A Networking Extension for the SoundScape Renderer

Master's Thesis

Technische Universität Berlin
Fakultät I - Geisteswissenschaften
Fachgebiet Audiokommunikation
Audiokommunikation und -technologie M.Sc.

Vorgelegt von: David Runge
[redacted] [redacted]

Erstgutachter:	Prof. Dr. Stefan Weinzierl
Zweitgutachter:	Henrik von Coler
Datum:	2017-07-28

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.
Berlin, den 2017-07-28

. . . .
David Runge

Abstract Different types of software have been designed in the field of spatial audio reproduction to cope with the needs of several rendering algorithms, conceived over the last century. Their feasibility and reproducability in a real environment have been evaluated and tested and their implementations extended, or improved accordingly.
Today, spatial audio rendering software is often a single-purpose application, that attends to the needs of a specific setup in place. Unfortunately, to some of these solutions, no further work is applied, leaving them unsupported at some point in time. The affected hardware setups range from those for large scale Wave Field Synthesis to smaller ones, found in scientific research, involving Dynamic Binaural Synthesis. All of them are in need of extensively supported and reliable software, that can even be used, once the operating system is changing. This can happen, if the hardware has to be upgraded due to old age or if new software is required, that can not be built on older operating systems.
In the following work, a set of current free and open-source realtime spatial audio renderers, actively used in scientific and artistic contexts, is evaluated for usability, applicability and realtime context: sWONDER, HoaLibrary (for PureData), 3Dj (for SuperCollider), WFSCollider and the SoundScape Renderer. The latter — in contrast to the other candidates, a multi-purpose renderer — is chosen for a rewrite of its network based messaging system, as the application implements many different rendering algorithms, while still being actively maintained.
Its new functionality allows it to be used in other environments: While rendering the same virtual audio scene, large scale multi-loudspeaker setups, in which several instances work collectively, are possible as well as networked individual setups.
The SoundScape Renderer's new networking extension, along with a client-server architecture, its messaging system with tests and workflow examples is elaborated. New setup possibilities are contrasted with the automation available through the network interface currently in use.
Closing, an outlook on future work with the help of the new networking extension and general improvement suggestions for the SoundScape Renderer are discussed.

Zusammenfassung Unterschiedlichste Software wurde im Feld der Raumklangsteuerung geschrieben, um die vielzähligen Algorithmen des letzten Jahrhunderts abbilden zu können. Die Praktikabilität und Reproduzierbarkeit dieser Algorithmen in einer realen Umgebung wurde evaluiert und getetest und ihre Implementationen entsprechend erweitert, oder verbessert.
Heutige Raumklangsteuerungssoftware ist häufig eine Einzweck-Software, welche einem spezifischen Aufbau dient. Unglücklicherweise werden einige dieser Lösungen nicht weiter entwickelt, was zukünftig zwangsläufig zu ihrer Unbrauchbarkeit führt. Die betroffenen Hardwareaufbauten reichen von großen Anlagen, wie jene für die Umsetzung einer Wellenfeldsynthese zu kleineren, in der wisschenaftlichen Forschung auffindbare, die Dynamische Binauralsynthese anwenden. All diese Hardwareumgebungen benötigen Software, die weitläufig unterstützt ist und verlässlich arbeitet, auch wenn die Betriebssysteme, auf denen sie genutzt werden, sich verändern. Diese Umstände können eintreten, wenn veraltete Hardware ausgewechselt werden muss, oder neuere Software benötigt wird, die nicht auf älteren Betriebssystemen einsetzbar ist.
In der nachfolgenden Arbeit wird die folgende Sammlung aus freien und quelloffenen Anwendungen zur Echtzeit-Raumklangsteuerung, die derzeit aktiv in künstlerischen und wisschenschaftlichen Umgebungen Anwendung finden, anhand ihrer Nutzungsmöglichkeiten, ihrer Anwendbarkeitkeit und ihrer Umgebung evaluiert: sWONDER, HoaLibrary (für PureData), 3Dj (für SuperCollider), WFSCollider und der SoundScape Renderer. Der letztere — im Gegensatz zu den anderen Kandidaten ein Mehrzweck-Werkzeug — wurde für eine Neuausarbeitung seiner Netzwerkfähigkeit gewählt, aufgrund seiner vielzähligen Implementationen von Raumklang-Algorithmen und seiner noch betriebenen Weiterentwicklung.
Seine neue Funktionalität erlaubt es ihm in anderen Umgebungen eingesetzt zu werden: Während die gleiche virtuelle Audio-Szene verräumlicht wird, ist die Anwendung nun gleichermaßen einsetzbar in einer Großanlage, in der viele Instanzen zusammen arbeiten und vernetzte, individuellen Aufbauten, die getrennt voneinander arbeiten.
Die neuen Netzwerkmöglichkeiten der Anwendung, zusammen mit einer Client-Server Architektur, sowie seines Nachrichtensystem, anhand von Arbeitsabläufen und Tests, werden ausgearbeitet. Neue Möglichkeiten der Vernetzung werden der derzeitig verwendeten Netzwerkschnittstelle gegenüber gestellt.
Abschließend werden ein Ausblick auf weitere Arbeit anhand der neuen Netzwerkschnittstelle und generelle Verbesserungsvorschläge für den SoundScape Renderer diskutiert.

Acknowlegdements

I would like to thank my supervisors Prof. Dr. Stefan Weinzierl and Henrik von Coler for their tremendous efforts in making this work happen.
Special thanks to Henrik for pushing in the right moments.

I would also like to thank Marc Voigt for bouncing off ideas every once in a while and helping in realizing a test setup. Hopefully this work will be useful for the setups you have to maintain in the future.

Thanks to Matthias Geier for being relentless and yet supportive.

Special thanks to my family for their loving support over all of this time spent.
Thank you Nanni, for taking my mind off.

Thanks to Sabine and Peter for reading through all of this nonsense.

To all, whom I forgot: You know, who you are. I could not have done it without you.

Contents

1 Introduction

From the early days of stereo audio reproduction onwards, different kinds of spatial audio reproduction techniques have been developed and established, ranging from plain stereophony to three-dimensional, multi-channel setups. Their applications range from research to artistic and conventionally commercial fields, such as cinema and home entertainment.

