18th Linux Audio Conference

Proceedings

25-27 Nov 2020
Online conference (originally Talence, France)

https://lac2020.sciencesconf.org

Table of contents

Papers

PIPEWIRE: A LOW-LEVEL MULTIMEDIA SUBSYSTEM

Wim Taymans *

Principal Software Engineer
Red Hat, Spain
wim.taymans@gmail.com

ABSTRACT

PipeWire is a low-level multimedia library and daemon that facilitates negotiation and low-latency transport of multimedia content between applications, filters and devices. It is built using modern Linux infrastructure and has both performance and security as its core design guidelines. The goal is to provide services such as JACK and PulseAudio on top of this common infrastructure. PipeWire is media agnostic and supports arbitrary compressed and uncompressed formats. A common audio infrastructure with backwards compatibility that can support Pro Audio and Desktop Audio use cases can potentially unify the currently fractured audio landscape on Linux desktops and workstations and give users and developers a much better audio experience.

1. INTRODUCTION

In recent years, a lot of effort has been put into improving the delivery method for applications on Linux. Both Flatpak [1] (backed by Red Hat and others) and snappy [2] (backed by Canonical) aim to improve application dependencies, delivery and security.

Due to the increased security policy of these sandboxed applications, no direct access to system devices is allowed. Access to devices needs to be mediated by a portal and controlled by a daemon.

Linux and other operating systems have traditionally used a daemon to control audio devices. For consumer audio, the Linux desktop has settled around PulseAudio [3] and for Pro-Audio it has adopted JACK [4].

The initial motivation for PipeWire [5] in 2017, came from a desire to support camera capture in a sandboxed environment such as Flatpak. PipeWire was initially conceived as a daemon to decouple access from the camera and the application, not very different from the existing audio daemons. Later on, the design solidified and with input from the LAD community it went through a couple of rewrites and gained audio functionality as well.

PipeWire provides a unified framework for accessing multimedia devices, implementing filters and sharing multimedia content between applications in an efficient and secure way. This framework can be used to implement various services such as Camera access from browsers, Screen sharing, audio server, etc.. The design allows to run PulseAudio and JACK applications on top of a common framework, essentially providing a way to unify the Linux Audio stack.

This paper will focus on the Audio infrastructure that PipeWire implements.

* This work was supported by Red Hat

2. LINUX AUDIO LANDSCAPE

Audio support on Linux first appeared with the Open Sound System (OSS) [6] and was until the 2.4 kernel the only audio API available on Linux. It was based around the standard Unix open/close/read-/write/ioctl system calls.

OSS was replaced by the Advanced Linux Sound Architecture (ALSA) [7]from Linux 2.5. ALSA improved on the OSS API and included a user space library that abstracted many of the hardware details. The ALSA user-space library also includes a plugin infrastructure that can be used to create new custom devices and plugins. Unfortunately, the plugin system is quite static and requires editing of configuration files.

OSS — and also ALSA — both suffer from the fact that only one application can open a device at a time. Some hardware can solve this by doing mixing in the audio card itself but most consumer cards or even pro audio cards don't have this functionality. ALSA implements a software mixer as a plugin (Dmix) but its implementation is lacking and its setup inflexible.

2.1. First sound servers

EsoundD (or ESD) was one of the first sound servers. It was developed for Enlightenment and was later adopted by GNOME. It received audio from multiple clients over a socket and mixed the samples before writing to the hardware device. Backend modules could be written for various sound APIs such as ALSA and OSS.

The first sound servers used TCP as a transport mechanism and were not designed to provide low-latency audio. Applications were supposed to send samples to the server at a reasonable speed with some limited feedback about the fill levels in the server.

BSD has another audio API called `sndio` [8]. This is a very simple audio API that can also handle midi. It is based on Unix pipes to transport audio and has, like ESD, no real support for low-latency audio.

2.2. Pro audio with JACK

The JACK Audio Connection Kit (JACK) was developed by Paul Davis in 2002 based on the audio engine in Ardour. It provides real-time and low-latency connections between applications for audio and midi.

JACK maintains a graph of applications (clients) that are connected using ports. In contrast to the previous audio servers, JACK will use the device interrupt to wake up each client in the graph in turn to process data. This makes it possible to keep the delay between processing and recording/playback very low.

There are 2 implementations of the JACK API with different features. Work is ongoing to bring the JACK2 implementation to the same level as JACK1, eventually rendering JACK1 obsolete.

JACK is missing features that are typically needed for regular desktop users such as format support, power saving, dynamic devices, volume control, security etc.

2.3. Consumer audio with PulseAudio

PulseAudio is a modern modular sound server. In contrast to other sound servers on Linux it handles the routing and setup of multiple devices automatically and dynamically. One can connect a bluetooth device and have the sound be routed to it automatically, for example.

It is possible to write rules into a policy module to perform various tasks based on events in the system. One can for example, lower or pause audio streams when an incoming call is received.

PulseAudio is optimized for power saving and does not handle low-latency audio very well, the code paths to wake up a client are in general too CPU hungry.

Most desktops nowadays install PulseAudio by default. And it is possible to let PulseAudio and JACK somewhat coexist. PulseAudio can automatically become a JACK client when needed although this will cause high CPU load with low-latency JACK setups.

3. PIPEWIRE MEDIA SERVER

When designing the audio infrastructure for PipeWire we need to build upon the lessons learned from JACK and PulseAudio. We will present the current design and how each part improves upon the JACK and PulseAudio design.

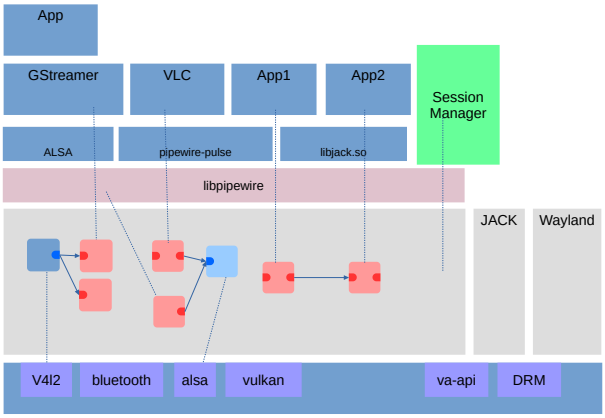


Figure 1: *Overview*

Figure 1 shows where PipeWire is situated in the software stack. It sits right between the user-space API to access the kernel drivers and the applications. It takes on a similar role as PulseAudio or JACK but also includes video devices, midi, Firewire and bluetooth. It allows application to exchange media with each other or with devices.

Applications are not supposed to directly access the devices but go through the PipeWire API. There is a replacement JACK and PulseAudio server that go through PipeWire to provide compatibility with existing applications. There is no urgent reason to port applications to PipeWire unless they would want to use newer features that cannot be implemented in the other APIs.

The core functionality of PipeWire is simple and consists of:

- Provide a set of core objects. This includes: Core, Client, Device, Node, Port, Module, Factory, Link along with some helper objects.
- Load modules to create factories, policy or other objects in the PipeWire daemon or client.
- Allow clients to connect and enforce per client permissions on objects.
- Allow clients to introspect objects in the daemon. This includes enumerating object properties.
- Allow clients to call methods on objects. New objects can be created from a factory. This includes creating a link between ports or creating client controlled objects.
- Manage the negotiation of formats and buffers between linked ports.
- Manage the dataflow in the graph.

An important piece of the infrastructure is the session manager. It includes all the specific configuration and policy for how the devices will be managed in the PipeWire graph. It is expected that this component will be use-case specific and that it will interface with the configuration system of the environment. PipeWire provides a modular example session manager and work is ongoing to create an alternative session manager called WirePlumber [9].

In the next subsections we cover the various requirements and how they are implemented in PipeWire.

3.1. IPC

A sound/media server needs to have an efficient and extensible IPC mechanism between client and server. The PipeWire IPC (inter process communication) system was inspired by Wayland [10]. It exposes a set of objects that can have properties, methods and that can emit events.

The protocol is completely asynchronous, this means that method calls do not have a return value but will trigger one or more events asynchronously later with a result. This also makes it possible to use the protocol over a network without blocking the application.

PipeWire has a set of core built-in interfaces, such as Node, Port and Link that can be mapped directly to JACK Client, Port and connections. It is also possible to define new interfaces and implement them into a module. This makes it possible to extend the number of interfaces and evolve the API as time goes by.

Extensibility of the protocol has been lacking in JACK and to some extend PulseAudio as well.

3.2. Configuration/Policy

Configuration of the Devices and nodes in the PipeWire daemon as well as the routing should be performed by a separate module or even an external session manager.

With PulseAudio, the setup and policy was loaded into the daemon with modules and requires editing of configuration files to change. Modules can also only be developed inside the PulseAudio repository, which makes them very inflexible and not adaptable to the specific desktop environment.

JACK has very limited setup, it can normally only load and configure 1 hardware device for capture and playback. It is up to other processes (session managers, control applications) to add extra devices dynamically (`netjack`, `zita-a2j`, `zita-n2j`, ...).

PipeWire chooses the external session manager setup, like JACK but makes it possible to choose what services to run in the daemon and which in the session manager. Devices, for example, can run inside the daemon for better performance or outside of the daemon for more flexibility (Bluetooth devices need encoding/decoding that is better run outside of the daemon).

The session manager can export any kind of device, including Bluetooth, Firewire, ALSA, video4linux and so on.

3.3. Security

JACK and PulseAudio make it possible for clients to interfere with other clients or even read and modify their data. PipeWire fixes this problem with a permission system that is enforced at the PipeWire core level.

When a client connects, it can be frozen until permissions are configured on it. This is usually done by an access module when it detects a sandboxed client. Usually a session manager will configure permissions on a client based on its stored permissions or based on user interaction.

PipeWire enforces that clients can't see or interact with objects for which they don't have READ permission. Client can't call methods on objects without EXECUTE permissions and WRITE permission is needed to change properties on an object.

It is also important that clients can only see the shared memory they need. This is implemented in PipeWire by only handing `memfd` file descriptors to clients that require the data. Seals are used to make sure that clients can't truncate or grow the memory in any way and cause other clients or the daemon to crash.

3.4. Format negotiation

PipeWire uses the same format description as used in GStreamer [11]. This allows it to express media formats with properties, ranges and enumerations. It is possible to easily find a common format between ports by doing a generic intersection of formats.

Format conversion, however, is not something that should be done often in a real-time, low-latency pipeline. It should typically only be done when writing to or reading from the actual hardware. The PipeWire audio processing graph uses a common single format between all the processing nodes. The format is not hard-coded into PipeWire but configured by the session manager and is currently the same format as used by JACK: Float 32 bits mono samples.

PipeWire uses a generic control format to transport midi and other control messages. This can include timed property updates, OSC or CV values.

Flexible format negotiation is a requirement to implement features like pass-through over HDMI or AAC decoding on the bluetooth devices. The session manager will usually define how this will work, for example, pass-through will require exclusive access to the device because mixing of encoded formats is not possible.

3.5. Dataflow

After a format is negotiated, PipeWire negotiates a set of buffers backed by memory in `memfd`. These buffers are then shared between nodes and ports that need them by passing the file descriptor around. `eventfd` is used to wakeup nodes when they need to process input buffers and produce output buffers.

`timerfd` is used to measure when a devices will be empty/-filled. The timeout is adjusted based on the fill level of the device and a DLL. By using a timer, we can also dynamically adjust the period size based on client requirements. It is also possible to write the device wakeup using the traditional IRQ based approach but that does not provide flexible period adjustments.

When a device needs more data (or has more data, in case of a source), the graph is woken up. PipeWire uses the same concepts as JACK2 to schedule the processing graph. It keeps track of dependencies between nodes and nodes are informed about the peer nodes they are linked to. When processing starts, all nodes without dependencies are scheduled (sources). When they complete, dependencies are satisfied on their peer nodes, which are then scheduled, and so on until the whole graph is completed. Nodes that complete can directly wake up their peers by signaling the `eventfd` without having to wake up the PipeWire daemon.

This allows for the same latency and complexity as JACK and significantly better performance than PulseAudio.

3.6. Automatic slaving

PipeWire will automatically manage the master/slave relationship between devices. For this it uses a priority property configured on the device node by the session manager.

Devices are only slaved to each other when their graphs are interconnected in some way. This is an improvement compared to JACK, which requires all devices to be slaved to one master, even if they don't need to be. It allows PipeWire to avoid resampling in many cases.

The clock slaving and resampling algorithm is inspired by zita-ajbridge [12]. It however runs in a single thread and uses a DLL to drive the resampler by matching its device fill level to the graph period size. This results in exceptionally good rate matching, far superior to what PulseAudio manages and with lower latency than what zita-ajbridge does.

3.7. Transport

PipeWire expands on the JACK transport feature with the following additions:

- multiple transports at the same time. Each driver has its own transport, when drivers are slaved, the transport of the master becomes the active one. This makes it possible to avoid slaving and resampling when the driver graphs are not linked in any way.
- Seeking is supported in other formats than audio samples. Seeking in beats or bars is possible.
- Clients can know about new position changes in advance. There is a queue of pending position changes that clients can look at.
- Sample accurate looping.

4. SESSION MANAGER

The PipeWire daemon is usually configured to start up with a minimal set of modules. All devices and policy are typically loaded and configured by an external session manager. This usually include a factory for devices and a factory for making links.

PipeWire includes a modular example session manager that can be used as a basis for a custom session manager.

The session manage usually also implement the session manager extension API that introduces concepts of Session/Endpoint/EndpointStream and EndpointLink. These interfaces are used to group and configure nodes in the graph and allows PipeWire/SessionManager to provide similar concepts to what PulseAudio uses.

5. API SUPPORT

Legacy application should run unmodified on a PipeWire system. Depending on the API, a plugin or a replacement library is used for this purpose.

5.1. ALSA

There is an ALSA plugin that interfaces directly with PipeWire to support older ALSA-only applications. See Figure-3 for an example of aplay streaming to PipeWire.

5.2. PulseAudio

PulseAudio support was initially implemented with a reimplementation of `libpulse.so` and some other pulse libraries that interfaces with PipeWire directly. This however proved to be more complicated and error prone than expected.

The lastest PipeWire version implements PulseAudio support with a minimal reimplementation of the PulseAudio protocol in a separate daemon. This provides excellent compatibility even for Flatpak applications and turns out to be considerably less complicated to implement.

See Figure 2 for a screenshot of `pavucontrol` running on top of the PipeWire PulseAudio replacement daemon.

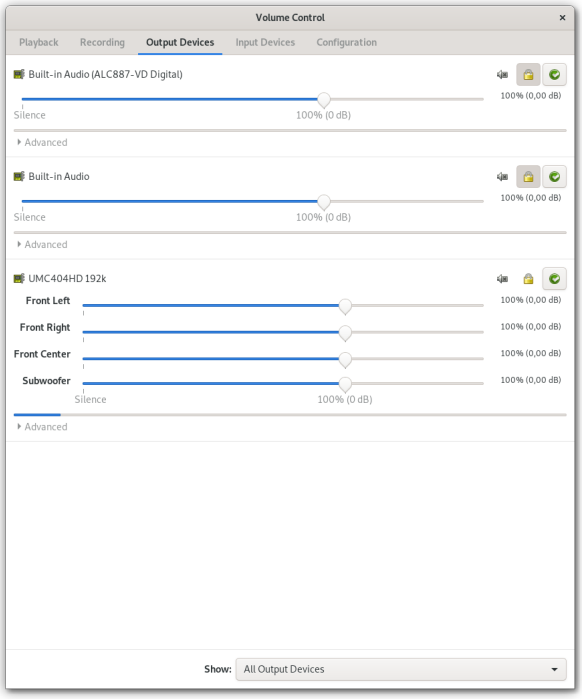


Figure 2: *`pavucontrol` on PipeWire*

5.3. JACK

JACK is supported with a custom `libjack.so` library that maps all jack method calls to equivalent PipeWire methods. See Figure-3 for an example of catia running on top of the PipeWire `libjack.so` replacement. The figure also shows how VLC (using the PulseAudio API), aplay (using the ALSA plugin) and paplay (using the PulseAudio API) can coexist with JACK applications.

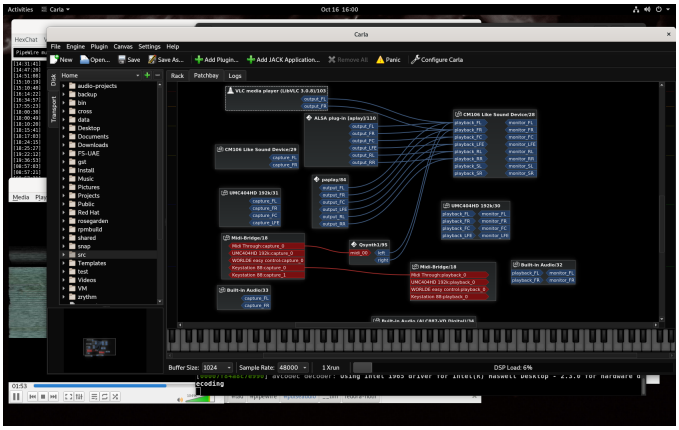


Figure 3: *Catia on PipeWire*

6. USE CASES

In addition to the audio use case that we covered in the previous section, in this section we briefly touch upon the other use cases that PipeWire handles.

6.1. Camera access

In sandboxed applications, it is not allowed to directly access the video camera. Browsers provide a custom dialog to mediate access to cameras but this task would be better handled by the lower layers in order to have a unified access control mechanism but also a common video processing graph.

PipeWire can provide a video4linux source that applications can use to capture video from the camera. This has many benefits such as:

- Access can be controlled by PipeWire. Revoking access is easy.
- Resolutions and format are managed by the session manager. Based on the profile and requirements of the apps using the camera.
- Filters can be applied.
- The camera can be shared between applications.

GNOME has created a portal DBus API [13] to negotiate access with the camera (what camera to use) and create a session with limited permissions for this stream.

Figure-4 shows Cheese and a GStreamer pipeline sharing the captured video of a `video4linux` camera served through PipeWire.

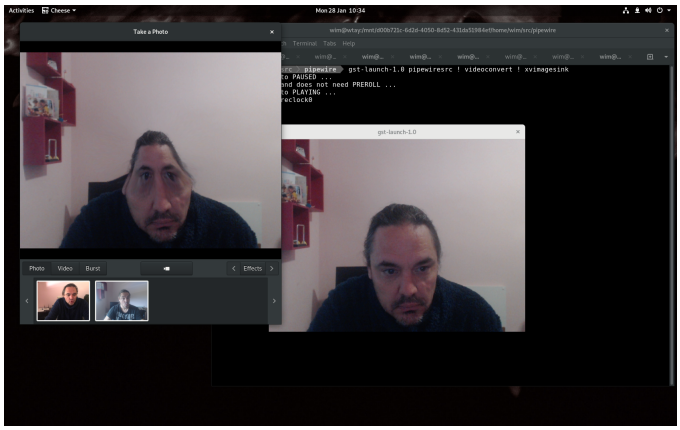


Figure 4: *Camera access and sharing*

6.2. Screen sharing

Under Wayland, it is for security reasons not possible to grab the contents of the screen. This makes it impossible to implement screen sharing or remote desktop on top of Wayland without some extra work.

GNOME has implemented a portal (DBus API) that can be used to request a PipeWire stream of the desktop. The portal will ask the user what kind of screen sharing to activate (windows, area or whole desktop along with what monitor etc) and will then set up a PipeWire session with the stream. The `fd` of the session is passed to the application. Using the PipeWire security model, only this stream is visible to the application and data can flow between the compositor and the application. See Figure-5 to see a GStreamer pipeline rendering the captured screen of a Wayland session.

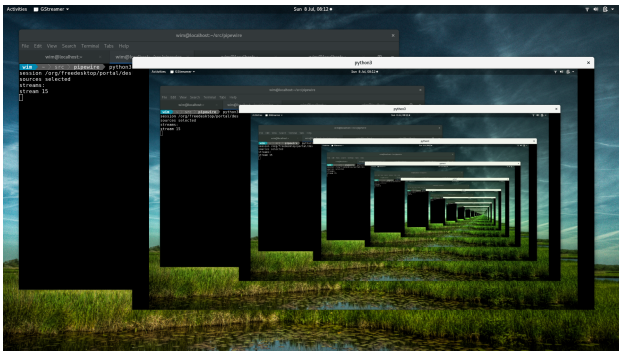


Figure 5: *Wayland screen sharing*

6.3. Video processing

Some effort has been put into the Video processing part of PipeWire. Currently there is a Vulkan compute source that can generate video in RGBA float 32 linear light format. We expect video filters to be made at a later stage, enabling the same kind of features JACK gives but on video streams.

6.4. Adoption

PipeWire has been in Fedora 27 since 2017 to implement Wayland screen sharing. FireFox, Chrome (WebRTC) have support to implement the screen sharing with native PipeWire API using the DBus portal.

Since Fedora 32 (early 2020), the redesigned version 3 with audio support has been shipped but not enabled by default.

On September 4th 2020 [14], a tech preview can be enabled in Fedora 32 and Fedora 33 to test out the audio functionality. This resulted in quite a few new features and bugfixes reported by early testers.

Currently, a plan is developing to try to enable PipeWire as the default Audio service in Fedora 34 (april 2021) and to phase out PulseAudio and JACK.

