are input streams, and y the result of the mix. This reduces the volume of each stream as follows:

$$x_i(t) \mapsto \frac{1}{N} x_i(t)$$

This is not a problem as long as N stays constant in time, but that's not the case. When a new stream is inserted, *i.e.*, the number of streams becomes $N+1$, the i-th stream changes:

$$\frac{1}{N} x_i(t) \mapsto \frac{1}{N+1} x_i(t)$$

its amplitude makes a relative[11] step of:

$$\frac{N}{N+1}$$

For small values of N the step is large (*i.e.*, very audible) and it converges to 1 (*i.e.*, no step) for large values of N. This is a problem because audio servers are mainly used with one or two streams, so this option is not acceptable.

The other possibility would be to fix M, the maximum number of streams and to systematically scale inputs by $1/M$. But when there's only one input stream (most of the time), the dynamic range (thus quality) is unnecessarily reduced, which is not acceptable either. Actually, this depends on user requirements, so we used the following compromise: we let the user choose M the maximum expected inputs and we scale inputs by $1/M$ if $N < M$, else we scale them by $1/N$. By default $M = 1$ to keep the full dynamic range when there's a single output, but if volume steps are annoying, the user can pick a larger value of M.

4.7 MIDI control of the volume

We allow the volume of each client to be controlled individually. Rather than adding yet another API, and yet another utility to control the volume, the `aucat` server exposes a MIDI port and uses the standard MIDI volume controller messages to adjust the volume. Besides the advantage of being simple, this approach allows volume to be controlled not only from common MIDI programs, but also from MIDI hardware. For instance a control surface with motorized faders can be used as it was a mixing console; such hardware is a much more appropriate input device than the mouse and keyboard of the computer. MIDI also allows saving and restoring the mixer state into a standard MIDI file, or to create volume automation by simply playing a file with volume change events.

The drawback of this approach is that the user sitting behind her/his control surface doesn't know which fader corresponds to which application. Indeed, when a new application connects to the server, a unused fader is allocated to it but its name is not exposed yet[12]. To mitigate this problem, faders allocations are persistent: when an application is started, it gets the same fader it used the last time it was started. As a desirable side effect, the volume is restored allowing the user to keep her/his per-application volume settings persistent.

4.8 Synchronization of audio streams

Keeping in sync two audio applications means that the n-th sample seen by one application is played or recorded simultaneously as the n-th sample seen by another application. This property must be preserved across transient underruns or overruns. Obviously this requires that applications start simultaneously on the same hardware device[13]. From the `aucat` server standpoint, this is easily achieved by atomically connecting the application to the mixer and/or demultiplexer, see fig. 1. Streams that require to start synchronously, are not started immediately; they are put in a "ready to start" state, and once all streams are in this state, they can be started atomically on user's request.

Applications don't need to explicitly request this feature. It is enabled at server start-up on a per subdevice basis. Typically, the user can create two subdevices for the same hardware device: the default one and an extra device with synchronization enabled. Depending on which subdevice the application uses, it will be synchronized to other applications or not.

We use MIDI Machine Control (MMC) protocol to start and stop groups of applications. This avoids defining a new API and developing new tools. Most software or hardware sequencers support this protocol, and it's likely that the user will need to control applications from such software or hardware anyway. MIDI controllers often have start/stop buttons too.

This implementation of MMC has three states:

- stopped – incoming streams are not actually started, but put into a "ready to start" state. Even if all streams are ready to start, they are not started until a MMC start message is received.

- starting – incoming streams are not actually started, but put into a "ready to start" state. Once all incoming streams are ready to start, they are atomically started and the server moves to the "running" state. If a MMC stop message is received, then the server moves back to the "stopped" state.

[11] We hear relative changes of the amplitude only: we can't hear a rustling near a jackhammer, but we easily hear it in a quiet room.

[12] This is being worked on.

[13] Different audio interfaces use different clock sources, which slightly drift; the drift is accumulated over the time which sooner or later makes devices go out of sync enough to be audible. That's true even for devices of the same model from the same vendor, using the same electronic components. Audio interface clocks cannot be adjusted, through certain professional interface can use external clock sources. Nevertheless we focus on features working on any device.