With the rise of dynamic two and three-dimensional rendering algorithms (see 2.1), the need for specialized software, implementing them, grew. Opposed to encoding of spatial information of sources in only two channels (static in the case of commercially produced audio for radio and film) encoding for massive multi loudspeaker systems would not be feasible, when done statically, or not applicable in the case of dynamic setups, reacting to user input in realtime.

Early dedicated hardware implementations, such as the *Halaphon*, designed by Hans Peter Haller and Peter Lawo (Haller, 1995, p.78f), started out as basic spatial dispersion systems for quadrophonic loudspeaker setups, based on amplitude pannings using envelopes. Due to huge interest from artists in this new technique, these systems were soon expanded to cope with eight and more channels.
A notable piece, making use of a later revision of the *Halaphon*, is Luigi Nono's *Prometeo*. For it, the componist developed the *coro lantissimo*: A choir singing at a great distance. To achieve the effect — in a usually very dampened orchestra house — the spatialization system was used to add between eight to 15 seconds of reverberation time in the prolog and up to 20 seconds in the second part of the piece. This enabled a sung *fivefold pianissimo* and a *triple pianissimo* (respectively) to be perceived as coming from a larger distance than the room's dimension (Haller, 1995, p. 91f).
This early example of a spatial audio renderer already illustrates the close vicinity of applied scientific research in experimental electronic music studios and that of artistic work, facilitating live electronics.

With the fast technological development of computer systems, the dedicated solutions shifted more into the digital domain and finally towards software solutions. This effectively allowed a stronger focus on specializing and refining the algorithms in use.

Spatial audio rendering software exists for different Operating Systems (OSs), in several stages of completeness and feature richness, while covered

by free (see 2.7) and non-free licenses. The following work focusses on free software, used in scientific research and artistic contexts. Several spatial audio renderers, currently in use, were evaluated and compared (see 2), of which one was chosen for extension.
Some spatial audio renderers are single-purpose applications, conceived for a specific (and often quite rare) loudspeaker setup, such as those used for Wave Field Synthesis (WFS) or Higher Order Ambisonics (HOA). An example of this is the large scale system at Technische Universität Berlin (TU Berlin) (Audiokommunikation, 2017) or HAW Hamburg.

The SoundScape Renderer (SSR) is a multi-purpose spatial audio renderer, developed at the TU Berlin. To improve its usability and networking capabilities, a new networking extension was developed, facilitating an Open Sound Control (OSC) based messaging system, that incorporates features for distributed processing in massive multi-loudspeaker setups.

2 Free and Open-source Spatial Audio Renderers

JACK Audio Connection Kit (JACK) (Davis, 2016) is a low-latency audio server, that allows for software using its environment to connect their in- and outputs with any other application using it. It is licensed under the GNU General Public License (GPL) and can be built for various OSs (e.g. Linux, macOS, Windows). As of today, a plethora of applications exist, that extend JACK's functionality graphically, or make use of it musically and productively. Due to the large set of audio drivers it can use (i.e. Advanced Linux Sound Architecture (ALSA), coreaudio, freebob, oss sun and portaudio) and its general availability, the audio server has become the de-facto standard for free and open-source, production ready applications on all major OSs.
To date there exist five (known of) free and open-source spatial audio renderers, which are all JACK clients:

- sWONDER (Baalman, 2007), developed at the TU Berlin, Germany
- WFSCollider (Snoei et al., 2016), developed by the Game Of Life Foundation (Foundation, 2016), The Hague, Netherlands
- HoaLibrary for PureData (Pd) (Guillot et al., 2017b) developed at the Centre de recherche Informatique et Création Musicale (CICM), Paris, France
- 3Dj for SuperCollider (Pérez-López, 2014), developed at the Universitat Pompeu Fabra, Barcelona
- SSR (Quality & Usability Lab, Telekom Innovation Laboratories et al., 2016), developed at the Quality & Usability Lab, Telekom Innovation Laboratories, TU Berlin and Institut für Nachrichtentechnik, Universität Rostock and Division of Applied Acoustics, Chalmers University of Technology

Different concepts and contexts apply to all of the renderers, which are briefly explained in the following sections, prefixed by a section about spatial audio rendering algorithms and followed by one about free software and its pitfalls.

2.1 Spatial Audio Rendering Algorithms

In the following subsections several spatial audio rendering algorithms are introduced briefly. As they serve as a mere introduction, they were merged where applicable.

2.1.1 Dynamic Binaural Synthesis and Dynamic Binaural Room Synthesis

Binaural Synthesis (BS) describes a stereophonic audio reproduction, in which — usually using headphones — acoustic signals are recreated at the ears of the listener.
For humans, sound source localization and distance estimation takes place according to auditory cues from each ear. The signals perceived by inner and outer ear are correlated by the brain, to account for locations in all three dimensions and their distances from the listener.
The differences between the cues perceived by each ear can be measured as a Head Related Impulse Response (HRIR) for every human individually (as it is dependant on physiology). Its Fourier transform, the Head Related Transfer Function (HRTF), can then be used to modify audio signals to become a directional audio source, perceived as in free field conditions.
Binaural Room Synthesis (BRS) is a special form of BS, in which Binaural Room Impulse Responses (BRIRs), encode all of the virtual source's characteristics, such as position, alongside the room's acoustic characteristics. This way, recordings from real rooms can be reproduced authentically.
HRIRs and BRIRs are by default applied seperately for each ear. Therefore, if a resolution of 1° is desired, it can be achieved by a set of 720 impulse responses, that are applied to the source with the help of a head tracker, measuring the azimuth of the listener towards it.