7. CONCLUSIONS

We showed how PipeWire provides a performant and secure multimedia subsystem in Linux. With lessons learned from existing consumer and pro audio solutions, PipeWire unifies the audio stack and provides a future proof foundation for all kinds of new exciting multimedia applications.

Future work will involve deploying PipeWire in distros and learning how to improve the design. More research and experience is needed for writing the session manager and how this will integrate with the desktop configuration.

More work is being done on experimenting with scripting languages to define the policy and routing in a flexible and reusable way.

8. ACKNOWLEDGEMENTS

Many thanks to the LAD community (and in particular Robin Gareus, Paul Davis, Len Ovens and Filipe Coelho) for letting me pick their brains and putting me on the right track.

Many thanks to my employer Red Hat, who sponsored the development of PipeWire.

9. REFERENCES

[1] Flatpak Comunity, "Flatpak," `https://github.com/flatpak/flatpak`.

[2] Canonical, "Snappy," `https://snapcraft.io`.

[3] Lennart Poettering et al., "Pulseaudio," `https://pulseaudio.org/`.

[4] Paul Davis, "Jack audio connection kit," `http://jackaudio.org/`, 2002.

[5] Wim Taymans, "Pipewire - multimedia processing," `https://pipewire.org`, 2017, [Online].

[6] Hannu Savolainen, "Open sound system," `http://www.opensound.com`.

[7] Jaroslav Kysela, "Advanced linux sound architecture," `http://alsa-project.org`, 1998.

[8] Alexandre Ratchov and Jacob Meuser, "sndio: Openbsd sound system," `http://www.sndio.org`, 2008.

[9] George Kiagiadakis and Julian Bouzas, "Wireplumber - session / policy manager implementation for pipewire," https://gitlab.freedesktop.org/pipewire/pipewire.

[10] Kristian Høgsberg, "Wayland," https://wayland.freedesktop.org/.

[11] The GStreamer Comunity, "Gstreamer api documentation," https://gstreamer.freedesktop.org/documentation/gstreamer/.

[12] Fons Adriaensen, "Zita-ajbridge," http://kokkinizita.linuxaudio.org/linuxaudio/zita-ajbridge-doc/quickguide.html, 2012.

[13] Freedesktop Comunity, "A portal frontend service for flatpak," https://github.com/flatpak/xdg-desktop-portal, 2016.

[14] Christian F.K. Schaller, "Pipewire late summer update 2020," https://blogs.gnome.org/uraeus/2020/09/04/pipewire-late-summer-update-2020/, 2020.

[15] PipeWire comunity, "Pipewire - gitlab freedesktop," https://gitlab.freedesktop.org/pipewire/pipewire.

OMAI: AN AI TOOLKIT FOR OM#

Anders Vinjar *

https://www.avinjar.no
Artistic Research Residency at IRCAM, 2019–2020
anders@avinjar.no

ABSTRACT

OMAI is a toolkit for composers wanting to explore the use of artificial intelligence and machine learning in computer assisted music composition. The OMAI library for the OM#CAC-application implements techniques for data classification, prediction and generation, in order to integrate these techniques in composition workflows.

Examples are provided using simple musical structures, highlighting possible extensions and applications.

A brief description of OM# - a new CAC environment derived from OpenMusic - is included. OM# is a visual programming language dedicated to musical structure generation and processing, available on Linux, MacOSX and Windows platforms.

1. INTRODUCTION

As soon as computers were conceived, composers entered the labs and started to explore potentials for computation and representation in search for new creative options. Computer assisted composition (CAC) became part of the emerging field of artificial intelligence (AI) [1].

Artificial intelligence and machine learning are commonly used in research on computational creativity [2], "autonomous" generative and/or improvisation systems [3, 4, 5], or real-time performance monitoring and interaction [6]. However, apart from a few examples [7, 8], machine learning and AI are rarely explored by composers as a means for composing music, and current techniques to assist composition tasks (e.g. [5]) generally do not operate directly in compositional workflows and environments.

CAC systems provide explicit computational approaches through the use of end-user programming languages [9]. OM#[10], derived from OpenMusic, is a recent newcomer amongst the visual programming environments for music and sound, allowing users to process and generate scores, sounds and many other kinds of musical structures, handle scheduling, input/output and interaction with external systems, and provide a platform for general programming and scripting.

This article presents ongoing work exploring the use of AI and machine learning techniques in the OM# environment. In contrast to approaches aimed at automatic creation or machine classification systems, the aim of OMAI is to provide useful techniques to aid in the composers workflow. The approach is a "composer-centered" machine learning approach [11] allowing users of CAC systems to implement experimental cycles including pre-processing, training, and setting the parameters of machine learning models for data generation, decision support or solving other generic problems.

The toolkit is designed to be used in a bottom-up workflow, supporting creative tasks where there's not one correct answer. The goal is to provide pragmatic tools, having an adequate interface for end-users while retaining an open-ended environment in OM#s graphical programming environment.

* This work was supported by IRCAMs Artistic Research Residency Program

2. OM#

OM# (om-sharp[1]) is a computer-assisted composition environment derived from OpenMusic: a visual programming language dedicated to musical structure generation and processing.

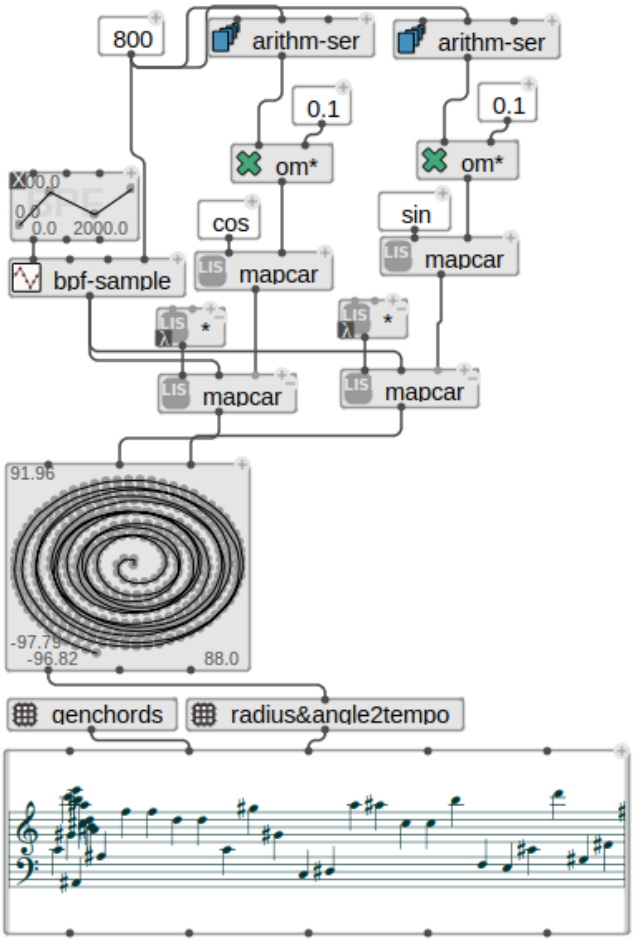


Figure 1: OM# — graphical programming

[1] https://cac-t-u-s.github.io/om-sharp/

The visual language is based on Common Lisp, and allows to create programs that are interpreted in this language. Visual programs are made by assembling and connecting icons representing Lisp functions and data structures, built-in control structures (e.g. loops), and other program constructs. The visual language can therefore be used for general-purpose programming, and reuse any existing Common Lisp code. A set of in-built tools and external libraries make it a powerful environment for music composition: various classes implementing musical structures are provided, associated with graphical editors including common music notation, MIDI, OSC, 2D/3D curves, and audio buffers.

OM# is available for Linux, macOS and Windows. The first successful port of OpenMusic to Linux was done in 2013[12], and since then the development of both OpenMusic and OM# has taken place on these 3 platforms. This software is free, distributed under the GPLv3 license.

3. TOOLS AND ALGORITHMS

The OMAI library for OpenMusic provides basic tools from the domain, with elementary algorithms to classify vectors in a multidimensional feature space [13].

3.1. Vector Space

A generic data structure called VECTOR-SPACE is used to store vectorized data and information necessary to train and run machine learning and classification models. The structure is simple and generic; it is initialized with a list of entries (key, value) for a hash-table of vectors, where keys can be strings or any other unique identifiers for the different vectors.

Feature-vectors are also stored as hash-tables using descriptor names as keys. Descriptor names can also be input to the VECTOR-SPACE initialization for facilitating visualization and query operations. A graphical interface allows the 2D and 3D visualization of vectors in the feature space, selecting two or three descriptors as projection axes (see Figure 2).

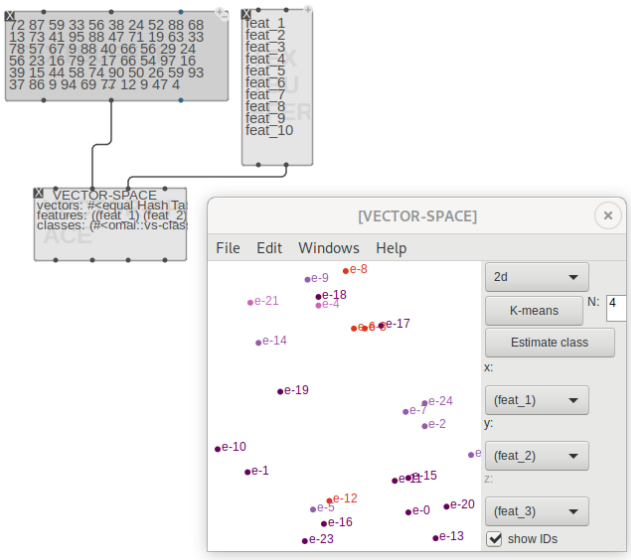


Figure 2: Simple 2-D vector space visualization.

3.2. Clustering and Classification

Within the VECTOR-SPACE, distance-functions are used to retrieve information and compute similarity between feature-vectors. These operations can be applied in various algorithms for automatic clustering and classification.

The *k-means* algorithm performs "unsupervised" clustering by grouping feature-vectors around a given number of centroids. This process can be done in a visual program (see Figure 3) or interactively from within the VECTOR-SPACE graphical interface (as in Figure 2).

Supervised classification approaches (based on preliminary labelling information) are also available. In our generic model, a *class* is represented by a unique label and a list of IDs corresponding to known members of this class (this is typically determined during a preliminary training stage). Based on this information (which implicitly labels all known class members), it is possible to compare any unlabelled vector with centroid feature-vectors of the different classes, or its similarity with an established neighbourhood in the multidimensional feature space (*k-NN*). Such comparison allow to determine a measure of likelihood for this vector belonging to a certain class.

3.3. Musical Descriptors

An extensible set of descriptor algorithms allowing to extract features from objects are included with the system, many of which are aimed at musical content, pitch attributes, harmonic attributes, chord sequences, temporal attributes, variability, repeatability at various levels, degree of recurring figures etc. These features can be combined freely to constitute the N-dimensional vectors representing musical data in different OMAI algorithms.

The set of provided feature extractors can easily be extended using OM#s graphical programming environment or by coding in Lisp.

An example application could be: given a set of existing or generated material, extract a feature-vector for each of these using a selection of features, the system would cluster the input material accordingly.

For a composer these clusters could represent musical contrasts, variations, similarities etc. Having the ability to generate new material without necessarily knowing or caring exactly how it was generated, only that it ends up together with other material within a certain cluster, greatly reduces the amount of time needed to search for wanted solutions.

4. AI AND CAC, MUSICAL COMPOSITION, ARTISTIC NEEDS

Algorithms for ML, HMM, kNN, Neural Networks, Viterbi and more are part of the OMAI project. Tools and editors based on these algorithms are being developed as part of OMAI.

The Machine Learning and Clustering approaches described above are useful together with other AI algorithms to handle musical data traditionally worked with in CAC applications. They can also provide effective handles for more ill-defined, but arguably important and readily perceivable musical features such as "texture", "structure", "entropy" etc., musical qualities many modern composers use much time trying to control.

A possibly interesting observation of this project is that often rather simple versions of more advanced techniques together with an

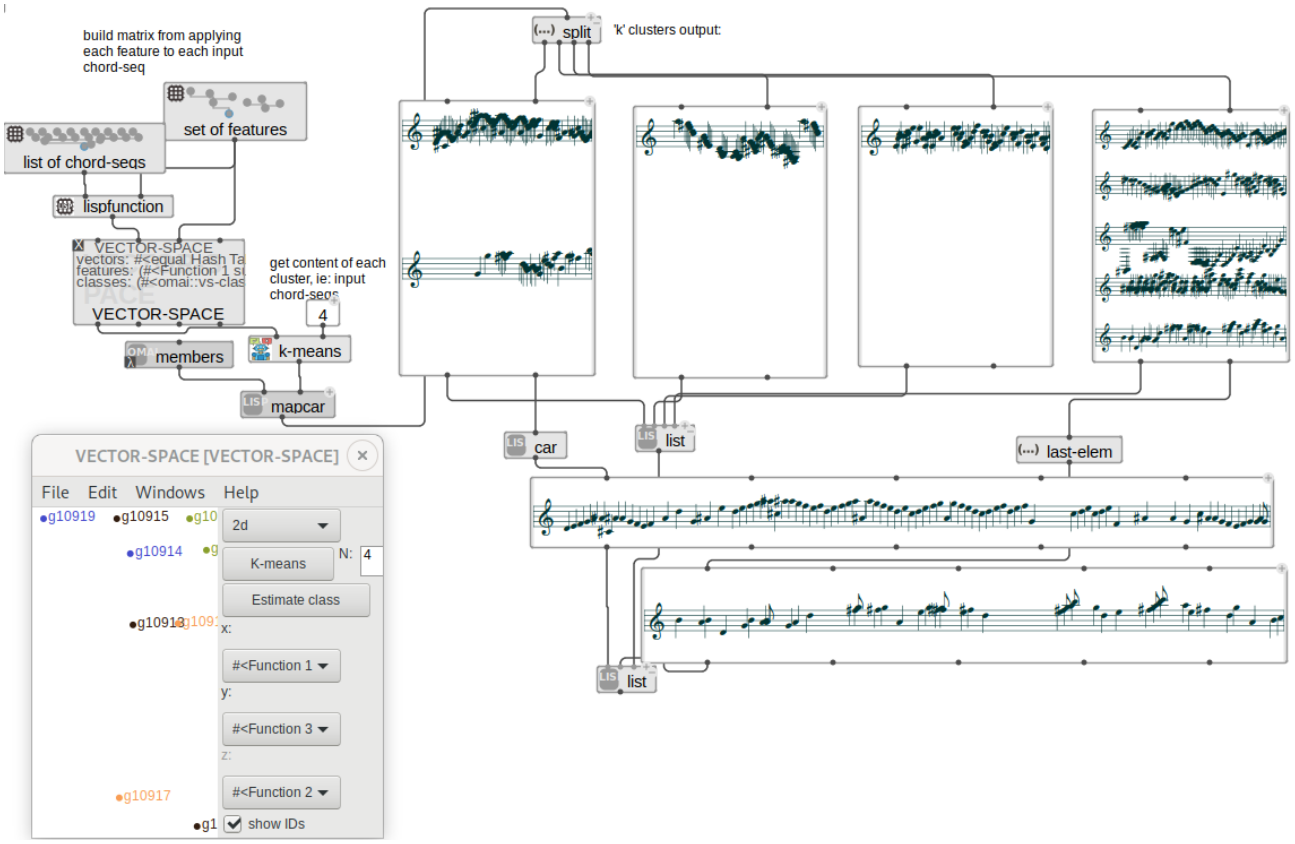


Figure 3: An example of clustering applied to a set of input structures. The output of the k-Means algorithm is 4 clusters. One sample from each cluster is displayed at the bottom

exploratory approach is very useful in the context of creative work. Stochastic methods are everywhere, and here as well, exact answers and reproducibility is often not particularly useful as such. In contrast: degrees of precision, possibly even errors, new solutions within certain constraints - can all be potential triggers for cool things to happen while composing.

Where a researcher most often would make sure to use large amounts of data to train an HMM or Neural Network, this is not necessarily interesting in our context. Just as Markov generation models seldom provide interesting results in composition work beyond 2-3 order, an ANN can output interesting results already after 2-10 iterations, and the *difference* between 2 given iterations are by itself very useful, e.g. to provide variants, development, embellishment etc.

Miller Puckette is quoted in the preface to "The OM Composer's Book"[14], illustrating some of the challenges:

> "CGM (Computer Generated Music, ie. Audio) is in effect building instruments (which were previously made of wood and the like), but CAC is in effect making the computer carry out thought processes previously carried out in human brains. Clearly, a piece of wood is easier to understand than even a small portion of a human brain. ... Ultimately, CAC researchers will have to settle for much less than a full understanding of even a single musical phenomenon. The best that can be hoped for is partial solutions to oversimplified versions of the real problems."

Good AI integrated in CAC tools may help bridge the gap between the composers mind and the systems they work with.

5. MODELLING, GENERATIVE ALGORITHMS

The OMAI system has been used by the author during recent composition work, e.g. to optimize fingering positions while scoring for guitar, and extracting generative patterns from analysis of input musical sequences using HMMs (Hidden Markov Models).

While developing the models used in this particular piece, the resulting scores have been evaluated along the way together with a professional guitarist to verify their level of 'guitaristicity'.

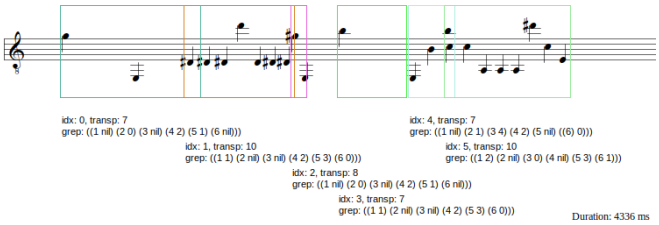


Figure 4: *Tabulature composition, generative algorithms use models of hands and fingers to suggest possible sets of notes*

6. RESOURCES AND DOWNLOAD

The sources of OMAI are open source, and distributed under the GPLv3 license, available along with documentation and examples at:
`https://github.com/openmusic-project/OMAI`

7. CONCLUSIONS

A project developing AI-based tools and techniques for Computer assisted composition and creative work, OMAI, is presented. The tools are part of OM# and the OpenMusic family of CAC environments.

Acknowledgments

This work is supported by IRCAMs Artistic Research Residency Program and the Norwegian Cultural Council.

8. REFERENCES

[1] Lejaren A. Hiller and Leonard M. Isaacson, *Experimental Music: Composition With an Electronic Computer*, McGraw-Hill, 1959.

[2] Philippe Pasquier, Arne Eigenfeldt, Oliver Bown, and Shlomo Dubnov, "An Introduction to Musical Metacreation," *Computers in Entertainment*, vol. 14, no. 2, 2016.

[3] F. Ghedini, F. Pachet, and P. Roy, "Creating Music and Texts with Flow Machines," in *Multidisciplinary Contributions to the Science of Creative Thinking (Creativity in the Twenty First Century)*, G. E. Corazza and S. Agnoli, Eds. Springer, 2015.

[4] Gérard Assayag, Shlomo Dubnov, and Olivier Delerue, "Guessing the Composer's Mind: Applying Universal Prediction to Musical Style," in *Proc. International Computer Music Conference (ICMC'99)*, Beijing, China, 1999.

[5] Léopold Crestel and Philippe Esling, "Live Orchestral Piano, a system for real-time orchestral music generation," in *Proceedings of the Sound and Music Computing Conference (SMC'17)*, Espoo, Finland, 2017.

[6] J. Françoise, N. Schnell, and F. Bevilacqua, "A Multimodal Probabilistic Model for Gesture-based Control of Sound Synthesis," in *ACM MultiMedia (MM'13)*, Barcelona, Spain, 2013.

[7] David Cope, *Experiments in Musical Intelligence*, A-R Editions, 1996.

[8] Shlomo Dubnov and Greg Surges, "Delegating Creativity: Use of Musical Algorithms in Machine Listening and Composition," in *Digital Da Vinci: Computers in Music*, Newton Lee, Ed. Springer, 2014.

[9] Gérard Assayag, Camilo Rueda, Mikael Laurson, Carlos Agon, and Olivier Delerue, "Computer Assisted Composition at IRCAM: From PatchWork to OpenMusic," *Computer Music Journal*, vol. 23, no. 3, 1999.

[10] Jean Bresson, Dimitri Bouche, Thibaut Carpentier, Diemo Schwarz, and Jérémie Garcia, "Next-generation Computer-aided Composition Environment: A New Implementation of OpenMusic," in *International Computer Music Conference (ICMC'17)*, Shanghai, China, 2017, Proceedings of the International Computer Music Conference.

[11] Marco Gillies, Rebecca Fiebrink, Atau Tanaka, Baptiste Caramiaux, Jérémie Garcia, Frédéric Bevilacqua, Alexis Heloir, Fabrizio Nunnari, Wendy Mackay, Saleema Amershi, Bongshin Lee, Nicolas D 'alessandro, Joëlle Tilmanne, and Todd Kulesza, "Human-Centered Machine Learning Workshop at CHI'16," in *Proc. CHI'16 – Extended Abstracts on Human Factors in Computing Systems*, San Jose, USA, 2016.

