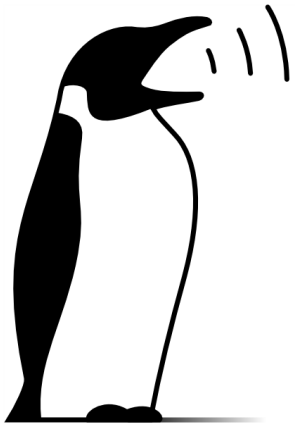
Proceedings of the

Linux Audio Conference 2018

June 7th - 10th, 2018

c-base, in partnership with the
Electronic Studio at TU Berlin
Berlin, Germany



Published by
Henrik von Coler
Frank Neumann
David Runge

http://lac.linuxaudio.org/2018

Published online on the institutional repository of the TU Berlin:
DOI 10.14279/depositonce-7046
https://doi.org/10.14279/depositonce-7046

Credits
Layout: Frank Neumann
Typesetting: LaTeX and pdfLaTeX
Logo Design: The Linuxaudio.org logo and its variations copyright Thorsten Wilms ©2006, imported into "LAC 2014" logo by Robin Gareus

Thanks to:
Martin Monperrus for his webpage "Creating proceedings from PDF files"

Partners and Sponsors

Linuxaudio.org

Technische Universität Berlin

c-base

Spektrum

CCC Video Operation Center

MOD Devices

HEDD

Native Instruments

Ableton

Foreword

Welcome everyone to LAC 2018 in Berlin!

This is the 15th edition of the Linux Audio Conference, or LAC, the international conference with an informal, workshop-like atmosphere and a unique blend of scientific and technical papers, tutorials, sound installations and concerts centering on the free GNU/Linux operating system and open source software for audio, multimedia and musical applications.

We hope that you will enjoy the conference and have a pleasant stay in Berlin!

Henrik von Coler
Robin Gareus
David Runge
Daniel Swärd
Heiko Weinen

Conference Organization Core Team

Henrik von Coler
Robin Gareus
David Runge
Daniel Swärd
Heiko Weinen

Conference Website and Design

David Runge

Paper Administration and Proceedings

Frank Neumann

Organization of music program, installations, and workshops

Henrik von Coler
David Runge

Concert Sound

Henrik von Coler
Jonas Margraf
Paul Schuladen

Review Committee

Fons Adriaensen	Huawei Research, Germany
Henrik von Coler	Technische Universität Berlin, Germany
Götz Dipper	ZKM, Karlsruhe, Germany
Robin Gareus	Germany
Harry van Haaren	OpenAV, Ireland
Joachim Heintz	University for Music Drama and Media Hanover, Germany
Björn Kessler	RISM (Répertoire International des Sources Musicales), Germany
Romain Michon	CCRMA, Stanford University, United States
Martin Rumori	Institute of Electronic Music and Acoustics, Graz, Austria
Bruno Ruviaro	Santa Clara University, United States
Steven Yi	Independent, United States
IOhannes Zmölnig	IEM, University of Music and Performing Arts (KUG), Graz, Austria

Music Jury

Andre Bartetzki
Henrik von Coler
Goetz Dipper
David Runge

Workshops

Day 2

Joao Pais	*Introduction to pmpd*
Marten Seedorf, Simon Steinhaus	*The levTools – a modular toolset in purr data for creating and teaching electronic music*
Louigi Verona	*Djing with FLOSS: Mixxx Workshop*
Hermann Voßeler	*Inbuilt Musicality*
Albert Gräf	*Getting Started with Purr Data*

Day 3

Uroš Maravić, David Vagt	*One Hour Challenge*
Will Godfrey	*Yoshimi Live*
Filipe Coelho	*Carla Plugin Host - Feature overview and workflows*
Joao Pais	*Understanding and being creative with Pure Data's data structures*

Day 4

Daniel James, Christopher Obbard	*How to create real-time audio appliances with Debian GNU/Linux*
David Runge	*Pro-audio on Arch Linux (revisited)*
Rui Nuno Capela	*QjackCtl Considered Harmful*
Uroš Maravić, Tres Finocchiaro	*LMMS 1.2: Changes and Improvements*

Music Program

Opening Night

Louigi Verona	*Minimal House DJ Set*
Superdirt	$Superdirt^2$

Tape Night

Massimo Vito Avantaggiato	*ATLAS OF UNCERTAINTY*
Anna Terzaroli	*Dark Path #2*
Magnus Johansson	*Iammix*
Helene Hedsund	*Bus No. 1*
Massimo Fragalà	*Memorie*
Michele Del Prete	*Spycher*
Andre Bartetzki	*SHIFT*

Performance Night

Alex Hofmann	*COSMO*
Claude Heiland-Allen	*mathr performs with Clive*
José Rafael Subía Valdez	*Tessellations*
Krzysztof Gawlas	*Pick It Up*
Elektronisches Orchester Charlottenburg	*Rotation II*

Installations and Demonstrations

Jaime E Oliver La Rosa	*Caracoles IV*
Marcello Lussana	*Sentire*

Table of Contents

Using Perlin noise in sound synthesis

Artem POPOV
Gorno-Altaysk,
Russian Federation,
art@artfwo.net

Abstract

Perlin noise is a well known algorithm in computer graphics and one of the first algorithms for generating procedural textures. It has been very widely used in movies, games, demos, and landscape generators, but despite its popularity it has been seldom used for creative purposes in the fields outside computer graphics. This paper discusses using Perlin noise and fractional Brownian motion for sound synthesis applications.

Keywords

Perlin noise, Simplex noise, fractional Brownian motion, sound synthesis

1 Introduction

Perlin noise, first described by Ken Perlin in his ACM SIGGRAPH Computer Graphics article "An image Synthesizer" [Perlin, 1985] has been traditionally used for many applications in computer graphics. The two-dimensional version of Perlin noise is still widely used to generate textures resembling clouds, wood, and marble as well as procedural height maps.

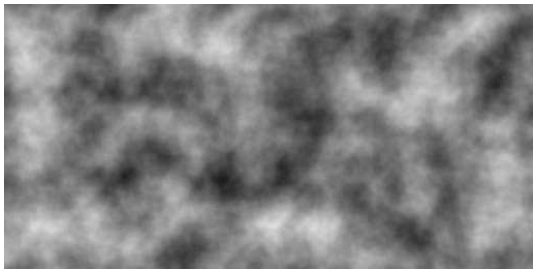


Figure 1: 2D Perlin noise as rendered by Gimp plugin "Solid noise"

Despite its popularity, Perlin noise has been seldom used for creative purposes in the fields outside the world of computer graphics. For music applications, Perlin noise has been occasionally used for creating stochastic melodies or as a modulation source.

This paper is focused on synthesizing single-cycle waveforms with Perlin noise and its successor, Simplex noise. An overview of both algorithms is given followed by a description of fractional Brownian motion and several techniques for adding variations to noise-based waveforms. Finally, the paper describes an implementation of a synthesizer plugin using Perlin noise to create musically useful timbres.

2 Perlin noise

Perlin noise is a *gradient noise* that is built from a set of pseudo-random gradient vectors of unit length evenly distributed in N-dimensional space. Noise value in a given point is calculated by computing the dot products of the surrounding vectors with corresponding distance vectors to the given point and interpolating between them using a smoothing function.

Sound is a one-dimensional signal, and for the purpose of sound synthesis Perlin noise of higher dimensions is not so interesting. While it is possible to scan Perlin noise in 2D or 3D space to get a 1-dimensional waveform, it's necessary to make sure the waveform can be seamlessly looped to produce a musically useful timbre with zero DC offset.

For one-dimensional Perlin noise, the noise value is interpolated between two values, namely the values that would have been the result if the closest linear slopes from the left and from the right had been extrapolated to the point in question [Gustavson, 2005]. Thus, the noise value will always be equal to zero on integer boundaries. By sampling the resulting 1-dimensional noise function, it's possible to generate a waveform that can be looped to produce a pitched tone (Figure 2).

3 Simplex noise

Simplex noise is an improvement to the original Perlin noise algorithm proposed by Ken Perlin

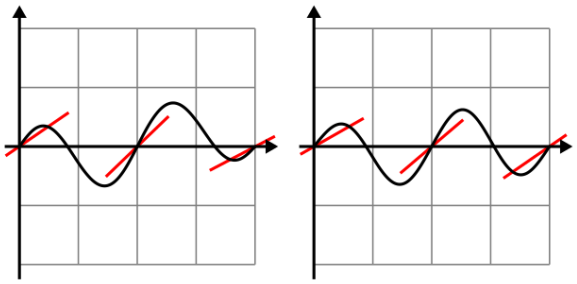


Figure 2: Perlin noise (left) and Simplex noise (right) with the gradients used for interpolation

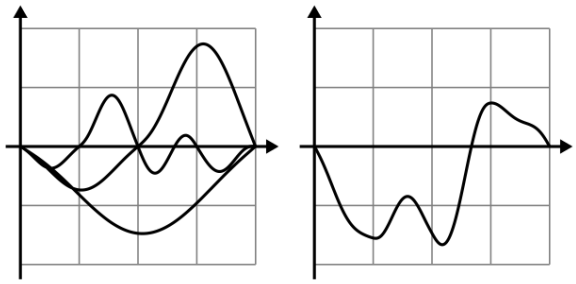


Figure 3: 3 octaves of Perlin noise (left) summed to generate a fBm waveform (right)

himself [Perlin, 2001]. The advantages of simplex noise over Perlin noise include lower computational complexity, no noticeable directional artifacts, and a well-defined analytical derivative.

Simplex noise is created by splitting an N-dimensional space into simplest shapes called simplices. The value of the noise function is a sum of contributions from each corner of the simplex surrounding a given point [Gustavson, 2005].

In one-dimensional space, simplex noise uses intervals of equal length as the simplices. For a point in an interval, the contribution of each surrounding vertex is determined using the equation:

$$(1 - d^2)^4 \cdot (g \cdot d) \tag{1}$$

Where g is the value of the gradient in a given vertex and d is the distance of the point to the vertex.

Both Perlin noise and Simplex noise produce very similar results (Fig. 2) and are basically interchangeable in a sound synthesizer[1]. For brevity, Perlin noise or noise will be used to refer to both algorithms for the scope of this paper, since Simplex noise is also invented by Ken Perlin.

4 Fractional Brownian motion

Fractional Brownian motion (fBm), also called fractal Brownian motion is a technique often used with Perlin noise to add complexity and detail to the generated textures.

Fractional Brownian motion is created by summing several iterations of noise (*octaves*), while successively incrementing their frequencies in regular steps by a factor called *lacunarity* and decreasing the amplitude of the octaves by a factor called *persistence* with each step [Vivo and Lowe, 2015].

$$fBm(x) = \sum_{i=0}^{n} p^i \cdot noise(2^i \cdot x) \tag{2}$$

Lacunarity can have any value greater than 1, but non-integral lacunarity values will result in non-zero fBm values on the integer boundaries. To keep the waveform seamless in a sound synthesizer, lacunarity has be an integer number. A reasonable choice for lacunarity is 2, since bigger values result in a very quick buildup of the upper harmonics (Eq. 2).

Fractional Brownian motion is often called Perlin noise, actually being a fractal sum of several octaves of noise. While typically the same noise function is used for every octave, different noise algorithms can be combined in the same fashion to create *multifractal* or *heterogeneous* fBm waveforms [Musgrave, 2002].

5 Waveform modifiers

5.1 Gradient rotation

One technique traditionally used to animate Perlin noise is gradient rotation [Perlin and Neyret, 2001]. When gradient vectors in 2- or more dimensional space are rotated the noise is varied while retaining its character and detail. This technique has been used for simulating advected flow and other effects. A similar technique can be applied to 1-dimensional noise to introduce subtle changes to the sound.

Rotating gradients is a computationally expensive operation and cannot be used with 1-dimensional noise, since the noise is built from linear gradients instead of directional vectors.

[1] In some cases Perlin noise adds additional low frequency harmonics to the sound which may or may not be desirable.

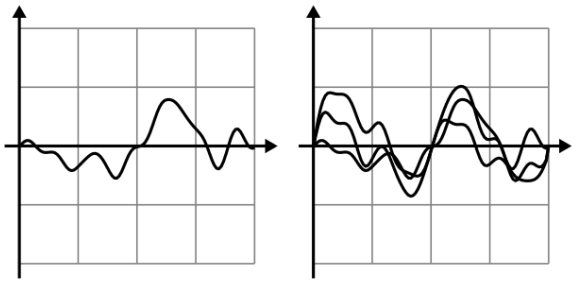


Figure 4: Gradient offsets applied to fBm (left) modify the waveform (right) while preserving the timbre

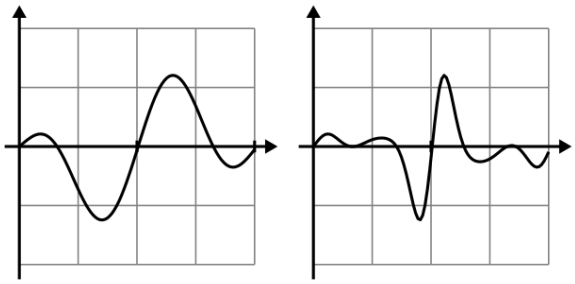


Figure 5: Simplex noise (left) with domain warping (right)

It is still possible to apply this technique to 1-dimensional noise by adding a variable offset value to the gradients and symmetrically wrapping it when the maximum allowed gradient value (1) is reached.

$$g' = \begin{cases} 2 - g, & g > 1 \\ -2 - g, & g < -1 \end{cases} \tag{3}$$

In a sound synthesizer, gradient rotation does not change the timbre significantly. It does alter the amplitudes of the upper harmonics slightly (Fig. 4), adding variations that can be used in a polyphonic (poly-oscillator) synthesizer.

5.2 Domain warping

Another classic technique for adding variation to Perlin noise is called domain warping. Warping simply means that the noise domain is distorted with another function $g(p)$ before the noise function is evaluated.

Basically, $noise(p)$ is replaced with $noise(g(p))$. While g can be any function, it's often desirable to distort the image of *noise* just a little bit with respect to its regular behavior.

Then, it makes sense to have $g(p)$ being just the identity plus a small arbitrary distortion $h(p)$ [Quílez, 2002]. In the most basic case the distortion can be the noise itself (Eq. 4).

$$f(p) = noise(p + noise(p)) \tag{4}$$

For the purpose of sound synthesis it is better to expose warping as an adjustable parameter. Warping modulation can be implemented by adding a coefficient that is used to control the warping depth (Eq. 5).

$$f(p) = noise(p + noise(p) \cdot w) \tag{5}$$

Figure 6: Andes, a JUCE-based synthesizer using Perlin noise

Since the domain of noise is distorted with the noise itself, the symmetry of the waveform will remain generally the same as seen on Fig. 5.

6 Implementation

The presented ideas have been implemented as a basic synthesizer plugin called *Andes* (Figure 6). The plugin has been developed using the JUCE[2] framework and is currently available in the form of VST, AU, and standalone program for Windows, MacOS, and Linux[3].

At the time of writing, Andes supports gradient rotation, basic warping, up to 16 octaves of noise, and adjustable persistence, which allows a usable range of unique sounds to be produced. The sound of noise is susceptible to aliasing at higher frequencies, but oversampling has not been implemented so far.

The resulting sounds resemble early digital synthesizers, but also have a unique character to them and can be described as "distinctively digital".

[2] `https://juce.com`
[3] `https://artfwo.github.io/andes/`

6.1 Predictable randomness

A synthesizer plugin cannot have completely randomly sounding timbres when the synthesizer is used in certain contexts such as a multitrack DAW project. The amplitude and timbre of the synthesizer cannot change in unpredictable ways to make sure the track won't break the mix.

Predictable randomness in Andes is achieved by saving the random seed for generating gradients in the plugin state (preset). The 32-bit Mersenne Twister 19937 generator from C++ standard library is used explicitly to make sure the random numbers generated from the same seed will stay the same across different architectures and platforms.

The set of gradients covering the entire allowed range of octaves is created and stored in memory every time the plugin is instantiated or when a new seed is created using the plugin UI.

The additional advantage of using precomputed set of gradients is that computationally expensive random number generation is moved out of the audio processing code.

6.2 Output level normalization

A big issue with Perlin noise is normalizing the output level to fixed values. This issue is currently not resolved in Andes, but a possible direction to explore is early computing of peak values during the stage of generating gradients.

6.3 Waveform symmetry

The symmetry of waveforms is another thing to consider when developing a noise-based synthesizer.

Current Andes implementation uses completely random gradients. The first noise octave is built from 3 gradients (at points 0, 1, and 2). Sometimes, this results in cusps and unwanted distortion when the both outermost gradients are either positive or negative. Alternating signs for even and odd gradients in the gradient table can further improve the synthesizer usability.

Setting signs for even and odd gradients explicitly can also help reduce the domain range for the noise function.

7 Conclusions

Perlin noise, Fractional Brownian motion and multifractal synthesis are interesting directions to explore for sound applications. Although Perlin noise can be used to make sounds, the approach still remains to be improved. Noise level normalization is one of the biggest issues yet to be resolved.

The general idea of using unconventional, i.e. graphics algorithms in sound and music presents a lot of challenges, but also opens many different possibilities in both the technical and aesthetic aspects.

8 Acknowledgments

The author would like to thank Maria Pankova for helping the idea of making a noise-based synthesizer to emerge and for assistance with maths in the early stages of Andes development. Thanks also goes to Alexey Durachenko for suggesting useful optimizations to the Simplex noise implementation.

References

Stefan Gustavson. 2005. Simplex noise demystified. `http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf`.

F. Kenton Musgrave. 2002. Procedural fractal terrains. In *Texturing and Modeling: A Procedural Approach*, chapter 9.

Ken Perlin and Fabrice Neyret. 2001. Flow noise. In *Siggraph Technical Sketches and Applications*, page 187, Aug.

Ken Perlin. 1985. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, July.

Ken Perlin. 2001. Noise hardware. In M. Olano, editor, *Real-Time Shading ACM-SIGGRAPH Course Notes*, chapter 2.

Íñigo Quílez. 2002. Domain warping. `http://www.iquilezles.org/www/articles/warp/warp.htm`.

Patricio Gonzalez Vivo and Jen Lowe. 2015. Fractal brownian motion. The Book of Shaders, `https://thebookofshaders.com/13/`.

SpectMorph: Morphing the Timbre of Musical Instruments

Stefan Westerfeld
Freiburg, Germany
stefan@space.twc.de

Abstract

SpectMorph is an open source software which performs morphing of the timbre of musical instruments. This allows creating sounds that smoothly transition from the timbre of one instrument to the timbre of another instrument. There are three steps necessary to obtain the final sound. In the analysis, we use the fourier transform to create models of the spectrum of the input samples. During synthesis a time domain signal can be obtained from these data. An algorithm for morphing the spectral models of multiple instruments is the core of our method. Synthesis and morphing can be done in real-time. After the description of the theoretical background, we provide an overview of the features of the SpectMorph plugin.

Keywords

Morphing, timbre, audio, spectral modelling

1 Introduction

The starting point for SpectMorph[1], our morphing software, are recordings of musical instruments. Typically samples for many different notes per instrument are used, to provide natural sound quality for different notes. From these samples we build spectral models, which are a description of the timbre of each instrument.

Once the analysis data is available, the software can combine the timbre of multiple instruments. A simple use case would be a smooth transition from a pan flute to a trumpet sound. Since we really combine spectral models, this is usually better than crossfading the samples, and does not have the undesirable phase cancellation a direct time domain approach has.

Combining the sounds of instruments can be done in different ways, and support for morphing more than two instruments is implemented. The software has been carefully optimized to allow real-time usage. For Linux, the usual plugin formats, LV2 and VST are supported, as well as a standalone JACK client and a plugin for BEAST. The VST plugin is also available for 64-bit Windows. At the time this paper was written, a port for macOS is being developed, but is not yet ready for end users.

Our goal is that musicians should be able to work with whatever tools they usually use, and the real-time morphing should integrate with these tools.

In addition to this paper, [Westerfeld, 2017] (german) provides a much more detailed description of how SpectMorph works.

2 Analysis of the Samples

In [Serra, 1989] and [Serra and Smith, 1990], the authors present a method called spectral modelling synthesis, which is the theoretical foundation of our analysis step. This produces a spectral model of the sound as sum of a deterministic (sine components) and a stochastic (noise) part.

2.1 Splitting the Signal into Frames

The perceived timbre of samples of musical instruments slowly changes over time. Our goal is to model the structure of the spectrum, as a morphable representation of the timbre. To capture the slow gradual change, the first analysis step is to split our input signals into frames of constant length.

Typically we use overlapping frames of 40ms duration, but for low notes the frames will be longer. If we look at a plot of one single pan flute frame as shown in figure 1, we can see that the signal is almost periodic within these 40ms.

The next analysis step is designed to capture this regularity by representing the frame signal as sum of (periodic) sine functions.

2.2 Modelling the Frame as Sum of Sine Waves

Since our signal is almost periodic, it can almost be represented as a sum of a number of sine

[1] `http://www.spectmorph.org`

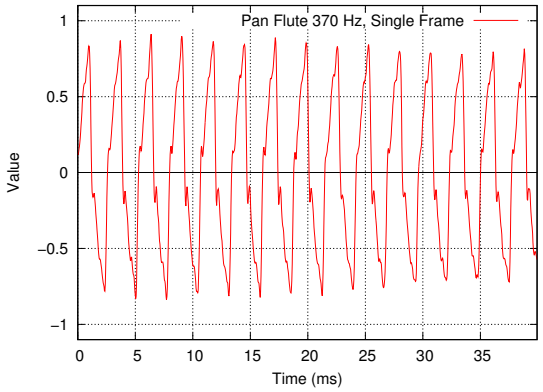


Figure 1: Single analysis frame of the pan flute

waves. In this analysis step, we try to find parameters to decompose the frame signal, called $x(t)$ into a periodic part $d(t)$ and some non-periodic rest $e(t)$.

$$x(t) = d(t)+e(t) = \sum_{p=1}^{P} A_p cos\left(2\pi \frac{F_p}{F_s} + \Phi_p\right)+e(t)$$

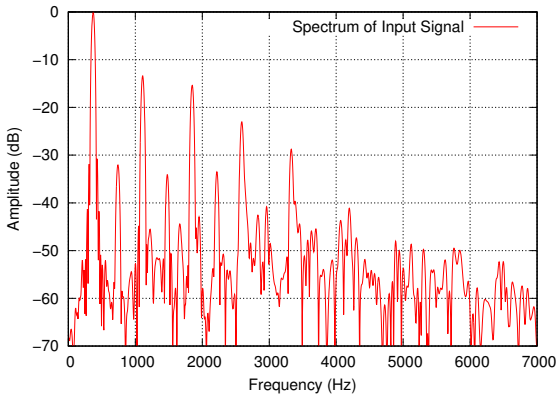


Figure 2: Spectrum of pan flute analysis frame

Figure 2 shows the spectrum of our pan flute analysis frame. Each sine component corresponds to one peak in the spectrum. In other words, if we say that the sound is made up of partials, we want to find the frequency F_p, amplitude A_p and phase Φ_p of each of these partials, and in the next step deal with whatever remains ($e(t)$).

To do this, we compute the (Hann-) windowed fourier transform of each frame signal. We zero-pad the input signal to get a higher frequency resolution, and use a power-of-2 FFT for efficiency reasons. The parameters for frequency, amplitude and phase can be derived by picking the peaks from the spectrum. A peak is a local maximum in the spectrum, however only some peaks are relevant (that is, correspond to a sine component).

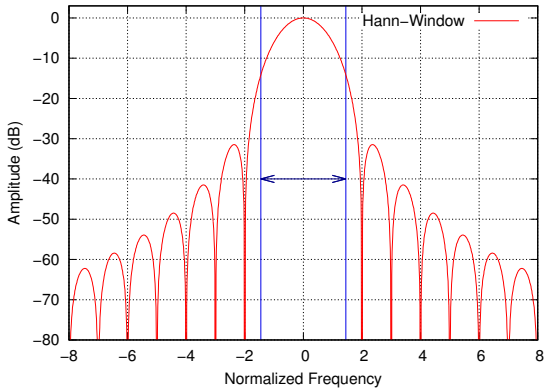


Figure 3: Peak Width and Hann-Window Transform

Since we have multiplied our input data with a hann window before using the FFT, an ideal sine component would look like figure 3 in the spectrum. This would correspond to a peak width of four. In practice a sine component will never be completely ideal, and we also have to consider that multiple sine components added together interfere. Still, we found that checking for a peak width[2] of at least 2.9 provides a good criterion for picking the relevant peaks on the different instrument samples we tested.

As a second (global) criteria we compare the magnitude of the peak with the biggest peak that we found in all frames. If the relative peak magnitude is less than -90 dB, we also consider the peak irrelevant. Note that these two criteria are intentionally chosen too permissive (rather than too strict), because keeping more peaks than necessary will not affect overall sound quality, whereas ignoring too many peaks as irrelevant would.

Finally, each peak corresponds to a sine component with a certain amplitude, frequency and phase, found by interpolation of the three values around the FFT bin with the peak maximum (as for instance described in [Serra, 1989]). At this point, the spectrum of the sum of all sine signals will be very similar to the input spectrum. Figure 4 shows this spectrum. These

[2]The peak width is computed based on how many bins the peak occupies from the local minimum before to the local minimum after the center. This value is normalized relative to frame length, fft size and zeropadding.

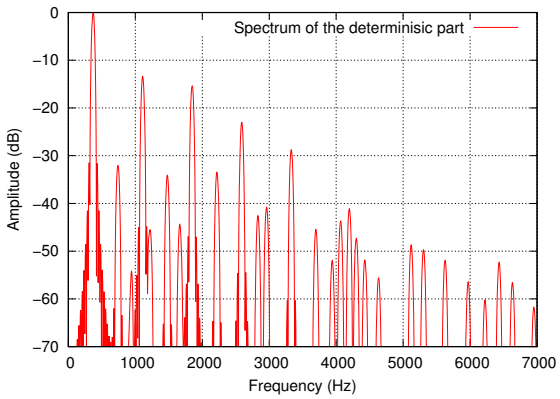


Figure 4: Spectrum of the sum of the sine waves

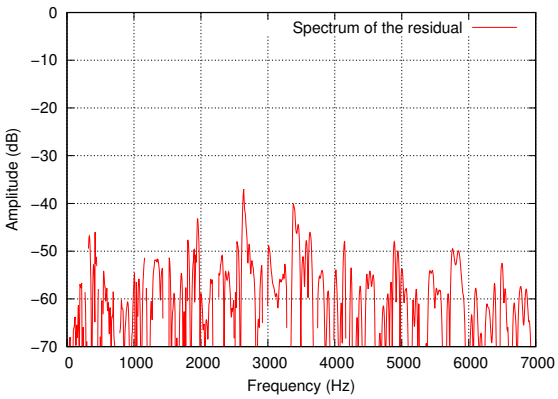


Figure 5: Spectrum of the residual

partials make up most of the sound, and therefore are the most important part of the model of the timbre. So a good answer to the question how a pan flute sounds (at a certain point in time), is: as a sum of a number of sine waves with these frequencies, amplitudes and phases.