2.1.2 (Higher Order) Ambisonics Amplitude Panning and Near-Field-Compensated Higher Order Ambisonics

Ambisonics Amplitude Panning (AAP) and HOA are spatial rendering algorithms, that reproduce audio on multi-speaker setups. Those are usually circular or spherical.
Depending on a loudspeaker's position in the setup, relative to the spheres's center (the listening area or *sweet spot* (Wierstorf, 2014, Fig. 1.4)), a linear combination of all loudspeakers is used to achieve a localized representation of a virtual sound source.
The relatively small listening area can be extended by using additional sets of loudspeakers, which in turn lead to more spatial aliasing.
Due to the perceptibility of localization cues, mentioned in 2.1.1, it is required to apply spatial equalization for the rendered sources, to account for differences in low- and high-frequency localization capabilities of the human ear.
For ambisonics, plane-wave sources are assumed, which means their distance

is infinite. Due to the proximity effect, this leads to a bass boost in the listening area. Near-Field-Compensated Higher Order Ambisonics (NFC-HOA) accounts for this by a set of driving functions, applying a per speaker near-field compensation.

2.1.3 Vector Based Amplitude Panning

Vector Based Amplitude Panning (VBAP) is another rendering method for multiple loudspeakers. Up to three loudspeakers are used to reproduce a virtual sound source in a three-dimensional setup, while only two are needed in a horizontal one.
It enables for “virtual source positioning in a three-dimensional sound field formed by loudspeakers in an arbitrary three-dimensional placement“, while being ”computationally efficient and accurate“ (Pulkki, 1997, p. 464).
However, according to Geier and Spors (2012) ”VBAP has a very small sweet spot, out of which localization of sources is distorted towards the nearest active loudspeaker“ and ”works best for circular setups“.

2.1.4 Wave Field Synthesis

WFS is a spatial audio rendering technique, which is based on the Huygens-Fresnel principle. It states that any wave front can be synthesized by the superposition of elementary spherical waves.
Setups mainly focus on horizontal, preferably spatially discrete, speaker arrays of rectangular or circular shape as the human hearing is most capable of localizing acoustic sources in this plane.
According to Wierstorf et al. (2012), localization is accurately and evenly distributed in the listening area with loudspeaker spacings of up to 40cm.
Although WFS does not suffer from a pronounced sweet spot, and spatial aliasing is distributed over a relatively large listening area, compared to e.g. NFC-HOA, the spatial sampling artifacts may still be perceived as coloration of the sound field, which can be improved by prefiltering especially high-frequency content (Wittek, 2007).
Due to the relatively high amount of loudspeakers (and thereby computing power to calculate as many audio channels) needed for a medium to large-scale setup, WFS is not yet very widely distributed.

2.2 sWONDER

sWONDER (Baalman, 2007) consists of a set of C++ applications that provide BS and WFS rendering. In 2007 it was specifically redesigned (Baalman et al., 2007) to cope with large scale WFS setups in which several (computer) nodes, providing several speakers each, drive a system together.
In these setups each node receives all available audio streams (which represent one virtual audio source respectively) redundantly and a master application signals which node is responsible for rendering what source on which speaker.
It uses OSC for messaging between its components and for setting its controls. Additionally, it can be controlled through a Graphical User Interface (GUI), that was specifically designed for it.
Sound sources can be moved dynamically, or according to an Extensible Markup Language (XML) based score.
For example sWONDER has been in use for the medium and large scale WFS systems in the Electronic Music Studio (von Coler and Pysiewicz, 2017) and lecture hall H0103 (Audiokommunikation, 2017) at TU Berlin and a medium scale system at the Wave Field Synthesis Lab at HAW in Hamburg (Fohl, 2013).
The included convolution engine fWonder is applied in "Assessing the Authenticity of Individual Dynamic Binaural Synthesis" (Lindau, 2014, pp. 223-246).
Unfortunately, the spatial audio renderer has not been actively maintained for several years. Hence it is limited to its two rendering algorithms and has many bugs, that are not likely to get fixed in the future.

2.3 HoaLibrary (PureData extension)

The HoaLibrary is "a collection of C++ and FAUST classes and objects for Max, PureData and VST destined to high order ambisonics sound reproduction" (Guillot et al., 2017a). By the extension for Pd (Puckette, 1997), it enables for HOA reproduction, while harnessing the rich feature set of the audio programming language still enables for implementing other forms of spatial rendering alongside the HoaLibrary.
Pd is OSC capable with the help of extensions, such as *mrpeach*[1] or *IEMnet*[2].

[1] https://puredata.info/downloads/mrpeach
[2] https://puredata.info/downloads/iemnet

2.4 3Dj (SuperCollider Quark)

3Dj is a SuperCollider Quark conceived in the course of a Master's Thesis at Universitat Pompeu Fabra, Barcelona (Pérez-López, 2014) for interactive performance live spatialization purposes. It implements HOA and VBAP rendering (Pérez-López, 2014, p 45) and uses a specific scene format (Pérez-López, 2014, pp. 45–46) to allow sound sources to have static, linear, random, brownian, simple harmonic and orbital motion.
Due to being a language extension to sclang, 3Dj can be used in conjunction with other spatial rendering algorithms provided by SuperCollider or any of its Quarks.
SuperCollider is OSC enabled by default, which renders 3Dj a dynamically accessible solution.

2.5 WFSCollider

WFSCollider was built on top of SuperCollider 3.5 (McCartney, 2017) and as its name suggests, it is an application for WFS reproduction. It “allows soundfiles, live input and synthesis processes to be placed in a score editor where start times, and durations can be set and trajectories or positions assigned to each event. It also allows realtime changement of parameters and on the fly starting and stopping of events via GUI or OSC control. Each event can be composed of varous objects (“units”) in a processing chain“ (Snoei et al., 2016). According to its current manual, it is also capable of using a VBAP renderer for other multi-speaker setups (Sauer and Snoei, 2017, p. 8).
”WFSCollider is the driving software of the Wave Field Synthesis system of the Game Of Life Foundation“ (Foundation, 2016). In multi-computer setups, it can synchronize the involved processes and a dynamic latency can be introduced to account for high network throughput (Sauer and Snoei, 2017, p. 22). By nature WFSCollider is OSC capable and extendable by what sclang has to offer. Its scores are saved as SuperCollider code, as well. It is currently only tested on macOS and is based upon a several year old version of SuperCollider.