[12] Anders Vinjar and Jean Bresson, "OpenMusic on Linux," in *Linux Audio Conference*, Karlsruhe, Germany, 2014.

[13] C. D. Manning, P. Raghavan, and H. Schütze, *An Introduction to Information Retrieval*, Cambridge University Press, 2009.

[14] Miller Puckette, ," in *The OM Composer's Book 1*, Jean Bresson, Carlos Agon, and Gérard Assayag, Eds. Editions Delatour France / Ircam-Centre Pompidou, 2006, cote interne IRCAM: Agon06a.

APPLICATIONS OF JUPYTER NOTEBOOKS FOR AUDIO PLUGIN DEVELOPMENT

Travis Skare
CCRMA
Stanford University, USA
travissk@ccrma.stanford.edu

Jonathan Abel
CCRMA
Stanford University, USA
abel@ccrma.stanford.edu

ABSTRACT

Notebook interfaces in computing, introduced in the late 1980s, are in active modern use by data science and machine learning communities. Related to literate computing, notebooks encourage interleaving expository text with data, code, and figures, making for intuitive presentation of results. During development, they allow for nonlinear or exploratory development, and encourage building on prior research. We consider the application of such notebooks in audio plugin development and analysis, providing short example notebooks covering scenarios in DSP tutorials, white-box testing, black-box testing, and automation of third-party tools. While noting these workflows have been supported by commercial tools for decades, we exclusively use a range of FOSS languages and tools in our samples.

1. INTRODUCTION

We consider the scenario of developing a new phaser effect plugin for digital audio workstations. This involves tasks such as researching what a phaser is, learning how it operates, perhaps exploring the underlying filters we will use, generating some sound samples from competing products, and then moving to write and debug our own plugin.

We demonstrate a set of workflows using Jupyter[1] notebooks to explore accomplishing these tasks with notebook interfaces. Many of these workflows will be immediately familiar to researchers fluent in MATLAB, which offers a similar notebook paradigm, and indeed supports all these use cases with relevant tools and toolboxes. However, in this work we concentrate on open-source tools, libraries, and languages, without loss of generality.

A study of the field of the phaser effect is orthogonal to this work, since it is only an example. For very brief context, the effect is accomplished by applying a chain of time-varying allpass filters with a feed-forward signal path. This results in a pleasant modulation-class effect commonly used on electric guitar, electric piano, synthesizers, and more. We provide sound examples in our first example notebook, and more detail on the history and approaches is available in[2, 3], or in DAFX[4][1].

Our notebooks during development of this hypothetical plugin include:

- Python–Shows combining prose and code to generate an allpass filter, and plotting the filter response.
- Julia[5]–calling C++ instances of an STK[6] biquad class via a foreign function interface and wrapper library. This may be considered a "virtual breadboard" for development; we may edit our native C++ directly and have it driven with test signals by a higher-level language.
- Python–loading an arbitrary LADSPA plugin for which we may or may not have the source, and driving it with a test signal. This can be used for black-box validation of our own plugins or studying third-party effects.
- Faust[7] (via Python and the shell)–driving external tools in the course of implementation of a phaser. Faust is a very powerful domain-specific time-domain language and lets us code up a phaser in a few lines of code–in fact the standard library includes such modulation effects as primitives! Because the language does not yet integrate with Jupyter notebooks directly via a kernel or similar, we demonstrate shelling out to the Faust tools to have a notebook act as a build automation or report generation tool

These notebooks have been uploaded as part of this work; readers may wish to browse them after scanning this paper. However the descriptions in the paper are intended to stand on their own, and a screenshot of the first notebook is provided. The direct URL at the time of writing is https://github.com/tskare/lac2020demo; a redirector has been set up at at https://bit.ly/2TqcQuG in case this changes in the future, which is not expected.

We note some source repository browsers have notebook-viewing facilities for `.ipynb` and similar formats, which is convenient; we use GitHub for this work to demonstrate. In case the viewer does not support audio widgets, `phaserdemo.mp3` is provided as a standalone file for this case.

A side note on machine learning: Machine Learning and Deep Learning are very popular topics across many domains of research. Such notebooks are a common workflow in ML and Data Science; many getting started guides use them. Because they are so prolific, we intentionally avoid discussing ML workflows in this work and aim to stay within the digital audio effect development domain.

Finally, to aid in conveying motivation and use case, a demonstration video developed for the conference presentation will be provided/linked with the repository.

1.1. Installation

Readers may follow along by installing the following software:

[1] Section 2.4.2 in the Second Edition

Python: likely already installed on your system. We use Python 3 for this work; noting that Python 2 has been officially sunset as of January 1, 2020. We would suggest that if you do not have Python installed already, consult your system administrator or package manager, as this may affect your system in a wide manner.

Conda (*Optional)*: This work was developed using Conda, a package manager that is supported on Linux, MacOS, and Windows, and allows switching between different environments for different projects. https://docs.conda.io/en/latest/

Jupyter via `conda` or `pip`: https://jupyter.org/install

Julia via Conda, your package manager of choice, or from their homepage at https://julia-lang.org. The Julia Project homepage may offer the most up-to-date version; we updated to 1.3.1 before paper submission.

STK, the Synthesis Toolkit in C++[6]. This is for following along with the second example. A mirror is available at https://github.com/thestk/stk

Julia's **CxxWrap** package via the built-in package manager (press right-bracket, `]`, in the Julia REPL and enter "`add CxxWrap`"). This is for easily wrapping C++ classes. This is only one of a handful of methods for wrapping or calling C/C++ code. This seemed to work better than the built-in libraries when running inside Jupyter, but readers are encouraged to evaluate the others for their use case.

Faust via some package managers, or the Faust Homepage at https://faust.grame.fr/.

Next, we present the four use cases.

2. USE CASE: EDUCATION (PYTHON)

In this section we explore a standard use of notebooks, presentation of interleaved prose, code, and results. We note that this use of notebook interfaces is common in other domains.

Here we explain a simple digital phaser effect. In the opening paragraphs, we list some commercial phaser effects from MXR, Electro-Harmonix, and Eventide, and fetch a Creative Commons image of an MXR Phase 90 pedal from the web (local filesystem works as well and is better for posterity).

We provide an inline audio example of the phaser, so the reader may immediately understand what our desired end result may sound like.

A screenshot of the second half of this notebook is presented in Figure 1 [after the main paper text and bibliography]. Note we interleave explanatory text, an equation, Python numerics code, and filter response plots.

There, we explain the digital allpass filter that will be a building block for our implementation. The introduction links to resources we cite here, to guide readers to deeper study. LaTeX-style equations are rendered in the Markdown prose via MathJax.

Finally, we use `SciPy`'s `freqz` implementation to obtain the frequency and phase response of the first-order digital allpass filter. We leverage the example code from the `freqz` documentation to plot the frequency and phase responses inline in the notebook.

Readers who wish to dive deeper may download the notebook and experiment. For instance, they may wish to alter the allpass parameters g_i, or extend the notebook from an introductory level to a deep-dive level by adding text discussing virtual analog considerations.

While again we emphasize this exploration is a standard use of data science notebooks, rather than a novel work, we call out the benefits of interleaving product images, underlying equations, study of the building blocks, implementation, and sound example in a single browser window. Beyond display in a browser, JupyterLab also allows exporting notebooks with code, data, results, and commentary all "baked in" to slide-style presentations, HTML, lecture-note-style PDFs, or LaTeXwhich could be integrated directly into an academic paper.

Tools also exist to host live versions of the notebook, or have multiple researchers working in the same session; these are outside the scope of this paper.

3. USE CASE: WHITE-BOX PRODUCTION C++ ANALYSIS (JULIA)

Next, we consider the case of using notebooks to provide a "report" on production C++ code.

A variety of workflows for plugin development exist. Anecdotally, we hear it is common to prototype in a high-level language such as MATLAB before porting algorithms to optimized code, usually in C++. In recent years, Mathworks has even introduced compilation direct to plugins to facilitate prototyping and experimentation directly in DAWs.

In this section we propose use of notebooks to call C++ code in development. The hypothetical code under test is considered "white box;" that is, in this section our imaginary company has developed both the notebook and the plugin code. We may be porting C++ from a Matlab prototype, and wish to make sure inputs and outputs match, or we may be building our plugin from scratch and would benefit from a test bench that drives the plugin and obtains various plots, inputs, and outputs for analysis, or sharing with our development team.

The Julia language is used without loss of generality; we note that in the next section we will call C++ from Python for a different application. While outside the scope of this paper, interested readers might consider the `cffi` module (C Foreign Function interface) in Julia, or CPython extension capabilities. Finally, C++ interpreter kernels exist for notebook computing and we could write our plugin code directly in the notebook.

Development of this notebook is fairly straightforward. We imagine a use case is that we are debugging the Biquad filter present in the Synthesis Toolkit (STK)–this may be found in `src/BiQuad.cpp` in the STK repository. Julia supports multiple ways of calling C/C++ code, including a built-in `ccall`[2], designed to be a low-overhead, no-glue method of calling C and Fortran numerics libraries. Other methods such as `Cxx` and `CxxWrap` packages may be added from the built-in package manager. We use the latter, `CxxWrap`, currently available via GitHub[3], and installable via the built-in package manager as discussed in Section1.1.

[2]https://docs.julialang.org/en/v1/manual/calling-c-and-fortran-code/

[3]https://github.com/JuliaInterop/CxxWrap.jl

3.1. C++ Work Required: Wrapping the Class

The CxxWrap approach requires some prep work outside of the notebook, but this is straightforward. We write some standardized glue code then use CMake to build a `.so` shared library.

We must define a function to expose our method as a Julia module. This is as follows; not all methods are included for brevity.

```
#include "jlcxx/jlcxx.hpp"
JLCXX_MODULE define_module_biquad(
    jlcxx::Module& mod)
{
  mod.add_type<stk::BiQuad>("STKBiQuad")
    .constructor<>()
    .method("setCoefficients", &stk::BiQuad::
        setCoefficients)
    .method("sampleRateChanged", &stk::BiQuad
        ::sampleRateChanged)
    .method("setB0", &stk::BiQuad::setB0)
    .method("setB1", &stk::BiQuad::setB1)
    .method("setB2", &stk::BiQuad::setB2)
    .method("setA1", &stk::BiQuad::setA1)
    .method("setA2", &stk::BiQuad::setA2)
    .method("tickOne",&stk::BiQuad::tickOne);
}
```

Whereas most functions like `setCoefficients` are native STK functions, `tickOne` was added to work around specifying an overloaded function in the call to `add_type`. STK's `tick` has several variants and this was currently the easiest way we found to disambiguate between them. Also, this way we do not depend on familiarity with the `StkFrames` class and only deal with primitive types in the notebook. We provide our modified source in `BiQuadJulia.cpp` and associated CMake file; the only other addition for the sake of completion is a `tickOne` implementation. A minimal one:

```
float BiQuad::tickOne(float in) {
  StkFrames frames(1, 1);
  frames[0] = in;
  StkFrames framesout = tick(frames);
  return framesout[0];
}
```

On the module side, loading the library begins with the minimal:

```
module STKBiquad
  using CxxWrap
  @wrapmodule("/home/$USER/src/third_party/
      stk/src/lib/libbiquadtestlib", :
      define_module_biquad)

  function __init__()
    @initcxx
  end
end
```

Now we may drive and plot our C++ function in Julia.

Next, we explore loading shared libraries from another language, Python:

4. USE CASE: BLACK-BOX BINARY PLUGIN ANALYSIS (PYTHON)

We may wish to drive and analyze a plugin on a "virtual lab bench." Perhaps we wish to black-box test our build artifacts to validate with a test suite, or perhaps we wish to script an analysis of which third-party saturation plugins alias when running at 44.1kHz, for example.

In this section we load and drive a simple LADSPA plugin. A simple v1 plugin is launched directly as a standard library; we note a complete production workflow would instantiate and call LV2 plugins through the Liiv library. We note the existence of Python-LADSPA projects on GitHub; we did not evaluate these so that our notebook requires no dependencies beyond the built-in `ctypes`.

Here, we enumerate available LADSPA plugins from the commandline (via the `listplugins` program included with the SDK) and then in our notebook, use the `ctypes` module to load any of those plugins. We declare the LADSPA interface to `ctypes` in terms of relevant structures and functions, then may load the library and create a plugin in memory. LADSPA is fairly unique in that the plugin libraries expose only one function, which retrieves a reference to the Nth plugin in the library. The reference structure in turn contains function pointers which allow connecting control and sound buffers, reading metadata, and processing audio data.

Because a LADSPA wrapper may be more immediately useful than our notebooks, we include the wrapper code directly in Listing 1. Users may load and call into a plugin with Python code such as:

```
plugHandle = 0
# Load the second plugin in a shared library.
plugPtr = loadPlugin(
  '/myhome/dev/testplugin.so', 1)
plugInst = plugPtr[0] # dereference pointer
print("Plugin: %s by: %s, (c) %s" % (
    plugInst.Name,
    plugInst.Maker,
    plugInst.Copyright))
print('ports:')
for i in range(plugInst.PortCount):
    print("%s - %s" % (
      plugInst.PortNames[i],
      plugInst.PortDescriptors[i]))
```

As we provided type information to `ctypes`, runtime type checking is performed. LADSPA typedefs were included in the wrapper to help avoid type confusion and increase readability.

5. USE CASE: CALLING EXTERNAL TOOLS (FAUST)

A final, fourth notebook considers the case where we would like to use the exploratory, cross-media notebook paradigm but have existing tools and do not want to use C foreign function tools or write a new notebook kernel.

As a concrete case, consider that we wish to report on the architecture and results of a Faust plugin in development.

We do note the existence of `faust_python`[4] from 2015

[4] https://github.com/marcecj/faust_python

and a wrapper for Julia widgets that leverages this, currently in development over the last months[5].

Development of this notebook is perhaps the most straightforward. A notebook cell that begins with the exclamation point operator will execute that command in the shell.

```
!echo hi world
```

will output "hi world", for example. We can use this functionality to display a `.dsp` file in development, call `faust` to compile it, invoke `faust2svg` to generate the system diagram, and display that artifact with IPython's native SVG rendering support in the notebook.

This may be seen as build automation, though extending things a bit further, it could be used to combine algorithm descriptions, relevant Faust code, plots of system response, and generated audio demos. There are opportunities for significant further work here, as described in the next section. On its own, this style of notebook can demonstrate that processes spanning multiple tools may be combined and automated in place of a Makefile or script. We can glue together existing workflows quickly, and spend more time on exploration and development of our hypothetical plugin–a common goal among all these processes.

6. FURTHER WORK

The "virtual test bench" that runs LADSPA plugins would ideally be extended to use LV2 and/or VST, as development has moved to those platforms for Linux (for MacOS, AudioUnit is worth considering).

There are many opportunities for extending the Faust notebook. As mentioned, there are some open-source libraries in the field for loading plugins or wrapping Faust's compilation functionality with the Python foreign function interface. This could be investigated, toward having the full Faust development workflow available to a notebook. We could also call the excellent Faust web-based tools and compiler as an API, or have those system-local, to be able to develop, build, and test actual plugin binaries within one notebook. Especially once inline coding opportunities are added, this could be the framework for a set of interactive articles on introductory effects plugin signal processing.

7. CONCLUSIONS

We suggested the use of notebook workflows, popular in data science and machine learning communities, for subtasks involved in plugin development. Both Python and Julia were used at different times, and we shelled out to Faust to demonstrate driving tools not yet integrated in the notebook ecosystem. Markdown provides prose and equation support for all notebooks.

As a secondary tangible result, we provide generic wrapper code for loading LADSPA v1 plugins in Python.

8. ACKNOWLEDGMENTS

Thanks to the anonymous reviewers and conference organizers, especially for this unique year. And of course to the authors and contributors to all the FOSS software mentioned in these workflows. They combine to make audio development an exciting area for research, education, and hobby development.

[5] https://github.com/hrtlacek/faustWidgets

9. REFERENCES

[1] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing, "Jupyter notebooks – a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt, Eds. IOS Press, 2016, pp. 87 – 90.

[2] Julius O. Smith III, *Physical audio signal processing: For virtual musical instruments and audio effects*, W3K publishing, 2010.

[3] Julius O. Smith III, "An allpass approach to digital phasing and flanging," in *ICMC*, 1984.

[4] Udo Zölzer, *DAFX: digital audio effects*, John Wiley & Sons, 2011.

[5] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah, "Julia: A fresh approach to numerical computing," *SIAM review*, vol. 59, no. 1, pp. 65–98, 2017.

[6] Perry R Cook and Gary P Scavone, "The synthesis toolkit (STK).," in *ICMC*, 1999.

[7] Yann Orlarey, Dominique Fober, and Stéphane Letz, "Faust: an efficient functional approach to dsp programming," 2009.

Allpass filters

The phaser effect is obtained by combining an even number of allpass filters with a feedforward gain path. The allpass filters are formed as follows with time-varying parameter $g_i(n)$:

$$\text{Allpass}_i = \frac{g_i + z^{-1}}{1 + g_i z^{-1}}$$

Then we can plot frequency and phase response via `SciPy` and plot via `Matplotlib`:

[38]:
```
import numpy as np
from scipy import signal
import matplotlib.pyplot as plt
g = 0.5
w, h = signal.freqz(b=[g, 1], a=[1, g], worN=1000)
```

[39]:
```
fig = plt.figure()
plt.title('Digital allpass frequency response')
ax1 = fig.add_subplot(111, label='freqresp')
plt.plot(w, 20 * np.log10(abs(h)), 'b')
plt.ylabel('Amplitude [dB]', color='b')
plt.ylim([-1,1])
plt.xlabel('Frequency [rad/sample]')
ax2 = ax1.twinx()
angles = np.unwrap(np.angle(h))
plt.plot(w, angles, 'g')
plt.ylabel('Angle (radians)', color='g')
plt.grid()
plt.axis('tight')
plt.show()
```

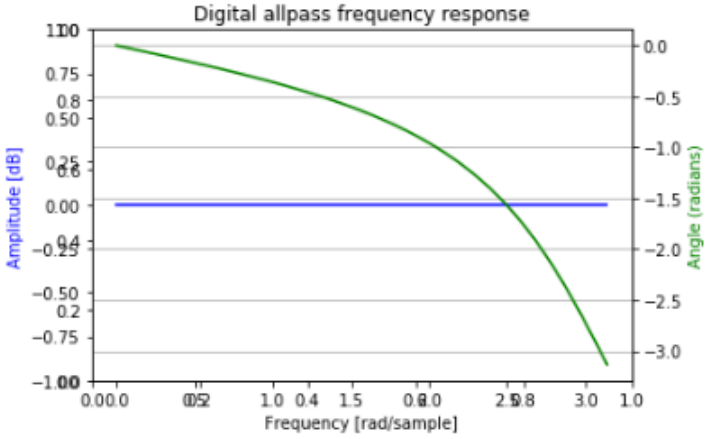


Figure 1: *A screen capture of the second half of Notebook 1, described in Sec. 2*

```
import ctypes

# Declare interfaces to the structures and functions we'll call.

# typedefs and constants
LADSPA_Data = ctypes.c_float
LADSPA_Properties = ctypes.c_int
LADSPA_Handle = ctypes.c_void_p

LADSPA_PortDescriptor = ctypes.c_int
kLADSPA_PORT_INPUT   = 0x1
kLADSPA_PORT_OUTPUT = 0x2

LADSPA_PortRangeHintDescriptor = ctypes.c_int;
# hint constants omitted so this fits on one page; please reference the .h file.

class LADSPA_PortRangeHint(ctypes.Structure):
    pass
LADSPA_PortRangeHint._fields = [
    ("HintDescriptor", LADSPA_PortRangeHintDescriptor),
    ("LowerBound", LADSPA_Data),
    ("UpperBound", LADSPA_Data)
]

class LADSPA_Descriptor(ctypes.Structure):
    pass
LADSPA_Descriptor._fields_= [
    ("UniqueID", ctypes.c_long),
    ("Label", ctypes.c_char_p),
    ("Properties", LADSPA_Properties),
    ("Name", ctypes.c_char_p),
    ("Maker", ctypes.c_char_p),
    ("Copyright", ctypes.c_char_p),
    ("PortCount", ctypes.c_ulong),
    ("PortDescriptors", ctypes.POINTER(LADSPA_PortDescriptor)),
    ("PortNames", ctypes.POINTER(ctypes.c_char_p)),
    ("PortRangeHints", ctypes.POINTER(LADSPA_PortRangeHint)),
    ("ImplementationData", ctypes.c_void_p),

    # Interface is via function pointers in the struct.
    ("instantiate", ctypes.CFUNCTYPE(LADSPA_Handle, ctypes.POINTER(LADSPA_Descriptor),
        ctypes.c_ulong)),
    ("connect_port", ctypes.CFUNCTYPE(ctypes.c_int, LADSPA_Handle, ctypes.c_ulong)),
    ("activate", ctypes.CFUNCTYPE(ctypes.c_int, LADSPA_Handle)),
    ("run", ctypes.CFUNCTYPE(ctypes.c_int, LADSPA_Handle)),
    ("run_adding", ctypes.CFUNCTYPE(ctypes.c_int, LADSPA_Handle, ctypes.c_ulong)),
    ("run_adding_gain", ctypes.CFUNCTYPE(ctypes.c_int, LADSPA_Handle, LADSPA_Data)),
    ("deactivate", ctypes.CFUNCTYPE(ctypes.c_int, LADSPA_Handle)),
    ("cleanup", ctypes.CFUNCTYPE(ctypes.c_int, LADSPA_Handle))
]

# The actual library has only one function.
# The argument, |index|, can choose one of N plugins in the library.
# Indices beyond that range are NULL.
def loadPlugin(name = '/usr/lib/ladspa/delay.so', index=0):
    plugin = ctypes.CDLL(name)
    plugin.ladspa_descriptor.argtypes = [ctypes.c_ulong]
    plugin.ladspa_descriptor.restype = ctypes.POINTER(LADSPA_Descriptor)
    return plugin.ladspa_descriptor(index)
```

Listing 1: *Code to define the LADSPA interface in Python via* `ctypes`.