2.3 Modelling the Residual

The sine signals usually make up most of the sound, but they cannot represent noisy aspects of the sound. For instance, a flute sample will have some breath or air noise, and a violin has some noise created by the bow. So far, we have only modelled the deterministic part $d(t)$ of the signal. To get the part of the signal that we did not yet describe, we simply subtract the spectrum of the sum of all sine waves from the original spectrum.

If our sine signal $d(t)$ perfectly matches our original signal $x(t)$, nothing would remain. However, if we missed something, the subtracted spectrum will contain just the missing part. For our pan flute frame, the residual spectrum is shown in figure 5.

As we have a model of the periodic part already, we assume that what remains is some kind of noise. So to complete our timbre model, we use 32 perceptually spaced frequency bands and store just the average level of the noise that we find in in each of these bands. The noisy part of each frame is then stored, along with the sine parameters, and provides a spectral model that includes both, $d(t)$ and $e(t)$.

2.4 Issues with Transients

The analysis algorithm described so far produces very good results for many different input signals. However, if the signal contains transients, the quality can be low at the time of the transient. Transients are fast changes in the signal. For instance for a piano attack sound, there is silence, and then suddenly there is a loud signal. This attack happens much faster than the size of one frame. But we only store parameters per frame, so the sharp attack of the original signal gets blurred over one analysis (and later synthesis) frame. The piano resynthesis will have a much softer attack than the original.

So far, we have not found a good strategy for handling transients. As a brief example consider the following method: do analysis as mentioned before, but for frames with transients, keep original sample data. While this is not too complicated to implement, and while this definitely will improve the quality (and preserve the sharp attack of a piano), the problem is that for these frames we would not be able to do proper morphing, as the description of the signal is no longer parametric in a form that allows combining multiple input signals.

Ideally, we would have an analysis strategy that preserves transients, but in a way that still allows morphing. Fortunately, even without special casing transients, there are many instrument sounds that only change slowly so that there are no quality issues caused by transients.

3 Synthesis

The goal of the synthesis is to compute a time domain signal from a sequence of spectral models. These spectral models consist of a set of sine frequencies, amplitudes and phases, and 32 noise bands. Similar to the analysis step, synthesis takes place in synthesis frames, which are overlapping, and added together to produce a

time domain signal.

3.1 Additive Synthesis and Inverse FFT

Since we want to use the synthesis in real-time, performance is important. Although we could theoretically simply add up all sine waves of each frame, to get $d(t)$, this could easily result in 100 or more sine computations and additions per output sample value. Instead, we compute the spectrum of the frame by adding one peak per sine component, and then use an inverse FFT, as described in [Rodet and Depalle, 1992].

The computation of the noise part of the output consists in setting up a spectrum of suitably chosen random values according to the 32 perceptual bands, and performing an inverse FFT. In SpectMorph, the sine and noise part are computed together, so we only need one single inverse FFT per synthesis frame.

3.2 Reconstruction of the Phase

Before, we used frequencies F_p, amplitudes A_p and phases Φ_p to describe the sine components that are part of one analysis frame. A more or less technical detail is that we can (and have to) avoid using the phase Φ_p completely during synthesis. This is also described in [McAulay and Quatieri, 1984] and [Serra, 1989].

To compute a phase value for a sine component that is to be synthesized in the current synthesis frame, we look at the last synthesis frame. If a component with similar frequency[3] can be found in the last synthesis frame, then the phase in this synthesis frame is chosen to continue the sine component of the previous frame. This avoids interference and possible cancellation of sine components of adjacent synthesis frames, while only using F_p and A_p for synthesis.

Any sine component in the current frame that was not found in the last synthesis frame starts with a phase of zero.

4 Morphing

4.1 Input and Output Parameters

Once we have transformed our samples to spectral models during analysis, the input for the morphing algorithm is the description of two spectral models, each with the parameters shown in table 1. The two input frames are from two sources, source A and source B, so we use superscript α and β for the parameters, for instance F_1^α (first frequency of source A) or A_1^β (first amplitude of source B).

	Parameters
Frequencies	$F_1, \ldots, F_P$
Amplitudes	$A_1, \ldots, A_P$
Noisebands	$NOISE_0, \ldots, NOISE_{31}$

Table 1: Spectral Model Parameters for one Frame

From this input, the morphing stage should produce one single spectral model with frequencies, amplitudes and noise band values. A parameter $\lambda \in [0, 1]$ controls the morphing. A value of $\lambda = 0$ means that only source A is audible, $\lambda = 1$ means that only source B is audible, $\lambda = 0.5$ corresponds to a 50%/50% mix, and so forth.

4.2 The Stochastic Part

Since the computation of the stochastic part (noise part of the signal) is simple, we start with this. The 32 noise band parameters of the output can be computed as

$$NOISE_b = (1 - \lambda) \cdot NOISE_b^\alpha + \lambda \cdot NOISE_b^\beta, \qquad \text{for } b \in [0, 31]$$

For $\lambda = 0$, only the noise component of source A is used. For $\lambda = 1$, only the noise component of source B is used. If λ is between 0 and 1, the amplitude of the corresponding noise bands is interpolated linearly.

4.3 Matching corresponding Partials

Figure 6 is one example for the positions of the partials of a frame from source A and a frame from source B. To be able to perform the morphing, the first step is to find matching components. If partials match, they are assigned to each other. However each entry is at most used once, no partial is assigned to more than one entry.

To get good results, our algorithm starts with the louder partials. Since they will be clearly audible in the output, it is important that they get a close match.

We also use a frequency similarity criteria. Let G be the fundamental frequency of the note, we ensure that

$$\delta = |F_q^\beta - F_p^\alpha| \leq \frac{G}{2}$$

[3] We consider two frequencies to be similar, if the frequency difference is less than than 5%.

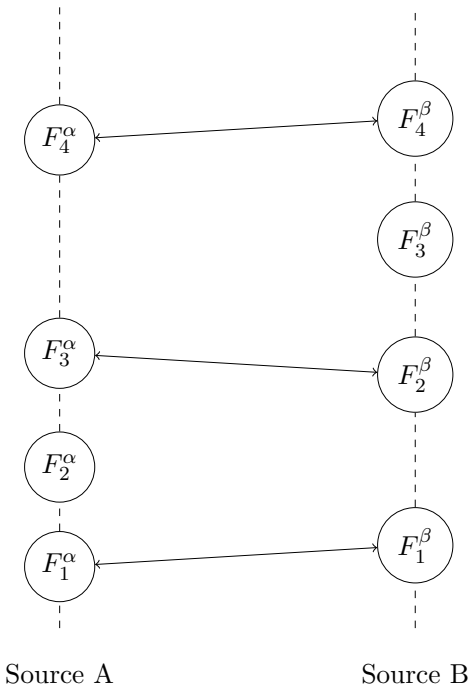


Figure 6: Matching Partials of the two input frames from Source A and Source B

which means that partials can only be assigned if they are closer to each other than half the fundamental frequency.

At the end of this stage, some partials are assigned to each other (like F_1^α and F_1^β), whereas other partials remain without a matching frequency in the other frame (like F_2^α).

4.4 Computing the Amplitudes

Once we have assigned the partials of the frames from source A and source B to each other, the output amplitudes can be found using

$$A = (1 - \lambda) \cdot A_p^\alpha + \lambda \cdot A_q^\beta$$

for partials p and q which have been assigned in the previous step.

For partials that remain without matching entry in the other frame, we simply use zero as amplitude for the interpolation.

We also implement an alternative way of dealing with amplitudes, which should be closer to how human loudness perception works: dB-linear amplitude interpolation. To do this, the amplitudes are converted to dB before the interpolation step, and converted back afterwards.

4.5 Computing the Frequencies

After the previous description of how noise band parameters and amplitudes are computed, the first idea would be to use the same strategy here, so

$$F = (1 - \lambda) \cdot F_p^\alpha + \lambda \cdot F_q^\beta$$

and keep the frequency exactly as it is for partials that have not been assigned.

However, this leads to one undesirable effect: for partials from the analysis stage that are not very loud, it does not matter much if their frequency is wrong. They are inaudible anyway.

If such a partial gets assigned to a very loud partial, the output frequency can easily get wrong, for if for instance $\lambda = 0.5$, half of the frequency output value F is determined by the almost inaudible partial.

So in practice, if partials do not have the same volume, we ensure that the louder partial has more influence on the output frequency.

Let A_p^α be the louder partial ($A_p^\alpha \geq A_q^\beta$), then we use as frequency:

$$F = F_p^\alpha + m\lambda(F_q^\beta - F_p^\alpha)$$

where m is a factor that depends on both amplitudes:

$$m = \frac{A_q^\beta}{A_p^\alpha}$$

If both amplitudes are equal, so that $m = 1$, we get the same result as the approach at the start of the section.

$$F = F_p^\alpha + 1\lambda(F_q^\beta - F_p^\alpha) = (1 - \lambda)F_p^\alpha + \lambda F_q^\beta$$

If one amplitude is a lot louder than the other, we have $m \approx 0$:

$$F \approx F_p^\alpha + 0\lambda(F_q^\beta - F_p^\alpha) = F_p^\alpha$$

So if a stable (louder) partial is combined with an almost inaudible partial, the factor m will ensure that the louder partial almost completely determines the frequency.

4.6 Grid Morphing

For grid morphing, instruments are placed on grid points of an WxH (width W, height H) grid. The simple case is that we have a $2x2$ grid, with four instruments A, B, C and D. In this setup, we now have two control parameters that correspond to the X- and Y-position on the plane.

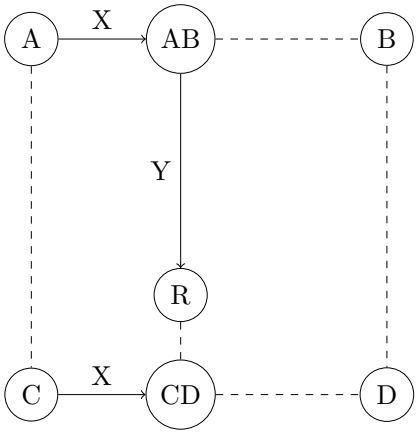


Figure 7: Grid Morphing of four Instruments

Our job is to compute a resulting output sound R as result of setting our X- and Y-position. This is shown in figure 7. To compute R we proceed as follows: as a first step we combine instrument A and B with the control value X, which produces sound AB. Then we combine instrument C and D, again with control value X to get CD.

As last step we can combine AB and CD with the control value Y to R. To summarize this algorithm: we can morph of four instruments on a plane using the morphing of two input frames, which we already described. To do it, we simply use this algorithm three times.

5 Plugin Features

In the last sections we've given the theoretical background how SpectMorph works internally. We'll now summarize some of the relevant topics for end users, which will usually use one of the SpectMorph plugins, LV2, VST or BEAST (or the JACK client).

5.1 Standard Instrument Set

As we've seen, SpectMorph itself is based on sample data. It can morph instrument sounds, but only after an instrument has been described as a set of samples, and analysis of these samples was performed. Typically we need one sample per semi-tone, or at least one sample every few semi-tones, in order to produce good reproduction quality. The tools required to build instruments from samples are distributed as a part of SpectMorph. However, it is a bit of work to create your own instruments.

To address this issue, SpectMorph currently ships with 14 ready-to-use instruments, like trumpet, oboe, pan-flute, saxophone and so on. All of the samples we used were free. Many were taken from the *Iowa Musical Instruments Samples*[4], we also recorded some samples ourselves, and added some instruments from the *Fluid R3 SoundFont*[5].

5.2 Using Morphing

The user interface of SpectMorph supports the two use cases mentioned before. The simple case involves combining two instruments using morphing, in the UI this is called "Linear Morph". For a linear morph, the user can choose the instruments from a list of instruments, usually from the standard instrument set. In the simplest case, an UI slider is used to control the morphing, so dragging the slider will gradually change the sound from the first to the second instrument.

Grid morphing as mentioned previously is also supported, which allows using an X/Y control pad to control the position on the grid with the mouse.

5.3 Automation / Control

If users create music using the plugin in sequencers, it is often desirable to exactly specify how the morphing should be performed, alongside with the notes to be played (rather than controlling the morphing with the mouse like it could be done during live performances). So we support automating the control value, so that the timbre can be controlled by the sequencer. For X/Y morphing, two control values can be used, to automate the position on the plane.

Besides these possibilities, SpectMorph implements an LFO operator (low frequency oscillator), which will change the control value periodically. This feature is also useful for live performances, to get interesting slowly changing sounds.

5.4 After Morphing

So far, we've described how a sound is produced, usually combining two or more standard instruments using some control values. This can be used as it is, or be modified with some optional additional steps before the output sound is generated.

One refinement is the unison effect, which adds up a few detuned copies of the spectral

[4] `http://theremin.music.uiowa.edu/MIS.html` - public domain license

[5] packaged by many linux distributions, for instance ubuntu: `https://launchpad.net/ubuntu/xenial/+package/fluid-soundfont-gm` - MIT license

model of the sound. This makes the sound more fat, and can be seen as imitation of multiple musicians playing the same notes using the same instrument.

Another refinement is using an ADSR envelope, which adds a custom volume envelope, replacing the natural volume envelope the sound has. Finally we implemented support for portamento and vibrato.

Although these optional post-morphing operations, that modify the output sound, can produce interesting possibilities, it is also obvious that using such post-morphing refinements has a cost: the sound will no longer be as natural as possible. For instance, giving a trumpet a quick exponential volume decay is a new variant of the sound, but it will sound less like a trumpet.

6 Conclusions

The algorithms presented in this paper, implemented in SpectMorph, produce realistic reproduction of instruments, based on building spectral models of samples. For many musical instruments, such as bassoon, trumpet, saxophone, oboe and so forth, the quality of the analysis step will be very high.

There are however some cases, in which the spectral modelling approach does not work well. Whenever the peaks in the spectrum are too close, the peak finding algorithm will not work properly. While spectral models for natural sounds usually provide good quality, typical synthetic sounds, such as a synthetic saw waves with unison cannot be analyzed properly.

We already discussed that there are issues with transients, such as the sharp attack of a piano, in section 2.4. For all sounds where the analysis step provides good quality, the morphing steps described in section 4 provide realistic transitions between the timbre of the instruments. This provides composers with a way of creating sounds that is not available with samplers or synthesizers.

In SpectMorph, much care has been taken to ensure not only good quality of the sounds, but also fast computation. Morphing and synthesis are reasonably fast so that high polyphony is available during real-time usage.

The SpectMorph LV2/VST/BEAST plugin (and the JACK client) supports creating new sounds by morphing existing ones, and includes many standard instruments. The plugin integrates into whatever sequencer or live performance environment the composer wants to use, and provides flexible ways of controlling the morphing parameters, as well as post-morphing refinements.

References

Robert J. McAulay and Thomas F. Quatieri. 1984. Magnitude-only reconstruction using a sinusoidal speech model. In *Proceedings IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 27.6.1–27.6.4.

X. Rodet and P. Depalle. 1992. Spectral envelopes and inverse FFT synthesis. In *Audio Engineering Society Convention 93*.

Xavier Serra and Julius O. Smith. 1990. Spectral modeling synthesis: A sound analysis/synthesis system based on a deterministic plus stochastic decomposition. *Computer Music Journal*, 14(4):12–24.

Xavier Serra. 1989. A system for sound analysis/transformation/synthesis based on a deterministic plus stochastic decomposition. Dissertation STAN-M-58, Center for Computer Research in Music and Acoustics, Stanford University.

Stefan Westerfeld. 2017. Morphing der Klangfarbe von Musikinstrumenten durch Spektralmodellierung. `http://edoc.sub.uni-hamburg.de/informatik/volltexte/2018/236/`.

RSVP, a preset system solution for Pure Data

José Rafael SUBIA VALDEZ
Edinburgh University
Alison House
Edinburgh
EH8 9DF
Scotland
Rafael.Subia@ed.ac.uk

Abstract

This paper describes the logic and process behind the development of the RSVP preset library for the Pure Data programming environment. The library aims to tackle the lack of a native preset system in Pure Data. Projects like Kollabs[1], CREAM[2], sssad[3] and others, have produced different solutions for this issue. However, after experimenting with these, it became clear that a different approach was required to fit personal needs. This led to the creation of the RSVP library which will be described in detail. During the development of this project, a feature request for PD was identified, and that will also be shared here. This paper will offer a detailed description of how the system works, but will not go into extensive Pure Data patch descriptions. Instead it will focus on how the code is structured and will describe how the system functions with the users' own projects.

Keywords

state-saving, GUI, interpolation, external, abstraction

1 Introduction

The flexibility that Pure Data [Puckette, 1996] has as a programming environment is immense; the fact that a *Graphical User Interface* or "GUI" is part off its workflow concept is very interesting as a programming language. Pure Data, and its "prettier sibling", MAX [Zicarelli, 1990], allow users to program in a different style than Supercollider [McCartney, 1996] or ChucK [Wang and Cook, 2002] to name a few. PD incorporates the idea of "connection" that is well known among musicians with stage experience. However, it does not contain an easy and rapid preset mechanism such as the one found in MAX. Having to tackle this issue led to the development of other ways of interacting with a patch. Consequently, this required the use of some interesting tricks to overcome the lack of this particular feature. Nevertheless, a preset system is very helpful for musical purposes even if not used extensively.

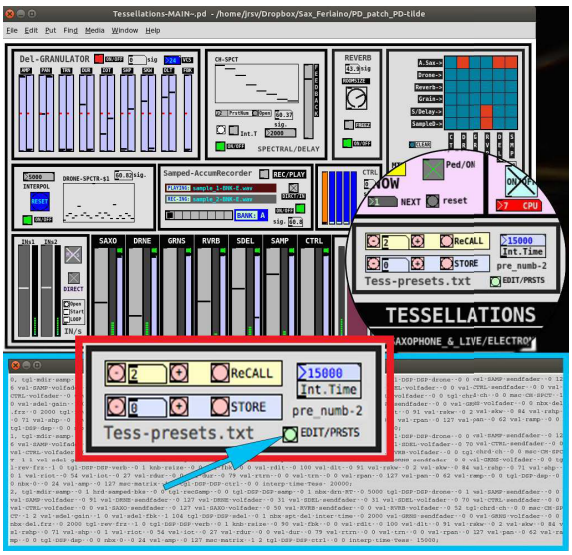


Figure 1: Patch of *Tessellations* for alto sax and computer, developed with RSVP

Over the years, different techniques implemented by other users were tested and incorporated in complex patches produced for personal use. There are some complete and powerful libraries like Kollabs [Weger, 2014], that allow different types of interpolation between current and to-recall values. The CREAM [Guillot, 2014] library and its GUI programmed as *externals*, also offers interesting interpolations, even working with its *c.breakpoints*[4] object. The design is very similar to the one in MAX, including the commands on the *c.preset* such as "shift + mouse-click" to *save* a preset and "mouse-click" to *recall* it.

[1] `https://github.com/m---w/kollabs`
[2] `https://github.com/CICM/CreamLibrary`
[3] `http://puredata.info/downloads/sssad`
[4] c.breakpoints is a GUI external that allows the creation of different breakpoint functions

However, the method developed by *rodrigo@anorg.net*[5] was the closest to the type of preset management envisioned. This solution used the *pool*[6] object to recall data into the patch. Although it had no interpolation methods, and needed to be used in "looped" (see Fig. 2) connection with the GUI, its structure was a great starting point for this project. Still, these discoveries were never completely adequate, as they are workarounds to a problem that ideally should be solved in the source code itself.

Moreover, the testing and experimentation of the different solutions was done when implementing them in specific projects. The judgement made on each was based entirely on particular situations. This means that these libraries could be better implemented or may run "smoother" if the programming had been done on a faster system or with more time available. Issues based on installation and implementation, fast editing as well as CPU or GPU consumption in the equipment available, played an important part on the amount of usage they received. Consequently missing key features were identified and it was decided that a new and custom solution was required. This resulted in the creation of the RSVP library.

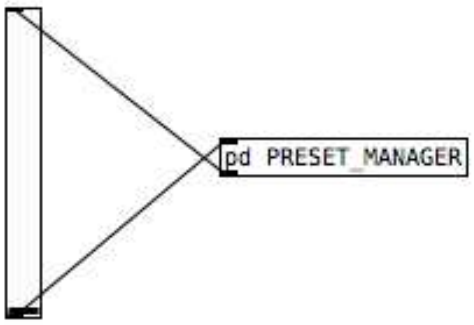


Figure 2: looped connection that some preset managers offer

2 How it works

The main idea when designing RSVP was to develop a "light and flexible (as possible)" library to meet general needs. The library had to easily be incorporated in projects by avoiding the "looped" connections (see Fig. 2), it had to implement a way to edit presets from within the patch and include a basic interpolation method between values. Another goal was to include a single click call and recall strategy with a simplified interface. This way the user would not have to struggle with loading and naming files, as well as opening n number of subpatches of settings. To accomplish this, the project was divided in three main parts:

- GUI/single click saving
- Rapid patching
- Managing the presets with automatic creation/loading of files containing the data

2.1 GUI/single click saving

The design of GUI objects, which contain the ability to store and recall presets, must be based on the easy creation of the objects and easy recall of the presets. Thus it was decided to link the Data and the GUI, as opposed to Kollabs, which is based on the principle that separates the GUI from the data processing [Weger, 2014]. A mechanism of state saving based on native vanilla GUI objects was programmed by creating a wrapper around these. The wrapper would save the state of a variable when it received a global "save preset" type message or bang that would register the value into a *coll* [7] object with the unique ID of the abstraction that generated it. This would simplify the recording of the data by the values inside *coll*, and allow easy recalling of the values by routing it to the abstraction based on an "ID" given when created.

2.2 Unique ID (keep score of data with iemguts)

A fundamental component for the development of RSVP was the iemguts[8] library and the *stat*[9] object. After experimenting with *dollarsign-zero*[10] to create unique IDs, it was understood that *dollarsign-zero* number is only unique for each session; once the file is closed and opened again, that unique number changes. Consequently, it would make the already saved data useless if saved in a previous session. Using the *canvasname* external of the iemguts library, allows the query of window names and arguments,

[5] only remaining information found on the author

[6] `https://grrrr.org/research/software/pool/`

[7] `https://puredata.info/downloads/cyclone`

[8] `https://puredata.info/downloads/iemguts`

[9] `https://puredata.info/downloads/hcs`

[10] A mechanism to create a unique ID inside Pure Data to help with the creation of abstractions

and enables the creation of unique IDs to save the data. The *canvasname* external provides the name of the parent patch, and by using *canvasargs* and providing the necessary information, it is possible to have multiple instances called (see Fig. 3). On the other hand, the *canvasindex* external provides a way to keep count of the number of instances. This is crucial for the deletion of GUI abstractions and the synchronisation of all the Data to be stored inside the *coll*.

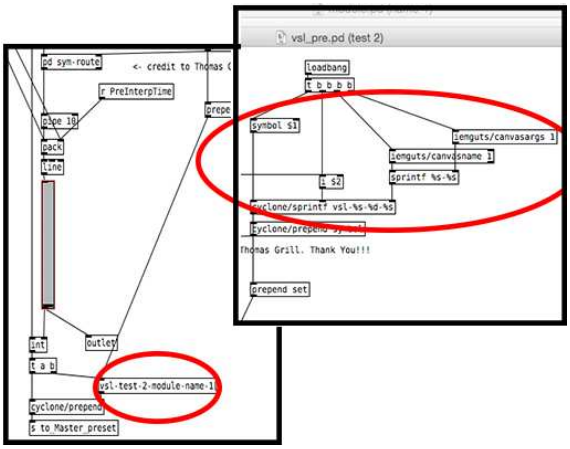


Figure 3: use of *canvasname* & *canvasargs* in RSVP

The system works by adding the name of the parent patch to the initial unique name created, either by hand or using the *GUI-creator* that comes with RSVP. If there is no parent patch, a unique name is expected but not necessary, although that code, if no unique name is provided, will not work correctly if multiple instances of it are used in the same patch. The unique name (using dollarsign number/local variable) allows the correct use of multiple copies of code in a patch. By using this standard method to handle the appointment of IDs, RSVP allows flexibility to use presets in different ways, including nesting or multiple instances.

2.3 Track of Instances and Deletions

One of the biggest challenges of creating a library that records states, is to correctly map values if an instance of the object recorded is deleted. Many basic preset solutions made with Pd involve the use of an array to record values. Although this is a very fast and efficient way to build a preset system, this method does not take object deletion into account. When objects are deleted, the array is resorted and those deleted are "popped" out, causing the values in the array to shift their positions thus corrupting the data. In order to solve this, it was decided to implement a way in which all values are recorded every time a preset is saved. Consequently, the list of values in the *coll* object with values of the new object IDs created in the patch is updated, without any that may have been deleted.

RSVP builds a unique message in Pure Data -with all the variables recorded- that is later pushed into a preset slot in the *coll* object. This is achieved by using the "add2" message to a blank message box that receives all the values when the "SaveMaster" message sent from the *PresetManager* abstraction is received in each RSVP object. The challenge in this part of the program is to know when to push the message into the *coll*. To accomplish this, the *canvasindex* from iemguts library is extremely important. The object keeps track of all the number of RSVP abstractions being used. In this way, the patch knows when all values have reached the empty message as its length must be double the number of instances. Consequently, the message can be pushed into the *coll* object in the *PresetManager*. The presets are saved in a text file in the same directory where the preset manager is being called from. It creates a text file with the SUFFIX "-preset" added.

2.4 Recalling

Every GUI abstraction accepts only values that correspond to the ID tag given for the preset. The *sym-route*[11] abstraction routes messages like the built-in *route* object, but accepts *symbols* instead of *floats* as the *type* of data to process. This abstraction receives each pair of values from the *coll* object when a preset is recalled. The value is routed properly when the ID values match and advances to the interpolation stage achieved with the *line* object.

3 GUI Abstractions

The RSVP library mirrors all vanilla GUI objects plus the *breakpoints*[12] and the *knob*[13] externals. With this selection, most needs of a typical simple patch are covered. Every abstraction has the functionalities of the original "wrapped" object plus an interpolation time that will be sent from the PresetManager. Their names also try to resemble the object native to

[11] This abstraction was coded by Thomas Grill
[12] Part of the tof library
[13] Part of the flatgui library

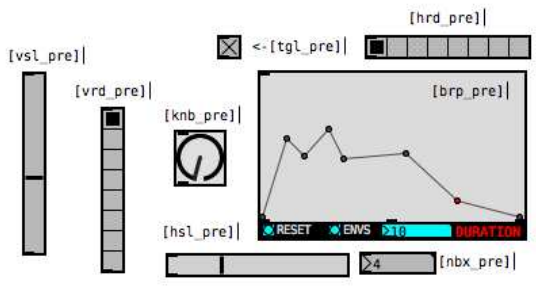


Figure 4: GUIs available in RSVP

vanilla in order to create the objects more easily (see Fig. 4). The suffix "_pre" to the available objects creates the wrapped version with the exception of the breakpoints and knob object, which are part of other libraries and are abbreviated to brp_pre and knb_pre respectively.

3.1 The Breakpoints Abstraction

The breakpoints abstraction brought some complications to the saving and recalling technique that was being implemented. While the *PresetManager* handles pairs of messages formed by the abstraction's unique ID and the value, the breakpoints object allows the creation of an envelope with a list of values. By adding a *coll* object to the brp_pre abstraction, lists can be stored and saved independently thus allowing the storage and recalling of objects that use more than one value The values of the internal *coll* are stored in a different text file with the file extension ".brp".