2.6 SoundScape Renderer

The SSR, written in C++, is a multi-purpose spatial audio renderer, that runs on Linux and macOS. Based on its underlying Audio Processing Framework (APF) (Geier et al., 2012), it is able to use BS, BRS, AAP, WFS, NFC-HOA and VBAP. However, all rendering algorithms with potentially

orthogonal sound fields, are currently only available in 2D (Geier et al., 2008). It can be used with a Qt4 based GUI or headless (without one), depicting the virtual sources, their volumes and positions. If a loudspeaker based renderer is chosen, the GUI also illustrates which speakers are currently used for rendering a selected source.
The SSR, since its conception, had a history of conducting psychoacoustic experiments with it (Geier and Spors, 2010).
Current scientific research with the BS and BRS renderers were done by Ackermann and Ilse (2015), Böhm (2015) or Grigoriev (2017). The WFS renderer has been improved by the work of several research papers, dealing with enhancements of spatial aliasing, active listening room and loudspeaker compensation and active noise control (Spors et al., 2008) and analyzing and pre-equalizing in 2.5-dimensional WFS (Spors and Ahrens, 2010). The loudspeaker based renderer was also used for psychoacoustic experiments, such as the one found in Koslowski (2013)
The SSR uses XML based configuration files for reproduction (i.e. how something is played back) and scene (i.e. what is played back). The Audio Scene Description Format (ASDF) however is not (yet) able to represent dynamic setups.
The application can be controlled through a Transmission Control Protocol (TCP)/Internet Protocol (IP) socket. OSC functionality can only be achieved using the capabilities of other applications such as Pd (Puckette, 2016) in combination with it.
Unlike sWONDER or WFSCollider, the SSR is not able to run medium or large-scale WFS setups, as it lacks the features to communicate between instances of itself on several computers, while these instances serve a subset of the available loudspeakers.

2.7 Why Free Software Matters and What Its Pitfalls Are

Free software is the terminology for software published under a free license. Licenses, such as the GPL are considered free, because they allow for anyone to copy, modify and redistribute the source code (under the same license).
Research is a field of work, in which reproducability is very important, as findings need to be independently verifiable. Scientific work is published and shared (sometimes also under free licenses, such as Creative Commons (CC)) amongst research groups of institutions all around the world. In an ideal world, all scientific research would be published under a free documentation license, such as the GNU Free Documentation License (FDL), allowing

access to anyone, free of charge.
The software used in scientific institutions is unfortunately rarely free (e.g. word processing, statistics, mathematical calculations, realtime audio synthesis and audio production) and additionally mostly bound to proprietary OSs, such as Microsoft Windows or Apple's macOS, preventing interoperability, development and an open society.
However, free software enables students and researchers to learn from the source code directly (if they can and want to), to modify (and improve) it and to share their findings. More than with proprietary software, it is possible to have a community develop around it, that takes care of the project for a long time.
Free software nonetheless can not be considered superior. It is after all only a way of developing software and not a way to grade its efficiency or code quality. Additionally it has to be noted, that especially in a scientific context it can happen, that software is conceived by an institution, put to use, but later lacks the developers to drive the project onwards (e.g. sWONDER). Therefore, a high responsibility lies with these institutions, as they need to ensure further development on systems, not easily accessible to the public, or not feasible for home use (e.g. WFS). This situation however also holds a great opportunity for cooperation.
As the development of free and open-source software is driven by its users and its contributors, its main goal should be to build a large and dedicated community at some point. Only this way new features can be developed, while taking care of bugs in the already existing source code.
Extending a software's functionality and improving its usability, such as that of the SSR, can therefore be seen as an important step towards a more diverse user base and in effect ensuring its further development.

3 Implementation

This section covers the implementation of a networking interface for the SSR and the considerations leading to it. The application was chosen to be extended by an OSC based networking interface, because it runs on multiple OSs, offers a wide set of rendering algorithms (in various stages of completion) by using the APF (Geier et al., 2012), is used extensively in scientific research, has the future possibility to run medium and large scale WFS setups and was still actively maintained by its creators at the time of writing. Software, such as the HoaLibrary (see 2.3) or 3Dj, (see 2.4) were not considered, as they were too reliant on their environment (i.e. Pd or SuperCollider) and only implemented a small set of spatial audio renderers, while sWONDER was additionally unmaintained for a long period of time (see 2.2) and WFSCollider bound to a non-free operating system (see 2.5).

3.1 Outline

Initially, the aim was to extend the SSR's features in the scope of creating a replacement for the aging sWONDER software, enabling it to run networked instances to drive a medium or large scale WFS setup. However, the approach appeared too narrow, as the application offers many different rendering algorithms. A networking extension therefore would have to be available to all of them with an equal feature set. Additionally, extending a rendering framework by a networking feature, with the help of only one of its engines proved to be linked to a massive, but avoidable overhead (see 3.1.1). The SSR, being a multi-purpose spatial audio renderer, can be used in diverse setup scenarios (see 3.4.4). Therefore not only classic server-client relationships (see 3.1.2), but also client-only and local (see 3.1.3) setups have to be taken account of. In addition, the case of medium and large scale loudspeaker based rendering setups and their specifics have to be considered (see 3.1.4).

3.1.1 Prelimenaries

In preparation to this work, an implemention of a side-by-side installation to the OS currently driving the WFS system setup of the Electronic Studio at TU Berlin (von Coler and Pysiewicz, 2017) was attempted for testing purposes.
Arch Linux (Vinet and Griffin, 2017) was installed and configured to run the medium scale setup. Unfortunately, the proprietary Dante (Ellison, 2017) driver for Linux, offered by Four Audio (Thaden, 2017), creates non-trivial circumstances for using it on an up-to-date Linux kernel, due to ALSA Ap-

plication Programming Interface (API) changes not accounted for.
While the current OS — an Ubuntu (Shuttleworth, 2017) Studio 2012 Long Term Support (LTS) — still runs well in its own parameters, its support has run out and it is therefore becoming harder, if not impossible, to build newer software on it, using newer versions of free software compilers.
For research purposes however, it is desirable to be able to try new kernel and software features on a regular basis. It is essential to find the most stable and secure setup possible involving realtime enabled kernels and building new versions of (spatialisation) software.
The hardware of the large scale setup at TU Berlin in lecture hall H0104 was being updated and unusable at the time of writing. However, in the future it will become a valuable candidate for testing of the sought after SSR features, as its setup involves no Dante network, but is instead run by several rendering computers connected to Multichannel Audio Digital Interface (MADI) and Alesis Digital Audio Tape (ADAT) lightpipe enabled speaker systems.
Although a WFS setup for testing purposes was eligible, it is generally not required for implementing the features described in the following sections and subsections, as they can be tested using two machines running Linux, JACK and a development version of the SSR.