PD-FAUST MACKIE CONTROL

Albert Gräf

IKM, Music-Informatics
Johannes Gutenberg University (JGU) Mainz, Germany
aggraef@gmail.com

ABSTRACT

The paper describes faust-mcp, a Pd abstraction which interfaces Faust to control surfaces utilizing the Mackie Control Protocol (MCP). It builds on the author's Pd-Faust software which enables you to run dsp programs (such as synthesizers and effects) written in Grame's Faust programming language inside Pd. The add-on can be used to control Faust dsps in Pd using MCP-compatible controller hardware and software.

1. INTRODUCTION

Grame's Faust is a functional programming language which greatly facilitates the programming of audio processing and instrument plugins [1]. Faust programs can be compiled to native code for an abundance of different signal processing environments and plugin standards. Pd-Faust is a plugin which allows Faust programs to run in Miller Puckette's graphical real-time patching software Pd[1]. It offers dynamic loading (and reloading) of Faust modules, MIDI[2] and OSC[3] control and sequencing, as well as automatic GUI generation (in the form of graph-on-parent subpatches), cf. [2].

Faust dsps typically offer a number of different controls for various parameters, such as the cutoff frequency and resonance of a filter, oscillator and envelope parameters of a synthesizer, etc. These are represented in a Faust program by means of so-called UI (user interface) elements, cf. [3]. While Pd-Faust lets you generate Pd GUIs for all UI elements of a Faust dsp in an automatic fashion, it is often desirable to control such parameters by means of some external, physical control surface instead. To these ends, Faust lets you map MIDI messages to each UI element by corresponding meta-data in the UI element specifications. For instance:

```
res = hslider("res [midi:ctrl 20]", 3, 0, 20, 0.1);
cutoff = hslider("cutoff [midi:ctrl 21]",
                 6, 1, 20, 0.1);
```

The controller mappings are in the square brackets following the control names. The `midi:ctrl tag` specifies the kind of MIDI message to be received by the program, in this example CC20 for the resonance and CC21 for the cutoff control, respectively. These input values can then be used in the Faust definition of the dsp as needed, e.g., for computing the required filter coefficients. UI elements for output (so-called bargraphs) are available as well, and all of these can be mapped to different kinds of MIDI messages (pd-faust only supports MIDI CC bindings at this time, however).

[1] http://puredata.info/
[2] http://midi.teragonaudio.com/
[3] http://opensoundcontrol.org/

So Faust dsps can already be controlled by plain old MIDI controllers with a few knobs or faders quite easily, by just adding a small amount of meta-data to the Faust program. However, this method quickly becomes unwieldy when using a lot of different Faust programs in the same patch, since most MIDI fader boxes won't easily accommodate a large amount of parameters, and remapping the controls is often a tedious task. Thus some form of automatic mapping of the controls is needed, and you also want to be able to quickly switch between different banks of controls.

As luck would have it, this kind of functionality is readily provided by so-called DAW (digital audio workstation) controllers, and there is an established MIDI-based protocol for these, the Mackie Control Protocol (MCP). This is what faust-mcp uses to interface pd-faust to compatible controllers. In the paper, we give a quick introduction to MCP, discuss how the faust-mcp package utilizes it, illustrate faust-mcp's usage with an example, and finally discuss some future work to further improve the interface.

2. MACKIE CONTROL

DAW controllers were invented to ease the operation of digital audio workstation (DAW) software [4]. They often resemble a mixer control surface, which seems sensible because mixing is a big part of what a DAW program does, and most musicians and studio engineers will be well familiar with that kind of interface.

Thus DAW controllers typically have a number of faders and knobs used to input track parameters such as volume, panning, sends, etc., along with buttons for playback control and various other functions. On the output side, they may also provide useful feedback through motor faders indicating the current values, LED strips showing meter values in real-time, a timecode display, and "scribble strips" (little LCD displays) to denote track and parameter information. The knobs and faders are typically organized into banks of 8 which can be switched at the push of a button to accommodate a large number of different parameters (which is why you need the scribble strips to figure out which tracks and parameters are actually represented on the control surface at any one time).

The first DAW controller was produced by the mixer manufacturer Mackie for Logic by emagic, and was subsequently modified to support a number of other DAW programs, see Fig. 1 [5]. The Mackie Control also set the de facto standard MIDI protocol for this kind of gear, although there are some alternatives, most notably the HUI protocol developed for Digidesign's Pro Tools.

Nowadays, DAW controllers can take many shapes and forms, ranging from tiny gadgets just providing playback con-

Figure 1: Mackie Control [5].

trols, keyboard controllers with added knobs and faders, and even foot controllers with switches and expression pedals, to full-blown mixer-like control surfaces. Most of these speak the Mackie Control Protocol (MCP), which is also supported by most DAW programs these days. Some prominent examples of these are the Mackie Control Universal, the Icon Platform M, Behringer's X-Touch series, as well as the Presonus Faderport controllers. There are also software implementations on mobile platforms, such as humatic's TouchDAW[4], which emulates a full Mackie-compatible DAW controller on Android devices, and can be connected to PCs either via USB or LAN (using RTP-MIDI or ipMIDI in the latter case); see Fig. 2.

MCP is in fact just a subset of MIDI, so it can be transmitted over any kind of MIDI connection, but it uses MIDI in its own, somewhat idiosyncratic way. Here is a brief summary of the most important features relevant for our purpose:[5]

- The knobs are usually rotary encoders which transmit relative changes in sign-bit encoding (thus, e.g., the CC values 1 and 65 denote an incremental change by +1 and -1, respectively). This includes pan (mapped to CC16 to CC23), and often there's also a big jog wheel (CC60) used to change the position of the playback cursor on the timeline.
- The faders emit pitch bend messages on the first eight MIDI channels (rather than MIDI CC) to take advantage of the 14 bit resolution these messages provide.
- The buttons used to control playback and other functions emit note messages such as note 94 and 93 for transport control start and stop. Thus MCP *always* needs a separate MIDI connection to the DAW where *only* MCP data is transmitted, lest you risk the knobs being pushed triggering actual notes in some synthesizer plugin.
- MCP controllers also *receive* data to properly set the current values of encoders and faders. In addition, on the back connection, channel pressure (monophonic aftertouch) messages are employed to denote meter values, MIDI CCs 66 to 73 represent the timecode display, and sysex messages encode the contents of the scribble strips.

[4] `https://www.humatic.de/htools/touchdaw/`

[5] Although MCP is widely used, there doesn't seem to be an official specification of the protocol anywhere on the internet. However, a fairly comprehensive overview of the protocol (albeit without the feedback messages) can be found at `http://www.jjlee.com/qlab/MackieControlMIDIMap.pdf`.

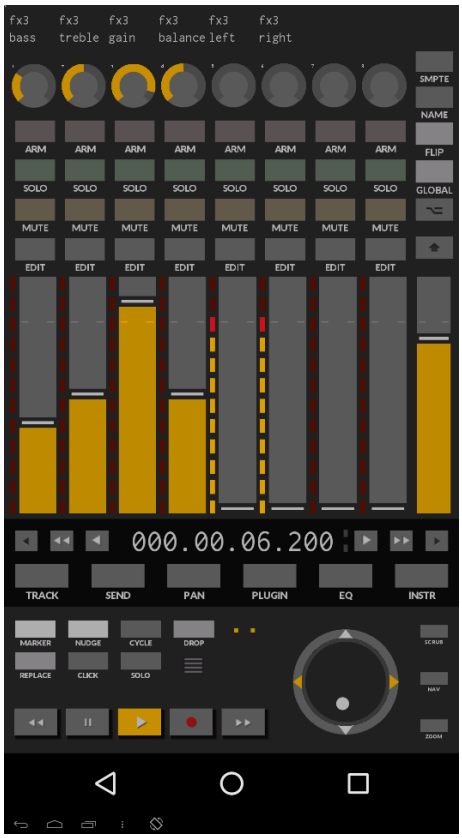


Figure 2: TouchDAW running on Android.

It is also worth noting here that while the encoders, faders, and transport controls should work the same with any DAW, the other (button) controls are much less standardized and may vary a lot in function depending on the DAW program that you use. Therefore many Mackie-compatible controllers ship with overlays for popular DAWs. Likewise, TouchDAW lets you configure the target DAW and changes some of its button layout and labeling accordingly.

3. THE FAUST-MCP PACKAGE

faust-mcp is distributed as open-source software on Github.[6] The package contains a Pd abstraction mcp.pd, along with some helper abstractions and externals, and a few examples. To use it, you'll obviously need Pd to run the patches, an installation of Grame's Faust compiler (and gcc) to compile your

[6] `https://github.com/agraef/faust-mcp`

Faust programs, and an MCP-compatible controller. We have tested the package with the Behringer X-Touch controllers (including the X-Touch One and Mini), the Studiologic Mixface, the Korg nanoKontrol2, and humatic's TouchDAW, but any MCP-compatible controller should work according to the capabilities it offers.

faust-mcp is built on top of pd-faust, and the accompanying externals are written in the author's Pure programming language [6], so both pd-faust and pd-pure need to be installed and enabled in Pd. Sources and binary packages for all of these can be found on the Pure website, which also provides detailed installation instructions.[7]

4. HOW IT WORKS

Basically, faust-mcp is a specialized MIDI mapper which translates MCP to standard MIDI control change (CC) messages and vice versa. Apart from the requisite MIDI bindings in the Faust programs, no manual setup is required; once the patch has been loaded, the mcp.pd abstraction keeps track of all the MIDI controls in all Faust dsps and configures itself accordingly in a fully automatic fashion. Note that in the current implementation, *only* controls with MIDI bindings will show on the MCP surface.

The faders and encoders of the MCP device are linked to the MIDI controls of your Faust dsps, so moving them changes the controls of the dsp accordingly. Conversely, changing the controls in the Pd GUI sets the controls of the device (if it supports feedback). In faust-mcp, the faders and encoders in each strip can be used interchangeably (and will move in lockstep if the device supports feedback), to accommodate any kind of MCP device which has any faders or encoders at all. *Passive* Faust controls (bargraph elements which output control values rather than reading them) are also supported and will be displayed using the meter strips of the MCP device if it has those (for instance, you can see these in the `left` and `right` strips of the `fx3` unit in Fig. 2).

The controls are organized into banks of eight faders and encoders. The abstraction provides as many banks as needed to represent all MIDI controls of all Faust dsps, ordering the controls by increasing MIDI CC numbers. The usual bank and channel controls on the MCP device can be used to switch between different banks as needed, so that all controls with MIDI bindings become accessible.

Scribble strips are also supported (as can be seen at the top of Fig. 2); they will show the name of the Faust units and controls assigned to each fader and encoder, or display the corresponding parameter values. Also, if you're using the included midiosc.pd abstraction, the transport keys of the device can be used to control playback. There are a number of other useful features like these, which will be described in Section 6.

The faust-mcp package contains a few examples which can be run straight from the source directory. The sources also include a small collection of sample Faust instruments and effects in the dsp subdirectory. Before running any of the examples, you'll have to compile these with the Faust compiler.

[7] See `https://agraef.github.io/pure-lang/`. Binary packages for Arch, Debian, and Ubuntu can be found on the Open Build Service, please check the "Pure on Arch" and "Pure on Debian/Ubuntu" wiki links on the website.

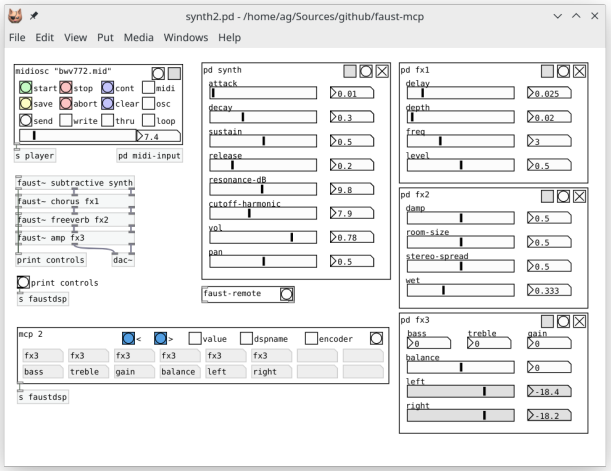


Figure 3: Pd patch running faust-mcp.

A Makefile is included, so you can just type `make` in the dsp folder to do this. The provided examples have all been set up so that the MCP device is expected to be connected to Pd's second MIDI port, so you'll have to configure your Pd MIDI connections accordingly. The included README file describes this in more detail.

Of course, you can also use the abstraction in your own patches. To do this, it's enough to copy the mcp folder to the directory containing your patch and Faust modules, or to any folder on Pd's library search path. To insert an instance of the abstraction into your patch, create an object (`Ctrl+1`) and type `mcp` followed by the MIDI port number to which the MCP device is connected.[8] Then connect the abstraction's single outlet to whatever `faust~` objects you wish to control, or just send it to the `faustdsp` receiver which is read by all Faust modules present in the patch. In either case, MIDI CC data emitted by the abstraction is encoded in the author's SMMF Pd message format[9], which is also the format used by pd-faust to encode all MIDI messages.

For instance, `mcp 2` connects to the device on Pd's second MIDI port. In principle any of Pd's MIDI ports can be used there (port 1 being the default). But as we already mentioned, MCP uses note and control data in its own peculiar way, thus you should make sure that live MIDI input to the Faust dsps is kept separate from the MCP data.

5. EXAMPLE

Fig. 3 shows the synth2.pd example from the faust-mcp package; please also revisit Fig. 2 to see how the same patch looks on the MCP device (TouchDAW in this case). In both figures the third (and last) bank of controls is shown. This example illustrates all the various elements: several `faust~` objects along

[8] Note that only a *single* instance of the mcp patch is needed for any running Pd instance, not one per toplevel patch!

[9] `https://bitbucket.org/agraef/pd-smmf`

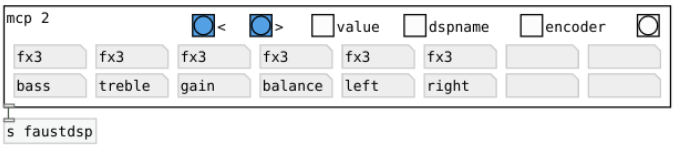


Figure 4: mcp abstraction closeup.

with their Pd GUIs, the `midiosc` abstraction which can be used to play back a MIDI file and record automation data, and the `mcp` abstraction itself. Note that faust-mcp ships with a special version of the midiosc abstraction which has been modified so that the MCP transport controls can be used with it.

Let's have a closer look at the `mcp` abstraction in the example (cf. Fig. 4). It shows a mirror of the scribble strips, as they will render on the MCP device, as well as a few buttons and toggles in the top row. All these functions are also available using corresponding controls on the MCP device, as described below; in the following list we give the equivalent MCP functions in parentheses.

- The first two bang controls, labeled < and >, switch to the previous and next bank of eight faders, respectively. (MCP: bank left/right keys)
- The `value` toggle, when engaged, shows the current values of the controls in the top row of the scribble strips. (MCP: touch a fader, or push an encoder)
- The `dspname` toggle switches the scribble strips between showing the instance and the dsp name of the Faust unit. (MCP: F1 key)
- The `encoder` toggle switches between two alternative display styles (`fan` and `pan`) for the encoder LED rings. Fan style (the default) shows an arc from 0 to the current value, while pan style shows just a single tick between min and max markers. (MCP: F2 key)
- The bang control on the right resets the internal state of the abstraction and re-displays the scribble strips. (MCP: F3 key)

Note that the controls in the abstraction are not meant to replace a real MCP device; they merely provide you with the most essential functions in case your MCP device lacks some of these controls. Also, the facsimile of the scribble strips will be helpful if your device has no display.

In the following section, we discuss the meaning of all available MCP controls in some detail.

6. CONTROLS

The primary purpose of the mcp abstraction is to take controller input from the mixer strips (faders and encoders) of your device and map them to the corresponding MIDI control changes of the Faust units in your patch. It also does the reverse translation, providing feedback to the MCP device (moving motor faders or lighting up LEDs if your device has any of those) if you change the Faust controls in the patch. In addition, the abstraction offers various other useful functions, mostly accessible through special keys on the MCP device:

- **Bank changes:** As already mentioned, the controls are organized into banks of size eight (which matches the number of strips on most MCP devices). The bank left and right buttons can be used to switch between these, so that all Faust controls become accessible. The channel left and right buttons, if available, move through the controls one strip at a time; this is useful, in particular, with single-strip devices like the X-Touch One.
- **Scribble strips:** Instance/dsp and control names are shown in the scribble strips of the device (if available), and touching the faders or pushing the encoders toggles the value display in the top line of each scribble strip.
- **Special dsp controls:** Each Faust dsp has three special controls, which correspond to the buttons in the upper right corner of the generated Pd GUI (cf. Fig. 3): `record` (a toggle which arms the unit for recording of OSC automation data when used with the midiosc abstraction), `reset` (a bang control which resets all controls to their initial values), and `active` (a toggle which turns the unit on or off). With the mcp abstraction these are assigned to the record, solo/select and mute buttons of the device, respectively. The rec and mute buttons also provide feedback, i.e., the buttons light up when the option is engaged. In the case of `mute` this actually means that the unit is *de*activated, so the corresponding GUI toggle is *off*. Pressing the select or solo button simply resets all controls of the dsp to their initial values, without lighting any buttons.

  Note that the special dsp controls always apply to the dsp *as a whole*, so pressing the button on *any* strip currently assigned to a given dsp will change the mute or record status of *all* the other buttons currently assigned to the same dsp.
- **Display options:** The following options are assigned to some of the function keys of the MCP device: F1 switches the scribble strips between instance and dsp name of the Faust units; F2 switches the encoder style between `fan` and `pan`, as discussed in the previous section; and F3 tells the abstraction to update its internal state and re-display the scribble strips (which can be used to force an update of the display, e.g., after editing and reloading Faust units).
- **Playback and transport:** When used with the included (modified) version of the pd-faust midiosc player, the transport controls will work as follows: the rewind key moves the playhead to the beginning of the MIDI file, fast forward moves it to the end; stop stops, and play toggles playback; record toggles the player's OSC automation recording; cycle toggles the player's loop function; and the big jog wheel and the cursor left/right keys move the playhead in smaller and larger steps, respectively. In addition, the function keys F4, F5 and F6 are assigned to some special OSC recording functions (`save`: save the currently recorded automation data to a text file; `abort`: delete the automation data of the current take; and `clear`: delete the entire automation sequence). Please check the pd-faust documentation for more details on how these operations are used.[10]

[10] `https://agraef.github.io/pure-docs/pd-faust.html`

- **Timecode:** When used with the midiosc player, the timecode display shows the time (in h/m/s/tenths of seconds) of the current playhead position.

Obviously, some of these functions may or may not be available depending on the MCP device that you have. The Mackie, Faderport 8 and X-Touch devices should enable all features, but some lesser MCP devices may not offer transport or function keys, push encoders, fader touch detection, scribble strips, or a timecode display.

Finally, let us mention in passing that even if your MIDI controller does *not* have built-in MCP support, chances are that if it has enough faders, knobs and buttons, you can make it work as an MCP-compatible device using the author's midizap program [7]. For instance, faust-mcp works just fine with the Akai APCmini, or even the Harley Benton MP-100 foot controller, using the corresponding MCP emulations included in the midizap distribution.

7. FUTURE WORK

While faust-mcp is perfectly usable already, we still consider it work in progress. Here are some things we may want to address in future versions:

- The most notable limitation right now is that faust-mcp only covers dsp controls which already have MIDI bindings. This simplifies the implementation a lot. However, another option would be to go through pd-faust's OSC layer instead. This would allow arbitrary controls to be mapped, without having to configure MIDI bindings beforehand.
- It would be nice to offer more layout options (i.e., how "pages" for different Faust units are organized, and how the controls are ordered).
- Currently controls mapped to the same MIDI CC in different Faust units will be mapped to the same MCP control. This is an outright bug and will hopefully be fixed by the time you read this.
- faust-mcp is currently hard-wired to use 8-fader banks, which is what most dedicated DAW controllers offer. But there are devices with smaller and larger bank sizes (as well as extender units which can be added to existing DAW controllers), so it makes sense to provide alternative versions of the mcp abstraction to accommodate all common sizes.
- For DAW controllers without motorized faders, the current fader positions will often be way off from the actual Faust control values, especially after bank switches. The usual way to deal with this is a "pickup" (a.k.a. "takeover") mode which makes sure that controls start moving only when the fader "picks up" the actual value. Obviously, it would be nice to have this in faust-mcp as well, at least as an option.[11]
- There should be some form of musical timecode display. Currently only physical time in h/m/s/tenths is shown. This is due to limitations in the current pd-faust implementation which doesn't report musical time.