3.2 The Miscellaneous Abstraction

In addition to the GUIs offered, RSVP includes a "msc_pre" abstraction which can be used to save different values in a sub preset and be recalled by the PresetManager. This abstraction allows the use of RSVP to write other types of data in case the native RSVP abstractions cannot fulfil certain needs. The msc_pre abstraction was initially created to store variable amounts of points of the breakpoints abstraction explained above. It was later duplicated as an independent abstraction to offer a way of recalling data in objects not native to RSVP. The "msc_pre" abstraction creates an additional textfile, with the file extension ".msc", that records the presets assigned specifically to this abstraction.

The abstraction is linked to the *PresetManager* by receiving the number of the internal preset to recall, in the same way as the brp_pre abstraction. The main purpose for the creation of this abstraction is offer the possibility of a modular preset systems, but it also allows the use of the library with other abstractions or externals from different developers. In the example (see Fig. 5), the msc_pre object is used with the *matrixctrl*[14] object. Different ways of using the library with the msc_pre abstraction and a new "local" feature, are currently being tested and are discussed further on this paper.

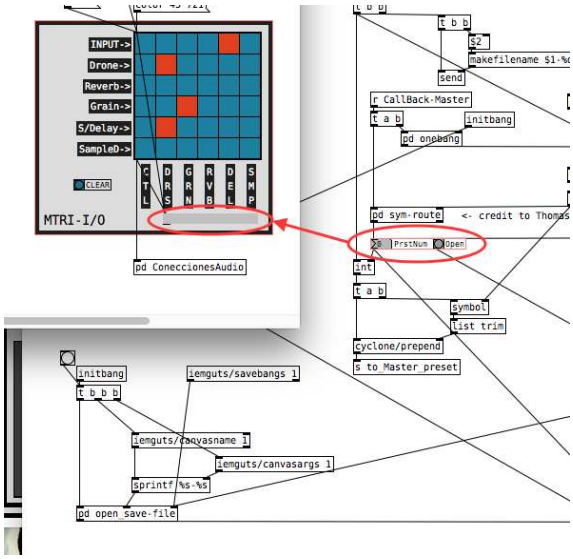


Figure 5: RSVP msc_pre object working with jmmmp's *matrixctrl*

4 Usage

4.1 Rapid Patching with the Help of the *GUI-creator* Abstraction

Initially, the intention was to hack Pd's Tcl/TK frontend and link the "put" action of the main menu to the creation of every GUI that comes with RSVP. Eventually, it was concluded that developing a Dynamic Patching abstraction named *GUI-creator*, would be the best solution[15] to develop the idea quickly (the initial idea is still being researched with the use of the Tcl/Tk plugin API).

The abstraction creates the RSVP GUIs with the click of a button. It takes care of the sequential SUFFIX that is entered for the unique ID and allows for the quick creation of abstractions

[14] `puredata.info/downloads/jmmmp`

[15] I decided to wait until having good results once RSVP was finished to start thinking on further developments.

if developing something that needs a matrix of knobs or toggles[16], for example.

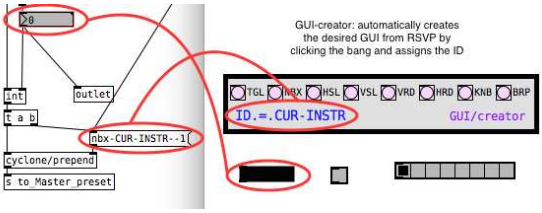


Figure 6: *GUI-creator* assigns the ID and increments the SUFFIX as it creates objects

An interesting problem surfaced while developing the *GUI-creator* abstraction. In situations when the user deletes the *GUI-creator* (it is intended to be deleted after use), and for some reason needs to add more RSVP objects with it. The *GUI-creator* first queries how many instances of that object already had been created previously and then continues to increment the SUFFIX number from that value on (see Fig. 6). To provide the abstraction with this number, the same abstraction creates a text file and stores the number and type of RSVP abstractions it creates. This record is used to initiate a new count, starting from this number plus one, when a new instance of *GUI-creator* is called.

The *GUI-creator* abstraction, only needs to keep count of instances created. If the number of instances recorded is different than the number present in the patch, then an instance or instances of RSVP abstractions were created but later deleted. In this case the value in the SUFFIX is more than the true number of RSVP objects used. However, RSVP uses that number as the SUFFIX of the ID allowing an infinite amount of unique IDs by incrementing on the previous known total.

RSVP works by keeping count only of instances created and values saved the moment a preset is recorded. The library was developed around the idea of how to discard the data that becomes obsolete when a preset is rewritten. RSVP takes care of this by using the destructive editing feature of the *coll* object to purge obsolete data. Furthermore, if obsolete data exists because *coll* has not been updated and this data is recalled, then the values are not processed as there are no instances of the sym-route abstraction linked to that ID.

[16]Video Examples: `http://www.jrsv.net/pure-data-preset-system`

4.2 *PresetManager*

The *PresetManager* is the module that controls the state saving and the value recalling of the stored data. The module consists of two main parts that take care of the saving and recalling of the values. The abstraction contains a GUI with visual feedback when an action is taken. It also allows the recording and recalling of any given position in the *coll* and contains the interpolation time control in a number box. Finally, the patch also allows the user to display the values for fast queries and/or editing in a "popup" window (see Fig. 7),. The *PresetManager* will save the contents of the *coll* object every time the patch is saved.

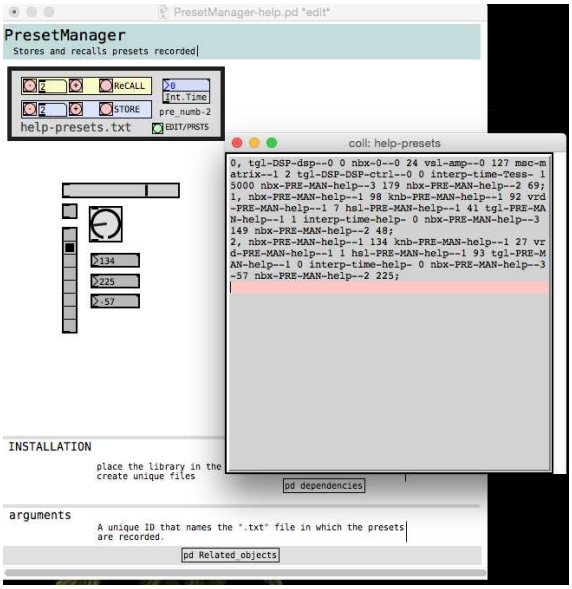


Figure 7: PresetManager help file with the popup winddow displaying the presets

4.3 Dealing with Multiple Instances

The user can call multiple instances of projects using RSVP abstractions by allowing the creation of an instance ID. This works following the way that Pd uses \$0 and \$*n* to create local and global variables. This feature was created in case the user needs two modules that use RSVP in the same master patch. A similar type of use can be observed in projects like *Automatonism*[Eriksson, 2017] or *Context*[Goodacre, 2017] that allow the creation of modular instruments to connect as desired in a patch. RSVP takes care of this by allowing an argument to set a name on creation.

4.4 Customizing RSVP

The current version of RSVP works as a local library inside a project. This means that the folder containing RSVP should be copied and placed in the main directory of the file being used as a main patch. The reason for this is that the library uses GUI objects that alter the source code of the abstractions if modified; causing the GUI to change for all files calling the library. For this reason, RSVP is intended to work as a local library letting the users customize the abstractions source code with the colors and sizes set for each specific project.

The flexibility that RSVP offers is based on the ability to modify the graphical properties of the abstractions. The modifications are as extensive as what a user can modify to the wrapped GUI objects. Extending the objects available is as easy as duplicating the source file of the GUI and using it as a template to be modified. The user can then apply all changes and save the personalised abstraction under a custom name or keep the changes to the original. If the object uses a new name, it will still be compatible with the RSVP preset system when called.

4.5 Modularity: implementation of "Local Presets"

The RSVP library offers different ways to have local presets stored in modular projects. This provides added flexibility as RSVP can be used with the user's own GUI design. With the use of msc_pre abstraction, it is easy to achieve any type of modular state saving. To make this user friendly, a "local" method inside msc_pre and brp_pre (see Fig. 8) was programmed thus giving a straight forward way of using RSVP like this. The objects can receive a message with a "local" flag and the values 1 or 0 to turn on/off the ability to read and write the values of their local *coll* object. This means that the user can program modules using their own GUIs and write their local presets such as timbres of a synthesizer in a msc_pre that will not be controlled by the *PresetManager* in the parent patch.

5 What is next? ...the "to-do" list

The RSVP library was created in such a way that improvements could be made later according to structural areas of the system. Different ideas are already being tested, including the possible switch from native GUI to native lookalikes done with Data Structures. This move to

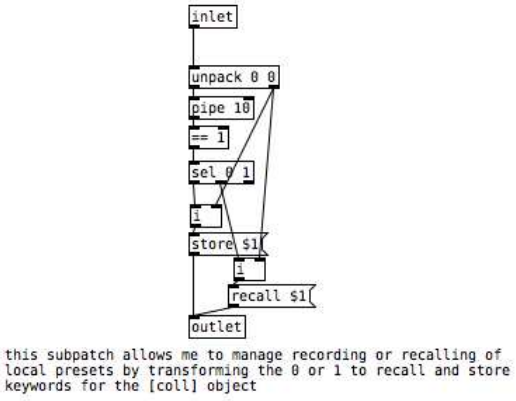


Figure 8: part of the patch that allows the recalling and recording of local presets

data structures might help develop a GUI with design changes and more importantly, have the data stored in the data structure itself.

Additional modifications are being considered to parts of the library in charge of the data storing. The implementation of the *text* object instead of the *coll* external in RSVP would make the library close to being vanilla-friendly.

The last improvement currently considered for the RSVP library is the implementation of the *propertybang* object of the iemguts library. By using this object, the library could be modified to run as a global library and be installed as any other offered, making the customization of the GUI easier to implement.

6 Envisaging a META section in the ".pd" file

During development of RSVP, it became evident that having the information stored as simple text is very useful. Future development is trying to use less files to store and recall information consequently centralizing everything into a single file. Other ideas include saving data inside the ".pd" file itself. This way the RSVP data could be accessed using the *text* object within the patch. Being able to store in a META section, opens additional possibilities for Pd users. Information such as credits, licensing and state saving could be stored with the patch. Other options may include building an abstraction by running a patch or a place to store scripts for externals like py/pyext, pdlua or pdlisp.

It is possible to read the Pd file as text inside Pd. Unfortunately, when Pd loads a file

with extra information on it, it produces warnings every time text that is not part of a normal Pd file is found. This means that any information saved in the patch as simple text to the ".pd" file is ignored. Using a simple text editor it is easy to write anything in the file and successfully save it, however when the same file is opened again in Pd, it will produce warnings and ignore that information if saved from Pd. This is why a META section for the file could be implemented. One solution could have an "EOF" (End of File) to stop Pd from reading the information that is stored in a META section.

The META section of the file would not be read by Pd when loading. It would instead be accessed as a plain text file inside Pd, with a *text* object and a message/method META sent to one of its inlets. Once the META section is accessed, the user could read it line by line and modify the information retrieved as needed. This feature would allow projects like RSVP and others to extend Pd to fulfil other needs easily.

7 Conclusion

The RSVP library offers a rapid way of saving different states in Pure Data. The design of the library was based on other solutions, but there was a desire to simplify and modify them to better solve various needs in different musical projects. RSVP offers a number of tools that help with the creation and performance of Pd patches and intends to have a solid, simple and fast way of managing presets.

The advantage of using a wrapper in its design allows the system to be easily modified and used in other versions of Pd. It can improve tools that other flavors of Pure Data have and gives space for quick development in modular areas of the system.

While this project was an attempt to solve something that Pd has been missing, it also proved that Pure Data is a flexible system that can take care of complex programming challenges like the one described. Nevertheless the library still lacks key feayures such as different types of interpollations and vanilla *friendlier* code.

Unfortunatelly, RSVP cancells usefull functionalities found in the "Properties" menu of its native GUI Objects. It is crucial for further development to address these setbacks and reinstate this functionalities *via* the wrapper. Finally, the code needs to be optimized as there are some processes that could be done more efficiently.

The RSVP library is in constant change and currently in an *Alpha Stage*. Anyone is welcome to download and use it in projects, but the developing changes according to user experience. Because RSVP is a local library, backwards compatibility is not a serious problem but more testing is needed to offer proper support. The library can be found in the URL address:

- `https://github.com/JRSV/RSVP`

Some videos introducing its features are hosted on the following website

- `http://www.jrsv.net/pure-data-preset-system`

8 Acknowledgements

RSVP uses third party libraries developed by people from the community. I wish to acknowledge the developers of libraries used in this project and individuals that helped answer questions on the pd mailing list and social media, including Matt Barber, Liam Goodacre and Thomas Grill. RSVP uses the following libraries that can be downloaded and added through the "deken" manager:

- HCS
- iemguts
- iemlib
- flatgui
- cyclone
- tof
- zexy

RSVP was developed thanks to the support of the University of Edinburgh in Scotland during the course of my PhD. I wish to thank my supervisor Dr. Michael Edwards for advice and guidance.

References

Johan Eriksson. 2017. Automatonism. `https://www.automatonism.com/the-software/`.

Liam Goodacre. 2017. Context Sequencer. `https://contextsequencer.wordpress.com`.

Pierre Guillot. 2014. CreamLibrary: A set of PD externals for those who like vanilla... but also want some chocolate, coffee or caramel. https://github.com/CICM/CreamLibrary.

James McCartney. 1996. SuperCollider SuperCollider. https://supercollider.github.io/.

Miller Puckette. 1996. Software by Miller Puckette. http://msp.ucsd.edu/software.html.

Ge Wang and Perry Cook. 2002. ChucK => Strongly-timed, On-the-fly Music Programming Language. url-http://chuck.cs.princeton.edu/.

Marian Weger. 2014. Kollabs / DS - a state-saving system with scene morphing functionality for Pure Data. https://iem.kug.ac.at/fileadmin/media/iem/projects/2014/weger.pdf.

David Zicarelli. 1990. Max Software Tools for Media | Cycling '74. https://cycling74.com/products/max/.

Open Hardware Multichannel Sound Interface for Hearing Aid Research on BeagleBone Black with openMHA: Cape4all

Tobias Herzke[1,4] and **Hendrik Kayser**[1,2,4] and **Christopher Seifert**[3,4] and **Paul Maanen**[1,4] and **Christopher Obbard**[5] and **Guillermo Payá-Vayá**[3,4] and **Holger Blume**[3,4] and **Volker Hohmann**[1,2,4]
[1]HörTech gGmbH, Marie-Curie-Str. 2, D-26129 Oldenburg, Germany
[2]Medical Physics, Carl von Ossietzky Universität Oldenburg, D-26111 Oldenburg, Germany
[3]Institute of Microelectronic Systems, Leibniz Universität, D-30176 Hannover, Germany
[4]Cluster of Excellence "Hearing4all"
[5]64 Studio Ltd, Isle of Wight, UK
info@openmha.org

Abstract

The paper describes a new multichannel sound interface for the BeagleBone Black, *Cape4all*. The sound interface has 6 input channels with optional microphone pre-amplifiers and between 4 and 6 output channels. The multichannel sound extension cape for the BeagleBone Black is designed and produced. An ALSA driver is written for it. It is used with the openMHA hearing aid research software to perform hearing aid signal processing on the BeagleBone Black with a customized Debian distribution tailored to real-time audio signal processing.

Keywords

Hearing aids, audio signal processing, sound hardware

1 Introduction

Hearing aids are the most common form of mitigation for mild and moderate hearing losses. Hearing aids help the wearer to follow conversations and acoustic events in different situations. In the complex acoustic environments that we encounter in our daily life, information about the acoustic scene is inferred at higher stages of the human auditory system and exploited in the brain for, e.g., speech understanding. A hearing loss causes — in addition to reduced sensitivity to soft sounds — a partial loss of this information. Effective signal processing algorithms are required for compensation. For this reason, improving signal processing in hearing aids is an active research topic.

Part of the work in hearing aid research is to develop novel signal processing algorithms that can be used in hearing aids to improve the hearing experience for hard-of-hearing people. Usually, simulations are run and evaluated in terms of objective measures after such an algorithm has been developed mathematically. Results from simulations do not necessarily reflect the benefit of the algorithm a) when integrated in a complete signal processing chain of a hearing aid and b) in a real-world scenario. To assess the usefulness of new hearing aid algorithms for hearing-impaired people, new potential hearing aid signal processing algorithms also have to be tested with hearing impaired test subjects in realistic situations. Running an algorithm under test on an end-user hearing device is practically infeasible as it requires access to a proprietary system of a hearing aid manufacturer, and a large effort for the down-to-hardware implementation is required on such devices. Instead, a software platform can be used to simulate the hearing aid processing chain. The open Master Hearing Aid (openMHA, [HörTech gGmbH and Universität Oldenburg, 2017], [Herzke et al., 2017]) is such a platform. openMHA can be utilized to conduct field tests of hearing aid processing methods running on portable hardware.

The following sections first introduce the software and hardware platforms utilizable to evaluate hearing aid algorithms with hearing-impaired test subjects. We work out the need for a custom multichannel sound interface for a small, portable computer. The subsequent sections report on the hardware design process that resulted in the *Cape4all*[1] BeagleBone sound interface, the sound driver development, and finally the possible usage of the sound interface for hearing aid research.

[1]developed in the cluster of excellence "Hearing4all"

2 Software and Hardware Platform for Hearing Aid Research

HörTech and the University of Oldenburg have developed the openMHA [HörTech gGmbH and Universität Oldenburg, 2017], [Herzke et al., 2017] software platform for the development and evaluation of hearing aid algorithms, where individual hearing aid algorithms can be implemented as plugins and loaded at run-time. The platform provides a set of standard algorithms to form a complete hearing aid. It can process audio signal in real-time with a low delay (<10 ms) between sound input and sound output. (The actual delay depends on the sound hardware used for input and output, configuration options like sampling rate and audio buffer size, and also on delay introduced by some signal processing algorithms.)

In its current version 4.5.5, the openMHA software platform can execute on computers with Linux and Mac OS operating system, e.g., in a laboratory environment. Toolboxes for generating virtual sound environments in a laboratory exist (e.g. TASCAR [Grimm et al., 2015]) but the sound environment in a lab — and even more the subject behavior in a lab environment — will always differ from real environments encountered by hearing aid users in real life. To test real-life situations, we have to go outside and into real situations with hearing-impaired users wearing a mobile computer that executes the openMHA and provides the first chance to test new algorithms in real-world situations. In the past, we have used laptops for this purpose but with the advent of small, ARM-based single board computers like the Raspberry Pi, BeagleBone, and several others these become an option for executing openMHA that imposes less weight to carry around for the test subjects. The processing power of these devices is significantly lower than that of PCs and laptops, which will always limit the extent and setup of algorithms that can be executed on such a mobile platform (compared to a PC).

openMHA is meant as a common platform to be used by different hearing aid research labs to combine their work. By providing a solid base platform, we want to encourage researchers to implement and publish their algorithms as openMHA plugins so that work can be shared and results can be reproduced by independent labs.

For this purpose, openMHA includes a toolbox library that already contains functions and classes useful to more than one algorithm to speed up implementation of new algorithms. As a key to usability of the software in different usage scenarios openMHA also includes several manuals for different entry levels ranging from plugin developments over application engineering based on available plugins and functionality to the application of the software in the context of audiological research and hearing aid fitting controlled through a graphical user interface (GUI). Step-by-step tutorials on the implementation of openMHA plugins as well as examples of configurations are provided to enable an autonomous familiarization for new users.

Some hearing aid algorithms — such as directional microphones — need to process the sound from more than one microphone per ear which is why a multichannel sound card is generally needed to capture the sound from all hearing aid microphones. Professional sound cards can be used for this purpose in stationary laboratory setups. Bus-powered USB sound cards can be used with laptops in mobile evaluation setups, but the choice of bus-powered interfaces with more than 2 input channels is limited. We have observed that the total delay between input and output sounds that can be achieved with USB sound cards is always larger than what can be achieved with similar sound cards with PCI or Expresscard interface. This difference in delay is in the order of 2 ms, which will already affect some hearing aid algorithms. We have also observed that the delay may vary from one start of the sound card to the next with USB sound cards, in the range of 1 ms, which is detrimental to some processing algorithms such as acoustic feedback reduction. (Feedback reduction algorithms are an essential part of a hearing aid processing chain and need the system to be as invariant as possible to work effectively.) The Inter-IC Sound (IIS or I^2S) bus — transporting sound data from the SoC[2] to the audio codecs with the AD/DA converters (and back) — is accessible on expansion headers on many of the single-board ARM computers, making it possible to create custom sound interface hardware.

Third parties already provide multichannel sound interfaces for popular boards like the BeagleBone Black and the Raspberry Pi. Of these two devices, the BeagleBone Black has the advantage of hardware support for multichannel

[2] Abbreviation for System on a Chip, the combination of a microprocessor and several peripherals (e.g. graphics unit, sound interface) on a single chip.

audio input/output. See Section 3.1 for details.

One multichannel sound interface option for the BeagleBone Black is the *BELA* cape [Moro et al., 2016]. It provides stereo in/out and additional 8 analogue data acquisition channels. These additional 8 analogue data acquisition channels can also be used to capture audio but do not provide anti-aliasing filters, and achievable sampling rates depend on the number of channels in simultaneous use. The *BELA* cape makes use of real-time hardware present on the BeagleBone Black. Audio processing algorithms can be compiled to execute on this real-time hardware, process the input channel data, and produce output channel data. Existing Linux audio processing applications using ALSA[3] or JACK[4][Davis, 2003] and common features of the operating system cannot execute on this real-time hardware.

Another multichannel audio interface developed for BeagleBone platforms is the *CTAG face 2|4* [Langer and Manzke, 2015], [Langer, 2015]. Its hardware design is available open-source from GitHub and drivers have been included in official BeagleBoard SD card images. Providing capabilities for multichannel signal processing this device is in principle suitable for hearing aid processing on the BeagleBone Black. A drawback that remains here is the necessity to add external power supply for the microphones connected to the device.

The *Octo Audio Injector* sound card `http://www.audioinjector.net/rpi-octo-hat` offers 6 input channels and 8 output channels for the Raspberry Pi. Although the Raspberry Pi offers no hardware support for more than two sound channels, this sound card manages to offer enough input channels to connect 2 hearing aids with 3 microphones each. A disadvantage of this sound card for hearing aid research is that additional external microphone preamplifiers are needed to raise the microphone signals to line level, which adds to the hardware that test subjects would have to carry around. An example setup for teaching hearing aid signal processing [Schädler, 2017], [Schädler et al., 2018] uses the stereo version of this sound card together with external microphone preamplifiers.

[3]Acronym for Advanced Linux Sound Architecture, name for a system of Linux kernel sound card drivers and user space API to exchange sound data with these drivers.

[4] Self-referencing acronym for JACK Audio Connection Kit, a user-space server application and library to connect inputs and outputs of audio applications and sound cards.

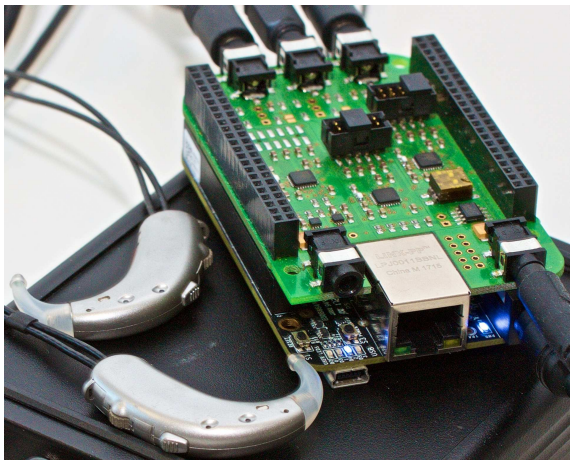


Figure 1: *Cape4all* with two hearing aids (each containing three microphones) connected.

3 Development of the Cape4all Multichannel Sound Interface for Hearing Aid Research

We have a need for a compact multichannel sound interface for a single-board ARM computer with integrated microphone pre-amplifiers for hearing aid research. Since such a multichannel sound interface was not available, we decided to develop such a sound interface ourselves.

3.1 Choice of ARM Board Basis for a Multichannel Sound Card

In the ongoing developments of the Cluster of Excellence "Hearing4all"[5] several audio interfaces were developed proving the inter IC sound (IIS or I^{2}S) in combination with the Analog Devices ADAU1761 [Analog Devices Inc., 2009] stereo audio codec useful [Seifert et al., 2015]. To gain multichannel capabilities, a time division multiplex (TDM) scheme specified for I^{2}S is used. The chosen ADAU1761 codecs support a TDM output scheme. To allow the usage in combination with an ARM-based platform and therefore with openMHA, the BeagleBone Black with native I^{2}S TDM support by the integrated McASP[6] interfaces was chosen.

3.2 Hardware Design

The *Cape4all* hardware was designed by the Leibniz University Hannover based on [Seifert

[5]`http://hearing4all.eu/`

[6]Abbreviation for Multichannel Audio Serial Port.

et al., 2015].

In addittion to the I²S TDM output capabilities the Analog Devices ADAU1761 audio codecs have integrated microphone amplifiers. Up to 3 microphones for each ear on a bilateral fitting are assumed in the context of hearing device development. Therefore, 3 stereo audio codecs are integrated on the *Cape4all* PCB[7] allowing up to 6 input and output channels simultaneously. Due to the TDM scheme, only five signal connections are required to transport and synchronize all 3 codecs with 6 input and output channels and the McASP interface of the BeagleBone Black.

The board provides standard stereo jacks for connecting off-the-shelf sound hardware as well as pin headers for custom designs. 3 stereo jacks are mounted on the board for the 6 input channels, and 2 additional stereo jacks for the first 4 output channels. The remaining output channels are only accessible through the pin headers. An on-board voltage regulator provides microphone bias voltage which can be switched on and off as needed and routed to different connectors. The bias voltage can be altered by exchanging on-board resistors. For more details, see the reference manual provided with the hardware design files and the driver as download from `https://github.com/HoerTech-gGmbH/Cape4all`. Figure 1 shows the hardware in use.

3.3 Hardware Tests and Design Revisions

In the testing process of previously built audio interface boards using the ADAU1761 stereo audio codecs, it was revealed that the internal components of the codecs create bus collision. The I²S TDM bus digital output pins of the codecs do not provide high-resistance state, driving the signal high or low preventing another codec to put data on the same signal. The documentation of the codecs did not give any details helping to avoid the bus collision. In order to avoid this, an OR-gate was added to the board design to merge the signals of the codecs to one signal. This solves the problem on voltage level but does not prevent timing collision due to wrong configuration of the codec outputs. The correct codec configuration is ensured by the ALSA driver (see Section 4). In normal TDM configuration, filling 6 of the available 8 timeslots, all 3 audio codecs are working correctly. For further details on I²S TDM signaling see [Seifert et al., 2013].

[7] Abbreviation for Printed Circuit Board.