3.1.2 Remote Controlling a Server

An SSR server instance in the notion of a medium to large scale reproduction setup is supposed to have a set of n (pre-)assigned clients. Generally, controlling it should be possible through either User Datagram Protocol (UDP) or TCP. Every message sent to it should be distributed to all of its clients (if applicable), preferably using the same protocol used to communicate with the server. The messaging system should be flexible and scriptable.
All audio inputs available to the server should be available to its clients as well. A server instance should be able to render audio just as a client would. It should be able to receive messages from its clients and act upon them (e.g. updating GUI elements).

3.1.3 Remote Controlling Clients

An SSR client can either be local (on the same machine) or somewhere on the same network, as the server or application controlling it. It should not make a difference, if the client instance is controlled by a server instance or any other application, implementing the messaging system it uses. A client should send an update to its server or the application controlling it, upon receiving a valid message.

3.1.4 Rendering on Dedicated Speakers

In a medium or large scale setup, n clients collectively render audio on l loudspeakers, while all should be using the same i inputs and each have c hardware outputs. l is preferably a multiple of c, but definitely larger than c.
As the described setups usually have too many loudspeakers for only one machine (i.e. a client), a system has to be conceived, by which each client will only render on its specifically assigned subset of size c of the n loudspeakers.

3.2 Publisher/Subscriber Interface

The SSR internally uses a Publish-Subscribe message pattern (PubSub), which is a design pattern to implement control through and over several parts of its components.
In Object-Oriented Programming (OOP), PubSub — also called observer, listener messaging — is usually comprised of a publisher class, handling the messages, without explicitly implementing how they will be used and a subscriber class, that allows for its implementations to subscribe to the messages provided. Filtering takes place to enable subscribers to only receive a certain subset of the messages.
The SSR implements a content-based filtering system, in which each subscriber evaluates the messages received and acts depending on its own constraints to implement further actions upon it.

The abstract class `Publisher` defines the messages possible to send and provides means to subscribe to them. The global `Controller` class is its only implementation within the SSR.
The abstract class `Subscriber` in turn defines the messages understood, while its implementations in `RenderSubscriber`, `Scene`, `OscSender` and `NetworkSubscriber` take care of how they are used.
This system enables a versatile messaging layout, in which components can call the `Publisher` functionality in `Controller`, which in turn will send out messages to all of its subscribers.
Depending on the design of an application, PubSub is not necessarily a one-way-road. As shown in Figure 1, subscribers can also be able to call functions of the `Publisher`, if the implementation permits it.
In the SSR this is possible, because each `Subscriber` holds a reference to the `Controller` instance and is therefore able to call its public functions.

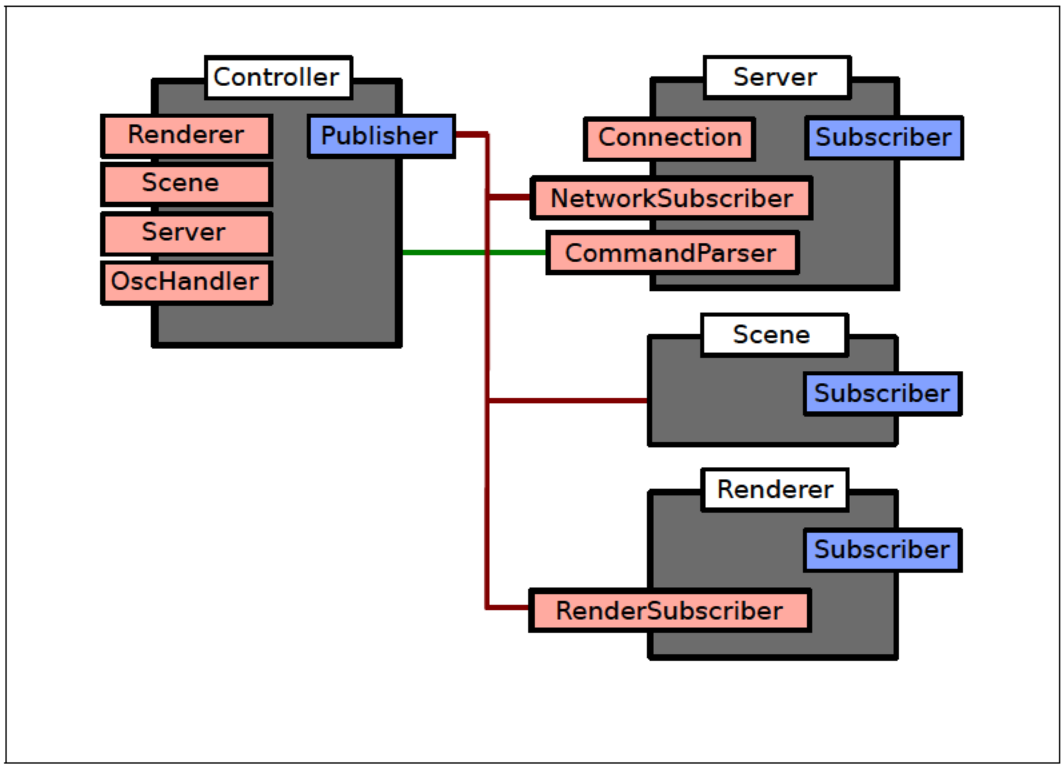

Fig. 1: A diagram depicting a simplified version of the PubSub used within the SSR with all original subscribers.
– Calls from Publisher to Subscriber – Calls from Subscribers to Controller (Publisher)

According to the principle of encapsulation in OOP, this type of functionality is handled by a separate class. In Figure 2, the `OscHandler` and `Server` instances delegate calls to `Controller` functionality to their utilities `OscReceiver` and `CommandParser` (respectively).