[11] As a remedy for the time being, if your controller doesn't have motor faders, then it may be safer to just use the encoders of your device instead, because these always emit changes relative to the current control value.

8. REFERENCES

[1] Yann Orlarey, Albert Gräf, and Stefan Kersten, "DSP programming with Faust," in *Proceedings of the 4th International Linux Audio Conference*, Karlsruhe, 2006, pp. 39–47, ZKM.

[2] Albert Gräf, "Pd-Faust: An integrated environment for running Faust objects in Pd," in *Proceedings of the 10th International Linux Audio Conference*, Stanford University, California, US, 2012, pp. 101–109, CCRMA.

[3] Yann Orlarey, Dominique Fober, and Stephane Letz, "Syntactical and semantical aspects of Faust," *Soft Computing*, vol. 8, no. 9, pp. 623–632, 2004.

[4] Colby N. Leider, *Digital Audio Workstation*, McGraw-Hill, Inc., New York, NY, USA, 2004.

[5] Mark Wherry, "Mackie control : DAW control surface," Sound On Sound, Dec. 2003, `https://www.soundonsound.com/reviews/mackie-control-universal`. Last access: Dec. 2019.

[6] Albert Gräf, "Signal processing in the Pure programming language," in *Proceedings of the 7th International Linux Audio Conference*, Parma, 2009, Casa della Musica.

[7] Albert Gräf, "midizap: Controlling multimedia applications with MIDI," in *Proceedings of the 17th International Linux Audio Conference*, Stanford University, California, US, 2019, pp. 113–120, CCRMA.

EXPRESS DATA PATH KERNEL OBJECTS FOR REAL-TIME AUDIO STREAMING OPTIMIZATION

Christoph Kuhr

Brühl, Germany
christoph.kuhr@web.de

Alexander Carôt

Anhalt University of Applied Sciences
Köthen, Germany
alexander.carot@hs-anhalt.de

ABSTRACT

Using a JACK media clock listener to synchronize JACK to an AVTP media clock talker results in performance issues when used with a raw Ethernet socket under Linux. The packet rate of a class A AVTP audio stream of 8 kHz triggers too many interrupts in the CPU. As a result a JACK audio cycle has only $125\ \mu sec$ to process the audio data of all JACK clients. This restriction prevents such a system from real-time signal processing. The extended Berkley Packet Filter in combination with the express data path kernel features, that are integrated in the Linux kernel since version 4.8, are investigated. We could optimize the media clock synchronization by using a eBPF XDP program for pre processing of the stream packets. Our described solution is meant as an alternative to the usage of generic raw sockets.

1. INRODUCTION

Soundjack [1] is a real-time communication software using peer to peer connections, to connect up to five participants with each other. The targeted user group consists mostly of musicians. It was first published in 2006 [2]. The interaction with live music over the public Internet is very sensitive to latencies, both round trip as well as one-way. A rehearsal environment for conducted orchestras via the public Internet is the the ultimate goal for this research. Up to 60 musicians and a conductor shall be able to play together live.

Signal processing requirements make a server network mandatory, that connects up to 60 UDP streams to each other and mixes them. A single optimized processing server could handle the process of mixing this amount of concurrent UDP streams with reasonably low latency. Future research, however, shall investigate the application of immersive audio technologies in real-time. A single server would not be able to handle such computational load, since any filter calculation has to be done more than 60 times. Thus, a scalable server network provides the required processing power for a subset of the streams. The audio signals are routed between the signal processing applications via JACK [3]. JACK is a professional and open source audio server, that allows applications to share sample accurate audio data with each other. The servers need to share the processed audio data amongst each and have to be synchronized in time. For this purpose the AVB technology defined by IEEE standards (IEEE 802.1AS, 802.1Qat, 802.1Qav and 1722) is used. The AVB standards extend generic Ethernet networks with precise time synchronization, network resource reservation and bandwidth shaping. These properties avoid the Soundjack client streams from interfering with each other and also ensures the sample accurate synchronization of audio data across multiple servers.

This media clock synchronization of the multiple JACK instances on all servers, with the JACK AVB media clock listener (*avb-mcl*) backend was presented in [4]. Further investigations have shown that JACK is not able to keep the media clock in sync, if the local processing demand rises to the intended amount. The reason for this is the asynchronicity between the AVB AVTP packet rate for stream reservation class A traffic with a transmission interval of $125\ \mu sec$. At a sampling rate of $48\ kHz$, each AVTP packet contains 6 audio samples. The JACK sample buffer, however, always has a size of the power of 2 (e.g. $2^6 = 64$ samples), which 6 is not. Thus, with any sample buffer setting, multiple AVTP packets have to be received in a single JACK audio cycle. With 64 samples 11 AVTP packets are required. This means, that for any one of the eleven AVTP packets, the kernel has to allocate meta data and switch the process context to call the user space application. A JACK audio cycle for 64 samples, which requires $1.3334\ msec$ at $48\ kHz$ to complete, is therefore interrupted any $125\ \mu sec$. But the situation is even worse, since the design of *avb-mcl* blocks until the next arrival of an AVTP packet. Consequently, it blocks 10 times and only leaves $125\ \mu sec$ for the processing of an audio cycle overall. This is exactly the duration between the arrival of the 11th packet and the deadline of the audio cycle. This makes it nearly impossible to deploy *avb-mcl* in a productive environment.

A common solution to this problem is the outsourcing of the packet reception into a different thread. However, this would require synchronization of the threads and would introduce latency by locking or busy waiting. The achievement of the lowest possible kernel latency for this desired behavior with classical methods, would require to write a specific kernel module. This is a difficult task due to several reasons. Another possible solution has found its way into the Linux kernel in 2016, which we will explore in this paper: eXpress data path (XDP).

1.0.1. extended Berkley Packet Filters and eXpress Data Paths

Network traffic nowadays may easily require bandwidths, e.g. $100\ Gbps$, of a computer system's data bus and CPU that a generic software stack is unable to handle. Thus, it makes it hard to process packets within a reasonable reaction time. The reason for this limitation can be found in the allocation of meta data for billions of packets per second by the kernel. Not every packet, however, requires handling by the software stack. Use cases exist, that can be significantly sped up by preprocessing of packets inside the kernel. For most software developers this meant to write their own kernel modules, which is a very delicate and complex process. Three different strategies to accelerate and optimize network packet processing on a Linux computer exist:

- Kernel Bypassing
- Customized Kernel Module
- extended Berkley Packet Filter (eBPF) with eXpress Data Path (XDP)

Kernel Bypassing disables all features of the kernel. Several techniques exist that can be used for kernel bypassing. All of which, however, require dedicated network adapters. A customized kernel module requires a significant development effort. The source code of kernel modules for network adapters easily contains tens of thousands of lines of code, that are carefully tuned. Adding even small features may create unforseen development and debugging effort. Therefore, these two strategies are not further discussed in this paper.

In 2014, the well known Berkley Packet Filter (BPF) kernel facility, to filter network packets in the kernel-space, has been rewritten and extended [5]. An extended Berkley Packet Filter (eBPF) [6] [7] program is a small snippet of code that is compiled to byte code by a just-in-time (JIT) compiler. It gets loaded into the kernel, which then executes this code in a dedicated virtual machine, explicitly handling only this code. Before this code is loaded by the kernel, a pre-verifier checks the code to avoid malicious code to be executed in kernel-space - i.e. it is checked, whether the program contains out-of-bounds memory accesses, loops or global variables. Loops require to be rolled out explicitly and global variables require to be stored in memory maps, that are shared with the respective user-space application.

In 2016, a patch set for high performance networking has been added [8]. The so called eXpress data path (XDP) has been merged in the Linux kernel in version 4.8. This new approach deals with network packets taken right from the NIC, before the kernel is setting up a socket buffer structure, and rejects unwanted, passes desired or redirects packets. A good example for the power of XDP is the defense of a denial-of-service attack. When such an attack is noticed it is possible to drop the packets inside the NIC with such an eBPF program. Thus, the CPU does not have to deal with them and the system stays operational. To use this feature, however, a network driver has to support XDP programs, which can then be accessed via the newly introduced `AF_XDP` socket type - with specialized hardware, the offloading of eBPF programs to the hardware is possible. But even without driver support for XDP programs, it can make sense to use XDP in software mode, as shown in figure 1. Driver mode has been described above. The software mode uses the network driver and allocates a socket buffer structure. For the given example XDP would not make much sense. On the other hand it enables new ways of preprocessing network packets, which can save a significant amount of CPU time for other tasks.

2. CONCEPT

We investigate two different use cases: AVB Listener JACK Client (*jackd_listener*) and JACK AVB Media Clock Listener Backend (*avb-mcl*). Both use cases have the same bottle neck with different consequences for the application.

The first use case is our proof of concept, since it involves all required functionalities: Integration into the build system, preprocessing of AVTP packets and sharing data between kernel- and user-space via memory maps, i.e. the audio samples. As build system the Linux native `make` is used. Both the existing application and the provided tutorials for XDP use the `make` build system [10]. The pre-processing involves three steps. In the first step, AVTP packets shall be filtered on arrival for their destination MAC address and stream ID. This step shall drop any packet that does not match the criteria and prevents a lot of context switches to the host applications waiting raw Ethernet socket. The second step is to store the audio samples contained in the AVTP packets in its integer representation to a memory map, that can be shared with the host application

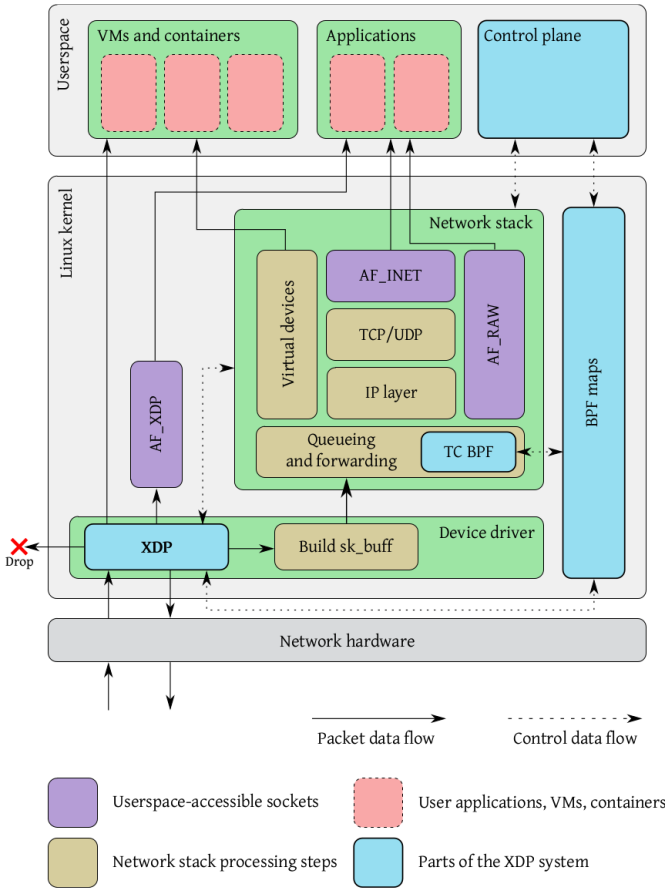


Figure 1: *Components of the XDP subsystem are shown in light blue and reside in the device driver as well as the network stack [9].*

in user-space. The final step is to pass the last AVTP packet, whose contained samples are required to completely fill a sample buffer to the host applications raw Ethernet socket. The raw Ethernet socket in the host application *jackd_listener* is waiting, it receives only AVTP packets for its own registered destination MAC address and stream ID. In fact only the last received AVTP packet is passed. On the reception of the last AVTP packet, it reads the integer-formated audio samples from the memory map, converts them to float format and writes them to a JACK ring buffer.

The second use case requires the integration of the XDP eBPF build process into the `Waf` build system [11], since `Waf` is the build system that is used to build JACK. The pre-processing involves two steps, namely the first and the second step of the first use case - filtering for destination MAC address and stream ID and passing only the last AVTP packet of a sample buffer period.

3. REALIZATION

A prerequisite for XDP and eBPF to work is a kernel later than version 4.8. We deployed a customized real-time kernel of version 5.2.17-rt9 in our test environment.

The implementation of an eBPF program with the host appli-

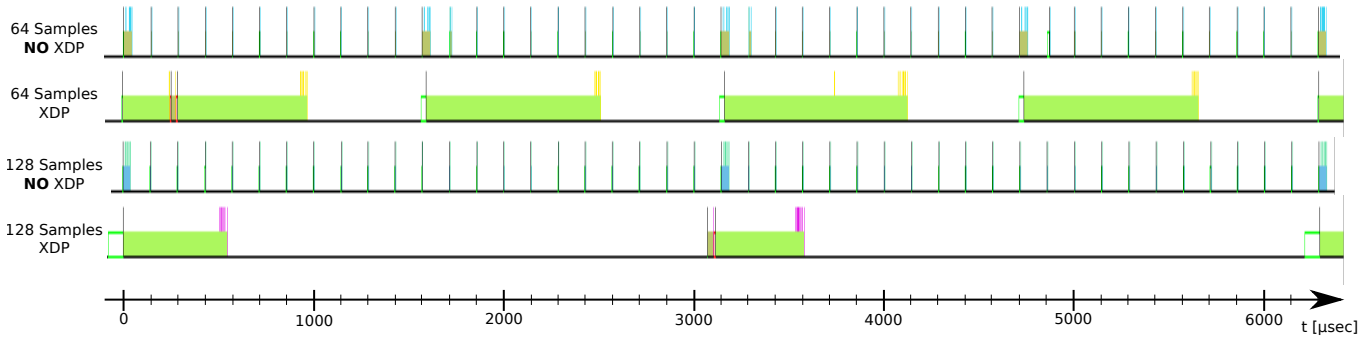


Figure 2: *Kernel traces showing the context switches of the JACK sound server for 64 and 128 samples per buffer with and without XDP.*

cation *jack_listener* has been used as proof of concept. An integration in the `make` build system already existed, which only required adoption to the host application. The `Waf` build system that is used for JACK, however, does not support the LLVM compiler framework [12]. Furthermore, it had not been possible to integrate the loading process of the eBPF program object file into the JACK backend. `Libbpf` [13] needs to find the `main` symbol of the application it is linked against, which could not be achieved until now. Thus it is necessary to compile the eBPF program in a preparing step and load it with a stand alone loader. If the `make` build system is used directly, as is the case for the *jackd_listener* application, the memory maps can be accessed by the host application via a file descriptor and a name string.

The eBPF kernel programs need to be customized, configured and compiled for each application that uses it. In which way parameters can be changed during runtime is still open for investigation.

After the eBPF program has been successfully hooked to the desired NIC, the generic command `ip link show 'dev'` can be used to verify this, i.e. the last line of the following console output.

```
$ ip link show enp5s0

enp5s0: <BROADCAST,MULTICAST,PROMISC,UP,LOWER_UP>
        mtu 1500 xdpgeneric qdisc mq state UP mode
        DEFAULT group default qlen 1000
 link/ether a0:36:9f:bd:95:46 brd ff:ff:ff:ff:ff:ff
 prog/xdp id 217
```

Regarding our first described use case the current lack of financial and in turn hardware resources prevents the required implementations with corresponding complexity: Theoretically a significant number of concurrent AVB listeners may be deployed on a single server, however, in order to test it our test environment lacks a significant amount of AVB talkers. A test would comprise the resulting streams to trigger the corresponding amount of interrupts by the NIC, each of which would be pre-processed by the eBPF XDP program that has been installed for that stream's listener application. Therefore, the evaluation of a setup, in which the *jackd_listener* application benefits from XDP is not possible at the moment. In contrast our second described use case represents a solid base for the technical implementation and evaluation as described in section 4.

4. EVALUATION AND DISCUSSION

The kernel network stack needs to keep working, although a filter is implemented with XDP. This might sound obvious, but it is a development experience worth noting. It is important for the XDP program to pass all Ethernet frames up to the kernel network stack even if they are not subject to our intentions. Filtering for a specific AVTP stream for example, requires PTP and MRP to keep receiving packets, otherwise the NIC does support neither IEEE 802.1AS nor IEEE 802.1Qat. Thus, such packets need to be passed up to the kernel stack and cannot be filtered, e.g. for debugging purposes. This becomes even more important when multiple XDP programs are attached to the same NIC. It has to be ensured that those XDP programs do not interfere with each other by filtering packets the other XDP program requires for its successful operations.

The runtime optimizations provided by the XDP eBPF are realized with kernel traces. A comparison of the context-switch scheduling events of the Linux kernel task scheduler is shown in figure 2. It shows the JACK sound server process with the *avb-mcl* backend configured at 64 and 128 samples per buffer, both with and without a XDP filter program attached to the AVB NIC. The impact of the XDP programs can be seen clearly. When a XDP program is attached to the NIC, fewer context-switches take place, which provides more CPU time to the JACK clients. The JACK clients context-switches are represented by the spikes at the end of the JACK sound server context-switches at the beginning of an audio cycle. Without XDP, those spikes appear much earlier in the cycle and have less time to complete, namely until the next JACK sound server context-switch $\approx 125\ \mu sec$ later.

During situations with heavy load generated by the entire system in a production scenario, the XDP improvements provide a much more robust signal processing and audio signal routing. No buffer over- or underruns occur. An in depth evaluation, however, does not provide further insights and is therefore omitted.

5. CONCLUSIONS

Integration into the `Waf` build system used for JACK is not possible at the moment, because `Waf` is not able to use the LLVM compiler framework.

The lack of ability to perform floating point operations in the kernel-space, is a limitation for further applications of XDP, i.e. for the first discussed use case. Otherwise, it would be possible to directly write the float-formated audio samples to the JACK ring buffer and eliminate any user-space interaction.

The JACK AVB audio stream listeners do not suffer from the asynchronicity between the JACK sound server and the AVB media clock, since multi threading and the JACK ring buffers decouple the

two clock domains. In a scenario where a massive amount of listeners is required, listener with XDP programs in place might be an improvement to listeners without XDP. This is still open for investigation.

For the JACK AVB media clock backend, XDP provides a significant improvement and solves the context-switching problems under load. Further investigations, however, revealed that this performance could as well be achieved with an appropriate handling of a generic raw socket. Thus, XDP represents a powerful and interesting alternative butaspects such as tedious debugging, lack of floating-point operations and the retrieval of hardware timestamps outweigh the benefits significantly.

6. FUTURE WORK

The workflow to create eBPF programs has to be improved. The name for each eBPF program, that shall be loaded, has to be unique in order for the host program to correctly address the memory map for the kernel-/user-space interactions. Furthermore, the parameters required at runtime, such as the stream ID, destination MAC address and sample buffer size, need to be passed to the eBPF program at runtime. Only then can JACK change internal parameters without the need for a newly compiled eBPF program.

At the moment, it is not possible to access hardware timestamps inside an XDP program. This is on the road map of the development teams, however, it might provide further optimization for an AVB network stack in the future.

In theory, XDP would allow to use a NIC (AVB is not required for this) with a raw Ethernet socket to implement a custom protocol. This way it may be used as an interface for digital signal processors that are equipped with an Ethernet interface as well. Signal routing could be done with XDP, so that the signal processing computations are offloaded to the digital signal processor. An user-space application would only manage the audio streams. This approach will be investigated in the future.

7. REFERENCES

[1] (2019, Feb. 8) Soundjack - a realtime communication solution. [Online]. Available: http://http://www.soundjack.eu

[2] A. Carôt, U. Krämer, and G. Schuller, "Network music performance (nmp) in narrow band networks," in *in Proceedings of the 120th AES convention, Paris, France*. Audio Engineering Society, May 20–23, 2006.

[3] (2019, Feb. 8) Jack audio connection kit. [Online]. Available: https://jackaudio.org

[4] C. Kuhr and A. Carôt, "A jack sound server backend to synchronize to an ieee 1722 avtp media clock stream," in *Proceedings of the Linux Audio Conference 2019*. Stanford, CA USA: Linuxaudio.org, Mar. 23–26, 2019.

[5] J. Corbet. (2014, Sep. 24) Linux weekly news (lwn.net): The bpf system call api, version 14. [Online]. Available: https://lwn.net/Articles/612878/

[6] D. Scholz, D. Raumer, P. Emmerich, A. Kurtz, K. Lesiak, and G. Carle, "Performance implications of packet filtering with linux ebpf," in *2018 30th International Teletraffic Congress (ITC 30)*, vol. 01, Sep. 2018, pp. 209–217.