3.4 Release as Open Hardware

The hardware design files are released under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License on GitHub `https://github.com/HoerTech-gGmbH/Cape4all`.

4 Driver development

The ALSA sound driver for the *Cape4all* sound interface was developed by 64 Studio.

As the Linux kernel already has support for both the McASP Audio Serial Port [Pandey et al., 2009] used on the BeagleBone Black and the ADAU1761 codec [Clausen, 2014] used on the *Cape4all*, the development by 64 Studio was to create a glue-driver explaining to the SoC the order the codecs are arranged on the *Cape4all*. The driver registers the cape as effectively one PCM device with three mixer sub-devices (corresponding to the three physical ADAU1761 codecs), each with their own set of controls in the ALSA mixer. Also, the driver sets up the codec's clock-path, TDM slots and various other default settings.

As the driver exposes the *Cape4all* as a regular ALSA device with three mixer sub-devices, each with their own ALSA controls, application software may communicate with these devices without any modifications.

4.1 Limitations

The McASP used on the BeagleBone Black is clocked from a 24.576 MHz crystal. This limits the available sample rates to be a whole divisor of this clock, for instance 24 kHz or 48 kHz is acceptable but 22.05 kHz or 44.1 kHz is not.

The ADAU1761 codecs do not directly support sharing 6 channels between 3 separate codecs on a TDM bus. As a workaround, the TDM mode for transferring 8 channels is used, where 2 channels contain no data. A consequence is that the sound card appears to have 8 channels in ALSA but only the first 6 channels, corresponding to the physical channels, should be used.

4.2 Release

The driver code is released as open source software under the GNU General Public License, Version 2 or later, in the same git repository as the hardware design files on GitHub, `https://github.com/HoerTech-gGmbH/Cape4all`.

5 Usage

As Linux distributions created by SoC development board manufacturers are typically not being suited to audio signal processing and contain a lot of applications that are not useful in this context, a custom Debian distribution has been prepared by 64 Studio. E.g. the JACK Audio Server contained in this custom distribution was built without DBUS support to allow the system to run without a GUI and the final Debian system was tweaked by 64 Studio for basic real-time performance. An image file containing this distribution is available for download together with the hardware design. It contains just the software needed to run openMHA, has device-tree and custom Kernel built-in as well as custom tweaks for increased real-time audio performance.

These steps are needed to prepare a BeagleBone Black for multichannel signal processing with openMHA and *Cape4all*:

- Download and copy image to SD-card
- Download and compile openMHA on the system
- Set up system for higher audio performance according to manual provided
- Start JACK Audio Server with settings according to the openMHA configuration to be run
- Read example configuration provided with openMHA and start processing

The openMHA processes can be accessed at runtime through a TCP/IP connection. This connection can be used to read out and change parameters of the running system. By this means it is possible to run a GUI on a laptop or tablet computer that can be used to control the processing parameters remotely. For details, refer to the openMHA application manual.

6 Conclusions

Cape4all is a working, multichannel sound interface for the BeagleBone Black with integrated microphone pre-amplifiers which makes it suitable for hearing aid research, where preamplifiers are essential and where a small form factor matters.

A working ALSA driver has been developed that takes care of the proper initialization of the codecs and the multichannel capabilities of the BeagleBone Black and then drives the multichannel sound exchange between user space applications and the codecs on the sound interface.

Both, the hardware design files and the driver, have been published with open licenses on GitHub, `https://github.com/HoerTech-gGmbH/Cape4all`.

In its current state, the *Cape4all* can be run together with a JACK Audio Server on a BeagleBone Black reliably with a 4 ms buffer (128 samples per channel) at a 32 kHz sampling rate. This is the state directly after driver development before any optimization towards shorter audio buffers has been performed. This current state is an important step towards our goal of a mobile hearing aid algorithm evaluation setup, but it needs to be improved to achieve the target overall audio delay below 10 ms between input and output sounds, considering that some of the algorithms will add a small algorithmic delay. Therefore, we are going to further optimize the driver in collaboration with 64 Studio after the initial release to enable smaller audio buffer sizes.

7 Acknowledgements

This work was supported by the German Research Foundation (DFG) Cluster of Excellence EXC 1077/1 "Hearing4all".

Research reported in this publication was supported by the National Institute On Deafness And Other Communication Disorders of the National Institutes of Health under Award Numbers R01DC015429. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health.

References

Analog Devices Inc. 2009. ADAU1761 – SigmaDSP stereo, low power, 96 khz, 24-bit audio codec with integrated PLL. `http://www.analog.com/static/imported-files/data_sheets/ADAU1761.pdf`.

Lars-Peter Clausen. 2014. `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/sound/soc/codecs/adau17x1.c`.

Paul Davis. 2003. Jack audio connection kit. http://jackaudio.org/.

Giso Grimm, Joanna Luberadzka, Tobias Herzke, and Volker Hohmann. 2015. Tool-

box for acoustic scene creation and rendering (TASCAR) – render methods and research applications. In *Proceedings of the Linux Audio Conference*, pages 1–7, Mainz. Johannes Gutenberg-Universität.

Tobias Herzke, Hendrik Kayser, Frasher Loshaj, Giso Grimm, and Volker Hohmann. 2017. Open signal processing software platform for hearing aid research (openMHA). In *Proceedings of the Linux Audio Conference*, pages 35–42, Saint-Étienne. Université Jean Monnet.

HörTech gGmbH and Universität Oldenburg. 2017. openMHA web site on GitHub. `http://www.openmha.org/`.

Henrik Langer and Robert Manzke. 2015. Linux-based low-latency multichannel audio system (CTAG face2—4). `http://www.creative-technologies.de/linux-based-low-latency-multichannel-audio-system-2/`.

Henrik Langer. 2015. Linuxbasiertes Mehrkanal-Audiosystem mit niedriger Latenz.

Giulio Moro, Astrid Bin, Robert H Jack, Christian Heinrichs, Andrew P McPherson, et al. 2016. Making high-performance embedded instruments with bela and pure data.

Nirmal Pandey, Suresh Rajashekara, and Steve Chen. 2009. `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/sound/soc/davinci/davinci-mcasp.c`.

Marc René Schädler, Hendrik Kayser, and Tobias Herzke. 2018. Pi hearing aid. *The MagPi (Raspberry Pi Magazine)*, 67:34–35.

Marc René Schädler. 2017. openMHA on Raspberry Pi. `https://github.com/m-r-s/hearingaid-prototype`.

Christopher Seifert, Guillermo Payá-Vayá, and Holger Blume. 2013. A multi-channel audio extension board for binaural hearing aid systems. In *Proceedings of ICT. OPEN. Conference ICT. OPEN*, pages 33–37.

Christopher Seifert, Guillermo Payá-Vayá, Holger Blume, Tobias Herzke, and Volker Hohmann. 2015. A mobile SoC-based platform for evaluating hearing aid algorithms and architectures. In *Consumer Electronics-Berlin (ICCE-Berlin), 2015 IEEE 5th International Conference on*, pages 93–97. IEEE.

MRuby-Zest: a Scriptable Audio GUI Framework

Mark McCurry
DSP/ML Researcher
United States of America
mark.d.mccurry@gmail.com

Abstract

Audio tools face a set of uncommon user interface design and implementation challenges. These constraints make high quality interfaces within the open source realm particular difficult to execute on volunteer time. The challenges include producing a unique identity for the application, providing easy to use controls for the parameters of the application, and providing interesting ways to visualize the data within the application. Additionally, existing toolkits produce technical issues when embedding within plugin hosts. MRuby-Zest is a new toolkit that was build while the ZynAddSubFX user interface was rewritten. This toolkit possesses unique characteristics within open source toolkits which target the problems specific to audio applications.

Keywords

Interface Design, LV2, VST, Ruby

1 Introduction

MRuby-Zest was created to address long standing issues in the ZynAddSubFX[1] user interface. The MRuby-Zest framework was built with 5 characteristics in mind.

Scriptable: Implementation uses a first class higher level language

Dynamically Resizable: Fluid layouts which do not have any fixed sizes

Hot Reloadable: Reloads a modified implementation without restarting

Embeddable: Can be placed within another UI without conflicts

Maintainable: Relatively simple to read and write GUI code

Several examples of the toolkit can be seen in Fig. 1, 2, 3, and 4.

Figure 1: Zyn-Fusion Add Synth

1.1 History

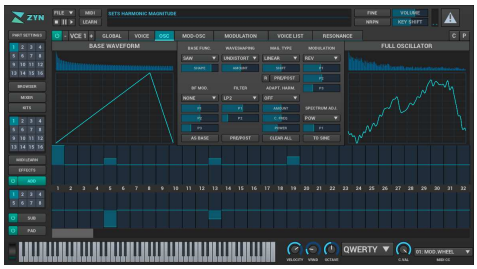
Historically the ZynAddSubFX interface was written in FLTK[2] and the user interface processed a number of usability issues as well as look and feel consistency issues. Additionally the multi-window FLTK design ZynAddSubFX previously used did not embed cleanly into plugin hosts. Mid 2014 a series of mockups by posted online by Budislav Stepanov[1]. The mockups provided an overhaul of the workflow of the GUI, but it was a new design which did not make use of any of the existing widgets, nor widgets available in other toolkits. Since the new interface was not small some tools would be needed to increase the speed of development.

Figure 2: Zyn-Fusion Kit Editor

The first prototypes were written in the Qt Meta Language (QML)[3; 4] QML is a domain

[1] `http://www.kvraudio.com/forum/viewtopic.php?f=47&t=412173`

Figure 3: Zyn-Fusion Oscillator



specific language commonly used to describe a group of components and properties within a user interface. In addition to purely describing components, QML can also define callbacks and new functionality for widgets using a scripting language. Within Qt, this scripting language is javascript.

While prototyping ZynAddSubFX's UI, the prototype frequently ended up accessing the C++ to QML layer of Qt which received much less documentation than the pure QML layer. Some of the logic/drawing routines for the program ended up in C++ portion which couldn't be effectively hotloaded, which slowed development. Additionally the barrier between C++ and Qt's javascript engine was non-trivial. Overall, this process highlighted that for the prototype and the version of QML used:

- QML's javascript was not sufficiently flexible when extending widgets
- QML's layout algorithms did not meet the requirements of the new design
- None of the QML components were heavily used beyond primitives (rectangles, component-repeaters, etc)

Figure 4: Zyn-Fusion Pad Synth

QML at a high level was useful, concise, and easy to dynamically manipulate. The infrastructure around it was limiting for the ZynAddSubFX use case. So, at this stage of prototyping the question was posed: "Why does QML need to be tied to Qt and the specific scripting language of Javascript?"

QML within Qt was script-able, layout routines were flexible enough that resize-ability wasn't a major issue, and it was built with hot loading in mind. Per embed-ability, Qt does not embed well; specifically, loading two plugins which use different Qt versions (e.g. Qt4/Qt5) is known to cause issues with symbol name conflicts and global variable conflicts. When initial prototyping was done with QML it was acknowledged that eventually the project may need to move away from Qt and MRuby-Zest was born. MRuby-Zest took the QML language, replaced the scripting language with Ruby, integrated it with the nanovg OpenGL rendering library, and began to leverage parameter metadata that ZynAddSubFX produces via the rtosc library[5].

1.2 Prior Art

The problem of creating a good looking embeddable GUI isn't a new task in the open source audio realm. Audio plugins are a challenging design space. Complex information needs to be presented to a reasonably non-technical audience in a way that they can quickly understand how to manipulate it. To facilitate this, an audio plugin needs to differentiate itself from other applications and provide a consistent and easy to understand visual and interactive language for the user to tune.

There's certainly plenty of tools based upon more standard toolkits like GTK or Qt. A few of the open source audio plugin toolkits include: AVTK[6], robtk[7], ffltk[8], DPF[9], rutabaga[10], JUCE[11], and a few PUGL based non-toolkit options also exist in some smaller applications.

Compared to these toolkits, MRuby-Zest desires to be generally built for larger more complex applications as well as having a distinct look and feel. Additionally the heavy use of Ruby scripting makes MRuby-Zest more geared towards rapid development of a large complex interface.

2 Implementation

The MRuby-Zest framework is implemented through a combination of different layers. This includes QML parsing/processing, OSC communication, event handling, and the widget classes themselves.

2.1 QML

QML is a domain specific language commonly used to describe a group of components within a user interface. More generally, QML defines a tree of objects, methods on object instances, a set of interrelated properties, and bindings for the properties. Within Qt, QML runs on Javascript on top of the normal tools that Qt provides. MRuby-Zest's QML uses Ruby for scripting, but otherwise shares most structural similarities.

Through the use of a dynamic language QML gains a number of properties which make interface development easier. First and foremost is the conciseness of the language. Using C++ a simple widget ends up being rather verbose:

Listing 1: C++ Widget

```
class SubWidget: public Rectangle
{
    public:
      SubWidget(void) {
          fooVar = "foo";
          barVar = true;
          structure = new Structure;
          model = new Model;
          structure->add_parent(this);
          model->add_parent(this);
      }

      ~SubWidget(void)
      {
          delete structure;
          delete model;
      }


      string fooVar;
      bool barVar;
      Structure *structure;
      Model *model;

      void fn(string args)
      {
          cout << args << endl;
          structure->method();
      }
};
```

With ruby methods/callbacks QML would look virtually the same. Indeed parsing all of the QML I had written thus far didn't depend upon the scripting language at all. With ruby it was possible to use QML to create something like:

Listing 2: QML Widget

```
Rectangle {
    id: window

    property String fooVar: "foo"
    property Bool barVar: true

    Structure { id: structure }
    Model { id: model }

    function fn(args) {
        puts args
        structure.method()
    }
}
```

And translate it to something similar to:

Listing 3: Ruby Widget Result

```
class Instance < Rectangle
    attr_reader :structure, :model
    attr_property(:fooVar, String)
    attr_property(:barVar, Bool)

    def initialize()
        add_child(@structure =
            Structure.new)
        add_child(@model =
            Model.new)
        set_property(:fooVar, "foo")
        set_property(:borVar, true)
    end

    def fn(args)
        puts args
        structure.method
    end
end
```

While this transformation may seem trivial, the organizational structure that QML's Qt Modeling Language provides is helpful at understanding complex widget hierarchies at a glance.

2.2 Hot-loading

When developing or maintaining a synth a considerable amount of time is spent on improving the user interface. GUI development can be slow going work and compared to other tasks it can be harder to obtain a fast feedback loop. Generally GUI development in these cases has the loop of:

1. Build - Compile from source
2. Open - Launch the application

3. Navigate - Get to the part of the application which is modified
4. Observe - See how the application behaves
5. Close - Close application
6. Modify - Change behavior
7. Repeat - From step 1 repeat

MRuby-Zest on the other hand makes it possible to load code into live instances of the user interface. Hotloading code in MRuby-Zest is possible since the vast majority of code can be relatively simply converted to Ruby code and loaded into the active Ruby VM during execution. Using hotloading the development loop becomes:

1. Build - Compile from source
2. Open - Launch the application
3. Navigate - Get to the part of the application which is modified
4. Observe - See how the application behaves
5. Modify - Change behavior
6. Repeat - From step 4 repeat until done
7. Close - Exit after desired behavior is obtained

Reducing the feedback loop makes it much easier to tune graphics, layout, and the feel of input handling.

2.3 OSC Communications

Different GUI toolkits have different approaches on communicating state to the rest of the application outside of the interface (the backend). MRuby-Zest leverages Open Sound Control (OSC) to communicate to in-process and out-of-process backends. This submodule is known as the OSC-Bridge.

The OSC-Bridge controls communication to the optionally-remote synthesis engine, and provides metadata for modeling parameters in the user interface. The OSC interface specifies the minimum value, maximum value, short names, tooltips, and other information about parameters that can be accessed. Additionally, this layer provides several mechanisms for tracking and synchronizing the value of remote parameters. These mechanisms abstract away synchronization mechanisms, simplifying the widget programming.

2.4 Drawing model & events

MRuby-Zest is an OpenGL based toolkit which uses PUGL[12] for platform specific event handling and nanovg[13] for a drawing API. OpenGL 2.1 (with the framebuffer extension) was used to simplify embedding and enable complex animations in future versions. NanoVG was used to simplify drawing vector graphics, which were necessary for simplified fluid resizing of the GUI.

When drawing in the MRuby-Zest toolkit, widgets are drawn depth first for each layer of the user interface. These layers are:

- the background - where most widgets are drawn
- the animation layer - simple drawings expected to update many times a second
- the overlay - drawing on top of the interface (e.g. modals/dropdowns)

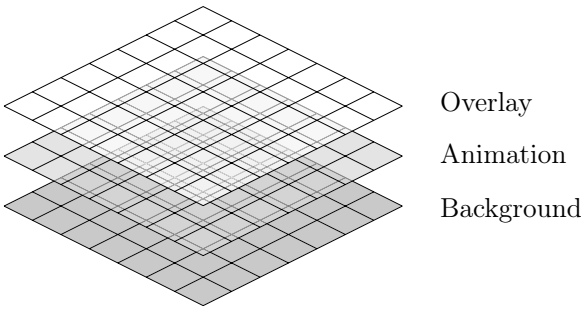


Figure 5: Framebuffer layers

Since the widgets define strict bounding boxes for drawing, redrawing can be cheaply done. First, the damaged part of the altered layer can be masked. Then, all widgets which intersect with the layer and damaged region are redraw. Finally, the three framebuffer layers are redrawn producing the final GUI.

On the event handling side, MRuby-Zest behaves fairly traditionally. At the time of writing MRuby-Zest responds to:

- Key presses/releases
- Mouse presses/releases
- Mouse drags
- Mouse hovering
- Window resizing

2.5 Widgets

The current version of MRuby-Zest has 182 widgets. These range from simple buttons, labels, and boxes to complex views of parameters. Two major types of widget that are available in MRuby-Zest are layout widgets and parameter controlling widgets.

In MRuby-Zest there are grid pack (Fig. 6), module pack (Fig. 7), tab pack, vertically packed, horizontally packed, and other layout specific widgets. Historically the resizing was taken care of by a constraint layout system which solved a set of linear-equations via GLPK[14], however this approach proved too computationally expensive and was removed to maintain a more consistent framerate.

Figure 6: Grid Layout

Figure 7: Control Rows Layout

There are also a wide array of options to represent parameters. This includes Knobs (Fig. 8), sliders (Fig. 9), drop downs (Fig. 10), buttons, plots (Fig. 11), text editors, piano keyboards, and more.

Figure 8: Knob Widget

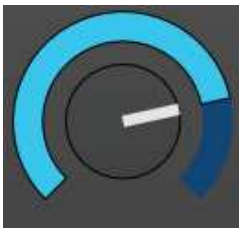


Figure 9: Horizontal Slider Widget

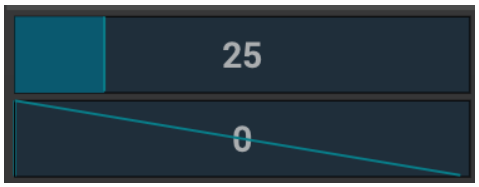


Figure 10: Drop down Widget

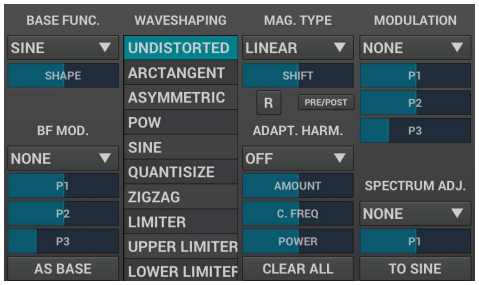
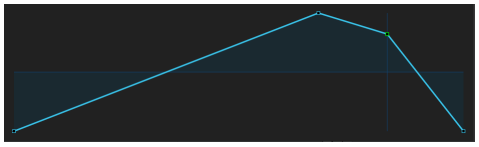


Figure 11: Envelopes/2D plotting Widget



3 Conclusion

Audio applications are a complex design and programming domain. Existing toolkits pose embedding challenges as well as difficulties in rapid development. MRuby-Zest provides one new approach to audio plugin GUI development and is available at `https://github/mruby-zest/` under a mixed MIT and LGPL license. Using MRuby-Zest, the ZynAddSubFX project has been able to build the new ZynFusion interface. This interface serves as a complex example of the MRuby-Zest framework and shows that the chosen approach can speed up development on non-trivial designs.

References

[1] N. O. Paul, M. McCurry, *et al.*, "Zynaddsubfx musical synthesizer." `http://zynaddsubfx.sf.net/`, 2018.

[2] B. Spitzak *et al.*, "Fast light toolkit (fltk)," 1998.

[3] H. Nord, E. Chambe-Eng, *et al.*, "Qt - software toolkit." `http://qt.io/`, 2018.

[4] Q. Contributors, "Qt - software toolkit." `https://doc.qt.io/qt-5.10/qtqml-index.html`, 2018.

[5] M. McCurry, "rtosc - realtime safe open sound control." https://github.com/fundamental/rtosc, 2018.

[6] H. van Haaren, "Avtk." https://github.com/openAVproductions/openAV-AVTK, 2018.

[7] R. Gareus, "robtk." https://github.com/x42/robtk, 2018.

[8] S. Jackson, "Infamous plugins." https://github.com/ssj71/infamousPlugins, 2018.

[9] F. Coelho, "Dpf." https://github.com/DISTRHO/DPF, 2018.

[10] W. Light, "Rutabaga." https://github.com/wrl/rutabaga, 2018.

[11] "Juce." https://juce.com/, 2018.

[12] D. Robillard, "Pugl - cross platform windowing abstraction layer." https://drobilla.net/software/pugl, 2018.

[13] M. Mononen, "nanovg - canvas api for opengl." https://github.com/memononen/nanovg, 2018.

[14] A. Makhorin, "Glpk linear programming kit manual." http://www.gnu.org/software/glpk/glpk.html, 2014.

[15] Y. M. Matsumoto *et al.*, "Mruby - embeddable ruby interpreter." https://github.com/mruby/mruby, 2018.

Camomile: Creating audio plugins with Pure Data

Pierre GUILLOT
CICM – EA1572
University Paris 8
Saint-Denis, France
guillotpierre6@gmail.com

Abstract

Camomile is an audio plugin with Pure Data embedded for creating, with patches, original and cross-platform audio plugins that work with any digital audio workstation that supports VST or Audio Unit formats. This paper presents an overview of the current functionalities of Camomile and the possibilities offered by this tool. Following this presentation, the main lines of future development are exposed.

Keywords

Pure Data, Plugin, DAW, VST, Audio Unit

1 Introduction

Camomile[1] is a free, open-source and cross-platform audio plugin with Pure Data[2] [1] embedded, used to control patches inside a large set of digital audio workstations – as long as they support VST[3] or Audio Unit[4] formats. Development for this tool started in spring 2015 with a view to address issues that are related to pedagogical uses, experimental purposes and creation contexts. To satisfy these objectives, several approaches have been explored, resulting in many prototypes that have preceded the current version of the plugin. This entire endeavour, the many functional specifications that have been defined, the major issues that have been encountered – such as support for multiple instances and multithreading in Pure Data, and linking Pure Data with the plugin –, the different solutions that have been proposed and the choices that have been made are all presented in detailed in [2][5]. As most of the technical barriers have been broken down, the main goal of this project is currently to offer a tool that can compete with standard plugins. Hence, following an overview of the many features already offered by Camomile, the paper exposes the remaining work that is needed to complete this plugin, and the perspectives of development.

In practice, Camomile can be viewed as a meta-plugin: a plugin that generates other plugins. To clarify this presentation, the term "meta-plugin" will be used for this plugin – which embeds Pure Data; while the resulting plugins, containing the meta-plugin and patches, and can be used in digital audio workstations will simply be called "audio plugins". Thus, this presentation of Camomile is organised along two distinct but complementary axes. The first axis is focused on the creation of the audio plugin using the meta plugin: defining its functionality, creating patches, setting up features and so on. The second axis focuses on using the audio plugins: support by digital audio workstations, graphical interfaces and so on. Nevertheless, to offer a clear understanding of the defining aspects of each axis, this presentation is inverted. First, audio plugins usage is presented to highlight the features offered to the final user. Secondly, a large set of the features which can be implemented during the creation process will be shown. Following this

[1]The plugin is available in the VST2, VST3 and Audio Unit format for Linux, Windows and MacOS. The binaries and sources are available on the Github repository github.com/pierreguillot/camomile (accessed January 2018). Since the version 1.0.0, the sources are distributed under the license GNU GPLv3. The sources of the anterior versions are distributed under the licence BSD 3.

[2]Pure Data is a free and open-source software, created by Miller Puckette at the University of California, San Diego msp.ucsd.edu/software.html (accessed January 2018).

[3]The digital audio plugin format VST (Virtual Studio Technology) 2 et 3 are developed by the Steinberg GmbH company steinberg.net (accessed January 2018).

[4]The digital audio plugin format Audio Unit is developed by the Apple Inc. company developer.apple.com/audio (accessed January 2018).

[5]The publication also presents the context in which this project took place and in particular the related projects such as PdVST and PdLV2 but also the parallel projects like PdDroidParty and PdParty [4].

presentation, future developments of Camomile and its general perspectives will be exposed.

2 Using plugins

Before presenting the different features available to create audio plugins, it seems necessary to introduce what is ultimately an audio plugin created with Camomile and how this audio plugin appears to the user. Indeed, the architecture of the audio plugin generated with Camomile is a bit particular, and its approach favours sharing patches, abstractions and other documents to be used in conjunction with the meta-plugin to generate plugins, rather than directly sharing audio plugins – mainly because the audio plugins are associated with specific formats and operating systems while the original documents are free from these restrictions, so this approach gives users more freedom. So understanding the architecture of an audio plugin and the result in digital audio workstations will allow users to get a better grasp of the process of creating plugins.

2.1 Generating and loading plugins

The flexibility and the dynamic aspect of the Camomile approach makes it a tool noticeably different from standard plugins. Indeed, the binary files offered in the distribution and the meta-plugins, are not designed to be used directly within a digital audio workstation[6]. They must be used to set up the bond between the digital audio workstation and patches in order to generate the new audio plugins (see Figure 1).

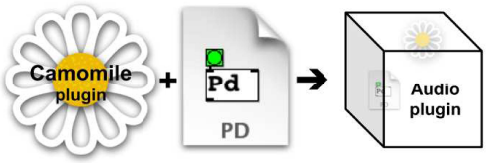


Figure 1: Schematic operation of the generation of an audio plugin from a Pure Data patch and the meta-plugin Camomile.

In practice, building an audio plugin with Camomile simply requires associating one of the meta-plugins provided by the distribution – according to the desired format and the type of plugin[7] – to a main patch and a set of additional and complementary contents – textual description of the audio plugin, abstractions, images and so on. Specifically, the operation consists in renaming the meta-plugin according to the main patch and also consists in respecting a certain hierarchy of the relative paths of the different files by creating a bundle[8]. Once this associated bundle is installed in the appropriate directory[9], the audio plugin is recognised by the digital audio workstations and is supported in the exact same way it would had been if compiled in a conventional manner (see Figure 2)[10]. The digital audio workstation then offers the ability to load one or more instances of the plugin and interacts with them in a conventional way with operations such as creating automations for the parameters, saving and recalling presets and so on.

