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.

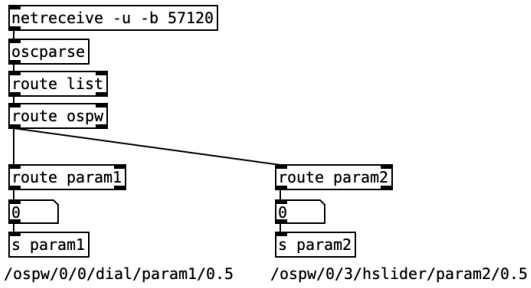

Figure 1: Pd patch with two OSPW parameters.

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.

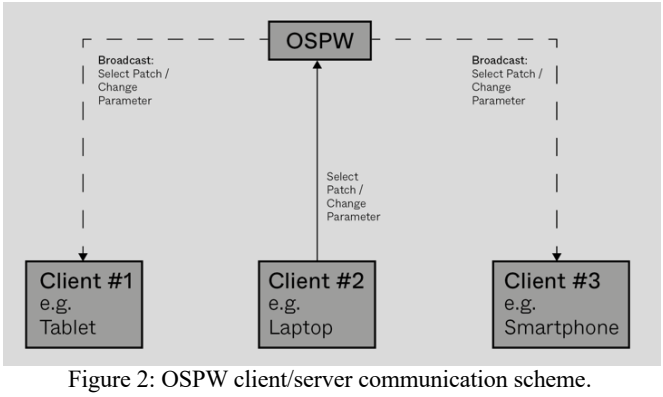

Figure 2: OSPW client/server communication scheme.

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.

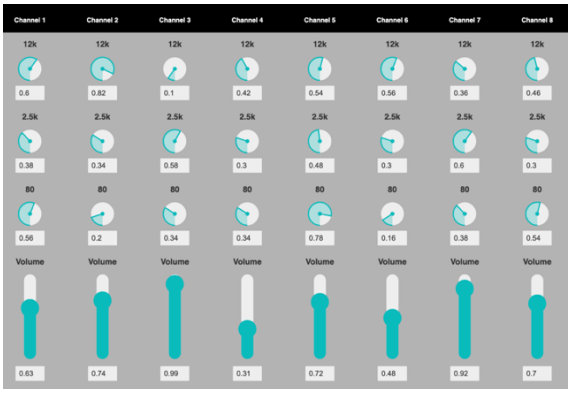

Figure 3: OSPW mixer demo.

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].

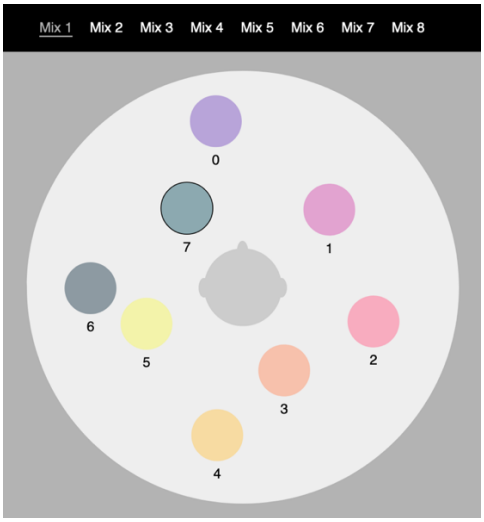

Figure 4: OSPW binaural monitoring.

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.