[7] (2019, Dec. 12) Bpf and xdp reference guide. [Online]. Available: https://cilium.readthedocs.io/en/latest/bpf/#bpf-and-xdp-reference-guide

[8] J. Corbet. (2016, Apr. 4) Linux weekly news (lwn.net): Early packet drop — and more — with bpf. [Online]. Available: https://lwn.net/Articles/315941/

[9] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The express data path: Fast programmable packet processing in the operating system kernel," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '18. New York, NY, USA: ACM, 2018, pp. 54–66. [Online]. Available: http://doi.acm.org/10.1145/3281411.3281443

[10] (2019, Dec. 12) xdp-project - xdp-tutorial. [Online]. Available: https://github.com/xdp-project/xdp-tutorial

[11] (2019, Dec. 31) Waf 2.0.18 - the meta build system. [Online]. Available: https://waf.io

[12] (2019, Dec. 31) Waf 2.0.18 documentation - waf tools - compiler_c. [Online]. Available: https://waf.io/apidocs/tools/compiler_c.html

[13] (2019, Dec. 12) libbpf. [Online]. Available: https://github.com/libbpf/libbpf

SYNTHBERRY PI: AN AUTONOMOUS SYNTHESIZER BASED ON RASPBERRY PI

Costantino Rizzuti

Artis Lab
Cosenza, Italy
costantinorizzuti@gmail.com

Fabrizio Rizzuti

Artis Lab
Cosenza, Italy
fabrizio.rizzuti@gmail.com

ABSTRACT

SynthBerry Pi is the first prototype of an autonomous synthesizer based on PDSynth. PDSynth is a toolkit for creating programmable digital synthesizers made using the Pure Data visual development environment. To run PDSynth synthesis architectures a Raspberry Pi mini computer was used. Eight slide potentiometers are connected to the mini computer to create a control surface that makes it possible to control PDSynth's architectures. SynthBerry Pi is, therefore, a compact standalone synthesizer capable of creating sounds by using Pure Data patches.

1. INTRODUCTION

In about the last twenty years, the miniaturization of computing systems and the growing diffusion of open software and hardware technologies allowed artists and designers to access technologies until then only available for technicians and engineers working in large university or business research centers. As Noble [1] mentions, all of this has created absolutely new and never seen conditions, making possible the emergence of new fields of research in art, design and also music such as: Physical Computing and Interaction Design. In fact, before that the idea of artists or designers writing code or designing hardware was almost unheard of. Today, not only it has become commonplace, but it has become an important arena of expression and exploration. Nowadays, this deep bound between technology and artistic creation is become a vital and vibrant phenomena that shapes both art and technology.

Even in computer music the growth of this technologies lead to interesting consequences like the positive convergences with already existing trends such as the practice of self-constructing synthesizers, the development of new musical interface and the building of experimental electronic musical instruments. These practices defined as *analog synthesizer do it yourself*, in its abbreviated form: *synth DIY* [2], aimed at the realization of electronic musical instruments, have had a great diffusion in recent years especially in relation to the increasing use of modular eurorack synthesizers.

This work presents *SynthBerry Pi*: a prototype of an autonomous synthesizer based on Raspberry Pi mini computer. The prototype uses Pure Data patches to generate and process sounds. The collection of patches used for this project is called *PDSynth*: a toolkit for creating programmable digital synthesizers. The Raspberry Pi mini computer was used to run PDSynth synthesis architectures in order to create a compact standalone synthesizer. SynthBerry Pi is equipped with an hardware control interface consisting of eight slide potentiometers that allow to change the parameters of the Pd patches.

Raspberry Pi[1] is a well known mini computer. The project is based on a Broadcom *system-on-a-chip* (BCM2836 for the Raspberry Pi 2, or BCM2837 for Raspberry Pi 3 and BCM 2711 for the latest Raspberry Pi 4 Model B), which incorporates an ARM processor, a VideoCore IV GPU and RAM memory (from 512 Megabytes, to 1 Gigabyte, up to 4GB for the latest version). The boards do not have neither hard disk nor a solid state memory unit, relying instead on an SD card for the boot and for the management of the non-volatile memory.

In the last years many audio projects have been realized around the Raspberry Pi platform [3, 4, 5, 6]. Moreover, Eurorack modules, such as the Terminal Tedium project and Nebulae 2 from Qu-Bit Electronix, use Raspberry Pi to create reprogrammable modules that can be used to implement audio processes developed through languages and development environments for audio (from C and C++ as programming languages, up to to Pure Data, SuperCollider and CSound as languages dedicated to audio).

2. PDSYNTH

PDSynth is a toolkit for creating programmable digital synthesizers.[2] The name derives from the acronym of the sentence: *Programmable Digital Synthesizer*. But also the acronym *PDSynth* allows to indicate a synth made with Pure Data (Pd) [7]: the well known visual development environment for multimedia applications used to implement the toolkit.

The development of this project started in the Autumn 2015 from an initial idea to create a series of easily interfaceable Pd patches capable of simulating the behavior of the essential modules of an analog synthesizer. The *PDSynth* modules implement different sound generation and processing systems and are all controllable via the *Open Sound Control (OSC)*[3] protocol. Users can easily create and interconnect the modules together to build high-level architecture for real-time sound synthesis and processing. The OSC protocol [8, 9] was choosen because it is becomed, along the years, a standard format for sharing data related to musical performance (parameters, sequences of notes, gestures) between musical instruments (mainly synthesizers and electronic instruments), calculators and other multimedia devices. This protocol, from the late 1990s, is becomed a valid alternative to MIDI, especially because it is open, flexible and extendable.

Open Sound Control was choosen because it allows to easily create a reliable and robust communication system among the various modules inside Pd allowing, also, a simple exchange of network messages to and from the outside. In fact, the OSC messages can be easily managed through the native message system provided by Pure Data. All this simplifies the creation of the control systems of the modules and allows to control the synthesizers through external

[1] For more information refer to the Raspberry Pi Foundation website: https://www.raspberrypi.org/.

[2] The PDSynth toolkit can be downloaded from Artis Lab website: https://www.artislab.it.

[3] For further information, refer to the project's official website: http://opensoundcontrol.org.

controllers and control surfaces. Moreover, the OSC protocol allows to have both an higher data resolution and greater parameter space than what is offered by the MIDI protocol.

A library of external objects was used to implement the control functions through the OSC protocol. The library provides useful objects only to realize the management of the OSC messages inside Pure Data, so for the transfer of OSC packets through the network, a second library, called *IEMnet*, was used. In particular, it is possible to use the `udpreceive` object to implement within Pd a server listening on a given port for receiving OSC messages.

The processing of messages received from the server can then be carried out using the objects provided by the OSC library. The `unpackOSC` object is useful for converting OSC packages, made up of binary data, into messages compatible with the Pure Data internal messaging system. Then the `pipelist` object is inserted to obtain a temporally coherent message scan in the case in which messages with a given *timestamps* are received. Finally, the OSC library provides an object, called `routeOSC`, which allows the addressing of messages according to a hierarchical structure defined by the address space. The arguments supplied to the object define a set of addresses to which corresponding messages can be routed.

2.1. PDSynth architecture

The development of the toolkit, unfortunately, is still in an initial phase, however it already provides a minimal series of modules that can be easily used to create and process sounds. At the beginning of the project, in fact, after the first phases of software development we decided to move the attention to the design and the construction of hardware devices to be used in combination with the toolkit. The modules currently available can be classified into three distinct categories (Signal generators, Filters, Envelope generators) which will be presented in detail below.

2.1.1. Signal generators

PDSynth currently offers five sound generation modules that emulate the behavior of classic analog synth oscillators. The modules offer the possibility of generating the following waveforms:

GENPULSE — band-limited pulse train generator;

GENSAWTOOTH — band-limited saw tooth wave generator;

GENSIN — sine wave generator;

GENSQUARE — band-limited square wave generator;

GENTRIANGLE — band-limited triangle wave generator.

The `GENSIN` module uses the Pure Data native object `osc~` for generating the sine wave. All other modules are based on reading data saved in tables (Wave Table Synthesis) [10].

The image in Figure 1 shows the patch of the `GENPULSE` module. The sound is generated by using the Pd object `tabosc4~`. It allows to read the data saved in a table using a polynomial interpolation of the third order (four points interpolation). The objects that manage the OSC messages are placed in the top right corner of the patch. The first object (`r OSCMessages`) is used to receive OSC messages sent in "*broadcast*" within Pure Data's native message infrastructure. The following object (`routeOSC /$1`) selects and sends on its leftmost outlet only the messages that have as the first tag of the address the name of the module. This can be defined during the creation of a new instance of the `GENPULSE` through the first argument of the patch (for example *Pulse1* in the upper part of the patch in Figure 1). The next `routeOSC` object allows to route properly the data related to the various parameters to its different outlets accordingto the OSC name address (`/Freq` — frequency, `/Amp` — amplitude, ...).

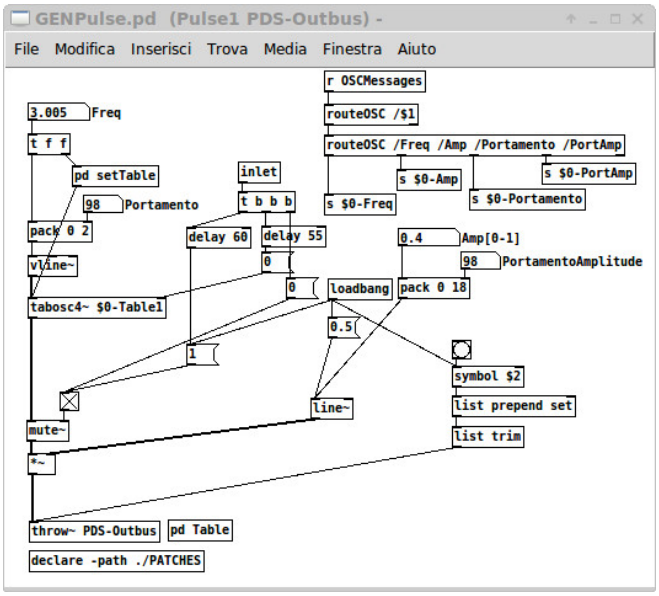


Figure 1: *GENPULSE - patch of the pulse train generator.*

The toolkit is based on digital sound generation techniques, so there is no real difference between audio band signal generators and LFO (Low Frequency Oscillator). Therefore, all generators can be used both in a frequency range below the threshold of audibility, as is typical for LFOs, and to produce audible sounds. For this reason, to achieve the generation of different waveforms we tried to make a compromise between the problems related to aliasing and the creation of signals with a frequency spectrum as wide as possible. After some initial experiments aimed at evaluating different approaches, we finally chose to generate band-limited signals by reading the waveform data stored in different tables. In order to be able to produce spectra that are very rich in harmonics, we decided to divide the audible frequency spectrum into eight distinct regions, each corresponding to a table containing the waveform with a suitably calculated harmonic frequency content to avoid aliasing phenomena.

The image in Figure 2 shows the subpatch of the `GENPULSE` module in which the eight tables are defined. Each table is related to a different region of the frequency spectrum: `$0-Table1` contains a waveform made of 511 partials that is used at the low end of the spectrum. While, at the opposite, `$0-Table8` contains a waveform generated using only three partials which is used to generate the sound in the upper part of the frequency spectrum.

All the signal generators, except the sinusoidal oscillator, use this approach based on the reading of eight tables with waveforms characterized by a different bandwidth. The change of the frequency parameter determines the selection of the appropriate table to be used for reading. The image in Figure 3 shows the `setTable` subpatch. It receives as input the frequency value in Hz and, by means of the conditional structure `if` contained in the object `expr`, it controls which is the table to be read according to the interval of frequencies in which the oscillator is called to operate.

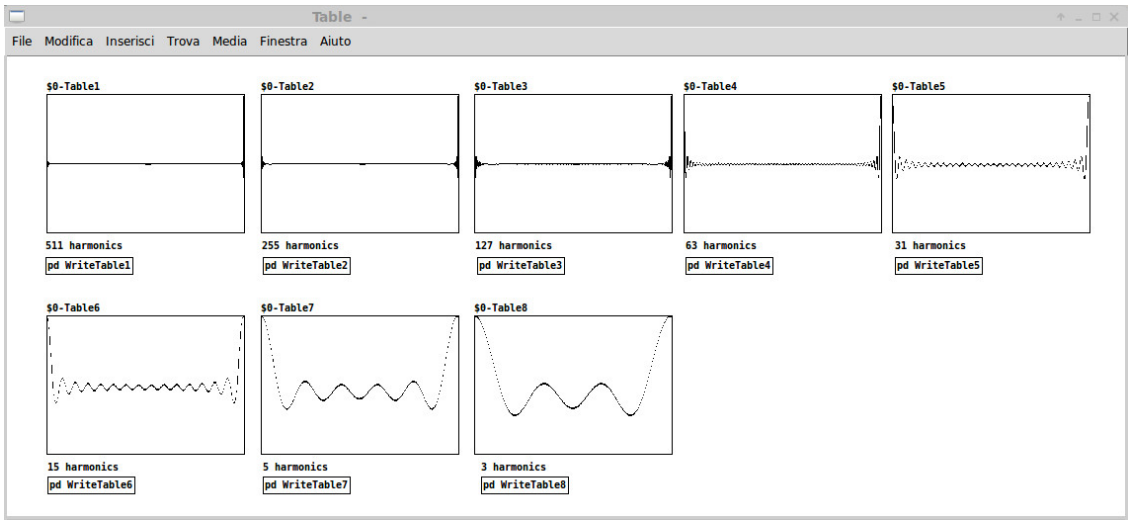


Figure 2: *GENPULSE — eight tables used for WaveTable synthesis.*

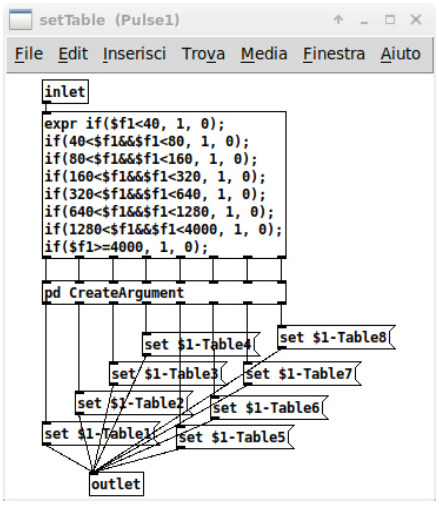


Figure 3: *GENPULSE — the setTable subpatch.*

The content of the wavetables shown in Figure 2 can be generated by using the Pd command `cosinesum` that allows to create a wave according to a sum of cosines harmonics (`sinesum` is the command in Pd to realize the sum of sine waves). The image in Figure 4 shows five subpatches used to create as many wavetables, each with its own number of partials (ie. `WriteTable8` — three partials, `WriteTable6` — fifteen partials, ...). The number placed after the `cosinesum` command define the length of the table to be generated expressed in number of points. In the image it is possible to notice that the number of points in the tables varies with the number of partials. In fact, as reported in the Pure Data manual, it is better to use tables composed of 512 points for waveforms containing up to fifteen partial. Above this threshold it is convenient to calculate the length of the table L as the number, power of two, greater than the product shown in equation 1, where N_p indicates the number of partial

$$L > 32 * N_p. \tag{1}$$

For example, to generate `$0-Table1` containing 511 partial we need to use 16384 points. To calculate the frequency values to be used for changing the table to be read as a function of the frequency value, it is possible to observe that from the previous relation the maximum number of partials can be obtained according to the size of the table. This value can be calculated by inverting the previous relationship and subtracting one as a safety margin as shown in equation 2:

$$N_p = \frac{L}{32} - 1. \tag{2}$$

Once this value is known it is possible to obtain the maximum reproducible frequency through the equation 3:

$$f_{max} = \frac{20000}{N_p} \tag{3}$$

it was decided to use the frequency of 20000Hz (with respect to the theoretical value of the Nyquist frequency equal to 22050Hz for the standard sampling frequency of 44100Hz) as an additional safety margin with respect to the occurrence of aliasing phenomena. Returning to the example of `$0-Table1`, we obtain therefore:

$$f_{max1} = 20000/511 = 39,1 \tag{4}$$

as it is visible in the image in figure 3 the first table is changed for a frequency equal to 40 Hz. This same procedure has also been applied for the calculation of all the other values defined to realize the change of the table to be read using the object `tabosc4~`.

The pulse waveform can be generated with a series of cosines in which all the partials have a uniform amplitude distribution. The value of the amplitude a_n can be computed according to equation 5, where N is the maximum number of partials [11].

$$a_n = \frac{1}{N}. \tag{5}$$

As shown in Figure 4, the command `cosinesum` is followed by the number of points of the table and by a list of numbers that defines the amplitude factor of each partial. In the case of the pulse waveform, this numbers are all the same and equal to the inverse of the number

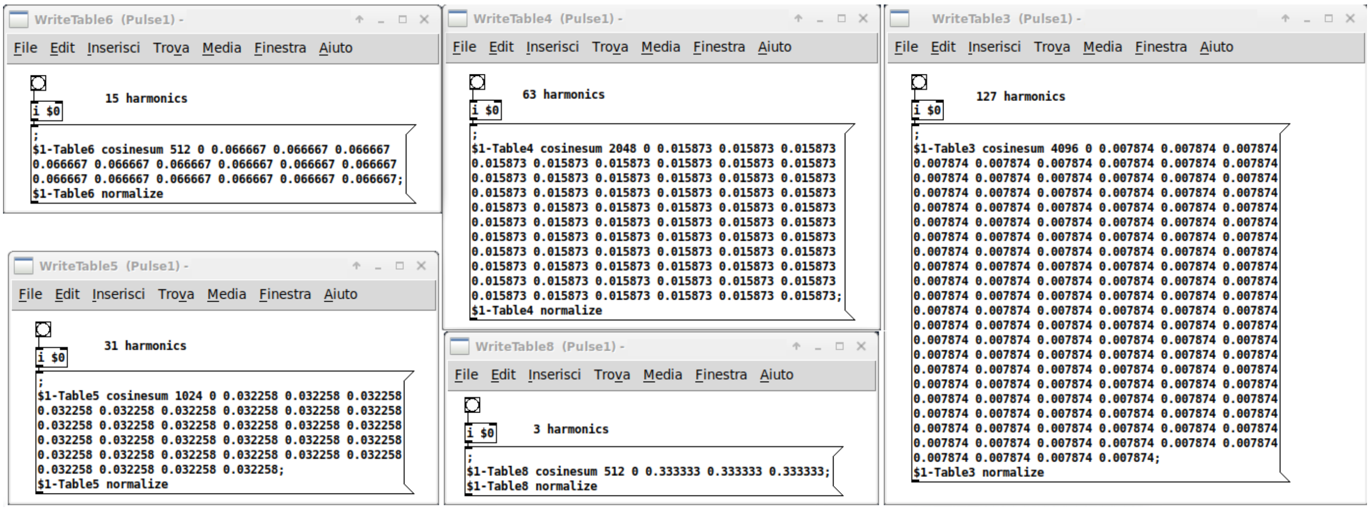


Figure 4: *GENPULSE — commands to create five wavetables.*

of partials. Creating this lists of amplitude coefficients for all the different waveforms is a long and repetitive process not so easy to do by hand. This is even more true, when it is needed to generate a very large number of partials as in the case of `$0-Table1`, as shown in the top right corner of Figure 2.

For this reason a Python script has been developed in order to automate the creation of a text file containing the list of amplitude coefficients. The script created for the pulse generator is shown below. In the first line, inside the `open` function, it is necessary to define the name of the text file where data will be stored. On the next line, the `MaxOrder` parameter defines the maximum number of partials to be generated. The content of the text files generated by this script can be easily copied and pasted into the Pure Data messages (see figure 4) used to populate the tables with the various waveforms.

```
out_file=open("Coef-Pulse.txt","w")
MaxOrder=511
x=round((1./MaxOrder),6)
out_file.write(str(0)+" ")
#The DC component is equal to 0
for i in range(1,MaxOrder+1):
        out_file.write(str(x)+" ")
out_file.close()
```

2.1.2. *Filters and sound processing*

In addition to sound generation modules, PDSynth provides also patches implementing filters. So far, three different filters of the fourth order have been created:

FLTBandPass – band pass filter based on the Pd object `vcf~`;

FLTHighPass – high pass filter based on the Pd object `hip~`;

FLTLowPass – low pass filter based on the Pd object `lop~`.

Figure 5 shows an example based on the `FLTBandPass` module. The patch realizes a small bank of filters, composed of three modules placed in parallel, used to filter white noise. Each filter is identified through a different OSC namespaces (`BP1`, `BP2`, `BP3`).

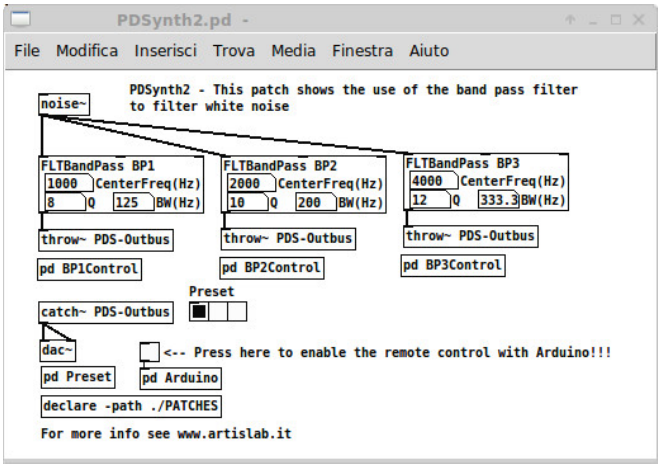


Figure 5: *Example patch of the FLTBandPass module.*

The structure of the filters patches is analogous to those of the generators with regard to the internal management of OSC messages. The filters differ from the generators only because of the presence of an audio inlet used to provide the input signal to be processed.

2.1.3. *Envelope generators*

Two modules were also developed to generate envelopes:

ENVTable — envelope generator defined through a table;

ENVADSR4 — ADSR type envelope generator with fourth order polynomial interpolation.