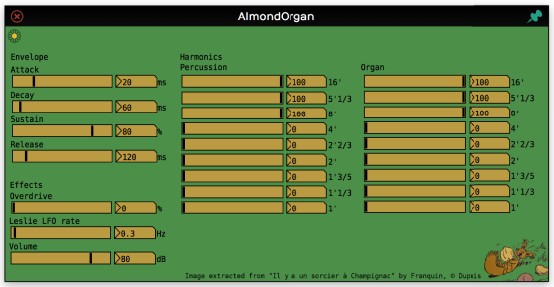


Figure 2: Representation of the graphical interface of the plugin *AlmondOrgan*, given as an example with Camomile, and generated from the patch.

2.2 User interfaces

Apart from the native representation of the digital audio workstation, which usually offers generic interfaces to represent and control parameters, this plugin has its own graphical interface. This window displays a representation of the main patch that potentially includes sliders, buttons, comments, or other user interface

[6]In practice, the digital audio workstation can nevertheless load the meta-plugins but without any patch, and so without any audio engine, they are useless.

[7]The distribution contains meta-plugin for each format – VST2, VST3 or Audio Unit – and each type of plugin – effect or instrument.

[8]On Linux and Windows, the meta-plugin and the patch must be placed in the same folder. On MacOS, all the files must be placed in the OS specific bundle of the plugin. These operations are presented step by step in the documentation of Camomile.

[9]The installation path of a plugin may depend on its format, the operating system or the preferences of the digital audio workstation. The documentation of Camomile helps to carry out this operation.

[10]This feature presented in detail in [2] ensures that the digital audio workstation manages each plugin created with Camomile independently, thus avoiding problems related to the management of presets or parameters but also to the sharing of projects and plugins.

components that are available in Pure Data (see Figure 2). This window makes it possible to represent the sound engine and interact with it, and also to communicate with the plugin. As will be shown later, the graphical user interfaces of the patch can be associated to parameters or specific actions like displaying a dialogue window to open or save files.

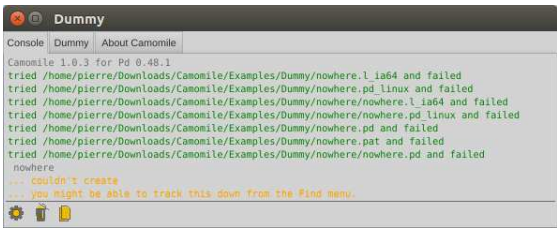


Figure 3: Auxiliary window of a plugin named Dummy illustrating the use of the console and the different types of messages.

In the upper-left corner of the interface, a button representing a chamomile flower is used to display an auxiliary window with three tabs (see Figure 3). The first tab corresponds to a console relatively similar to the one offered by Pure Data. This console receives the messages sent via the object print, the internal warnings of Pure Data – when an abstraction is not found for example – but also additional information related to the operation of the meta-plugin to facilitate debugging the patches. The console also allows you to copy, delete and filter messages according to their importance. The second tab displays information defined by the creator of the patch such as a description of the operations and how to use the plugin but also information related to credits or the plugin version. Finally, the last tab displays information related to Camomile, including legal information and credits related to different dependencies such as Pure Data, libPD[11] [3] and JUCE[12].

3 Creating plugins

Building a digital audio plugin with Camomile requires proper communication between the patch – the core of digital audio processing – and the digital audio workstation through the meta-plugin. For this purpose, Camomile offers several interfaces to use and handle a wide range of the usual features of digital audio plugins, such as parameters management, reading information from the play head, or creating the graphical user interface. These interfaces cover two aspects of plugin creation: properties definition for the plugin – such as its ability to handle MIDI events or the number and nature of its parameters – and communication between the patch and the digital audio workstation through the meta-plugin – so that the digital audio workstation or the plugin can interact with the patch and reciprocally the patch with the digital audio workstation – for example, to send and receive digital audio signals but also MIDI events, or to control parameters.

3.1 Plugin properties definition

To ensure optimal functioning within digital audio workstations, audio plugin properties are defined using a text file named after the meta-plugin and the main patch[13]. This properties file follows a syntax relatively similar to the FUDI[14] protocol where each line corresponds to a new statement and ends with a semicolon. So each statement can be used to define or to complete a feature or a property of the plugin. In order to ensure the proper functioning of the plugin, the console displays a warning if some properties have been wrongly defined, duplicated or omitted. Although in practice there is no hierarchy, these properties of the plugins can be organised according to categories.

First, properties are used to define general information, which is needed to generate the audio plugin and for it to function properly in digital audio workstations; such as the type of the plugin – to inform the user which meta-plugin to use for generating the plugin[15] – or the compatibility number – that corresponds to the version of the plugin with which the patch has been created and that is used to ensure compatibility with the patch[16].

[11]libpd is wrapper that turns Pure Data into an embeddable audio library libpd.cc (accessed February 2018).

[12]JUCE is an application programming interface oriented towards digital audio signal processing distributed by ROLI company juce.com (accessed January 2018).

[13]The documentation offers a full explanation on how to create and to use the properties file.

[14]FUDI is a network protocol invented by Miller Puckette for Pure Dat a en.wikipedia.org/wiki/fudi (accessed February 2018).

[15]The types can be effect or instrument and if the meta-plugin is not coherent with the type defined in the properties file, then the console displays a warning.

[16] If the version of the meta-plugin used is inferior to the compatibility version, then the console displays a warning.

Properties can also be used to activate extra functionalities that are originally deactivated for reasons of efficiency, for example if the audio plugin needs to handle MIDI events, play head information, or key event.

An important part of the options is focused on audio signal processing, like latency, which is implied by the plugin when using an FFT for example, or audio tail length – the time during which the output still produce audio after the input has been stopped – for reverberation effect for example. But the main audio property defines the audio buses supported by the plugin – the audio input and output configurations. The different audio plugin formats support dynamic audio buses layout, as well as multichannel and side-chains. Camomile offers a syntax that helps using these features. Thereby, an audio plugin can support several layouts of multichannel buses, for a sound spatialisation plugin for example, or the enabling or disabling of side-chains, for a compressor for example, so the process of the patch can be adapted depending on the buses layout submitted by the digital audio workstation[17] .

Another important aspect of an audio plugin is related to the control protocol of its state by the digital audio workstations using parameters. A parameter represents one or several aspect of the audio engine with a numerical value – that can be saved, restored, automated, etc. by the digital audio workstation. Camomile offers the possibility to create highly-developed parameters with names, labels, ranges of values, steps and so on to improve their use, their representation and their meaning.

At last, properties are used to define additional attributes which are not necessary for the proper functioning of the plugin, but which can be essential to its ease of use, such as the description displayed by the plugin in its tab on the auxiliary window, the reference to an image file that the plugin displays as background of the graphical interface or an option to automatically reload the patch when it has changed – useful during the creation process.

[17]All the audio buses layouts supported by the audio plugin must be defined at the first loading, so to support dynamic changes but also some specificities such as extra buses for side-chaining, this property must be pre-defined. More complex cases, like when the additional buses configurations depend on the main bus configuration, still need to be investigated. Furthermore, future versions could support a text description of the buses, like quadraphonic or ambisonic, to improve the specification of the configurations accepted by the plugin.

3.2 Communication between the plugin and the patch

Communication between the patch and the digital audio workstation through the meta-plugin is, for its part, ensured via a set of conventions and practices. First of all, the messages sent and received by the meta-plugin to and from the patch are synchronised sequentially to the audio thread depending on an order defined arbitrarily[18]. Overall, the meta-plugin first sends its messages, such as parameter values or MIDI events, then it processes the patch's digital audio chain, and finally it retrieves the messages sent from the patch to its address[19].

As defined by libpd, in a similar way to the applications PdParty or PdDroidParty, most of the communication can be handled within the patch using native objects: the objects *adc~* and the *dac~* for the audio signals[20], the objects *notein*, *noteout*, *ctlin*, *ctlout* and so on for the MIDI events and the objects *key*, *keyup* and *keyname* for the keyboard events. Furthermore, using a 'bus' receiver makes it possible to retrieve information about the current audio buses layout of the plugin when the audio starts – for example, to adapt the audio process. Using a 'play head' receiver during processing can be used to retrieve information such as tempo, time signature of the current bar, current position of the play head and so on, which could be indispensable for some synthesisers.

[18]Even if each Pure Data instance – each meta-plugin – can run in a separate thread, an instance can only be modified by only one thread, otherwise the behaviour is undefined and so potentially different from the one offered by the Pure Data application.

[19]The specific order of each message according to its type is fully explained in the documentation.

[20]In order to use directly the patch as an abstraction within the Pure data application, replacing the objects *adc~* and *dac~* by the objects *inlet~* and *outlet~* has been considered. Nevertheless, this solution didn't seem desirable because it prevents to receive or to send the audio signals from inside subpatches or abstractions and it makes more complicated the dynamic patching that could be useful to adapt the process to the audio buses layouts submitted by the digital audio workstations. Furthermore, the implementation of the meta-plugin becomes much more complex implementation especially to manage the audio block size in the main patch that would be no more necessarily predefined.

The patch is also used to define the plugin's graphical user interface. The bounds of the patch's visible area within the plugin interface is defined by the properties of the patch when using it as a graphical abstraction[21]. The graphical objects – such as the number box, the slider, the comment and so on – inside the area will be recreated by the plugin's interface and directly linked to their original object in such a way that no additional operation is necessary to communicate with the patch via the plugin (see Figure 4).

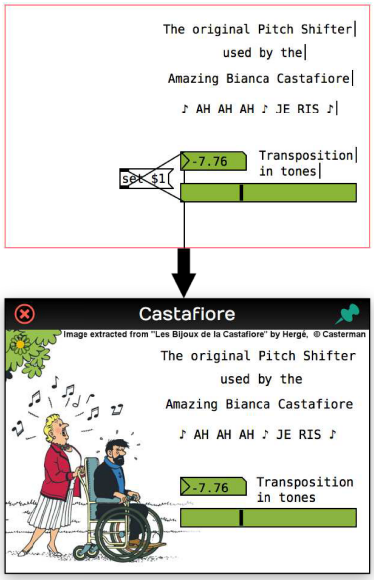


Figure 4: The part of the patch Castafiore that defined the graphical interface of the eponym plugin.

Inspired from the approach defined in PdParty, the plugin offers a replacement for the native Pure Data mechanisms such as the one offered by the objects *openpanel* and *savepanel* by displaying a dialog window to select files from the disk.[22]

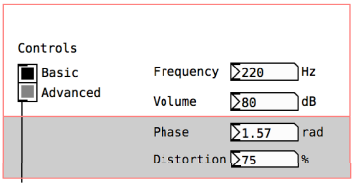


Figure 5: An example of a patch that illustrates the dynamic graphical interface of the plugin.

Moreover Camomile makes it possible to completely and dynamically redefine the graphical interface by changing its size, the objects and so on – a useful feature used to adapt the interface to the modes and requirements of the audio plugin (see Figure 5).

A specific aspect to the implementation of audio plugins is parameter management. Parameters values can be received using a 'param' receiver. But one could also want to modify the value of a parameter with the graphical interface – to record automations for example. This operation requires first to notify the digital audio workstation that the parameter will change, then to change the value – once or several times – and finally to notify the digital audio workstation that parameter modification has ended. If there are several graphical user interfaces and several parameters, these transactions become complicated to implement. Nevertheless the distribution offers a set of abstractions which can be directly connected to graphical objects to facilitate the setup of such mechanisms (see Figure 6).

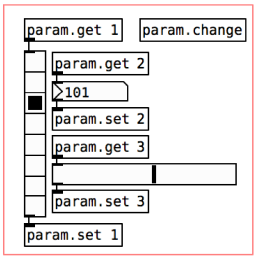


Figure 6: An example of a patch that shows how to link three graphical objects with three parameters. The abstractions *param.get* and *param.set* manage respectively the getting and the setting of each parameter and the abstraction *param.change* manages the global distribution of the messages within the patchs.

At last, when saving the digital audio workstation project or creating a new program, the meta-plugin automatically stores parameter states in an XML file. In addition, Camomile offers a mechanism to save and recall additional data via the patch that can be represented by parameters such as a system path or the content of an audio buffer.

With all these functionalities, Camomile offers the ability to create complex audio plugins with a large set of advanced features. Nevertheless, many developments can still be considered.

[21]The graphical abstractions are activated with the Graph-On-Parent option.

[22]A similar mechanism has been implemented to display the floating window of the object *array*. And the same feature is considered for the object *text*.

4 Perspectives

First of all, some native features of Pure Data relative to the graphical user interfaces are missing or can be improved, such as the implementation of the graphical object VU-meter or the improvement of the rendering of the graphical object labels. In order to get closer to standard plugins, it would be interesting to investigate the use of external images, which would replace drawing the graphical objects – using an image for the background of the object and one or more images for the foreground depending on the type of interface, it would be really easy to customise its representation. Another approach to offer more possibilities would be to implement the graphical part of the data structure of Pure Data [5], to draw and interact with more personal and original interfaces.

Support for external libraries is also very in demand by users. This feature could be a great improvement, this way someone could use an external as the audio processor of the plugin – optimizing the processes – and the patch as the interface with the meta-plugin and the digital audio workstation. Unfortunately, dynamic library loading seems to be restricted by the way Pure Data is embedded inside the meta-plugin[23] and by the fact that some of them are not directly compatible with multiple instance support[24]. Thus, direct integration of the most widespread libraries like the Cyclone [6][25] or the Zexy[26] libraries inside the plugin is considered. Nevertheless, this requires checking the compatibility of all objects and these dependencies could make Camomile difficult to maintain[27].

Offering a version of the plugin in the LV2[28] format is also considered, however the differences with the VST and Audio Units formats raise compatibility problems that still need to be explored.

5 Acknowledgements

The author would like to thank the whole community of Pure Data and libpd developers, especially Miller Puckette and Dan Wilcox, for their advice and explanations as well as the users of Camomile for their great feedback and suggestions. The author would like to also acknowledge the CICM and especially Alain Bonardi and Eliott Paris for their interest in the project, their comments and their advices.

References

[1] M. Puckette. 1997. Pure Data: Another Integrated Computer Music Environment *Proceedings of the Second Intercollege Computer Music Concerts,* p. 37-41, Tachikawa, Japan.

[2] P. Guillot. 2018. Camomile, Enjeux et Développements d'un Plugiciel Audio Embarquant Pure Data. *Actes des Journées d'Informatique Musicale*, Amiens, France.

[3] P. Brinkmann, P. Kirn, R. Lawler, C. McCormick, M. Roth and H.-C Steiner. 2011. Embedding Pure Data with libpd. *Proceedings of the Pure Data Convention*, Weimar, Germany.

[4] D. Wilcox. 2016. PdParty: An iOS Computer Music Platform using libpd. *Proceedings of the Pure Data Convention*, New York, USA.

[5] M. Puckette. 2007. Using Pd as a score language. *Proceedings of the International Computer Music Conference*, p. 184-187, Göteborg, Sweden.

[6] A. Torres Porres, D. Kwan and M. Barber. 2016. Cloning Max/MSP Objects: A Proposal for the Upgrade of Cyclone. *Proceedings of the Pure Data Convention*, New York, USA.

[7] I. I. Bukvic, A. Gräf and J. Wilkes. 2017. Meet the Cat: Pd-L2Ork and its New Cross-Platform Version "Purr Data". *Proceedings of the Linux Audio Conference*, Saint-Étienne, France.

[23] The reason of this restriction still need to be investigated.

[24] If a library goes beyond the 'public' API of Pure Data and uses internal structures that deal with the multiple instance support, some problems may occur.

[25] github.com/porres/pd-cyclone (accessed March 2018).

[26] The Zexy library is developed by IOhannes m zmölnig puredata.info/downloads/zexy (accessed March 2018).

[27] Using a monolithic approach by including the libraries [Bukvic & al., 2017] is one of the causes of the abandonment of the Pure Data variant, Pd-extended, puredata.info/downloads/pd-extended (accessed January 2018) originally maintained Hans Christoph Steiner.

[28] The LV2 format by D. Robillard is the successor of the LADSPA plugin format lv2plug.in/ns (accessed March 2018).

Ableton Link – A technology to synchronize music software

Florian Goltz
Ableton AG
Schönhauser Allee 6-7
10119 Berlin,
Germany

Abstract

Ableton Link is a technology that synchronizes musical beat, tempo, phase, and start/stop commands across multiple applications running on one or more devices. Unlike conventional musical synchronization technologies, Link does not require master/client roles. Automatic discovery on a local area network enables a peer-to-peer system, which peers can join or leave at any time without disrupting others. Musical information is shared equally among peers, so any peer can start or stop while staying in time, or change the tempo, which is followed by all other peers.

Keywords

Audio, Network, Peer-to-peer, Time, Synchronization

1 Overview of Common Sync Technologies

Synchronizing media devices has been a challenging task for a number of decades. This section provides an overview on existing standards and approaches. No single sync technology has been able to establish itself as a universal standard. Depending on the context and actual requirements of a scenario, one ore more of the existing standards are used.

1.1 SMPTE

In 1967, the Society of Motion Picture and Television Engineers released a standard for the synchronization of media systems [Rees, 1997]. In this standard, time is described as an absolute value separated into hour, minute, second, and frame. A master machine generates the clock signal and sends it to a variable number of clients. The clock signal can be sent across a dedicated channel or embedded as metadata within the media. SMPTE is still widely used today for synchronization of video and audio systems.

1.2 AES/EBU

The Audio Engineering Society and the European Broadcasting Union published the AES/EBU standard in 1985 [Laven, 2004]. It provides the same information as SMPTE but is optimized for audio equipment. AES/EBU can use a wide variety of transports, from XLR cables to S/PDIF.

1.3 MTC

Midi Time Code was released in 1987 and embeds the same data as AES/EBU, but is optimized to be transported via MIDI sysex messages. [Meyer and Brooks, 1987]

1.4 MIDI Beat Clock

Unlike the above standards, MIDI Beat Clock is a tempo-dependent signal. It consists of 24 pulses per quarter note. This is probably the most widely used sync signal in music software and hardware today.

1.5 JACK Transport

The Jack Audio Connection Kit Transport API [JackAudio, 2014] allows sharing sample accurate timecode between its clients. While Jack itself acts as a timecode master for its clients, Jack Transport allows all its clients to start and stop transport or seek the timeline. Using NetJack [Hohn et al., 2009], it is possible to connect multiple clients on a local area network to a master. This way transport controls can be shared among multiple applications running in different computers. NetJack however only allows audio output on the master machine.

1.6 OSC Sync

An OSC-based synchronization scheme has been proposed [Madgwick et al., 2015] which has a master send clock messages on a regular basis. This scheme targets networked use cases such as laptop orchestras.

1.7 Summary

All of the above technologies share the common approach of having a master provide a clock

signal to a number of clients, though the representation of time varies. Setting up such systems involves routing the signal from the master to the clients and/or configuring the master and clients to send and receive via the appropriate channels. In a master/client system, the master application is usually the only one that has control over tempo and transport state. As soon as the master fails, or the channel breaks, the clients are in an undefined state.

2 Link Design Criteria

Three criteria drove the development of Link:

- Remove the restrictions of a typical master/client system.
- Remove the requirement for initial setup.
- Scale to a wide variety of music applications.

These goals are achieved by designing a peer-to-peer system that sends multicast messages on a local network. Parameters are controlled mutually and all peers converge to the same shared timing information. The timing information is designed in such a way that peers with different capabilities such as a one-bar-looper or a fully-featured DAW can map the shared information to their specific needs. If peers are connected to a Local Area Network there is no further setup required.

3 Multicast Discovery

Link peers communicate using UDP multicast messages in a local area IP network. Each peer regularly sends messages that contain its unique peer ID and a snapshot of its current musical time. This way all peers and their state is known by each peer on the local network.

The incoming messages are processed by every receiver according to the same set of rules. If a receiver decides to adapt the timing information it has received, it updates its timing information and broadcasts accordingly. As a result of this peer-to-peer messaging, all peers on a network always converge to the same shared description of the current musical time.

Link regularly scans the available network interfaces on the host computer. When a new interface is discovered, multicast messages are sent and received on it as well. As a result, a Link peer that is connected to multiple networks can act as a relay: when the timing information from incoming messages on one interface is adapted, it is sent out on all available interfaces. This way, timing information is shared with peers that are not directly connected.

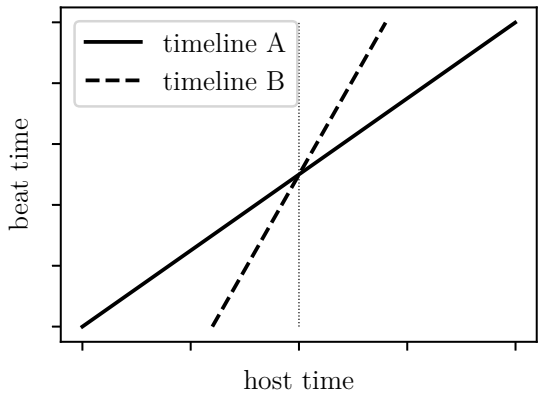


Figure 1: The new timeline B crosses the old timeline A at the host time of the tempo change

4 Timeline

Link describes the timing information of the session at a point in time as a tuple of three values: the *host time* that the hardware provides, a corresponding *beat time*, and a *tempo* that describes the change of beat time over host time. This tuple of values is referred to as a *timeline*. The system's beat time for a given host time and vice-versa can be calculated with a simple linear equation: `BeatTime/HostTime = Tempo`

When a peer intends to change the tempo at a specific host time, it creates a new timeline, describing a linear equation crossing the desired time point, and shares it with the network. When initializing Link, each peer creates such a timeline and immediately shares it with the network. This timeline then gets either adapted by other peers on the network, or the peer adapts a timeline it is receiving.

5 Host Time

Desktop operating systems usually provide calls that allow applications to ask for the current host time. Examples are `clock_gettime()` [IEEE, 2008], `mach_absolute_time()` [Apple, 2005] or `QueryPerformanceCounter()` [Microsoft, 2001]. The time stamps provided by those calls are based on information that the CPU or specialized hardware provides. Their quality can differ significantly in terms of accuracy and reliability. Additionally, the speed of the clock may depend on factors such as temperature and thus vary

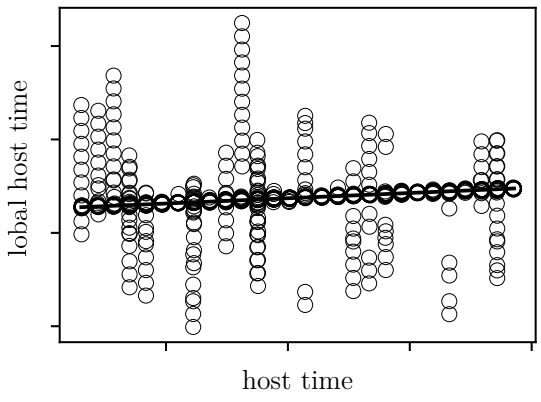


Figure 2: Measuring global host time against local host time in bursts

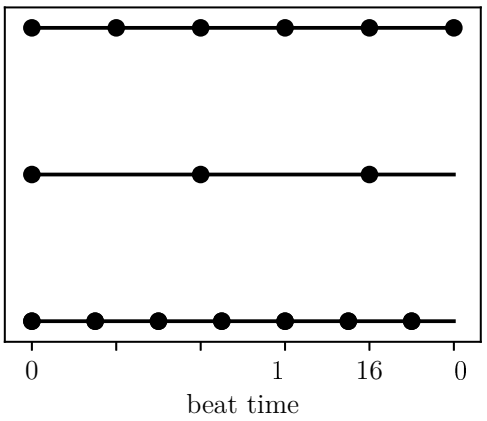


Figure 3: Alignment of timelines with quanta of 4, 8 and 3 beats

over time.

To be able to derive the session's beat time from the current host time, it is important that a peer has accurate knowledge of the system's host time. Some audio APIs provide accurate timing information in the audio processing callback. On other systems, it is necessary to query the systems' host time in the audio callback and filter it to get reliable information. The reference time Link uses is the "host time at speaker", which refers to the time the audio is actually perceived by the listener. To calculate this, software and hardware latencies must be incorporated into the host time provided by the system.

6 Global Host Time

Link establishes a reference host time that is shared between all peers in a session. This is referred to as the *global host time*. When a peer initializes Link and starts the initial timeline, its own host time is used as the reference. Every peer joining the session uses ping-pong messaging to calculate the offset of its own host time against this reference time. The result of this measurement, is a function that can convert the local host time of the peer to the global host time and vice versa. `globalHostTime = XForm.hostToGHost(localHostTime)`

As soon as a peer knows the global host time, it can function as a measurement endpoint for other peers. As a result, the peer that originally founded the session can leave, while the global host time is still maintained. Peers regularly measure their host time's offset to the global host time to compensate for speed variations.

7 Quantum

As mentioned above, one of the requirements for Link is to scale to music applications with different capabilities. This means it should work for applications that have different representations of musical time, e.g., loopers that only provide a simple one bar loop, or full featured DAWs that sequence a beat timeline and support different musical measures.

Link takes the approach of allowing each client to map the shared timeline to its own purpose, e.g., a looper can map Link's timeline to a position within its loop by calling `phaseAtTime(localHostTime, quantum)`. The quantum provided by the client describes the alignment grid in beats. A looper with a one bar loop in a 4/4 measure would provide a quantum of 4. Link also provides `beatAtTime(localHostTime, quantum)` which provides a monotonic timeline in a way that would typically be used by a sequencer.

Link guarantees that clients using the same quantum are phase synchronized. Peers with different quanta can form a polyrhythmic Link session, e.g., a peer using a quantum of 3 and another peer using a quantum of 4 would be share a downbeat every 12 beats.

8 Transactional API

Link provides lock-free `capture()` and `commit()` functions to be used in the audio thread, and a similar thread-safe pair of functions to be used in other threads.

The capture functions provide a snapshot of the Link session. This can be used to align the

client's audio to the shared timeline. In case the client wants to change the timeline, e.g., to change the tempo, the captured state can be modified and committed back to Link using the commit function. The new state will then be sent to the network and merged with the other peers' states.

9 Resources

Link is available as a header only C++11 library. It is dual licensed under the GNU-GPL and a proprietary license. The source code is currently available at `http://github.com/ableton/link`. Explanation of the concepts used in Link and technical documentation on the API can be found at `http://ableton.github.io/link`.

10 Conclusions

Existing technologies to synchronize music devices, as described in Section 1, are all based upon a master/client communication protocol. It is the master's responsibility to broadcast a signal according to the specification. The clients receiving the signal are dependent on the communication channel not being interrupted.

Link introduces a different approach to synchronize music devices. It creates a peer-to-peer network where all peers share a global time reference and a beat timeline. Any peer can introduce changes to the timeline in order to change the state of the session. To establish and maintain the shared state, it is important that all peers follow the same set of rules. In this sense, Link is not just a communication protocol, but a set of rules for multiple actors to create a shared musical session.

References

Apple. 2005. `https://developer.apple.com/library/content/qa/qa1398/_index.html`. Accessed: 2018-03-06.

Torben Hohn, Alexander Carôt, and Christian Werner. 2009. Netjack - Remote music collaboration with electronic sequencers on the Internet. LAC 2009, `http://lac.linuxaudio.org/2009/cdm/Saturday/22_Hohn/22.pdf`. Accessed: 2018-04-30.