- running – all streams are playing and/or recording synchronously. If a MMC stop message is received, then the server moves to the "stopped" state, but streams continue until all applications explicitly stop.

4.9 MIDI to audio synchronization

Non-audio applications can be synchronized to audio applications through MIDI. Indeed the MIDI Time Code (MTC) protocol allows multiple slaves to be synchronized to a master. The master sends a "start" event, and then sends clock ticks at fixed rate. Slaves use these clock ticks instead of using the system timer. There's no "stop" event, the master just stops sending clock ticks.

In the `sndio` case, the master is the audio server, and MIDI applications (or hardware) are its slaves. This is the most interesting when an MTC slave can be the MMC master, *i.e.*, when a single MIDI application (or hardware) controls audio streams and stays synchronized to them. This is a key feature allowing multiple simple programs to be used together to achieve complex tasks.

4.10 Overview of the API

The audio API [13, 14] to expose hardware devices or logical subdevices, mimics `read` and `write` systems calls when possible. This eases porting code using OSS, Sun or ALSA programming interfaces. The only significant difference visible by the user being worth mentioned is the device naming scheme [10]. Since audio device names can refer to a hardware device or to a subdevice exposed by `aucat`, which is not located on the file system, we can't use paths, so we use the following naming scheme:

```
type:unit[.option]
```

Possible values of the type of the audio devices are "`aucat`" and "`sun`" corresponding to `aucat` logical subdevices and real hardware respectively. Possible values for MIDI devices are "`midithru`", "`rmidi`" and "`aucat`" corresponding to software MIDI thru boxes, hardware MIDI ports and `aucat` control port respectively. The unit number corresponds to the character device minor number for hardware devices; for audio or MIDI devices created with `aucat` or `midicat` it corresponds to the server *unit number* specified on startup, typically 0. The "`option`" parameter corresponds to the subdevice name registered on server startup. The naming scheme is similar to the one used by ALSA.

5 Using the framework

5.1 Porting and writing applications

The `sndio` API is simple, and there's only one way to do one thing. Writing the actual code of a new `sndio` back-end for a simple application often takes less time than tweaking GNU autotools to integrate it in the application. Writing `sndio` back-ends is simplified further by the fact that certain tasks like synchronization are handled by `sndio` itself and do not require code in the application anymore. Less concepts, less thinking, less code, less bugs.

To the author knowledge there's no new code written primarily for `sndio` outside OpenBSD; `aucat` itself uses `sndio` to access the hardware though.

5.2 Setup examples

5.2.1 Audio server for desktop applications

This is the easiest use-case, there's no configuration file, neither specific options are required:

```
$ aucat -l
```

At this point multiple audio applications can use concurrently the default audio device, and any necessary conversions are performed on the fly.

5.2.2 Multistreaming

Suppose the audio hardware has 4 (or more) channels: first two wired to main speakers, and next two wired to headphones:

```
$ aucat -l -c 0:1 -s default -c 2:3 -s hp
```

At this stage applications can choose between two stereo devices: the default one corresponding to speakers and an extra device corresponding to headphones.

5.2.3 Creating a MIDI thru box

The following command:

```
$ midicat -l
```

creates a unconnected thru box. Applications can use it to communicate as they were running on separate machines connected with MIDI cables.

Furthermore, to share a MIDI keyboard across multiple applications, its MIDI port can be subscribed when the thru box is created:

```
$ midicat -l -f rmidi:4
```

where `rmidi:4` is the port of the keyboard.

5.2.4 Making programs cooperate

This requires to create a subdevice with MMC/MTC enabled[14]:

```
$ aucat -l -r 48000 -z 480 -t slave
```

Applications using this device, will be synchronized together. Any MIDI sequencer can be configured to use "`aucat:0`" as MIDI port. At this stage, pressing the "start" button of the sequencer will start all audio applications simultaneously, and – most importantly – the MIDI sequencer will stay in sync with all of them. Moving volume faders will adjust volumes of individual streams.

This use-case is important because this is the only way to use simple programs to record acoustic instruments (or voice) on top of a MIDI accompaniment. Without `aucat`, they can't be synchronized, meaning that the recording cannot be edited and replayed; the only option would be to use a big monolithic application handling both audio recording and MIDI playback.