3.3 IP Interface

The SSR from early on incorporated a network interface, that accepts specially terminated XML-formatted strings over a TCP port, called "IP interface". This has the benefit of reusing the same XML parser code in use for scene and reproduction description.
From the perspective of other available software, it is a downside though, that it is complicated to use, as a conversion to XML has to be attempted before sending a message to the SSR. Additionally, the message has to be linted (error checked) before sending and parsed again, after receiving an answer from the application.
The IP interface achieves to offer more or less direct access to the PubSub

(see 3.2). However, it has no notion of a networked setup and could therefore be described as a two-directional message system between two destinations. With it, only setups with up to n clients are possible.

3.3.1 OSC through PureData

To allow OSC communication, the SSR incorporates a Lua based Pd external. It uses two externals (*IEMnet*[3] and *pdlua*[4]) alongside a Lua library for parsing and creating XML (*SLAXML*[5]).

3.3.2 Sending and Receiving

As mentioned in section 3.2, the `NetworkSubscriber` class (part of the IP interface) implements the `Subscriber` interface. This implies that the network interface subscribes to the messages the `Publisher` (the `Controller` instance) has to offer. Every time a function of the SSR's `Controller` instance, that was inherited from `Publisher`, is called, it will issue the call on all of its subscribers, too. Every message, available to the SSR's 3.3 is therefore directly bound to its PubSub interface's set of functions.

[3] https://puredata.info/downloads/iemnet
[4] https://puredata.info/downloads/pdlua
[5] https://github.com/Phrogz/SLAXML

3.4 Open Sound Control Interface

The networking interface conceived in the course of this work was developed in several branches, using the git version control system (written by Linus Torvalds, now maintained by Junio Hamano[6]), publicly on Github[7]. Internally the liblo library (further explained in 3.4.2) was harnessed to implement OSC functionality (see 3.4.1) for the SSR.
After initial conversations with the current maintainer Matthias Geier through the project's Github issue tracker[8], different ideas were worked out to achieve a broad solution to the server-client and client-only setups and to get a better understanding of the underlying design. Initial attempts, such as the mapping of a networking setup in the scene description[9], proved too restrictive though, as it would allow the networking functionality only to renderers, that use loudspeakers and mix scene description with networking description.
A nearly configuration-less approach, based on subscribing clients on sending poll messages to them proved more open (in the sense that it can be interfaced with by any OSC-capable application or programming language) and have less configuration overhead. With it, a diverse set of setups can be achieved (further described in 3.4.4), which at the same time remain dynamically configurable (using a plethora of OSC implementations) and debuggable using tests (further explored in 4.2).

The main implementations of the interface, further described in the following subsections, can be found in the classes `OscHandler` (handling the OSC server), `OscReceiver` (handling incoming OSC messages and acting upon them in the context of the SSR instance) and `OscSender` (responsible for reacting to calls from the PubSub, as defined in 3.2 and sending of OSC messages to clients and server).
The class `OscClient` implements the representation of a client (or server) to the message interface. It holds information about the client's address and port, along with its `MessageLevel` (a concept elaborated in 3.4.5) and its alive counter (used to check, whether a given client is still available on the network).
As shown in Figure 2, the `OscSender` is another implementation of the `Subscriber` interface. This way, every call made through the `Publisher` (i.e. the `Controller`), will be made on the corresponding function in `OscSender` as well. With `OscReceiver` the OSC interface has direct access to the `Controller` and can make calls to it, on receiving a message.

[6] https://git-scm.com
[7] https://github.com/dvzrv/ssr
[8] https://github.com/soundscaperenderer/ssr
[9] https://github.com/dvzrv/ssr/tree/distributed_reproduction