IEEE. 2008. POSIX 1003.1-2008. `http://pubs.opengroup.org/onlinepubs/9699919799/functions/clock_getres.html`. Accessed: 2018-03-06.

JackAudio. 2014. `http://www.jackaudio.org/files/docs/html/transport-design.html`. Accessed: 2018-04-30.

Philip Laven. 2004. Specification of the digital audio interface. `https://tech.ebu.ch/docs/tech/tech3250.pdf`. Accessed: 2018-03-06.

Sebastian Madgwick, Thomas Mitchell, Carlos Barreto, and Adrian Freed. 2015. Simple synchronisation for Open Sound Control. `http://eprints.uwe.ac.uk/26049/1/03FinalSubmission.pdf`. Accessed: 2018-03-06.

Chris Meyer and Evan Brooks. 1987. MIDI Time Code and cueing. `https://web.archive.org/web/20110629053759/http://web.media.mit.edu/~meyers/mcgill/multimedia/senior_project/MTC.html`. Accessed: 2018-03-06.

Microsoft. 2001. Acquiring high-resolution time stamps. `https://msdn.microsoft.com/en-us/library/windows/desktop/dn553408`. Accessed: 2018-03-06.

Philip Rees. 1997. Synchronisation and SMPTE timecode. `http://www.philrees.co.uk/articles/timecode.htm`. Accessed: 2018-03-06.

Software Architecture for a Multiple AVB Listener and Talker Scenario

Christoph Kuhr and **Alexander Carôt**
Department of Computer Sciences and Languages, Anhalt University of Applied Sciences
Lohmannstr. 23, 06366 Köthen,
Germany,
{christoph.kuhr, alexander.carot}@hs-anhalt.de

Abstract

This paper presents a design approach for an AVB network segment deploying two different types of AVB server for multiple parallel streams. The first type is an UDP proxy server and the second server type is a digital signal processing server. The Linux real time operating system configurations are discussed, as well as the software architecture itself and the integration of the Jack audio server. A proper operation of the JACK server, alongside two JACK clients, in this multiprocessing environment could be shown, although a persisting buffer leak prevents significant jitter and latency measurements. A coarse assessment shows however, that the operations are within reasonable bounds.

Keywords

AVB, JACK, signal processing, public internet, multimedia streaming

1 Introduction

1.1 Soundjack and fast-music

Soundjack [1] is a realtime communication software that establishes up to five peer to peer connections. This software was designed from a musical point of view and first published in in 2009 [2]. Playing live music via the public internet is very sensitive to latencies. Thus, the main goal of this application is the minimization of latencies and jitter. The goal of the research project fast-music, in cooperation with the two companies GENUIN [3] and Symonics [4], is the development of a rehearsal environment for conducted orchestras via the public internet. 60 musicians and one conductor shall play together live. Further field of research is the transmission of low delay live video streams and motion capturing of the conductor.

1.2 Concept for a Realtime Processing Cloud

A specialized and scalable server infrastructure is required to provide the realtime streaming requirements of this research project. The service time property of an Ethernet frame arriving on a serial network interface at the wide area network (WAN) side of this server cloud, is of paramount importance for the software design. During the service time of a single UDP stream datagram, no concurrent stream datagrams can be received. Thus, the latencies of all streams arriving on such an interface are accumulated.

In addition to connecting the 60 streams to each other, the Soundjack cloud provides digital signal processing algorithms for audio and video streams. Digital signal processing is computationally expensive and may cause unwanted latencies. Thus, a GPU based signal processing in realtime will be investigated in this research project as well. A basic and scalable concept to address these two requirements is shown in fig. 1.

Audio Video Bridging / Time-Sensitive Networking (AVB / TSN) enables computer networks to handle audio and video streams in realtime. AVB is a set of IEEE 802.1 industry standards, operating on layer 2 of the OSI model [5].

- IEEE 802.1AS [6]
  Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks
- IEEE 802.1Qat [7]
  Virtual Bridged Local Area Networks - Amendment 14: Stream Reservation Protocol (SRP)
- IEEE 802.1Qav [8]
  Virtual Bridged Local Area Networks - Amendment 12: Forwarding and Queueing Enhancements for Time-Sensitive Streams

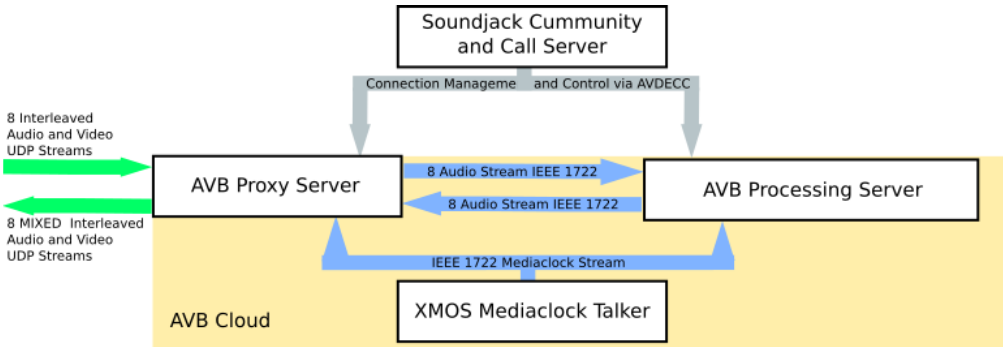


Figure 1: Soundjack Realtime Processing Cloud Concept

- IEEE 1722 [9]
  IEEE Standard for Layer 2 Transport Protocol for Time-Sensitive Applications in Bridged Local Area Networks
- IEEE 1722.1 [10]
  IEEE Standard for Layer 2 Transport Protocol for Time-Sensitive Applications in Bridged Local Area Networks

AVB extends a generic Ethernet computer network by the means of synchronization, resource reservation and bandwidth shaping. This way lower latencies and jitter, the avoidance of packet bursts and bandwitdh shortage are addressed.

AVB networks require special hardware for timestamping Ethernet frames with seperate bandwidth shaped transmission queues for AVB traffic. The Intel corporation provides the I2XX Series of NICs with the open source Open-AVB [11] driver.

Two server types are required for the Soundjack cloud, an AVB proxy server and an AVB processing server. Both server types are connected to the same AVB network segment. Each server is also connected to a non-AVB network segment, together with a Soundjack session server, which acts as an IEEE 1722.1 AVDECC controller endpoint. IEEE 1722.1 AVDECC traffic is not necessarily time-sensitive, thus a non-AVB network segment is used for command and control purposes. The Soundjack session server also provides the online services to the Soundjack client software and handles the connection management of public internet streams, establishes peer to peer and client-server connections.

All AVB servers are registered for a mediaclock stream, which is supplied by an XMOS/Atterotech development board [12]. The mediaclock stream maintains a constant mediaclock to synchronize the packet transmission times of the AVB servers. Without such a synchronization, each server would depend on the precise clock of an audio interface hardware, the CPU clock indicates too much jitter, which in turn would also require a central synchronization mechanism to provide a fully mediaclock-synchronized network segment.

2 Software Requirements for a Multiple AVB Listener and Talker

The AVB server software requires a proper configuration of the operating system and the AVB hardware support, to use the timestamping, bandwidth reservation and shaping. A multiprocessing design, as shown in fig. 2, takes care of all aspects required for multiple independent AVB talkers and listeners.

2.1 Operating System

The ability of Linux to communicate with raw sockets [13, p. 655] and also to be patched to operate in realtime mode, makes it the operating system of our choice. We decided to use the Linux Mint distribution release 18 Sarah, which is based on Ubuntu/Debian. Linux Mint 18 uses the Systemd init process, which makes it easier to dynamically handle OS services.

AVB requires three background services. A gPTP daemon, a MAAP daemon and a MRP daemon. Each requires super user permissions for raw socket communication.

In addition to the background services, a one-time-task to unload the generic Intel e1000/IGB kernel modul and replace it with the Open-AVB AVB IGB kernel module is required. The AVB talkers running on the system need the hardware transmit queues of the Intel I210 Ethernet NIC to be redirected to the bandwidth shaper transmit queues, so that the I210 NIC might use the FQTSS mechanism for enqueueing AVTP

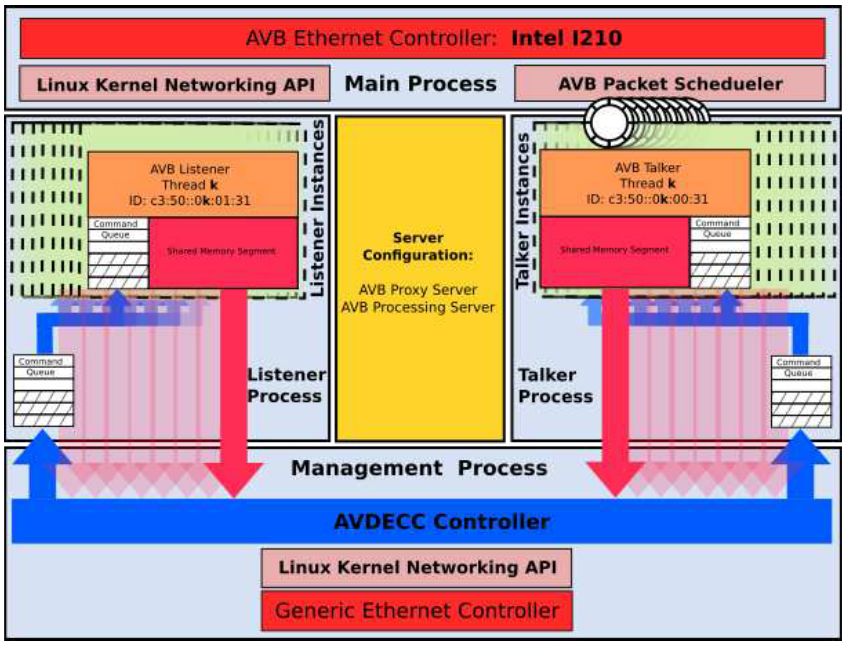


Figure 2: General AVB Server Architecture (MRP, Talker and Listener Processes)

packets from the DMA memory.

The Open-AVB project provides Shell scripts to setup those services.

The Linux kernel may be patched, configured and compiled for realtime operation [14]. A Linux realtime kernel with either the SCHED_FIFO or the SCHED_RR scheduling enabled, handles CPU tasks based on their priorities. A task requesting the CPU, that is scheduled by either scheduler, has a latency solely depending on tasks with a higher or equal priority. Examples for tasks that can still delay the execution of high priority task are DMA bus mastering, ACPI power management, CPU frequency scaling and hyperthreading techniques [15]. These interfering tasks have to be taken into account and carefully tuned, when configuring a realtime Linux system such as:

- Using POSIX realtime mutexes instead of spinlocks.
- Interrupt handlers are moved to the userspace process.
- Avoid priority inversion by priority inheritance.

Another scheduler was introduced in the Linux kernel version 3.14 [16], SCHED_DEADLINE. SCHED_DEADLINE is based on the earliest deadline first (EDF) scheduling enhanced by the constant bitrate server algorithm (CBS). The EDF scheduling with CBS was specifically developed for multimedia applications [17] [18]. This scheduler does not rely solely on the priority, but assigns an absolute deadline and a budget to each task. At each CPU cycle a specific budget is available to the scheduler. If a task is out of budget it is preempted to ensure the execution of another task. With EDF scheduling however, it is important to avoid deadlocks that result from CPU over-utilization. The kernel has an inbuilt mechanism to minimize the risk, by disabling CPU affinity for EDF-scheduled tasks.

The kernel we use in this project is mainstream release 4.8.6 with the realtime patch 4.8.6-rt5, which takes care of the above mentioned realtime mutexes, userspace interrupt handlers and priority inheritance. Besides patching the kernel for realtime operation, several optimization steps are performed. Kernel modules for unnesscary hardware support, e.g. most network interface drivers and peripheral device drivers were removed. Furthermore, the kernel module for NVidia's proprietary graphic adpater and CUDA driver was patched to be used with a realtime kernel.

The optimization of the OS mainly concerns AVB. Since the AVB implementation requires the Direct Memory Access (DMA) [19, p. 412] memory for operation, it is required to use the Memory Management Unit (MMU) in soft mode in `/etc/default/grub`, so that direct

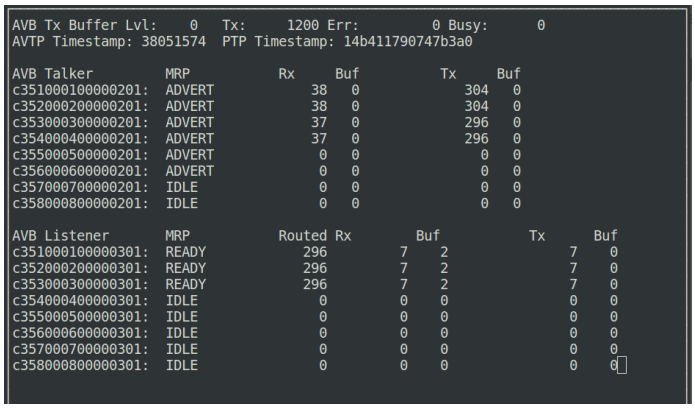


Figure 3: NCurses Shell User Interface

hardware addresses are used instead of a virtual address space, when necessary. The parameter `iommu=soft` prevents the usage of the IOMMU when communicating with the IGB DMA memory, but allows the Focusrite Solo to use it.

```
GRUB_CMDLINE_LINUX_DEFAULT="text iommu=soft"
```

It is also necessary to take care of the priorities for the interrupts, because it has a major influence on the task scheduling. The most important interrupt is the one of the NIC providing the mediaclock stream followed by the interrupts for the USB audio interface device. Linux provides the `/etc/default/rtirq` script to enforce those priorities, which are defined by the order of the RTIRQ_NAME_LIST attributes.

```
RTIRQ_NAME_LIST="enp4s0 enp4s0-TxRx-0
enp4s0-TxRx-1 enp4s0-TxRx-2 enp4s0-TxRx-3
snd usb snd_usb_audio enp2s0 i8042"
```

Further optimizations, e.g. the deactivation of the swappiness or configuring limits in a range a user might operate in, aim to increase the realtime responsiveness of the operating system as a whole [20]. Finally, the system memory is unlocked and realtime priority is assigned to the user-space application.

2.2 Software Architecture

The AVB server software is running five processes in parallel, to distribute processing time more evenly over the available CPU cores. The parent process forks four children and operates afterwards as mediaclock receiver and AVTP packet scheduler. The first child process is the management process and runs the AVDECC controller instance. All AVDECC operations are sent via command queues to a talker or listener instance, except the creation of talkers and listeners themselves. The creation of talkers and listeners is not covered by the AVDECC standard, thus a vendor specific command and response [10, p. 151] has been implemented. Status variables and stream states are written to and accessed by a POSIX shared memory segment. The management process also provides a terminal user interface, as shown in fig. 3, to monitor counters and states of the AVB streams in realtime. Talker and listener endpoint thread instances are created by the talker and the listener processes, respectively. The fifth process is the MRP process, handling all of the stream reservations of the endpoint instances. Figure 2 shows the general AVB server software architecture.

A talker instance reads audio and video data from some process and puts the data into the payload of an AVTP packet, which is then pushed to its circular buffer. The circular buffer is subsequently and continuously read by the AVTP packet scheduler. If the server is configured as AVB proxy server, it receives UDP streams from the assigned Soundjack client and thus provides audio and video data. If the server is configured as AVB processing server, audio and video data is provided by a realtime signal processing application.

A listener instance receives an AVTP stream from the Linux kernel network API. AVTP packets are filtered based on the destination MAC address and the ether type field with a Berkley Packet Filter (BPF) [21] [13, p. 705] mask. AVTP packets that match the filter expression are pushed to the circular buffer of the respective listener instance.

There are some aspects the software needs to take care of to make use of the realtime kernel. First of all, the memory required for dy-

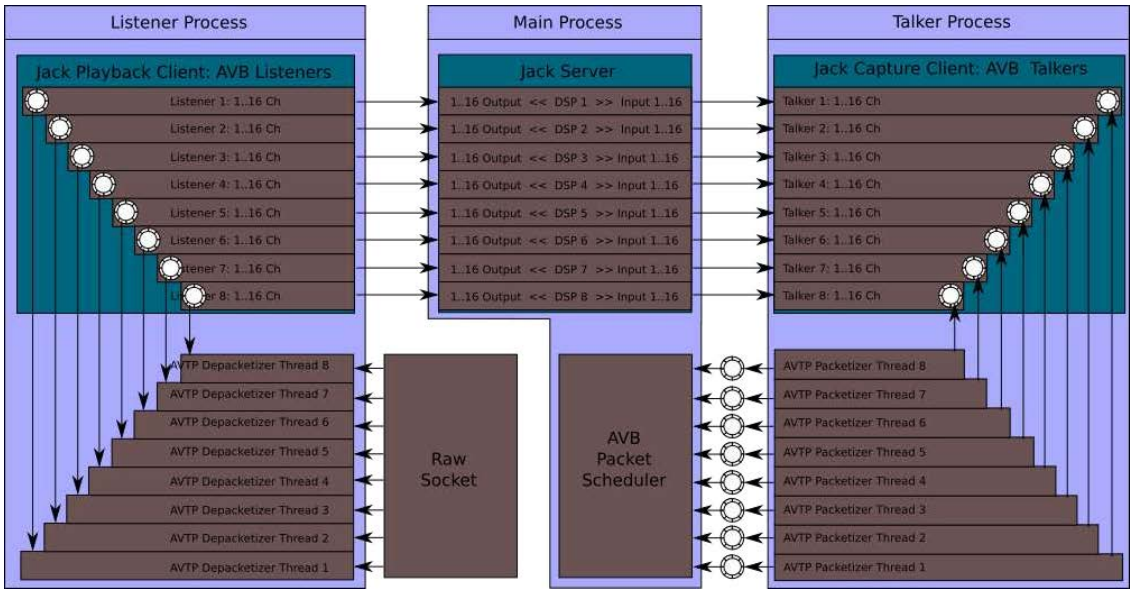


Figure 4: JACK Server and Clients

namic allocations at runtime has to be locked at application start. Otherwise, memory allocations would always be freed after their use and the application eventually crashes, due to memory page faults [22]. Secondly, locks have to be used to prevent the preemption of the task in time critical segments of code. The software also needs to be assigned a proper task priority, so that its scheduling takes place within the required deadlines.

2.3 Server Configurations

In case of the AVB proxy server configuration, the AVTP stream is converted to an UDP stream that is returning to a Soundjack client. In the case of the AVB processing server configuration, the audio and video data is provided to the realtime signal processing application.

2.3.1 AVB Proxy Server

IP packets are forwarded with best effort in the public internet. The Soundjack cloud in contrast, provides a fully managed and controlled AVB Ethernet network. The FQTSS amendment to IEEE 802.1Q prevents bursty traffic by the means of a credit-based bandwidth shaper, inside of the Soundjack cloud network segment. A proxy server is used as a wave trap to devide large and erratic UDP datagrams into more and smaller AVTP packets, that maintain a constant inter packet gap. With the credit-based bandwidth shaper the AVTP packets can travel inside the cloud network segment in a deterministic way.

The AVB proxy server accepts and returns UDP streams from and to Soundjack users, that have been assigned by the session server. To keep the latency introduced by the service times of the Ethernet NIC low, only eight streams are assigned to an AVB proxy server.

The UDP streams received on the WAN interface need to be transmitted in the AVB network segment at a different bitrate with a different payloading. A UDP datagram of a Soundjack stream contains 256, 512 or 1024 Bytes of raw audio. Lower amounts of bytes occur in cases of compression according to the chosen compression ratios. The resulting AVTP stream is sent from the AVB proxy server to the AVB processing server, which processes the eight streams and sends them back as AVTP packets with the same, but processed payload. In the opposite streaming direction, the AVB proxy waits untill sufficient AVTP packets are in a listener's circular buffer, constructs an UDP datagram and sends it back to the client, where it came from.

2.3.2 AVB Processing Server

The AVB processing server receives the audio and video streams from the AVB proxy server with a constant packet rate of $8kHz$. It executes signal processing applications for audio and video data.

Conventional audio signal processing like compression or equalization can be integrated with existing Linux tools, such as JACK [23]. JACK provides a realtime processing environment that is required to execute some DSP al-

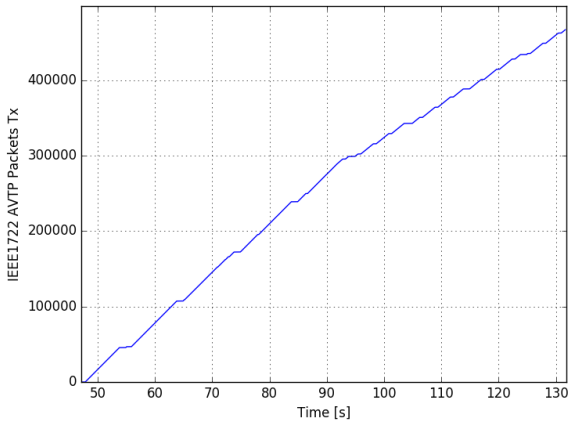


Figure 5: AVTP Stream Packets per Time

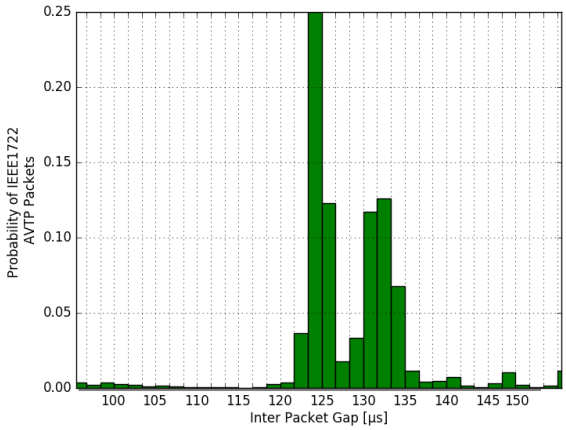


Figure 6: AVTP Stream Inter Packet Gap Probability Distribution

gorithms with LV2 plugins [24] or FAUST [25] applications. The design of the JACK server together with the two required JACK clients is shown in fig. 4. Before the other processes are forked, the main process starts the JACK Server. A Focusrite Scarlett Solo Gen2 [26] is used as audio interface hardware for the JACK server. Until now, JACK is running out of sync with the AVTP streams. After the forking of the listener and the talker processes, each process creates a JACK client. JACK ringbuffers are used by either client to communicate with the de-/packetizer threads respectively.

The JACK clients JACK ports are configured by the AVDECC process by means of the SET_STREAM_FORMAT [10, p.174] AEM command. JACK ringbuffers are created according to the channel count and sample format of the AEM command. This implies that channel count and sample format can only be changed before a AVB server session is established.

The listener AVTP depacketizer threads push audio samples from AVTP packets to the corresponding JACK ringbuffer, while the listener parent process pops the audio samples from the JACK ringbuffer and copies it to the JACK process graph. Audio samples are copied from the JACK process graph to the parent talker process, which in turn pushes the audio samples into the JACK ringbuffers of the talker AVTP packetizer threads.

The signal processing applications are connected to the talker and listener threads via JACK connections to the respective JACK clients.

Realtime audio production environments generally do not use graphics cards, as long as they are not involved in 3D rendering or video production processes. Thus, the graphics card is idle most of the time and can be utilized as an audio co-processor. Graphics card technologies made a lot of progress over the past years, which make modern graphics cards useable as co-processors for realtime signal processing [27]. More complex algorithms for processing audio and video data than the ones mentioned above shall be processed with a graphics card. Further applications such as a Viterbi decoder or virtual soundscapes and environments however, are still under development.

3 Evaluation and Discussion

For the evaluation of a general concept for a signal processing infrastructure no signal processing applications are applied yet, instead the JACK server creates loopback connections to allow the round trip transportation latencies and jitter to be tested. The current state of the AVB server application has a buffer leak, which leads to buffer overruns after ≈92 sec, as shown in fig. 5. The curve bends and the gradient decreases after this point, i.e. the IPG increases. This event marks the turning point between the two peaks exhibited the probability distribution of the transmitted AVTP packets shown in fig. 6 at 124μsec and 131μsec. Although the inter packet gaps of the AVTP stream are within reasonable bounds, the mean value of the PDF of 129.08μsec obviously cannot meet the defined inter packet gap for a SRP class A domain of 125μsec. The source of the buffer leak could not be determined yet.

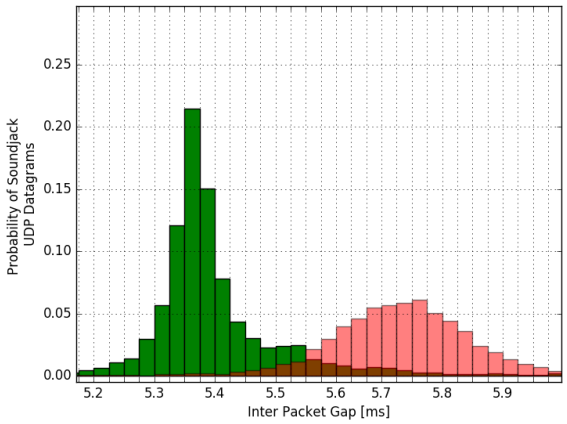


Figure 7: UDP Rx and Tx Inter Packet Gap Probability Distributions

An indicator for the source of the problem is shown in fig. 5. This figure shows the total amount of packets sent over time. The regions with a non-zero gradient show the constant flow of packets, while the gradients with value zero indicate some interrupt of the packet flow. This means that the talkers do not transmit for some period of time, which corresponds to the observation that the SRP states of the used switch ports toggle between listener ready and ask fail states.

Hence, no significant jitter and latency measurements could be done. Nonetheless, the magnitude of the observed end-to-end jitter and latency could be determined. The latency under this circumstances changes drastically when the buffers overrun from below 10msec to 300 − 600msec. Figure 7 shows the jitter of the Soundjack clients transmit UDP stream (green distribution) has a mean value of 5.35msec, which corresponds to 256 audio samples per UDP datagram. When leaving the Soundjack cloud, the Soundjack clients receive stream (red distribution) has a mean value of 5.75msec and a higher standard deviation.

The JACK server was running with 64 samples per period at 48kHz without causing xruns during the measurements.

4 Conclusions

The integration of the JACK audio server alongside two JACK clients into the multiprocessing software architecture of the AVB server went very well, although an already known but undocumented bug with libjackserver and libjack [28] required resolving. Only the feature to dynamically change the channel count of a Soundjack stream during the transmission had to be deactivated.
A buffer leak that could not be resolved yet, is accountable for an increasing round trip latency. The jitter and latency of the end-to-end UDP streams do not provide significant measurements yet, but the observed transmission behaviour is very close to the bounds defined in the AVB standards.

5 Future Work

The ongoing work is related to the localization of the buffer leak. Latencies and jitter can only then be evaluated in scenarios, where actual signal processing applications are applied to the audio streams. For the operation under heavy load with all AVB endpoints registered, the EDF scheduling has to be configured properly. It also might be neccessary to upgrade hardware components such as the CPU, since realtime computing by itself requires a lot of CPU utilization and leads to overhead by process context switching. Another item to be handled in the furture is the synchronization of the JACK server to the mediaclock stream.