The `ENVTable` module allows you to generate a time envelope by reading data contained in a table. Python scripts have been created for the generation of envelope tables with different temporal trends. The image in Figure 6 shows different two traits (attack and release) envelopes generated by Python.

The `ENVADSR4` module, instead, realizes a four-state ADSR envelope with fourth order polynomial interpolation. As shown by

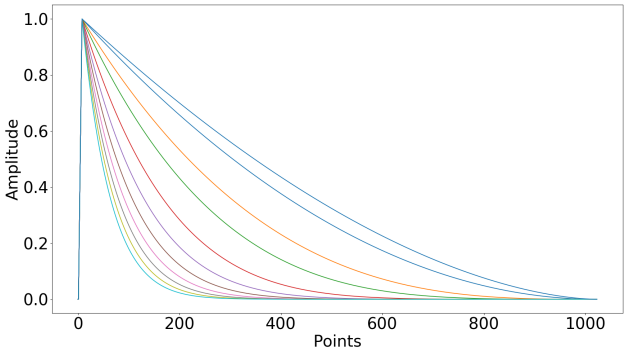


Figure 6: *Two piece polynomial envelope.*

Puckette [10] this type of interpolation allows to obtain a trend very similar to the logarithmic one, but with a reduced computational cost and a greater simplicity of implementation. As is known, the logarithm function diverges towards less infinite when the argument is approaching towards zero, which makes serious precautions necessary in the realization of logarithmic envelopes through truncation or approximation processes. On the contrary, the use of polynomial interpolation eliminates this problem by also offering the possibility of modifying the slope of the curves in a very simple manner by varying only the order of the polynomial used.

2.2. PDSynth-00

Starting from the initial idea to develop an exclusively software environment, we tried to use DIY controllers based on the Arduino prototyping platform to control the modules of the toolkit. This first experiments encuraged to broaden the vision of the toolkit by incorporating, therefore, both software development and the design of hardware devices to control the software architectures. The intent of the project has therefore been transformed into the creation of an environment for prototyping and developing portable electronic musical instruments and synthesizers.

On the hardware side, the project was oriented towards the realization of physical devices, equipped with potentiometers, sensors and other interaction systems. *PDSynth-00* (see Figures 7 and 8) is the first DIY prototype of a controller made by Artis Lab, in Spring 2016, that is born from the idea of an hardware device useful to control the synthesis and sound processing architectures created with the PDSynth toolkit.

Figure 7: *Rear panel of PDSynth-00.*

PDSynth-00 is a reprogrammable electronic musical instrument that can perform different functions depending on the software that is loaded into the Arduino board. It is equipped with six slide potentiometers and twelve buttons. Everything is contained in a simple and light container made of plywood shaped using a laser cutting machine.

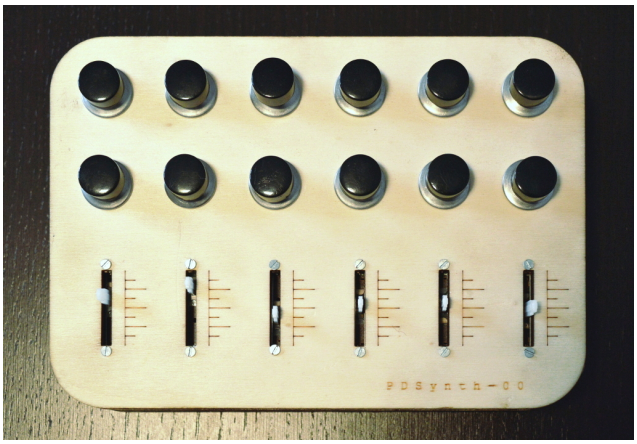


Figure 8: *Top view of PDSynth-00.*

PDSynth-00 can be interfaced with Pure Data through the Firmata protocol. By this way the data relating to the position of the six cursors and the status of the buttons can be sent to the program listening on the serial port and used to perform action or modify parameters inside the PDSynth patches.

3. SYNTHBERRY PI

SynthBerry Pi was born as a natural evolution of the PDSynth-00. The Arduino prototype is not autonomous, it is only useful for controlling the PDSynth modules running on a computer. SynthBerry Pi, enstead, integrates controller and computer through the use of a Raspberry Pi mini computer allowing to create an autonomous device able to generate sound that can be modified via a control surface.

3.1. The control surface

SynthBerry Pi is equipped with an hardware control interface consisting of eight slide potentiometers. The prototype was built, like the previous one, using two panels of plywood shaped with a laser cutting machine. Figure 9 shows the front view of the prototype.

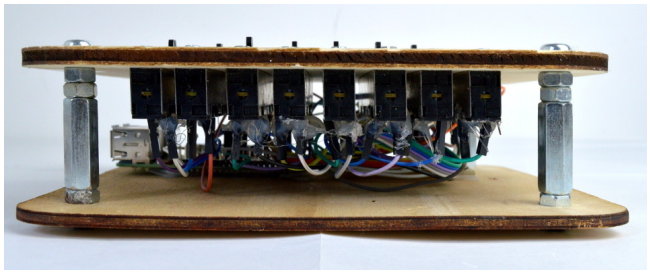


Figure 9: *The front view of SynthBerry Pi.*

The slide potentiometers are mounted on the front panel of the device, the assembly between the two panels of the prototype was

carried out through hexagonal steel spacers of suitable length. The image in Figure 10 shows the top view of the prototype.

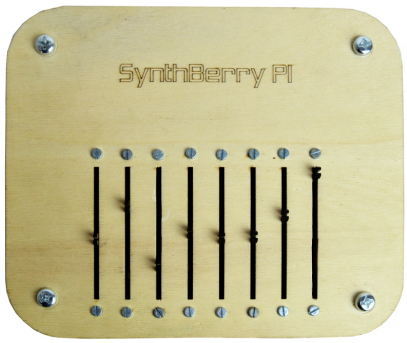


Figure 10: *Top view of the prototype.*

3.1.1. Hardware set up

Raspberry Pi is not equipped with analog to digital converters (ADC) allowing the connections of potentiometers. For this reason, the analog to digital converters *MCP3008* was used to read the voltages related to the positions of the slide potentiometers. The *MCP3008* is an integrated circuit that provides eight analog input channels with 10 bit digital resolution. Figure 11 shows the simulation of connections among the various components using a breadboard, while Figure 12 shows the circuit schematic. For simplicity, only one potentiometer has been inserted since all the others must be connected in a similar way to the ADC inputs.

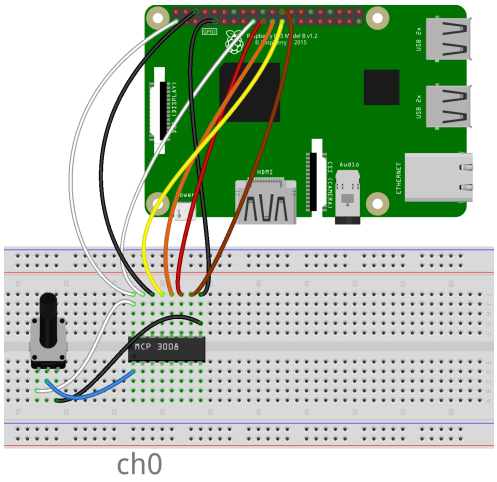


Figure 11: *Simulation of connections using a breadboard.*

The communication between Raspberry and the ADC is based on the *SPI (Serial Peripheral Interface)* serial communication protocol. SPI is a communication system between a microcontroller and other integrated circuits or between multiple microcontrollers. It is a communication standard, created by Motorola, in which the transmission takes place between a control device (called *master*) and one or more controlled devices (called *slave*). The master device controls the communication bus, emits the clock signal and decides when to start and end communication.

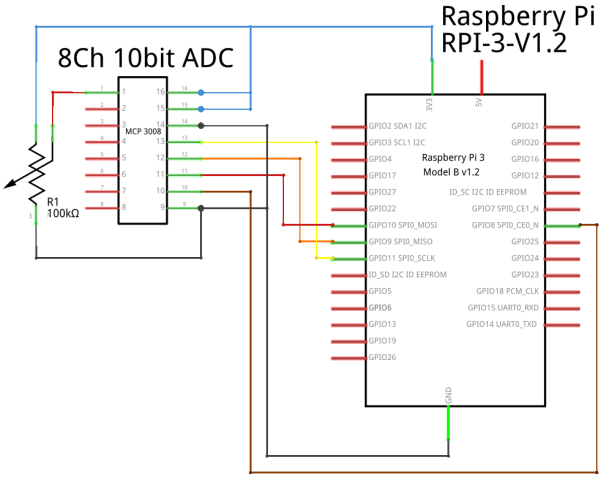


Figure 12: *Circuit schematic.*

The SPI communication system is commonly defined as four-wire, since for the transmission of data four distinct signals are generally used:

- **SCK**: Serial Clock (emitted from the master)
- **MISO**: Serial Data Input, Master Input Slave Output
- **MOSI**: Serial Data Output, Master Output Slave Input
- **CS**: Chip Select, Slave Select (issued by the master to choose which slave device to communicate with).

Chip Select is the only connection that is not always necessary in all applications since its just needed to manage multiple slave devices. A connection that defines the reference level of the voltage, often referred to as **GND**, must be added to these four wires.

3.1.2. Software set up

The reading of the data acquired by the ADC is realized through a Python script, which uses the *SPIDEV* library for the management of SPI devices. The following code shows a fragment of the Python script with the commands necessary to open the communication with the ADC and to perform a reading of the data through the `ReadChannel()` function. The `spi.xfer2()` function is invoked, inside `ReadChannel()`, to request the ADC to read the voltage value of a given channel.

```
#Open SPI bus
spi=spidev.SpiDev()
spi.open(0,0)
spi.max_speed_hz=1000000
#Function to read SPI data from MCP3008 chip
#Channel must be an integer 0-7
def ReadChannel(channel):
  adc=spi.xfer2([1,(8+channel)<<4,0])
  data=((adc[1]&3)<<8)+adc[2]
  return data
```

The transfer of the data read by the analog to digital converter, between the Python script and Pure Data, is achieved sending, on a specific port, local network messages. For this purpose we use the `pdsend` program provided within the standard Pd package, used as

a sub-process within the Python script. The following code shows the creation of the `subprocess p` which invokes the `pdsend` program used to send data on port 9000 of the local computer. The `send2Pd()` function is used to send messages through `pdsend`. The last line shows the use of the `send2Pd()` function that take as argument a string composed by the concatenation of two numeric values: the first to define the channel of potentiometer and the second to provide the ADC reading.

```
#Create a subprocess to send data to Pd
p=subprocess.Popen(["pdsend","9000"],
        stdin=subprocess.PIPE)
#Define the function to send data to Pd
def send2Pd (message=' '):
  print >> p.stdin, message
#How to use the function to send data to Pd
send2Pd('0'+str(pot_Volts0)+';')
```

A simple protocol was designed to send messages from Python to Pure Data keeping the data of the different potentiometers separate and easily differentiable. A list of two numbers is sent, the first is a label (from 0 to 7) useful for identifying the potentiometer, the second number is the numeric data obtained from the reading made by the digital converter. To receive data in Pure Data the `netreceive` object is used which opens a server listening on the port corresponding to that used by `pdsend`. The expedient used in the construction of the message sent by Python simplifies the sorting of data that can be easily accomplished through the native object of Pd `route`.

The data acquired by the ADC are filtered to reduce random fluctuations due to noise through the use of an average filter that generates an average output value every ten converter readings. The following code shows the simplified structure for reading and transmitting data of a single potentiometer. Within an infinite cycle, the `Count` counter is incremented and the values supplied by the ADC are read, divided by 1023 (to obtain numbers between 0 and 1) and accumulated on the `readPot0` variable. Every ten readings the `readPot0` variable is divided by ten, obtaining the average value. If the new value has undergone a variation compared to the previous one greater than 0.5%, the value of the `pot_Volts0` variable is updated. The value of this variable is then sent to Pd through the `send2Pd` function. After that the values of the counter and the accomulation variable are both reset to zero and the program execution is suspended for a time defined by the `delay` variable.[4]

```
while True:
 readPot0+=ReadChannel(pot0_channel)/float
     (1023)
 Count+=1
 if Count==10:
   readPot0=0.1*readPot0
   if abs(pot_Volts0-readPot0)>0.005:
        pot_Volts0=readPot0

   send2Pd('0'+str(pot_Volts0)+';')
   readPot0=0
   Count=0
 # Wait before repeating loop
 time.sleep(delay)
```

[4]In this way, using a value equal to $0.01s$ for the variable `delay`, the reading is made every $10ms$ and a new value is sent to Pure Data every $100ms$.

A script was created to start both Pure Data and the Python program to manage the ADC. Since we want to use the prototype as a common electronic instrument through the use of only the control surface we have chosen to use the Raspberry Pi *headless* without the connection of screen, mouse and keyboard. For this purpose, the script must be started automatically during the startup phase of the Raspberry Pi. To do this, a special service, launching the start script, has been created and set up to be started automatically during the initial phases of execution of the operating system.

3.2. The audio engine

The audio engine of the prototype is based on the use of a modified version of the first PDSynth sample patch. The patch offers the possibility to separately control the amplitude and the frequency of three oscillators generating different waveforms (square wave, pulses train and sawtooth wave). Furthermore, it provides a delay line with a feedback path. The delay line can be controlled by two parameters that can be modified in real time through the potentiometers of the prototype control surface: the delay time and the feedback coefficient. The eight potentiometers of the prototype have been associated to likewise control parameters of the patch according to the following scheme:

- A0 - square wave oscillator frequency;
- A1 - amplitude of the square wave oscillator;
- A2 - pulse generator frequency;
- A3 - amplitude of the pulse generator;
- A4 - frequency of the sawtooth oscillator;
- A5 - amplitude of the sawtooth oscillator;
- A6 - delay time;
- A7 - delay feedback.

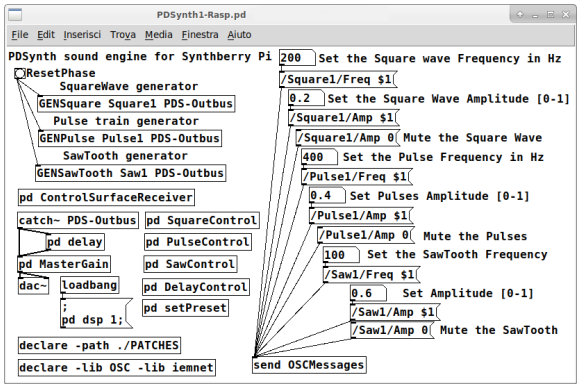


Figure 13: *Pd patch of the audio engine.*

Figure 13 shows the audio engine patch and the use of three signal generation modules. The modules are placed in the top left corner of the patch. The first argument of each object is a name that is used as an identifier for addressing OSC messages (*Square1*, *Pulse1*, *Saw1*). The second argument provided in the creation of the sound generators (*PDS-Outbus* in the image) allows to define the name of the *bus* on which the audio signals produced by the various generators will be accumulated. In this case, all the signals are collected by the `catch~ PDS-Outbus` object and sent both to the delay patch

and to the sound card output through the `dac~` object (in the lower left part of the patch in Figure 13).

On the right side of the patch, the *number boxes* allow to change the amplitude and the frequency of the various oscillators by sending OSC messages through the `send OSCMessages` object. The image shows also how to create the OSC message addresses to control the parameters of the modules.

3.3. Eurorack Module

In autumn 2019 a new version of the prototype was created in the form of a 18 hp eurorack module. The image in Figure 14 shows the front view of the module.

Figure 14: *The front view of the module.*

Figure 15 shows the internal structure of the module. An handcrafted PCB board is connected on the GPIO pins of the Raspberry Pi. The potentiometers are connected to the PCB where the ADC is also housed. This second prototype made in the form of a Eurorack module is completely analogous to the first prototype both in terms of hardware and software.

4. CONCLUSIONS

In this work we have presented *SynthBerry Pi* an autonomous synthesizer based on Raspberry Pi and Pure Data. The PDSynth toolkit was used as audio engine of the prototype. This toolkit provides a series of Pd patches that can be easily used as modules to create high level architecture to generate and process sounds. To run the PDSynth synthesis architectures a Raspberry Pi mini computer was used; a control surface made up of eight slide potentiometers was build to provide a suitable hardware device to play the instrument controlling in real time the sound parameters.

SynthBerry Pi was used in several live performances and also in studio recordings. In future work we intend to add a second ADC to the prototype to have eight more input channels useful for implement the control voltage (CV) of the synth parameters. Furthermore, we intend to create a more intuitive and powerfull control surface by also adding buttons, LEDs and rotary encoders.

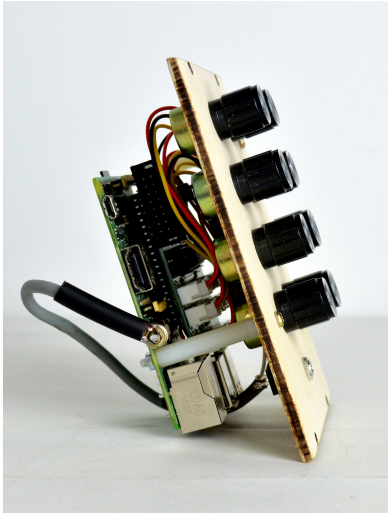


Figure 15: *The internal structure of the module.*

5. REFERENCES

[1] J. Noble, *Programming Interactivity. A Designer's Guide to Processing, Arduino and openFrameworks*, OReilly, Sebastopol, CA, 2009.

[2] R. Wilson, *Make: Analog Synthesizers*, Maker Media, Sebastopol, CA, 2013.

[3] J. Reuter, "Case study: Building an out of the box Raspberry Pi modular synthesizer," in *Proceedings of Linux Audio Conference (LAC14)*. Karlsruhe, 2014.

[4] F. Meier, M. Fink, and U. Zölzer, "The JamBerry - a standalone device for networked music performance based on the Raspberry Pi," in *Proceedings of Linux Audio Conference (LAC14)*. Karlsruhe, 2014.

[5] V. Lazzarini, Timoney J., and Byrne S., "Embedded sound synthesis," in *Proceedings of Linux Audio Conference (LAC15)*. Mainz, 2015.

[6] H. von Coler and D. Runge, "Teaching sound synthesis in C/C++ on the Raspberry Pi," in *Proceedings of Linux Audio Conference (LAC17)*. Saint-Etienne, 2017.

[7] M. Puckette, "Pure Data," in *Proceedings of International Computer Music Conference (ICMC97)*, pp. 224–227. Thessaloniki, 1997.

[8] M. Wright, A. Freed, and A. Momeni, "Open Sound Control: State of the art 2003," in *Proc. of the 2003 Conference on New Interfaces for Musical Expression (NIME-03)*, pp. 153–159. Montreal, 2003.

[9] M. Wright, "Open Sound Control: an enabling technology for musical networking," *Organised Sound*, vol. 10, no. 3, pp. 193–200, 2005.

[10] M. Puckette, *The Theory and Technique of Electronic Music*, World Scientific, 2007.

[11] C. Dodge and T. A. Jerse, *Computer Music*, Schirmer, New York, 1997.

Posters

Proceedings of the 18th Linux Audio Conference (LAC-20), SCRIME, Université de Bordeaux, France, Nov 25–27, 2020

OSPW 2.0 – AN OPEN SOURCE LINUX-BASED DSP SERVER FOR AUDIO APPLICATONS

Clemens Fiechter
Research & Development
Hochschule für Musik Basel FHNW
clemens.fiechter@students.fhnw.ch

Thomas Resch
Research & Development
Hochschule für Musik Basel FHNW
thomas.resch@fhnw.ch

ABSTRACT

The Open Signal Processing Workstation (OSPW) 2.0 is a Linux-based open software platform, designed for rapid prototyping and the development of digital signal processing (DSP) audio algorithms and corresponding user interfaces (UIs). Since audio interface and computer hardware can be chosen almost completely freely, the system can be easily integrated into any existing audio network and studio environment. Besides the necessary hardware components, OSPW 2.0 consists of the graphical programming environment Pure Data (Pd) for the signal processing, a script for the start-up procedure and initial configuration, and a webserver which generates browser-based UIs for an arbitrary number of remote clients automatically. All connected UI clients are synchronized among each other. This enables the simultaneous operation of applications by multiple users. Custom interfaces can be realized by extending the Javascript UI framework.

1. INTRODUCTION

The described system OSPW 2.0 is the successor of the OSPW 1.0, a project with a similar goal that never came into production [1]. The promising findings of the OSPW 1.0 were analyzed and evaluated and then adapted and re-implemented exclusively using open source technologies. In recognition of one of the first successful music DSP computation platforms, the ISPW [2], this prototype and the predecessor were named OSPW. To facilitate readability, the version number 2.0 will not be used in the remainder of this paper.

OSPW consists of a DSP server running Pd [3], that can be remotely controlled by any device on the same network that can execute a web browser. The web interface is automatically generated based on the underlying Pd patch. In contrast to hardware currently used in professional studio, broadcast or live sound environments which focus primarily on standard audio formats like two-channel stereo, or common surround formats (5.1, 7.1, etc.), algorithms developed for the OSPW are not bound to standard channel-formats. Depending on the sound card and the performance of the computer components used, massive multichannel operations can be realized; for example, high-order Ambisonics, Wavefield synthesis renderers or multiuser binaural monitoring applications.