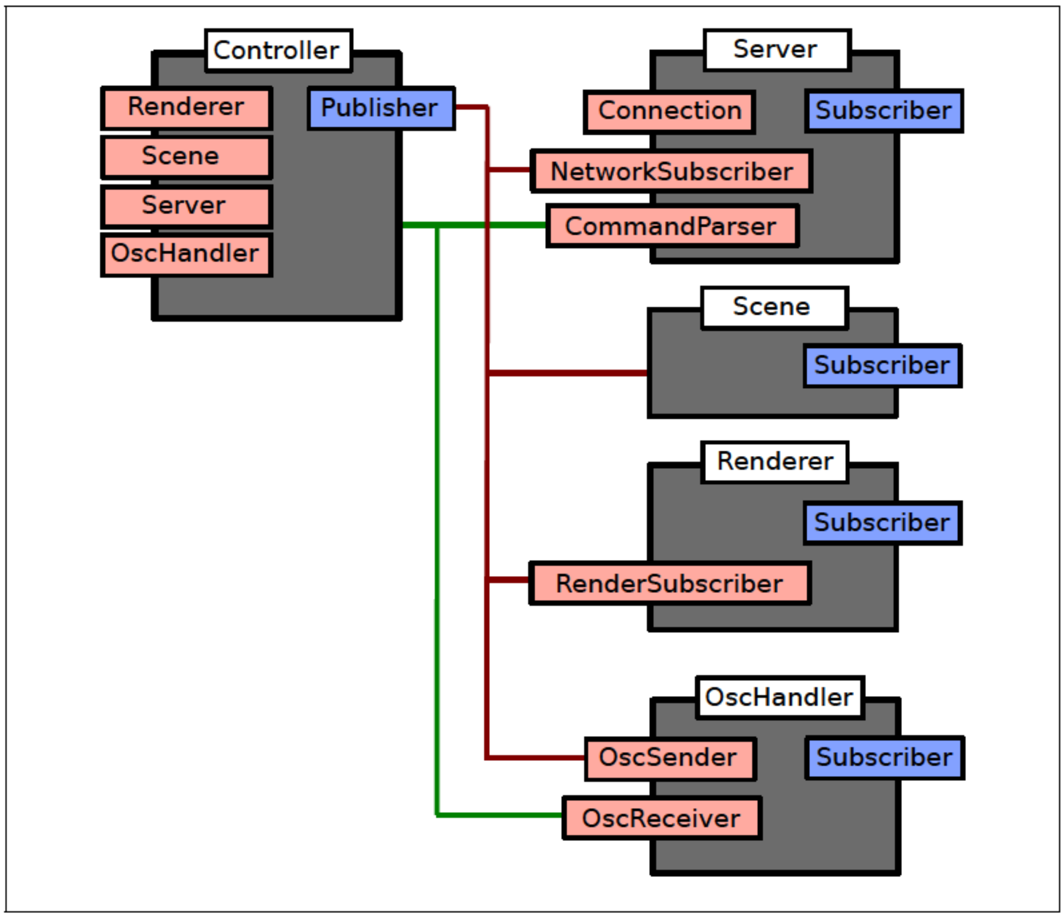

Fig. 2: A diagram depicting a simplified version of the PubSub used within the SSR with all subscribers. – Calls from `Publisher` to `Subscriber` – Calls from `Subscribers` to `Controller` (`Publisher`)

In its implementation approach the OSC interface follows that of the IP interface (see 3.3). However, it expands in creating a client-server architecture, controlled by message levels (further elaborated in 3.4.5), using a unified message interface (explained in 3.4.6).
SSR client instances only evaluate messages of server instances they are subscribed to. Server instances only evaluate messages of client instances, that are subscribed to them.

3.4.1 Open Sound Control

OSC is an "open, transport-independent, message-based protocol developed for communication among computers, sound synthesizers, and other multimedia devices" (Wright, 2002) developed at the Center for New Music and Audio Technologies (CNMAT). Its 1.0 specification was published by Matthew Wright in 2002 (Wright, 2002) and the protocol has found widespread im-

plementations (as libraries) in several programming languages. Many free and closed audio and video related applications (e.g. Ardour (Davis, 2017), Max/MSP (Cycling'74, 2017), SuperCollider (McCartney, 2017)) make use of it.
OSC's syntax is defined by several parts, which are discussed briefly in this section.

- Atomic data types, which are also reflected in type tags (see Table 1 for details)
- Address patterns (an OSC-string starting with a "/")
- Type tag string (a string, beginning with a ",", holding a set of type tags, describing a collection of atomic data types)
- Arguments, a set of binary representations of each argument
- Messages, consisting (in sequence) of an address pattern, a type tag string and n OSC arguments.
- Bundles, consisting of a set of Messages.
- Packets, the unit of transmission (sent over UDP or TCP), consisting of a message or a bundle.

According to the specification, applications sending OSC packets are considered a client and the ones receiving packets a server. Therefore, applications can both be client and server at the same time.

As shown in Table 1, only `int32`, `float32`, OSC-string and OSC-blob are considered standardized. However, most of the remaining non-standard types are implemented and used by many different clients. For implementing the SSR OSC interface, described in subsection 3.4.2 – 3.4.7, it was necessary to use the non-standard types *True* and *False* alongside the standard-types.

3.4.2 Liblo

Liblo (Sinclair, 2017) is an implementation of the OSC protocol for Portable Operating System Interface (POSIX) systems. It was initially developed by Steve Harris and is now actively maintained by Stephen Sinclair.
The library, written in C, offers a C++ abstraction layer and is released under the GNU Lesser General Public License (LGPL) v2.1 or greater. Additionally, there are wrappers for the Perl and Python programming languages.

OSC type tag	Type
`i`	`int32`
`f`	`float32`
`s`	OSC-string
`b`	OSC-blob
`h`	64 bit big-endian two's complement integer
`t`	OSC-timetag
`d`	64 bit ("double") IEEE 754 floating point number
`S`	Alternate type represented as an OSC-string (for example, for systems that differentiate "symbols" from "strings")
`c`	An ASCII character, sent as 32 bits
`r`	32 bit RGBA color
`m`	4 byte Musical Instrument Digital Interface (MIDI) message. Bytes from MSB to LSB are: port id, status byte, data1, data2
`T`	True. No bytes are allocated in the argument data.
`F`	False. No bytes are allocated in the argument data.
`N`	Nil. No bytes are allocated in the argument data.
`I`	Infinitum. No bytes are allocated in the argument data.
`[`	Indicates the beginning of an array. The tags following are for data in the Array until a close brace tag is reached.
`]`	Indicates the end of an array.

Tab. 1: Acronyms (type tags) for atomic data types, used in OSC messages and bundles (Wright, 2002). The first four types define the standard OSC type tags, which should be understood by all implementations. The remaining are non-standard types, that are implemented by most (e.g. liblo implements all but array and RGBA color type).

Due to its long standing availability and usage in many small and large-scale software projects, alongside its fairly straight forward implementability, it was chosen as the candidate for establishing an OSC interface for the SSR.
At the time of writing liblo's lastet stable release (0.28) was issued on 27th January 2014. Many changes and improvements have been applied to the codebase since then. One of them is the implementation of a `ServerThread` for the C++ abstraction layer, which runs a `Server` instance on a separate thread automatically.
In programming, threads are a way to implement simultaneous and/ or asynchroneous execution of code. The liblo `Server` class, at the core of the C++ side of the library, is responsible for assigning a network port to listen to for incoming messages, listening for messages, executing code on their arrival (i.e. callback handling) and sending messages to clients. Many applications facilitating liblo use OSC only as a messaging system. This usually means, that such an application itself is not single-purpose and is busy computing something else most of the time. Therefore it makes sense to run a Server instance on a separate background thread, to not interfere with the executional flow of the rest of the program.
The `ServerThread` class is able to free its ressources upon going out of scope (i.e. their ressources are not used by any object or function anymore), known

as Ressource Acquisition Is Initialization (RAII). For this reason, the latest development version, instead of the current stable version of liblo, was chosen for the implementation.

3.4.3 Starting the SSR

The SSR can be started with a rendering engine preselected (an executable postfixed by the supported rendering algorithm is provided by the software bundle — e.g. **ssr-wfs**) or by selecting one through the configuration file, when using the standard executable named **ssr**. This way, the following renderers become available: AAP, BS, BRS, generic, NFC-HOA, VBAP and WFS.
Additional features can be activated with the help of several flags to the executables. The customized ones, belonging to the OSC interface will be discussed in the following subsections. More information on the interplay between OSC messaging and the PubSub (see 3.2) can be found in 3.4.6.