6 Acknowledgements

fast-music is part of the fast-project cluster (fast actuators sensors & transceivers), which is funded by the BMBF (Bundesministerium für Bildung und Forschung).

References

[1] (2018, Apr. 23) Soundjack - a realtime communication solution. [Online]. Available: http://http://www.soundjack.eu

[2] A. Carôt, “Musical telepresence - a comprehensive analysis towards new cognitive and technical approaches,” Ph.D. dissertation, University of Lübeck, Germany, May 2009.

[3] (2018, Apr. 23) Genuin classics gbr, genuin recording group gbr. 04105 Leipzig, Germany. [Online]. Available: http://genuin.de

[4] (2018, Apr. 23) Symonics gmbh. 72144 Dusslingen, Germany. [Online]. Available: http://symonics.de

[5] H. Zimmermann, “Osi reference model - the iso model of architecture for open systems interconnection,” in *IEEE Transac-*

tions on Communications, Vol. 28, No. 4, Apr. 1980, pp. 425–432.

[6] *Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks*, IEEE Std. 802.1AS, Mar. 2011.

[7] *Virtual Bridged Local Area Networks - Amendment 14: Stream Reservation Protocol (SRP)*, IEEE Std. 802.1Qat-2010, Sep. 2010.

[8] *Virtual Bridged Local Area Networks - Amendment 12: Forwarding and Queuing Enhancements for Time-Sensitive Streams*, IEEE Std. 802.1Qav-2009, Jan. 2010.

[9] *Layer 2 Transport Protocol for Time-Sensitive Applications in Bridged Local Area Networks*, IEEE Std. 1722, May 2011.

[10] *Device Discovery, Connection Management, and Control Protocol for IEEE 1722 Based Devices*, IEEE Std. 17 221, Aug. 2013.

[11] A. Alliance. (2018, Apr. 23) Openavnu - an avnu sponsored repository for time sensitive network (tsn and avb) technology. [Online]. Available: https://github.com/AVnu/OpenAvnu/

[12] (2018, Apr. 23) Xmos ltd. / attero tech inc. [Online]. Available: http://www.atterodesign.com/cobranet-oem-products/xmos-avb-module/

[13] W. R. Stevens, B. Fenner, and A. M. Rudoff, *UNIX Network Programming, Vol. 1*, 3rd ed. Pearson Education, 2003.

[14] J. Kacur, "Realtime kernel for audio and visual applications," in *Proceedings of the Linux Audio Conference 2010*. Wittenburg, DE: Red Hat, Apr. 2010.

[15] (2018, Apr. 23) Howto: Build an rt-application. [Online]. Available: https://rt.wiki.kernel.org/index.php/HOWTO:_Build_an_RT-application

[16] (2018, Apr. 23) Linux programmer's manual sched(7). [Online]. Available: http://man7.org/linux/man-pages/man7/sched.7.html

[17] K. J. e. a. Ion Stoica, Hussein Abdel-Wahab, "A proportional share resource allocation algorithm for real-time, time-shared systems," in *Proceedings of the IEEE*, 1996.

[18] G. B. Luca Abeni, "Integrating multimedia applications in hard real-time systems," in *Proceedings of the 19th Real-Time System Symposium (RTSS 1998)*. Madrid, ESP: Scuola Superiore S. Anna, Pisa, Dec. 4–13, 1998.

[19] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005.

[20] J. JONGEPIER, "Configuring your system for realtime low latency audio processing," in *Proceedings of the Linux Audio Conference 2011*. ICTE department Faculty of Humanities, University of Amsterdam, 2011.

[21] S. McCanne and V. Jacobson, "The bsd packet filter: A new architecture for user-level packet capture," in *Presented at the 1993 Winter USENIX conference*. San Diego, CA: Lawrence Berkeley Laboratory, One Cyclotron Road, Berkeley, CA, Jan. 25–29, 1993.

[22] (2018, Apr. 23) Dynamic memory allocation example. [Online]. Available: https://rt.wiki.kernel.org/index.php/Dynamic_memory_allocation_example

[23] (2018, Apr. 23) Jack audio connection kit. [Online]. Available: https://jackaudio.org

[24] (2018, Apr. 23) Lv2 - open standard for audio plugins. [Online]. Available: http://www.lv2plug.in

[25] (2018, Apr. 23) Faust programming language. [Online]. Available: http://faust.grame.fr/

[26] (2018, Apr. 23) Focusrite audio engineering ltd. United Kingdom. [Online]. Available: https://us.focusrite.com/usb-audio-interfaces/scarlett-solo

[27] C. Kuhr and A. Carôt, "Evaluation of data transfer methods for block-based realtime audio processing with cuda," in *Proceedings of the 10th Forum Media Technology and 3rd All Around Audio Symposium*. St. Pölten, Austria: St Pölten University of Applied Sciences, Nov. 71–76, 2017.

[28] C. Kuhr. (2018, Mar. 06) Undocumented crash when using libjack and libjackserver #331. [Online]. Available: https://github.com/jackaudio/jack2/issues/331

Rtosc - Realtime Safe Open Sound Control Messaging

Mark McCurry
DSP/ML Researche
United States of America
mark.d.mccurry@gmail.com

Abstract

Audio applications which go beyond MIDI processing often utilize OSC (Open Sound Control) to communicate complex parameters and advanced operations. A variety of libraries offer solutions to network transportation of OSC messages and provide approaches for pattern matching the messages in dispatch. Dispatch, however, is performed inefficiently and manipulating OSC messages is oftentimes not realtime safe. Rtosc was written to quickly dispatch and manipulate large quantities of OSC messages in realtime constrained environments. The fast dispatch is possible due to the internal tree representation as well as the use of perfect-minimal-hashing within the pattern matching phase of dispatch.

The primary user of rtosc is the ZynAddSubFX project which uses OSC to map 3,805,225 parameters and routinely dispatches bursts of up to 1,000 messages per second during normal audio processing. For audio applications, rtosc provides a simple OSC serialization toolset, the realtime safe dispatch mechanisms, a ringbuffer implementation, and a rich metadata system for representing application/library parameters. This combination is not available in any other OSC library at the time of writing.

Keywords

Open Sound Control, Realtime, Intra-Process Communications

1 Introduction

Rtosc is a library which provides an OSC 1.1[Freed and Schmeder, 2009] compliant serialization/deserialization, along with a non-compliant matching algorithm. The serialization code was built with general realtime safe use in mind. The matching and dispatch algorithms were designed for simplified integration with existing realtime applications. Rtosc is available under the MIT license at `https://github.com/fundamental/rtosc`.

1.1 Motivation

Rtosc was originally motivated by the need of a messaging protocol within the ZynAddSubFX synthesizer [Paul et al., 2018]. A large number of parameters were directly exposed to the GUI in a manner which made lock-free audio generation difficult and overall make development of new functionality a slow drawn-out process. OSC has been a standard iter-process messaging option since 2002[Wright, 2002], though it was rarely used extensively inside of an application. This characteristic took me by surprise due to the simplicity of the OSC serialization which made it well suited for use in a low computational/memory overhead messaging protocol.

The two primary issues with other implementations of OSC are that they typically used dynamic memory and they had slow dispatch processes. The target for ZynAddSubFX involved processing data on the non-realtime threads as well as the realtime threads, so dispatch, reading messages, and writing messages needed to be done in an efficient realtime-safe manner.

1.2 Other Libraries

Currently there are a variety of OSC libraries available. Common issues with the available implementations at the time of initially writing rtosc were that:

- Many OSC implementation are incomplete
- Almost all OSC implementations did not focus on realtime safe implementation
- Almost all OSC implementations focus on network based inter-app communication
- Some OSC implementations had difficult to use APIs

Based upon their use of C/C++ and the adoption across Linux audio, the most notable

comparable library is liblo. The liblo project [Harris et al., 2018] has a solid reasonably complete implementation with an easy to use API. Other implementations such as oscpack [Bencina, 2016] were examined in initial development, however other C/C++ OSC implementations have limited adoption. Using ubuntu package dependencies as a measure of adoption, the liblo7 package has 42 directly dependent packages (outside of liblo subpackages) and oscpack1 has zero external packages (outside of dev/dbg subpackages).

While liblo has a number of excellent characteristics, it focuses on non-realtime serialization, dispatch, and networking tasks within OSC. For example, message serialization will involve memory allocation and deallocation from the heap, which can take a highly variable amount of time leading to possible time overruns, aka xruns, if used in a realtime context. While liblo acts as a point of comparison within this paper, it is important to note that it targets a different use-case with a number of tradeoffs, which make it suitable for some applications and rtosc for others.

2 C core

Rtosc is broken up into an easily embeddable C core, as well as a set of higher level C++ utility classes. The C core has a variety of methods for encoding/decoding and message matching, though to get started only three functions need to be used:

- rtosc_message(buf, size, path, arg-types, ...)
- rtosc_argument_string(msg)
- rtosc_argument(msg, i)

rtosc_message() will build a OSC message in a provided buffer and will encode all argument types in the OSC 1.1 standard. The types include: **i**:*int32*, **s**:*string*, **b**:*binary-blob*, **f**:*float32*, **h**:*int64*, **t**:*timetag*, **d**:*float64*, **S**:*symbol*, **r**:*rgb*, **m**:*4-byte-MIDI*, **c**:*int8*, **T**:*true*, **F**:*false*, **N**:*nil*, and **I**:*Inf*. rtosc_argument_string() will provide a list of types in an existing OSC message. rtosc_argument() will return the i-th argument through a union. The active union field can be determined via rtosc_argument_string().

Listing 1: Core API example

```
char buffer[128];
const char *value;
//Construct a simple message
rtosc_message(buffer, sizeof(buffer),
              "/test",
              "s", //1 string arg
              "Hello world");
//Say hello world
value = rtosc_argument(msg, 0).s;
printf("%s\n", value);
```

Outside of the simple serialization and deserialization routines there are a number of additional functions

- rtosc_amessage(buf, size, path, arg-types, args[])
- rtosc_message_length(msg, max_len)
- rtosc_itr_begin(msg)
- rtosc_itr_next(itr)
- rtosc_itr_end(itr)

rtosc_amessage() is the non-varargs extension of rtosc_message(), which is more suitable for non-C API bindings. rtosc_message_length() parses a message and verifies if a value message exists in the buffer which is max_len or fewer bytes. The rtosc_itr_* functions quickly iterate through long lists of arguments in complex messages.

3 Message Processing

One of the primary goals of any messaging library is to eventually handle the content of a message. Rtosc application focuses on a largely bi-directional communication between multiple different threads of execution. The four primary responses to a dispatched message are:

- reply - send a message to the client that sent the original message
- broadcast - send a message to all listening clients
- forward - take the current message unmodified and pass it to the next layer
- chain - send a new message to the next layer

Rtosc typically uses a REST-like API, so if OSC application receives "/volume" it should *reply* with the current volume. If "/volume +12.4"(dB) is received, then the OSC application is expected to set the internal volume to

+12.4 dB and then *broadcast* the response to all applications listening to the state of the OSC application in question.

This division between replies and broadcasts makes it possible to attach several different interfaces to a single stateful OSC application. For example, in ZynAddSubFX its graphical user interface will normally be communicating over OSC. While the GUI is running a debug interface, such as oscprompt[1], can be simultaneously connected to the same instance without generating any conflicts.

Chaining and forwarding messages come into play when there are multiple locations a message can be dispatched from. Since rtosc focuses on the realtime dispatch of messages a common configuration is that there is:

1. a dispatch layer on the non-realtime side for handling non-realtime operations such as file loading
2. a dispatch layer on the realtime side for handling most operations and parameter changes
3. a dispatch layer on the non-realtime side for handling responses from the realtime dispatch tree

The first dispatch layer here may choose one or more of several responses. On receiving a message, it can *reply* back to the original source of the OSC message, *forward* it onto the realtime side unchanged, or partially handle the method and *chain* a new message which would go to the realtime side rather than responding to some external client. As these messages are frequently being relayed between the realtime and non-realtime layers, rtosc provides an implementation of a ringbuffer to manage the inter-thread communications.

3.1 Dispatch

Dispatching messages to handlers is a non-trivial portion of each OSC connected application. At a high level dispatch is essentially:

```
handle(message):
 for each callback in callback-list
   if(match(message, callback.path))
     callback(message)
```

OSC complicates this process with pattern such as wildcards in messages. Rtosc, however, targets higher speed matching on a large number of callbacks, so patterns are not typically used in messages, but are used in callback path descriptions. Additionally, rtosc defines dispatch in terms of tree layers.

Consider the OSC path tree shown in Fig. 1. Paths like "/volume" or "/osc3/shape" can be matched to specific callbacks. "osc#5/" indicates a compound pattern consisting of the literal "osc", a number "0", "1", "2", "3", "4", and a trailing "/". Other paths can use optional argument constraints. For example, "detune::f" is composed of the path literal "detune" and then the argument specification ":", ":f" indicates that no arguments are accepted (":") as well as a single float32 argument (":f").

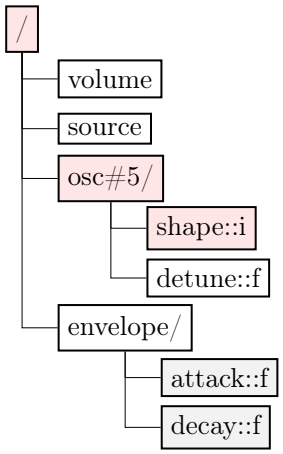


Figure 1: Example Dispatch tree

Other OSC implementations, such as liblo, tend to match an input message directly based upon the full callback path. An example can be matching a OSC message with path "/envelope/attack" or "/*/release" directly on possible destinations "/envelope/attack" or "/envelope/release". Rtosc, on the other hand, favors separate callback definitions/dispatches for each layer. Therefore one dispatch call would try to match "envelope/attack" against "volume", "source", "osc#5/" and "envelope/". Next rtosc would match "attack" against "attack::f" and "decay::f" in a second dispatch layer.

When subtrees are repeated this allows rtosc to have a much more compact representation of the dispatch tree as well as simplifying the difficulty of dispatching at any level. In the case of dispatching "/osc0/shape" (shown in red), the envelope subtree is never dispatched and thus no overhead is produced by the "attack::f" and "decay::f" nodes.

[1] https://github.com/fundamental/oscprompt

3.2 Metadata

Moving further outside of the OSC specification rtosc's dispatch structure provides a way to associate metadata with individual callbacks. Rtosc's metadata provides a list of properties which have optional values. Some of the most commonly used metadata properties and definitions are:

documentation - longer descriptions based upon the parameter

shortname - short name useful for labels in user interfaces

min - minimum value

max - maximum value

default - default value when not modified by user

parameter - signifies that this OSC address corresponds to a value which can be read or written to

enumerated - signifies that there are many symbolic values which map onto a series of integer values

map # - mapping of integer value onto a symbolic name for it

scale - specifies the mapping of values to the user perceived range of them (either "linear" or "logarithmic")

unit - states the units that a parameter is in (e.g. Hz, dB, cents)

3.3 Simplified port specification

As callbacks and metadata tends to be repeated, a some syntatical sugar is available for rtosc. Consider a relatively simple parameter accessors/setter with a minimum and maximum value. For a callback in rtosc's tree an associated parameter name and metadata are defined as shown in Listing 2.

Listing 2: Parameter set/get callback

```
{"foo::f", ":parameter\0"
           ":documentation\0"
           "=Foo␣parameter\0", NULL,
 [](const char *msg, RtData &data) {
   Obj *obj = (Obj*)data.obj;
   if(rtosc_narguments(msg)) {
       obj->foo =
           rtosc_argument(msg,0).f;
       if(obj->foo > 1.0)
           obj->foo = 1.0;
       if(obj->foo < -1.0)
           obj->foo = -1.0;
       data.broadcast(data.loc, "f",
           obj->foo);
   } else {
       data.reply(data.loc, "f"
           obj->foo);
   }
}}
```

The structure of different accessors are going to share a lot of similar code. Using some macros provided by rtosc, it is possible to instead write an abbreviated form:

Listing 3: Syntactic sugar callback

```
rParamF(foo, rLinear(-1.0, 1.0),
           "foo␣parameter")
```

Similar functionality is available via rParamI(), rToggle(), rOption(), and rArrayF(), as well as a few other macros.

Additional utility macros are available for metadata fields as well. One example is rOptions() which is used to define densely packed enums e.g. rOptions(Random, Freeverb, Bandwidth) would define Random as value 0, Freeverb as value 1, and Bandwidth as value 2. rProp(a) defines a generic property 'a' and adds it to the metadata. rMap(a, b) defines a property 'a' which has a value 'b'.

4 Extensions via rtosc messaging & metadata

The metadata associated with rtosc mapped parameters makes it possible to reflect upon the application. While this isn't the primary target of rtosc, there are some notable applications of the metadata so far.

4.1 Self-Documenting

One of the major impacts of having richly documented callbacks is that the API is self-documenting. Each individual OSC based action or parameter can be externally documented in terms of what arguments it requires, what responses should be expected, and what it maps to. At this moment, there are two means of exporting the data: osc-doc and a zyn-fusion specific JSON format. Osc-doc is an XML documentation specification proposed by `https://github.com/7890/oscdoc` and produces a searchable HTML representation of the API similar to doxygen. For Zyn-Fusion a JSON based variant of oscdoc was chosen to avoid the overhead of an XML parser.

Even if the metadata isn't exported to a new format, the existing compiled C-string format

can be transferred to other applications. Oscprompt is one such application and it displays metadata about OSC paths as well as the possible paths which can be tab completed using a reflection based approach.

4.2 Automations/MIDI Learn Support

One use of the metadata exposed through rtosc is a mapping from MIDI or plugin parameters to internal OSC mapped parameters. Given an OSC path (e.g. /part0/PVolume), it is possible to extract the expected type for any OSC messages, the minimum value, the maximum value, and the scaling (linear/logarithmic). Given the metadata, it's possible to define a reasonable default mapping and provide enough information for the user to be able to change the mapping to suit their desires. This functionality is currently being explored within ZynAddSubFX's use of rtosc.

4.3 Undo/Redo support

Within the model that rtosc provides, each OSC message will typically be an action, a state update, or a state read. Since the stream of OSC events contains the state updates that impact the sound engine, the same OSC events can be reused to encode state changes and denote which ones of them are reversible. Rtosc offers one system to capture undoable events and step through their history via undo/redo steps. This approach is similar to non-daw's OSC centric editing 'journal', which stores the programs state as a number of mutations via OSC messages [Liles, 2018].

5 Performance

Rtosc has the goal of providing the necessary information while using a minimum amount of resources. As such, it has been optimized rather extensively and is a very fast tool for handling OSC messages.

One easy point of comparison is against liblo. Liblo is one of the more commonly used OSC implementations within the open source realm; Though by design their API tends to end up allocating memory and producing small data structures. These data structures can produce a notable amount of overhead in OSC heavy systems.

To best compare these two libraries, they were both used to repeatedly encode or decode a message with moderate complexity. The message consisted of the path "/methodname" and arguments: "sif" "this is a string", 123, and 3.14. As can be seen by table 1 rtosc is notably faster in this scenario.

Table 1: Liblo comparison

Impl.	per op	ops per second	speedup
Decoding an average message			
liblo	218 ns	4,600,000	-
rtosc	53 ns	19,000,000	4.1x
Encoding an average message			
liblo	383 ns	2,600,000	-
rtosc	125 ns	8,000,000	3.1x
Dispatch message on single layer			
liblo	530 ns	1,900,000	-
rtosc	54 ns	19,000,000	10x

Rtosc is used in a few performance oriented applications, one of which being the sonic-pi project [Aaron, 2018]. Historically the sonic-pi project used the osc-ruby implementation and then upgraded to an internal subproject, samsosc, which was one effort in producing an optimized OSC implementation. After neither option was satisfactory, the sonic-pi project integrated rtosc via the fast_osc gem[Riley, 2017]. While this isn't an entirely fair comparison, as it crosses different implementation languages, however it provides another picture into the vast performance differences available in such small libraries:

Table 2: Sonic-pi performance stats

Impl	per op	ops per second	speedup
Encoding an average message			
fast_osc	1.2 us	800,000	9.6x
samsosc	3.8 us	260,000	3.1x
osc-ruby	12 us	83,000	–
Decoding an average message			
fast_osc	0.6 us	1,700,000	50x
samsosc	4.7 us	230,000	7.4x
osc-ruby	29 us	34,000	–

In this case, compared to existing options rtosc proved to be significantly faster at reading and writing messages even with the small amount of overhead needed to interface the Ruby and C code.

Beyond the stats recorded for single runs of operations in rtosc, liblo, sonic-pi, etc there are additional scaling behavior to consider. Rtosc's dispatch algorithm scales with the number of subpaths, so using the above numbers it's easy to get a rough approximation for expected DSP load from message dispatch. Message dispatch

time d_t, is roughly a function of path length p_l, time per dispatch layer l_t, and message decoding time m_t:

$$d_t = p_l \times l_t + m_t \quad (1)$$

and DSP load is a function of the rate of messages per second r and the expected average dispatch time per second d_t:

$$\text{dsp\_load} = \frac{r}{d_t} \times 100\% \quad (2)$$

Using the previously calculated timings we can see that even for complex systems with large numbers of messages the overhead of dispatch is low.

Table 3: Projected messaging overhead

path length	msg per second	DSP load
5	100	0.0032 %
20	100	0.011 %
20	10000	1.1 %
10	100000	5.9 %

The following assumptions are used to make the dispatch algorithm more scalable:

1. The tree structure of OSC paths is a way to partition methods.
2. Arrays of parameter should be represented by one port.
3. One OSC message should match one dispatch port.
4. More complex dispatch methods are preferable to a more complex dispatcher.

Item 1 limits the number of matches that need to be considered at each level. The second converts /par0 /par1 /par2 ... /par99 into /par#100. Given the third assumption, it is possible to use techniques such as perfect-minimal hashing to reduce the search space further. Perfect minimal hashing makes it possible to change the matching algorithm for callbacks C and message m from:

```
for c in C:
  if c match m:
     call(c,m)
```

into:

```
c = C[hash(m)]:
if c match m:
  call(c,m)
```

For large collections of parameters, these characteristics help speedup the algorithm immensely. ZynAddSubFX has 3,805,225 unique OSC paths, with an average depth of 6.11 with a maximum depth of 8 subpaths. If perfect hashing occurs at each level, then each input message would on average take 6.11 matches on subpaths, while a flat matching on all possible paths would result in $\sim 1,900,000$ matches with considerably more complexity per match.

Rtosc's approach does impose some restrictions on typical dispatch, however it scales based upon the number of subpaths, or layers. Other solutions will tend to scale based upon the number of possible paths. If the computational complexity is extrapolated from the simple tests for liblo this would result in an average message dispatch time on the order of 18.3 ms while rtosc's dispatch time would be around 380 ns. Equivalently this means the maximum dispatches per second would be ~ 2.6 million for rtosc and ~ 55 for liblo. ZynAddSubFX's use of OSC illustrates an extreme use case which rtosc is well suited for.

6 Conclusions

Prior to rtosc, OSC was a frequently used standard for communication between applications or devices, but library support was lacking for using OSC messages within a realtime safe application. Rtosc provides a realtime safe implementation of OSC messages, message dispatch, as well as several utilities applicable for audio applications. The core interface is written in portable zero-dependencies C code and as has been shown by this paper is performant when compared to other popular implementations. At this moment, the primary users of rtosc are ZynAddSubFX and sonic-pi, though the hope is that other programs will utilize rtosc for efficient and safe OSC message handling in the future.

References

Sam Aaron. 2018. Sonic-pi: The live coding music synth for everyone. `https://github.com/samaaron/sonic-pi`.

Ross Bencina. 2016. oscpack - open sound control packet manipulation library. `https://github.com/RossBencina/oscpack`.

Adrian Freed and Andrew Schmeder. 2009. Features and future of open sound control version 1.1 for nime. In *NIME*, volume 4.

Steve Harris, Stephen Sinclair, et al. 2018. liblo: Lightweight osc implementation. http://liblo.sourceforge.net.

Jonathan Moore Liles. 2018. Non daw. http://non.tuxfamily.org/.

Nasca Octavian Paul, Mark McCurry, et al. 2018. Zynaddsubfx musical synthesizer. http://zynaddsubfx.sf.net/.

Xavier Riley. 2017. fast_osc: A ruby wrapper around rtosc. https://github.com/xavriley/fast_osc.

Matthew Wright. 2002. Open sound control 1.0 specification.

Jacktools - Realtime Audio Processors as Python Classes

Fons ADRIAENSEN,

Huawei German Research Center,
Riesstrasse 25,
80992 Munich,
Germany,
fons@linuxaudio.org, fons.adriaensen@tonmeister.de

Abstract

This paper introduces a set of real-time audio processing blocks that can be used as components in Python scripts. Each of them is both a Jack client and a Python class. The full power of Python can be used to control these modules, to combine them into systems of arbitrary complexity, and to interface them to anything that can be controlled from Python. The rationale behind this approach, some of the the implementations details, and possible applications are discussed.

Keywords

Jack, Python, Audio measurements, Modular audio systems

1 Introduction

Jacktools is a set of real-time audio processing blocks using Jack for audio input and output, and wrapped as Python classes. Currently the set can be divided into two types of functionality: the first is aimed at audio measurement and more technical uses, while the second contains things like an audio file player, gain controls, equalisers, convolution matrices, etc. that can be combined into general-purpose audio systems.

The origins of Jacktools go back several years, when the author required a practical tool to test real-time implementations of audio DSP algorithms. What was needed was an easy way, without having to use a compiled language or write any low-level code, to

- generate complex and accurately defined audio test signals,
- output these via Jack to the tested module, at the same time capturing its outputs,
- analyse the captured signals and present the results in a convenient way.

Python[1], and in particular the numerical and scientific extensions (Numpy[2], Scipy[3], Matplotlib[4],...) provided the ideal environment for signal generation and analysis, only the second step was missing.

The result was JackSignal, a Python class that maps Numpy arrays to Jack ports. It took some research into Numpy's internals and some careful mixing of C and Python code, but in the end the implementation turned out to be straightforward.

JackSignal proved to be a very powerful and practical tool, and it also became clear that the same idea of combining Jack and Python could provide other useful things. The rest is history.

1.1 Overview

The complete Jacktools set contains at the moment more than sixty modules. Many of those implement proprietary algorithms developed in the context of the author's employment by Huawei Research, and unfortunately most of these can't be published.

As already mentioned, those that can be provided fall into two categories. Those intended for audio measurement include:

JackSignal Play and capture signals from/to Numpy arrays. Also provides looping and external triggering.

JackNoise Generates accurate white and pink noise.

JackNmeter Standard filters and detectors for noise measurement.

JackIecfilt IEC class 1 octave band and third octave band filters.

JackPll Phase locked loop, used to track low level drifting signals in noise. Also provides

[1] `https://www.python.org`

[2] `http://www.numpy.org/`

[3] `https://www.scipy.org/`

[4] `https://matplotlib.org/`

I,Q outputs of the phase detector as audio signals.

In the general-purpose category we have:

JackPlayer Multichannel, resampling audio file player. This will play anything that libsndfile can read.

JackGainctl Dezippered multichannel gain control, the DSP part of a fader.

JackParameq Multichannel parametric and 2nd order shelf equaliser.