DSP algorithms for OSPW are implemented with the visual programming environment Pd, which is widely used in academic and experimental musical contexts and environments. It provides an API in the programming language C and allows "intermediate" programmers and artists in the field of media technology to use the system through its easy-to-use graphical programming interface. Using Pd as an audio backend has the big advantage that it has been in use and extensively tested for decades. It supports parallel programming with multiple threads natively through the pd~ object [4]. The possibility of distributing different instances of an algorithm to all available processor cores makes optimal use of current CPUs and maximizes the available performance - one of the most important criteria for an external DSP server.

This paper starts with a brief discussion of related work in section 2. Section 3 describes the system design including necessary hardware and software components and basic usage of the OSPW. Section 4 outlines the implementation details of all components. Section 5 describes three implemented demo applications. As a part of this project, a repository with the source code including documentation and tutorials is available online [5]. This allows any interested person to set up his/her own custom version of the OSPW.

2. RELATED WORK

There are several commercial DSP systems available whose concepts are similar to those of the OSPW. SoundGrid by Waves Inc. is a DSP server that runs on a Linux machine with a general-purpose CPU [6]. The main difference to OSPW is that it is a closed-source proprietary product. Only the manufacturer's plugins and those of a few authorized companies run on the hardware. The UAD DSP devices by Universal Audio [7] follow a similar approach as SoundGrid. They work with special UA format plugins only. The Tesira platform is a highly configurable DSP server by the company Biamp [8]. It is also programmable with its own algorithms. However, the target group of these systems are not studio environments but rather multi room speech conferences and large-scale sound installation at exhibitions or hotels. Also noteworthy is the Bela project. It is an open platform for ultra-low latency audio and sensor processing [9]. It runs libpd [10] on an embedded computer with an additional, custom developed microcontroller board with sensor inputs. Bela doesn't provide user interfaces. Instead it is meant to be controlled by sensors. Mira in combination with Max/MSP is conceptually similar to the OSPW approach: a computer running the DSP combined with a remote application [11]. In contrast to the project presented in this paper, both tools are closed source. FreeDSP is a low-budget open source DSP module [12] which can be configured with the graphical programming environment SigmaStudio. Due to the limited number of inputs and outputs of the used DSP board, applications of this project are rather stand- alone effect processors.

3. SYSTEM DESIGN AND USAGE

This section provides a description of hard- and software components used for the OSPW prototype, brief instructions for the setup of the necessary software components and the basic usage. For details including a complete installation guide, please refer to the documentation on the Git repository [5].

OSPWs system architecture can be described as a server-client model where all signal processing is executed on the server hardware and an arbitrary number of clients can be connected for remote controlling and monitoring purposes. An external computer has to be used for the algorithm design in Pd. Once the design is completed, the user can transfer the code to the server, where it is analysed for automatic UI generation and executed. Any device with a browser running in the same network can be used as remote control for the loaded Pd patch. For OSPW server and remote client(s) to work together, they must be connected to the same network. The most elegant solution (which is also used in the prototype) is to configure the server as a wireless access point.

3.1. **Hardware**

The audio I/O of the OSPW platform utilizes the Advanced Linux Sound Architecture (ALSA). The prototype was built with the LX-Dante PCIe card by Digigram. Its 128 inputs and outputs offer flexible channel routing and enabled testing with many physical inputs and outputs. Although the card with its closed source Linux driver does not quite fit into OSPWs philosophy, it made an easy integration into the testing environment possible (a Dante-enabled mixing console). For a custom installation of a fully functional OSPW, basically any ALSA compatible soundcard can be used.

An x86 processor is not required but the target operating system must support the software components listed in the following section. For details on the prototype specifications regarding other hardware components please refer to the publication about the OSPW 1.0 [1].

3.2. **Software**

The software consists of two main parts: the audio backend and the OSPW server. The audio backend of the OSPW platform is based on a plain Pd Vanilla installation. Pd provides a graphical user interface and a C API for DSP development and control structures. The OSPW Server is a Node.js [13] server application which enables the user to control and interact with the running Pd instance. Several software components are necessary for a working OSPW installation:

- A Linux installation with ALSA support
- Pd
- Node.js
- The OSPW software package, containing scripts, the server and the demo applications.

The GUI control elements are generated with the open source framework NexusUI [14]. NexusUI is an open source project and already implements typical audio widgets such as sliders and dials.

3.3. **Usage**

After installing and setting up all the necessary components from the Git repository, the server is configured to start automatically with Linux's systemd init-system. After connecting a client to OSPWs network, the server's IP address has to be entered in the client's browser in order to render the main page. On this page, the user can either select one of the demos or choose one of his own uploaded Pd patches. After selecting an application, the server parses the corresponding patch and automatically serves the UI to the connected client(s). For each parameter to appear in the UI, a matching Open Sound Control (OSC) string must be included in the Pd patch. This is done as shown in figure 1 by placing a comment containing the string somewhere in the patch (ideally close to the corresponding parameter). The syntax for the string is /ospw*/x/y/widgettype/parameterName/initValue*:

- The string has to start with '/ospw'.
- *x* and *y* are grid coordinates for placing the object within a symmetric grid.
- */widgettype* defines the generated interface object. Possible values are *button, toggle, number, dial, hslider*, *vslider.*
- */parameterName* can be chosen freely and results in the rendered widget label.
- */initValue* initializes the interface object with the entered value.

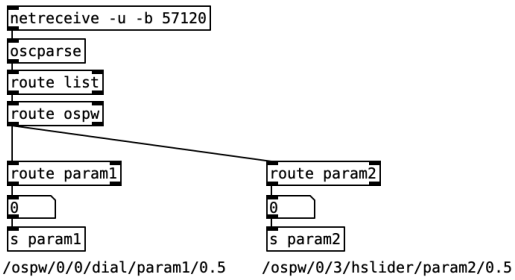


Figure 1: Pd patch with two OSPW parameters.

Alternatively, the automatic rendering can be set to a channel-based grid by placing a comment "*usechannellayout*" somewhere in the Pd patch. In this case, the grid coordinates are replaced by channel number and y position within this channel. In order to implement a custom GUI for the OSPW platform, the NexusUI framework has to be extended with new Javascript objects (widgets). The example GUI for the binaural headphone monitoring application (see section 5.2) is based on a pre-existing widget, a two-dimensional panning interface, which has been modified and given additional functionality specific to the application.

4. IMPLEMENTATION DETAILS

The Node program consists of two parts: the node server, and the index.html page. They interact with each other via web sockets.

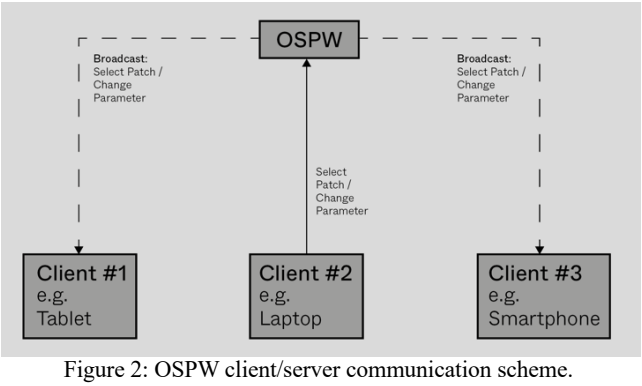


Figure 2: OSPW client/server communication scheme.

Any parameter change in any client is sent via OSC as Universal Data Package (UDP) to Pd. The port is configurable. The communication between server and clients works as follows:

- The server starts in the state 'No patch loaded'. Every client that connects to the server will render the main menu, allowing the user to select an application.
- Once a client selects an application, the server parses the corresponding Pd patch, searching for strings that start with /ospw (see figure 1) and stores all obtained data (widget type, position, name etc.).
- A broadcast message is sent to all connected clients. The GUI of the selected application will be rendered on all clients. The state of the server changes to 'Patch loaded'.
- If a new client connects to the server in this state, it loads the GUI of the current application.
- Each time a parameter is changed by a client, the new value is broadcasted to all other connected clients, allowing every client to update its interface. This way all connected clients are kept in sync with each other and can be operated at the same time.

5. EXAMPLE APPLICATIONS

Three exemplary applications have been implemented and will be described in the following section. The first two examples also serve as tutorials for OSPWs automatic interface generation and the creation of custom user interfaces. The third example is a mono-to10-channel convolution reverb and was used for evaluating and testing the parallel execution of several instances with the pd~ object.

5.1. Mixer

The first demo application is a simplified version of a digital mixing console. 16 audio input channels can be processed with a 3-band equalizer and the gain of the audio signal can be adjusted with a fader. The creation of a fully functional mixing desk was not the intention of this demo, it rather serves as an example and tutorial on how the OSPW server parses a patch and dynamically creates the corresponding interface, based on the information it finds in the patch.

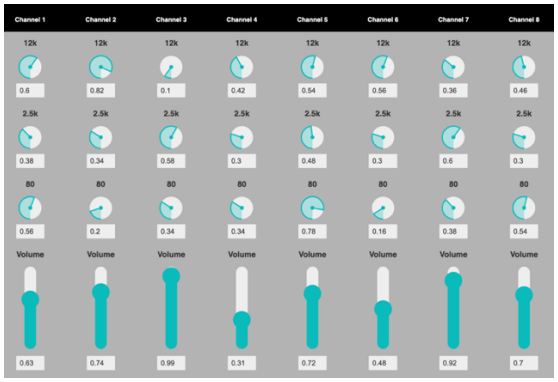


Figure 3: OSPW mixer demo.

5.2. Binaural

The second application is a binaural monitoring application for eight individual headphone mixes and serves as an example and tutorial for creating custom OSPW GUIs. The interface provides the user with eight circles (each representing a sound source) for each mix which can be placed in the virtual space around the listener as shown in figure 4 below. Each of the eight mixes can be chosen with the tabs on top of the GUI. The number of sound sources and mixes is only limited for this demo; in theory an infinite number of both sources and binaural mixes can be controlled (only limited by hardware resources). On every interaction with the widget, distance and angle of each source, in respect to the zero-degree axis of the listener, are calculated and sent to the DSP server. In addition to controlling the position of the sources, the circle in the middle representing the listeners' head can be controlled with an external head tracking device (for example the open hardware tracker described in [15]), thus providing the listeners with a dynamic binaural synthesis. The dynamic binaural rendering in Pd is realized with the vas_binaural~ object of the VAS library [16].

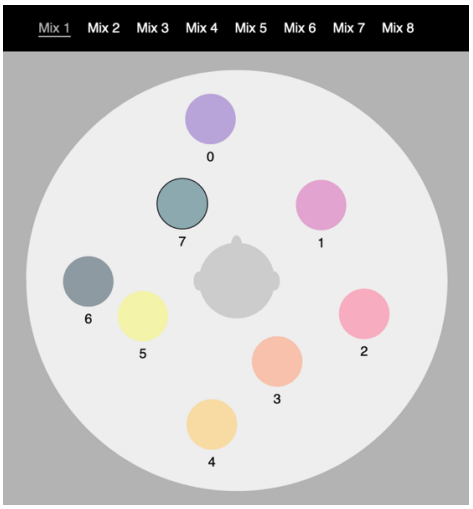


Figure 4: OSPW binaural monitoring.

5.3. Multichannel Reverb

The ten different channels for the convolution reverb were created by sampling the same reverb preset with different reverb and predelay times. The dry/wet parameter can be controlled for each channel individually. The convolution is realized with the vas_reverb~ object of the VAS library which performs a single-threaded non-equal partitioned convolution. Ten instances of the Pd patch performing the DSP are loaded from the main patch with the pd~ object in order to distribute the different reverb instances among all of the CPU cores.

6. CONCLUSION

OSPW is an easy-to-use open source DSP platform which can be built with off-the-shelf hardware components. The free choice of sound card (as long as it is ALSA compatible) makes the integration in any existing audio environment possible.

By using Pd as audio backend, the signal processing can be implemented both in the C programming language and graphically. The graphical access also enables "intermediate" programmers and artists in the field of media technology to use the system. Pre-existing Pd objects and patches of the large Pd developer community can be used as well. In order to automatically generate GUIs for existing Pd applications, only very slight patch modifications as described in section 3 are necessary.

The synchronization of all connected clients allows multiple users to use an application simultaneously. The first demo app presented in section 5 illustrates this in a simple manner. Several users can control a mixing console at the same time and even from different positions. This can be very interesting, especially for artistic applications such as a multi-player acousmonium. The second demo - the binaural monitoring application - can be realized at a fraction of the cost of a commercial solution and could be easily expanded to more binaural mixes and sources.

OSPW enables intuitive, network-based access to Pd. Finished patches are simply pushed into the designated folder and can then be selected and operated via remote client. Currently only the most important UI elements (dials, sliders and number boxes) are implemented for automatic interface generation. To ensure intuitive handling for more complex DSP algorithms, future updates should include more sophisticated UI elements such as multisliders or frequency domain editors (as they are usually used for filters). Also, a thumbnail view of the Pd patch that is currently running would be a nice feature in order to give the user an idea of what kind of DSP algorithm is currently executed on the server.

7. ACKNOWLEDGEMENTS

This work was supported by the OSPW 2.0 project, funded by the Maja Sacher-Stiftung.

8. REFERENCES

[1] H. Stenschke, T. Resch, P. Glaettli, R. Riedl, C. Fiechter, “OSPW (Open Signal Processing Workstation) - Development of a Stand-Alone Open Platform for Signal-Processing in AV-Networks”, *Audio Engineering Society Convention 142*, 2017.

[2] E. Lindemann, M. Starkier, and F. Dechelle. “The IRCAM Musical Workstation: Hardware Overview and Signal Processing Features”, *Proceedings of the 1990 International Computer Music Conference. San Francisco: International Computer Music Association*, 1990.

[3] M. Puckette, “Pure Data: another integrated computer music environment”, *Proceedings of the Second Intercollege Computer Music Concerts*, 1996.

[4] M. Puckette. “Multiprocessing for Pd”, [Online], URL: http://www.pdpatchrepo.info/hurleur/multiprocessing.pdf, [accessed 2019, December 27].

[5] C. Fiechter, T. Resch, “Git Repository of the OSPW 2.0”, [Online], URL: www.github.com/cfiechter/OSPW, [accessed 2020, August 30].

[6] Waves Inc., “SoundGrid Systems Website”, [Online], URL: https://www.waves.com/soundgrid-systems, [accessed 2019, December 27].

[7] Universal Audio, “Universal Audio Website”, [Online], URL: https://www.uaudio.com/, [accessed 2019, December 27].

[8] Biamp, “Biamp Tesire Website”, [Online], URL: https://www.biamp.com/products/tesira, [accessed 2019, December 27].

[9] G. Moro, S. Bin, R. Jack, C. Heinrichs, A. Mcpherson, “Making High-Performance Embedded Instruments with Bela and Pure Data” in *Proceedings of the International Conference of Live Interfaces,* 2016.

[10] P. Brinkmann, P. Kirn, R. Lawler, C. Mccormick, M. Roth, “Embedding Pure Data with libpd”, URL: https://www.uni-weimar.de/kunst-und-gestaltung/wiki/images/Embedding_Pure_Data_with_libpd.pdf, [accessed 2019, December 27].

[11] S. Tarakajian, D. Zicarelli, J.K. Clayton, “Mira: Liveness in iPad Controllers for Max/MSP”, *Proceedings of New Interfaces for Musical Expression (NIME),* 2013.

[12] S. Merchel, L. Kormann, “FreeDSP: A Low-Budget Open-Source Audio-DSP Module.”, *DAFx,* 2014.

[13] OpenJS Foundation, “Node.js”, [Online], URL: https://nodejs.org/, [accessed 2020, January 11].

[14] B. Taylor, J. Allison, W. Conlin, Y. Oh, D. Holmes, “Simplified Expressive Mobile Development with NexusUI, NexusUp and NexusDrop”, *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME),* 2014.

[15] T. Resch, M. Hädrich, “The Virtual Acoustic Spaces Unity Spatializer with custom head tracker”, *5th International Conference on Spatial Audio ICSA,* 2019.

[16] T. Resch, C. Böhm, S. Weinzierl, „VAS – A cross platform C-library for efficient dynamic binaural synthesis on mobile devices“, *AES, International Conference on Headphone Technology*, 2019.

Pict2Audio : Sound Generation by Hand-Drawn Images Analysis using Convolutional Neural Networks

Joséphine Calandra, Pierre Hanna, Pierrick Legrand, Myriam Desainte-Catherine

SCRIME, Bordeaux University, France

Scan and get more information !

The SCRIME

This project has been developed during Josephine Calandra's end-of-study internship as a student at ENSEIRB-MATMECA, which occured at the SCRIME, the Studio of creation and research in computer science and experimental musics.

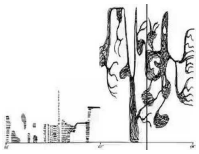


Figure: Extract from the music sheet Mycenae Alpha, Iannis Xenakis.

From electroacoustic music to new means of expression...

Electroacoustic music composition invites people to think about the instrumental composition in other ways. The use of transformed, synthetics or artificials sounds created by a computer as a tool leads to new reflections about the process of music creation, creation of new sounds and interaction between the tool and the composer. Since the XXth century, new tools, languages and edition software has been developed by research centers and companies.

...To the necessity to create a dedicated support

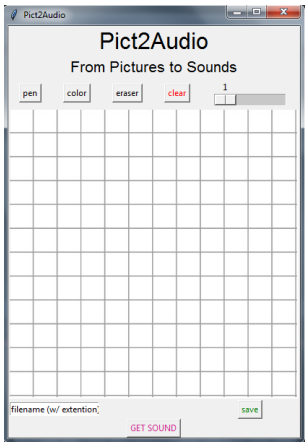
Composers from the XX^{th} and XXI^{rst} century develop their own tools and languages to compose. Nevertheless these tools may not be universal but application-specific, and they could be barely intuitive nor customizable. Sounds in edition software are fixed, the software being more a way to edit than to express the sound. A problem has emerged : how to create a universal tool that could facilitate the musical creation ? We would like to create an intuitive tool that leads to a natural expression of the composer who could create her/his own language but also manipulate it easily.

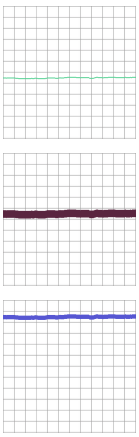
Pict2Audio : A tool for composition

- Matches drawing and sounds.
- Creates a customizable graphic language.
- Uses sound databases belonging to the composer.
- Provides a drawing interface *via* graphic tablet augmented with a stylus.

Technologies used

- Neural Networks developed in Python with the libraries Keras and Tensorflow.
- Use of Google Colab environment, that runs neural networks on Google servers.
- Graphic Interface developed in python using the library tkinter.



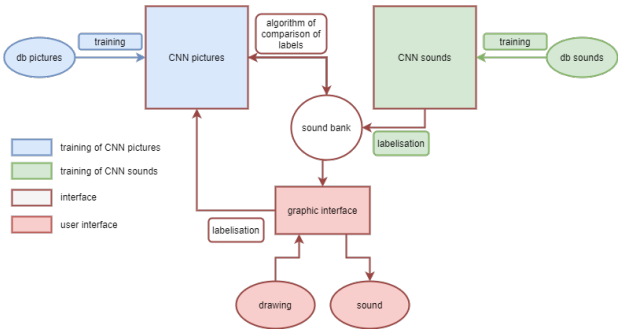


The system of Pic2Audio

The system is divided into two parts :

1) **The training of the system :** The drawer gives sounds to the system. These sounds are used to train a group of neural networks according to characteristics associated with the sound. Then, the neural networks associated with pictures are trained in the following way : the composer draws on the interface pictures such as visual characteristics correspond to audible characteristics.

2) **The use of the system :** once the neural networks are trained, the composer can draw anything that corresponds to the audible characteristic desired. The neural networks will analyse the picture, and the system will return the associated sound thanks to a dedicated matching algorithm.



Why do we use neural networks ?

Neural Networks are specifically efficient for classification of pictures and sounds. Moreover, it does not need to know the algorithms nor the architecture of the systems that generate the signals at the entry of the neural networks. This enables a possible abstraction of music theory and sound analysis. Then, the neural network detects the characteristics of the pictures, wich leads to a personalisation of the system.

Specific framework and improvement

In this context we limited our researches to the specific training of the neural network with a predetermined language where three visual characteristics are associated with three audible characteristics. The drawings are lines where the colors correspond to a tone, the height of the line corresponds to the height of the note and the thickness of the line corresponds to the volume of the sound. The sound databases are augmented databases of NSynth, the database proposed by Magenta, the music generation research project of the Google AI team.

Limits

To go towards a usability and universality of the system, it still has to be tested with various and complex drawings. Moreover, the creation of the databases can be laborious for the composer, so the databases should be automatically augmented. Also, there could be a bad training of the neural networks, so there is a need to create an assistance for the composer.

Perspectives

The future developments of this system could be the implementation of the polyphony, the management of time in the system, and the drawing and emission of sound in real-time.

And why not...

We could go further by imagining the system in 3 dimensions, coupled with an augmented-reality system. Moreover, even if Pict2Audio is aimed at helping composers, this could also be a tool used for pedagogy, to help learning music.

Author Index