[14] MTC uses 3 standard clock tick rates: 96Hz, 100Hz, 120Hz. By using a sample rate of 48kHz and 480 samples block size, we ensure that each block corresponds to a tick which gives maximum accuracy.

6 Conclusion and perspectives

6.1 Limitations and future work

There's no plan for much enhancements, one of the main project goals being to stay minimalistic, so it could be considered as almost complete. Nevertheless there's place for a lot of improvements:

First, more audio applications must be ported to `sndio`. As long as not all applications used on OpenBSD are converted to `sndio`, the kernel will have to expose the legacy Sun audio interface. Keeping compatibility prevents from simplifying kernel internals and forces us maintaining unnecessarily complicated code in `sndio` internals.

MIDI control should be extended to expose application names, and to control other parameters than the volume. Furthermore, it's not acceptable to have multiple mixer APIs: the current kernel mixer API and the MIDI protocol used by `aucat`. In the long term either both API will be made uniform or one of them will be deleted. At fist glance this might look like a complication of `aucat`, but in fact it should be an overall simplification of the audio subsystem. Finally, MIDI control is very simple but OpenBSD doesn't provide simple manipulation tools, which means that this feature is reserved to users owning the bulky MIDI hardware or using non-OpenBSD software. Simple programs should be developed to make these features easily available to anyone.

Given the data processing chain described in sec. 4.3.1, its not very hard to implement an interface to snoop audio streams and/or to inject data into them. This would allow simple programs to be combined in a more flexible way to achieve complex tasks. This would be one step further in the Unix philosophy applied to audio. For instance this would allow "recording what the device plays", monitoring, inserting effects, etc.

A longer term improvement would be to switch the kernel, `aucat` and the API to use shared memory instead of unnecessarily copying data. This requires reworking *all* device drivers, and the generic device independent kernel layer; `aucat` will have to use another communication mechanism than Unix-domain sockets.

6.2 Conclusion

We considered problems brought by audio support in desktop and music applications. Besides solving common problems of resampling, format conversions and mixing, the `sndio` framework attempts to address the problem of synchronization of audio and MIDI applications at system level, allowing simple application to cooperate to perform a more complex task.

Simplicity of the architecture, the design and the implementation were of first importance attempting to obtain a lightweight and reliable tool.

Acknowledgments

Thanks to Jacob Meuser for his help on `sndio`, for his work on kernel audio support, for the impressive amount of code he ported to `sndio` and for pushing me to the right direction. Thanks to all who donated machines and sound cards; to all who ported and helped porting code to `sndio`; to all who tested, reviewed and fixed the code. Thanks to Theo de Raadt and people who made `h2k8` possible, where certain ideas come up.

References

[1] M. D. McIlroy, E. N. Pinson and B. A. Tague, *Unix time-sharing system forward*, The Bell system technical journal, **57** (1978)

[2] Wikipedia, *Sound pressure* (2009) `http://en.wikipedia.org/wiki/Sound_pressure`

[3] J.O. Smith, *Digital audio resampling* (2009) `https://ccrma.stanford.edu/~jos/resample/`

[4] NetBSD `audio(9)` manual, *Interface between low and high level audio drivers* (2009)

[5] FreeBSD `sound(4)` manual, *FreeBSD PCM audio device infrastructure* (2009)

[6] *Advanced Linux Sound Architecture project* (2009) `http://alsa-project.org/`

[7] *PulseAudio home page* (2009) `http://pulseaudio.org/`

[8] L. Poettering, *The PulseAudio sound server* `linux.conf.au` (2007)

[9] Paul Davis' JACK presentation, Linux Audio Conference (2003)

[10] OpenBSD `sndio(7)` manual, *Interface to audio and MIDI* (2009)

[11] OpenBSD `aucat(1)` manual, *Audio server and stream manipulation tool* (2009)

[12] OpenBSD `midicat(1)` manual, *MIDI server and manipulation tool* (2009)

[13] OpenBSD `sio_open(3)` manual, *Interface to bidirectional audio streams* (2009)

[14] OpenBSD `mio_open(3)` manual, *Interface to MIDI streams* (2009)