JackKmeter Multichannel K-meter detector, provides RMS and peak level measurement.

JackMatrix Scalar gain matrix, can be used in many ways, e.g. signal distribution in complex audio installations.

JackMatconv Convolution matrix optimised for dense matrices of short convolutions, as used for microphone and speaker array processing.

JackZconvol General-purpose convolution matrix based on the zita-convolver library.

JackPeaklim Multichannel look-ahead peak limiter, similar to zita-dpl1.

JackAmbpan Up to 4th order Ambisonic panner.

JackAmbrot Up tp 4th order arbitrary axis Ambisonic rotation.

'Multichannel' here usually means 'up to 64 channels'. Also the various matrices can go up to 64×64.
An Ambisonic decoder, networked audio modules (compatible with zita-njbridge) and more dynamics processors are planned to be added.

1.2 Applications

The technical modules have been used to test real-time DSP code, measure speaker directivity patterns, find matched sets of microphone capsules, measure room acoustics, for long term monitoring of environmental noise, and many similar applications. The more general modules have so far been used mainly at the author's workplace, to set up complex demonstrations and listening tests using experimental algorithms. As an example, one listening test involved comparing three different Ambisonics to binaural rendering algorithms, each of them implemented as a Jacktools module, combined with several room simulation methods, again implemented in the same way. This required head motion tracking, so all the rendering had to be done in real time.

For both types of work, having everything under control of an interpreted general-purpose programming language such as Python has significant advantages. It provides not only whatever complex logic may be required, but also access to all system services, external hardware, databases, etc.

For measurements the complete process, including any off-line numerical calculations and up to the generation of a report can be automated. For the listening tests it was an easy exercise to create an ad-hoc graphical user interface providing the participants with exactly the amount of control and feedback required while hiding all the parts that they shouldn't touch or even see.

Other possible applications come to mind, e.g. artistic sound installations, and automated broadcasting systems.

2 Interfacing C++ and Python

All the real-time code for Jacktools is written in C++, so some way to interface this to the Python world is needed.

2.1 High level tools

A wide range of tools for interfacing C or C++ and Python is available. They all have a different scope and use widely diverse methods to achieve their aims.

Boost.Python [5] : 'A C++ library which enables seamless interoperability between C++ and the Python programming language'.

SWIG [6] : (Simplified Wrapper and Interface Generator) provides bindings for many languages, including Python, to C and C++.

Cython [7] : A compiler that extends the Python language and allows simple interfacing with C code.

Of these, **Boost.Python** certainly provides the highest level interface, offering a near transparent gateway between the two worlds, including

[5] `https://www.boost.org/doc/libs/1_61_0/libs/python/doc/html/index.html`
[6] `http://swig.org/`
[7] `http://cython.org/`

even Numpy's arrays (which have some peculiar traits, see below). **SWIG** is considerably simpler and more limited in scope, while **Cython** takes a completely different approach by mixing C-like and Python source code.

While **Boost.Python** would probably be able to do everything required, it would also be overkill for the relative simple functionality needed for Jacktools. In particular, we do not need nor actually want a one-to-one mapping between C++ and Python classes. The Python classes representing the various Jack clients only need to expose the functionality of the real-time code, not its implementation. Also, in Jacktools, the initiative is always at the Python side, the only exception being the C++ code performing a callback to a Python function, but this only happens after that function has been explicitly passed on by the Python code.

On the other hand, neither **SWIG** nor **Cython** seemed to offer much support for handling Numpy arrays nor for working with threads, so this would have to be done manually anyway. Given all this, the most suitable alternative seemed to use Python's C API directly. This had the added benefit of avoiding one more dependency, as well as being a most interesting exercise.

2.2 The Python C API

CPython, the only flavour of Python supported by Jacktools (there are also *Jython, IronPython and PyPy*, written resp. in Java, C# and a subset of CPython) is itself written entirely in C. All the C functions that create, destroy and manipulate Python objects are exported, available for use in extension modules, and quite well documented.

The C API [8], at a first look, seems quite overwhelming and complicated, containing probably thousands of functions. But it is not difficult to use once a few fundamental concepts are understood.

2.2.1 Reference counts

In Python everything is an object, and all objects are reference counted. Once the last reference to an object is deleted, the memory taken by the object can be reclaimed by the *garbage collector* which runs at unpredictable times. Normally all of this happens behind the scenes and automagically, which is one of the reasons why Python is so easy to use, at least regarding memory management.

When using the C API, the programmer has to take care of the reference counts, using the *Py_INCREF()* and *Py_DECREF()* functions. Failure to do this correctly will lead to memory leaks, or worse, stale pointers which will sooner or later trigger a violent crash. The rules are not difficult to understand, unless you want to use things like circular references (which are not used in Jacktools).

2.2.2 Numpy arrays

Numpy arrays are Python objects and are therefore reference counted, but they implement a second level of reference counting internally. This is because the actual data can be shared between array objects. This happens e.g. when an array is *sliced*, which is a very common thing to do in Numpy. For example if given a two-dimensional array **A** containing multichannel audio samples, we can create a vector containing the samples for channel **k** by writing **V = A[:,k]**. Numpy will do this without actually copying the data, it just creates a new *view* on **A**. That means that now two Numpy arrays, **A** and **V** are sharing the same data. Numpy arrays are implemented using the Python *buffer interface*, also used by Python for other array-like objects. It is here that the data sharing is implemented. To get access to the buffer of a Numpy array in C you need to call *PyObject_GetBuffer()* and after use the buffer must be released using *PyBuffer_Release()*. These two calls take care of the reference counting.

Since every Numpy array can be just a slice of another, nothing can be assumed regarding the actual placement of data elements in memory. For example, the samples in the vector **V** above may not be in consecutive memory locations. The *buffer interface* provides methods to find out the exact layout of the data elements in a Numpy array.

2.2.3 Threads

Python programs can be multi-threaded, but the interpreter is single- threaded, so only one thread can run at any time. This is implemented using the *Global Interpreter Lock* aka **GIL**. Multithreading in Python is cooperative: the interpreter will release the lock every so many bytecodes, giving other threads the opportunity to grab it and continue.

This has to be taken into account when using the C API, in two ways. First, when the C

[8] `https://docs.python.org/3/c-api/index.html`

code is actually called from Python and wants to call a blocking function, it must release the **GIL** and take it again on return. Second, when calling a Python function from C, the current thread must acquire the lock before doing so and release it when the callback returns. Apart from that, threads created in C can co-exist with Python without any problem.

3 Implementation

On the C++ side there is one base class *Jclient* which contains almost all of the code required to use Jack. It creates the Jack client and the ports, sets the callbacks, obtains the sample rate and period size, etc. It also handles the shutdown callback, and cleans up things when the client terminates. Finally it contains methods to connect or disconnect ports.

Some of the function members of *Jclient* are given a Python interface using the C API: this includes calls to obtain the current process state (more on this later), the Jack period and sample rate, and to manage ports and connections.

Each of the actual tools is a class derived from *Jclient*, implementing the process callback and any DSP code required, including methods to set parameters and obtain results. These members, and only these, are given a Python interface using the C API.

On the Python side there is also a class named *Jclient*, which is the base class for all the others. It provides access to those methods of the corresponding C++ class which have a Python interface. The actual Jacktools classes derive from this base class and again provide access to those methods of their corresponding C++ class that have a Python interface.

So the actual Python classes are defined in Python, and not in the C++ code. The C++ code only implements some methods which are used by the Python classes. It would be possible to define the Python classes directly in C++, but the current method is simpler and requires less code.

3.1 Connecting the two worlds

The remaining question is how the Python objects find their C++ counterpart when any of their methods are called. The mechanism used for this involves a Python class called *PyCapsule* which was first introduced in Python 3 and later backported to Python 2. A *PyCapsule* object is just a container for a C or C++ pointer. It allows Python code to store such pointers and hand them back to the C or C++ side whenever necessary. That is also their only possible use, as there is no way to interpret the contents of a *PyCapsule* on the Python side.

When the user's Python code creates e.g. a *JackGainctl* object, its *__init__()* calls a C++ function that creates the corresponding C++ object and returns two *PyCapsule* objects, one for the newly created C++ object and one for its *Jclient* base object. The Python object stores the first for later, and uses the second to call its base class *__init__()*. This again stores its *PyCapsule* for later use when calling C++ code.

The *PyCapsule* constructor on the C++ side also takes a pointer to a function that will be called when the last reference to the *PyCapsule* is deleted. So when the Python *JackGainctl* is deleted, this function is called and deletes the corresponding C++ object.

3.2 Process states

As explained in the previous section, a C++ object, which will be Jack client, is created whenever a corresponding Python object is created. On the C++ side things can fail e.g. when the Jack server isn't running, or later if Jack 'zombifies' the client. In those cases we still have a Python object, but one that is not usable. To handle this, all Jacktools classes share a common system which simply consists of maintaining a current state. All Jacktools classes have at least the following states:

PASSIVE : the object is an active Jack client, but the *process* callback does not access any ports. This allows to user to manually create ports. At the moment none of the published classes is using this state, all of them will create a fixed set of ports (depending on the number of inputs and outputs requested) and initialise in one of the two following states.

SILENCE : the object is an active Jack client, but the *process* callback outputs silence on all output ports. This state is typically used to further configure a processor that needs this, e.g. a convolution matrix. This is also a safe state to make port connections.

PROCESS : the object is performing its normal function as a Jack client. Some classes

e.g. *JackPlayer* have additional active states.

FAILED : the object will enter this state when initialisation or becoming an active Jack client fails.

ZOMBIE : the object will enter this state when zombified by the Jack server.

In the latter two states the only remaining option is to delete the Python object, as it can not recover from these states.

The state system allows complex systems to start up cleanly without making unexpected noises, or at least to fail in a controlled way. It also allows applications that have to run unattended to check things periodically and take some recovery action when anything goes wrong.

3.3 Documentation

All the Python code for Jacktools contains documentation in the form of 'docstrings' which can be read using Python's built-in help system. Also a collection of simple example applications (some of them written for testing the Jacktools classes themselves) is provided.

4 Conclusions

In the previous sections, the Jacktools set of Jack clients implemented as Python classes has been introduced. Some of the implementation aspects and choices have been discussed. It is hoped that this may be of interest not only to potential users, but also to developers of audio software that combines the powers of C, C++ and Python. In particular, in the author's opinion, exploring Python's and Numpy's C API has been a very rewarding exercise.

The Jacktools code package will be made available shortly before the start of the conference.

Distributed time-centric APIs with CLAPI

Paul Weaver and **David Honour**
Concert Audio Technologies Ltd.
Reading, UK
{paul, david}@concertdaw.co.uk

Abstract

Distributed control of applications by multiple simultaneous devices has traditionally been achieved via protocols such as MIDI or OSC. These simple protocols require additional semantics, often communicated out of band, in order to construct meaningful APIs.

We present the Concert Light-weight API (CLAPI) framework: a session-based pub/sub API framework that aims to simplify the definition and usage of semantic, time-centric distributed controls.

Keywords

API, Distributed, Pub/Sub, Semantics, Introspection.

1 Introduction

The Concert Light-weight API framework (CLAPI) is one component of our larger distributed DAW project. It grew from our need to control an audio engine from a heterogeneous mix of clients simultaneously, with event-driven feedback of the evolving state of the system.

1.1 Open Sound Control

Our original efforts sought to build semantics on top of Open Sound Control (OSC) [Wright, 2002]. We had hoped to use OSC to communicate instructions and state across the network. However, we ran into some issues with that approach:

- TCP/session-oriented support was lacklustre. This caused problems, for example, when considering TLS/authentication.
- Establishing bidirectional communication was hard (only OSC servers can receive messages), which would complicate NAT traversal.
- The semantics around variable length lists of values weren't standardised (if they were even present at all).
- Only OSC bundles could be timestamped (c.f. individual messages) and bundles could be nested. This meant we couldn't always derive timing information when required.
- Bundle nesting also meant we had trouble building sensible error semantics.
- The dispatch rules provided by existing libraries weren't particularly dynamic.

In other words, whilst OSC is a perfectly good protocol, it did not fit our pattern of component communication as well as we'd have hoped. This led us to consider the problem domain more broadly, and start experimenting with our own API protocol/framework.

2 Network API Paradigms

Before we consider our specific API requirements, we will briefly discuss three approaches for controlling remote systems. We consider both user-facing controllers, such as hardware devices or software GUIs, and autonomous systems connected to the network.

2.1 Fire and Forget

Conceptually, unidirectional protocols like OSC and MIDI [MIDI manufacturers association, 1996] are very simple. Once a connection has been established, control data is transmitted from the client (the party triggering the operation) to the server (the party doing the work) when an action is desired. For example, MIDI sends explicit instructional events like "Note On", which can be fairly directly translated into method calls on an instrument or captured by a recording device for later playback.

OSC is similarly intended to be used in an instruction-centric way, albeit that it is more agnostic in its design (no specific instructions/methods are defined in the protocol itself). Its human-readable metadata also offer a sub-

stantially more direct representation of semantic intent.

The unidirectionality of these protocols has some implications. They do not, for instance, provide a mechanism for applications to report errors. This is assumed to be noticed "out of band", frequently by the user. This "fire and forget" mentality implies, in the absence of such feedback semantics, a controller/executor relationship between the components of the system.

There have been attempts to create feedback semantics for these protocols [Portner, 2017], but they are far from universally supported.

Unidirectionality has the additional consequence that every receiver must handle (even if only by discarding) the union of all possible messages, due to the sender being unable to determine the recipient's capabilities.

2.2 Remote Procedure Calls and Request/Response

The most common kind of network API paradigm arranges the exchanges between client and server like that of a local function call: a request is made by the client to named method ("endpoint") on the server with some arguments and a response is received synchronously after the action is completed.

The most widespread example of this request/response pattern is HTTP [IETF, 1999], and many remote procedure call (RPC) API frameworks are based on top of it [Winer, 1999; W3C, 2000]. However, HTTP itself is an RPC protocol in its own right, with a fixed set of actions (HTTP methods such as GET, POST etc.) with their own arguments (headers), semantics and responses (exit code and potential body).

Feedback to method is improved over unidirectional communication as the client does not have to assume that the invocation was successful. This means that state can be kept in sync without any out-of-band communication, at the expense of more complicated client side handling.

The Representational State Transfer (ReST) philosophy [Fielding, 2000], of which HTTP is an embodiment, attempts to limit the proliferation of ad-hoc methods by structuring requests to the server in terms of *resources*, with a fixed set of methods providing predictable behaviours on those resources.

Formalisms like JSON schema [Andrews/IETF, 2017], and HAL [Kelly, 2011], aim to aid discoverability and introspection by building on top of ReST principles.

2.3 Publish/Subscribe

Another form of API that has been gaining traction, particularly in distributed systems, is the publication/subscription (*pub/sub*) model.

In APIs of this type clients (*subscribers*) communicate to providers (*publishers*) which data they wish to be informed about. Any subsequent updates about the data are then sent to the subscribers who have chosen to be notified.

Most commonly, pub/sub APIs are very scalable message queuing systems [ISO/IEC, 2014; RabbitMq, 2018; MQTT, 1999; OASIS, 2015; Hintjens et al., 2014a; Hintjens et al., 2014b], and clients connect to an API broker, rather than to the source of the data directly.

3 CLAPI's Paradigm

We introduced each of the above network API paradigms because CLAPI has a mixture of features from all of them.

At its heart, CLAPI is an idempotent pub/sub API framework. Providers publish state updates to an API broker (the "Relay") and interested clients subscribe to subsets of that state to receive updates.

Unlike traditional message queues, the Relay keeps a local cache of the application state in memory, so that subscribers are notified of the *current state* of data when they subscribe as well as any future updates.

CLAPI, however, is not just a broadcast system. Just as in traditional "fire and forget" systems, clients can push state update messages of their own, and the Relay forwards them to the provider of an API. Responses to these messages are not received synchronously, as in regular RPC, but rather through existing subscriptions.

These state update semantics give us a nice mix of properties for building an event-driven distributed application. Furthermore, CLAPI incorporates discoverability, introspectability and validation into the API framework from the ground up.

In the next sections we detail the mechanics of CLAPI and continue to contrast it with the three prevailing paradigms we have covered above.

4 Data model

The data communicated by CLAPI are conceptually held in the leaf nodes of a tree and are addressed by paths of names, such as

`/api/version`. The container nodes can also be addressed, e.g. `/api`, and can be thought of as containing data about the names and ordering of their children.

Each top level path (e.g. `/api`) is handled as an isolated API namespace and is "owned" by a single client, who is referred to as the *provider* of that API. The provider may not subscribe to their own API, but may subscribe to other APIs over the same connection. Other clients cannot directly modify the provider's API, but can publish update messages which are validated and forwarded to the provider only for handling.

Before a provider can publish any data, it must provide a collection of types that fully specify the form of the data at every path. Unlike most other API frameworks, this schema is also event-driven and updates can be published at any time. This allows providers, for instance, to expose only session-loading controls until a session is selected, or to defer providing type information for a plugin until after it has loaded.

4.1 Types of Time

There are two notions of time, often not explicitly distinguished, in session-based audio applications: *wall-clock time* and *project time*. CLAPI distinguishes between them explicitly.

Wall-clock time is the time we experience—the one shown by most clocks and watches. It is monotonically increasing and cannot be stopped.

Project time is the time between the start of the recorded work and an event occurring. It is mapped to wall-clock time by playback. This is useful for talking about the relative positions of events that will occur during playback (for parameter automation, for instance).

The values at the leaves of a CLAPI tree can change over project time, or they may be fixed. If the data may change, we refer to the node as a *time series* of *time points*. Time points consist of a pair of time value and tuple of data values, and are indexed in the series by UUID so that we limit the impact of messages crossing on the wire.

Times are stored in an NTP-inspired manner as a pair of 64- and 32-bit unsigned integers representing seconds past the Unix epoch and the sub-second fraction respectively.

The structure of a CLAPI tree is fixed over project time. Changes to both tree structure and project-time data can be made at any point in wall-clock time, and are always applied immediately.

Name	Constraints
enum	Option names (required)
time	
word32	Bounds
word64	Bounds
int32	Bounds
int64	Bounds
string	Regular expression
ref	Type name (required)
list	Item schema (required)
set	Item schema (required)
ordSet	Item schema (required)
maybe	Item schema (required)

Table 1: Value schema types

4.2 Schema

Leaf nodes in CLAPI are referred to as *tuples*, and consist of either a single heterogeneously typed tuple of values, or a time series thereof if the value is to change over project time.

Container nodes can either be *structs* (with a fixed set of heterogeneously-typed children) or *arrays* (with a variable set of homogeneously-typed children).

Because each of these entities has different constraints, there are three kinds of type definition in CLAPI, as detailed below.

4.2.1 Tuples

The type definition for a tuple consists of a documentation string, an ordered mapping of field names to value schema and an *interpolation limit*.

All documentation in CLAPI is intended for human consumption when exploring an API, and has no semantic meaning within the framework.

Each value schema consists of the type of value accompanied by any constraints on that type. For example, it is possible to specify that a value can be any 32-bit integer, or a list of strings that conform to a particular regular expression. The supported value types and their constraint options are shown in table 1. Note that container schema like *list* are defined recursively by constraining with an item schema that is itself another entry from the table.

CLAPI can express interpolation between the time-series data points in each tuple tree node. This means that applications do not have to

send dense streams of data to produce smoothly varying control values.

If the values in a tuple node can change over time, each tuple of values in the project-time series is associated with interpolation parameters. The permitted interpolations are:

Constant This tuple will remain as specified until the next time point.

Linear This tuple is linearly interpolated to the next time point.

Bezier This tuple is interpolated via a Bezier spline (parameters supplied by the user) to the next time point.

The interpolation limit, defined in the tuple type definition, specifies what kinds of interpolation parameters can be specified for each tuple. If the values in the tuple will not change over project time, the interpolation limit is specified as *uninterpolated*. Otherwise, because each of the above kinds of interpolation is more expressive than those that precede it, the interpolation limit simply takes the form of the most expressive interpolation type allowed for the tuple.

CLAPI does not attempt to restrict the choice of interpolation limit according to value types—it is perfectly possible for a provider to publish an API that states it can do Bezier interpolation on strings, and it's the provider's job to do whatever would be expected of it in that situation.

4.2.2 Arrays

The type definition for an array consists of a documentation string, and a type name and permission information about the children of the array. The type name specifies that any direct child nodes of this container node will be of the named type. We call the permission information the *liberty* of the child nodes. It is selected from the following enumeration:

Cannot The client cannot supply this data. Should the client create a new array element containing a path with this liberty, the provider will generate a value for it.

May Paths with this liberty are editable. Should the client create a new array element containing a path with this liberty without supplying a value the provider will generate a default value.

Must Paths with this liberty are editable. Should the client create a new array element containing a path with this liberty they must supply a value.

4.2.3 Structs

The type definition for a struct consists of a documentation string and an ordered mapping of child names to pairs of type name and liberty. Structs in the tree must always contain all their defined children. The liberty value, however, allows for partial definition of struct data by clients when inserting structs into array containers, which providers must then fill in. In other words, defining liberty values on structs allows us to nest structured data within arrays whilst keeping the semantics around defaults and read-only behaviour.

4.3 Attribution

Situational awareness is important in an application with collaborative control. That is, we want to know not only what changes have been made, but by whom. CLAPI attaches an *attributee* to each piece of data and each child in arrays, in order to keep track of who is doing what in the session.

4.4 Introspection

Because providers must publish a collection of types that fully specify the type of every path in their tree of data, and because the Relay publishes type information about the root node that contains all the providers' API namespaces, it is possible to explore the entire CLAPI data space beginning with a single subscription to the root node.

This means that CLAPI APIs are both discoverable and self-documenting, with a limited and consistent set of semantics—desirable properties we detailed in our brief discussion of ReST (section 2.2).

Type assignment messages are sent to clients when they first subscribe to a path to prevent them from having to infer the type of a path by traversing down from the root node type.

4.5 Consistency

Data updates received by clients must always lead to a self-consistent tree state. For example, tuples must contain data of the correct type, and data cannot be assigned to paths that are not reported to be contained in a parent node.

Therefore, multiple changes may be communicated together and applied atomically. This

is similar to bundles in OSC. Because of the dynamism of our type system, it is often required that type changes are accompanied by corresponding data changes.

The kinds of operations that can be performed in each set of changes differs with respect to client role and communication direction, due to the restrictions laid out in section 4. The kinds of information that can be transmitted between each party are outlined in table 2.

Given these consistency restrictions, and our general data type constraints, we include semantics for error reporting in our message exchange. Error message strings are keyed in relation to the API entity to which they pertain. We call this key the *error index* and it can take one of the following forms:

Global The error is not specific to any particular piece of data (e.g. an error decoding a message).

Type The error is specific to a type (e.g. referencing a type name that does not exist).

Path The error is specific to a path (e.g. attempting to assign invalid project-time-global data, or changing the child keys of a struct).

TimePoint Indices of this type contain the path and UUID for the point to which the error pertains (e.g. attempting to assign invalid data to a specific point in a time series).

5 Other concerns

5.1 Time

Sometimes it is important for a client to know when an event occurred even if that client was not connected when that event happened. CLAPI messages are timestamped to high precision so that the Relay may present its own API with information about the time differences between clients.

5.2 Topology in Larger Deployments

API providers can subscribe to other APIs within the same Relay, or even make connections to other Relays in order to collect information about remote systems that they may then choose to expose. This allows the formation of substantially more complex topologies withut the requirement for consensus algorithms in CLAPI.

6 Ecosystem

Our current implementation of CLAPI is written in Haskell. We have written library code that implements building blocks required to write a CLAPI application [Concert Audio Technologies, 2018b], including types for values, definitions and messages, as well as serialisation. We have implemented the Relay application using the library.

We have produced a dummy API provider in Haskell for testing purposes. The audio engine component of our application is currently written in a mixture of C and Haskell, with the Haskell portion providing the high-level API interaction and control plane.

We are looking to provide a framework for creating HTML5/WebSocket interactive frontends for CLAPI applications. These take the role of clients in the solution. This component is in the early stages of development at the time of writing [Concert Audio Technologies, 2018a].

We hope that the high degree of type introspection possible with CLAPI can assist in creating a UI by allowing the dynamic generation of widgets for controls. This should mean that clients and providers do not need always to be kept in tight version synchronisation. We aim to blend this dynamism with some explicit layout design in order to provide useful, customisable interfaces.

7 Future

We are currently prototyping our distributed DAW on top of the CLAPI framework. The design of CLAPI is heavily influenced by what we are trying to achieve in our application and vice versa. As both the application and CLAPI are still under very active development, we appreciate that some details may change between the time of writing and the conference.

We are curious as to whether the mixed-paradigm approach and features like validation, discoverability and introspection, which we have tried to incorporate in the CLAPI framework, are applicable to a wider range of applications outside our problem domain. We'd also like to explore further how these features impact the design of applications, and whether there are any technical considerations we may have overlooked in CLAPI's design.

Ultimately, we hope that CLAPI will be of use to the community, either directly, or by stimulating discussion about the kind of high-level features we want in our APIs in the future.

	Definitions	Type Assignments	Data Updates	Errors
Relay → *Client*	•	•	•	•
Client → *Relay*			•	
Relay → *Provider*			•	•
Provider → *Relay*	•		•	•

Table 2: Information each role can communicate to others in CLAPI

References

H. Andrews/IETF. 2017. Json schema specification. `http://json-schema.org/specification.html`.

Concert Audio Technologies. 2018a. A Prototypical CLAPI web GUI. `https://github.com/foolswood/elmweb`.

Concert Audio Technologies. 2018b. Clapi. `https://github.com/concert/clapi`.

Roy Fielding. 2000. *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. thesis, University of California, Irvine.

Pieter Hintjens et al. 2014a. Zeromq distributed messaging. `http://zeromq.org/`.

Pieter Hintjens et al. 2014b. Zeromq message transport protocol. `https://rfc.zeromq.org/spec:23/ZMTP`.

IETF. 1999. Hypertext Transfer Protocol 1.1 (RFC 2616). `https://tools.ietf.org/html/rfc2616`.

ISO/IEC. 2014. ISO/IEC 19464 - Advanced Message Queuing Protocol (AMQP) v1.0 Specificiation. `https://www.iso.org/standard/64955.html`.

Mike Kelly. 2011. Hypertext application language specification. `http://stateless.co/hal_specification.html`.

MIDI manufacturers association. 1996. MIDI 1.0 standard. `https://www.midi.org/specifications/item/the-midi-1-0-specification/`.

MQTT. 1999. MQTT homepage. `http://mqtt.org/`.

OASIS. 2015. Mqtt version 3.1.1 plus errata 01. `http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html`.

Hanspeter Portner. 2017. OSC additional semantics. `https://open-music-kontrollers.ch/osc/about/`.

RabbitMq. 2018. Rabbitmq amqp implementation homepage. `https://www.rabbitmq.com/`.

W3C. 2000. Simple object access protocol 1.1 specification. `https://www.w3.org/TR/soap/`.

Dave Winer. 1999. Xml-rpc specification. `http://xmlrpc.scripting.com/spec.html`.

Matt Wright. 2002. Open sound control 1.0 specification. `http://opensoundcontrol.org/spec-1_0`.