3.4.3.1 Client Instance By default the SSR is started as an OSC client on network port 50001 and only allows using ephemeral ports (in the range 49152–65535), suggested by the Internet Assigned Numbers Authority (IANA) according to Cotton et al. (2011). As shown in Listing 1, it is possible to use a different port, by defining it with the help of the **-p** flag.

```
ssr-binaural -p ‘‘50002’’
```

Listing 1: Starting the SSR using the BS renderer as an OSC client (default) on the non-standard port 50002.

Once started, the client instance waits to receive a poll message from a server instance (or an application, mimicking one), upon which it will subscribe to it. Only then is it possible for the server application to control the client instance to the full extent via OSC.

3.4.3.2 Server Instance With the help of the **-N** flag, it is possible to start the SSR as an OSC server. Additionally, the flag can be used in a future extension of the networking interface (see 4.3.1). Additionally, in Listing 2 flag **-C** is used to specify an initial client IP and its port (the flag actually accepts a comma-separated list of IP-port pairs).
The **-p** flag, for setting a specific port is also available, when starting a server instance.

```
ssr-aap -N “server” -C “127.0.0.1:50002”
```

Listing 2: Starting the SSR using the AAP renderer as an OSC server, with an initial client on localhost, port 50002 provided.

When the server instance starts, it instantly sends out periodic poll messages to all of its active clients. Clients provided by the **-C** flag are considered instantly active.
Additionally, it is possible for clients (SSR client instances, or OSC capable applications) to subscribe to the server instance, or be subscribed to it by another client, using a message level system further explained in 3.4.5. Every valid OSC message sent to the server instance will be delegated to all of its clients upon evaluation, again according to the aforementioned message level system.
If a client instance has not answered the sent out poll message of a server 10 times, it is considered to be unavailable and will be deactivated. No messages will be sent to it anymore, until the client subscribes/ is subscribed again.

3.4.3.3 Verbosity The SSR can be started with several levels of verbosity. These are accessed by using the flag **-v**, up to three times (i.e. **-vvv**). The higher the level of verbosity, the more messages will be printed by the application. This especially applies to the OSC interface part of the SSR, as most incoming and outgoing messages will be printed to stdout at a level of **-vv**. At a level of **-vvv**, additionally all incoming and outgoing messages, that are issued in very short intervals per default (see 3.4.5 for details) will be printed.

3.4.4 Setups

The SSR offers the possibility for many different OSC enabled client-server and client-only setups. They will be explained in the following subsections.
All examples provide audio input via a JACK client, which can be local (on each client’s or server’s host computer) or provided through external audio inputs from another host computer (e.g. through ADAT or MADI). However, this is not mandatory, as the SSR is capable of playing back audio files directly.
The differences between server and client messaging is further elaborated in 3.4.6.
A special networked setup, in which the server instance is not rendering any audio, is discussed in 4.3.1.

3.4.4.1 Client-Server, Shared Rendering In Figure 3, the setup shows *1* to n client instances, controlled by a server instance. All instances are receiving audio from an external JACK client or from reading local files. Collectively, the n clients and the server are rendering audio on a shared output system (e.g. WFS or HOA).
The server instance is controlled through its GUI, sends out OSC messages to all n clients and receives their updated information (again through OSC).

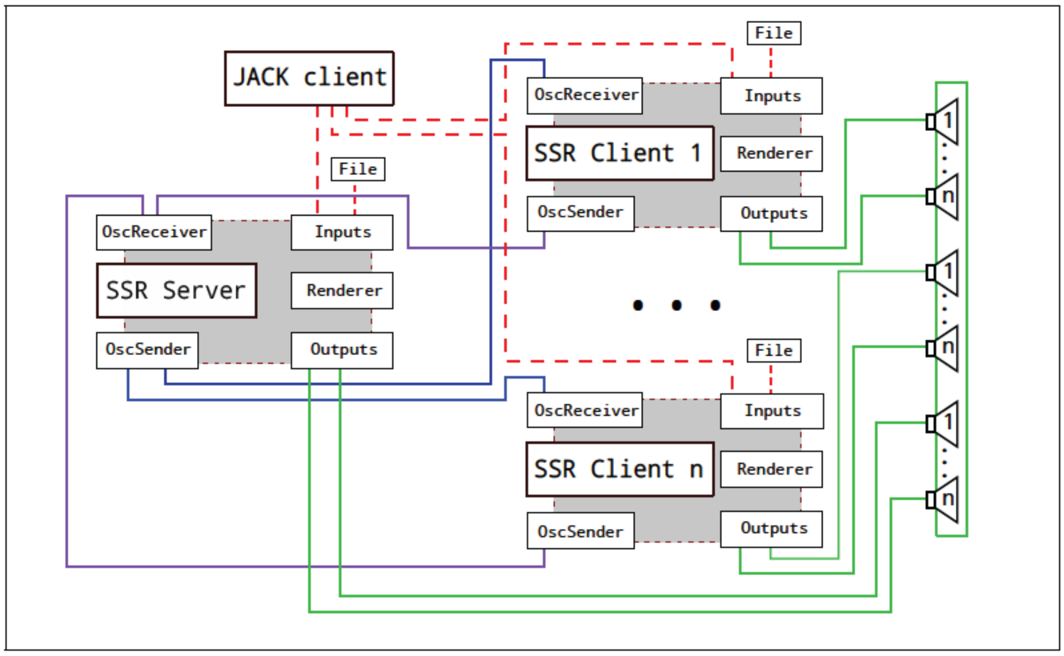

Fig. 3: A diagram displaying an SSR client/server setup, in which the server and the clients render audio collectively (e.g. WFS). The server instance is not controlled via OSC, but controls its clients through it.
– OSC input – OSC output – Audio input – Audio output

The setup shown in Figure 4 is similar to the previous one, with the exception, that the server instance is controlled by an external OSC capable application. This way, the server instance can also be run headless (without a GUI).
The set of n clients report back to the server instance, which in turn reports back to the OSC enabled application (acting as another client).

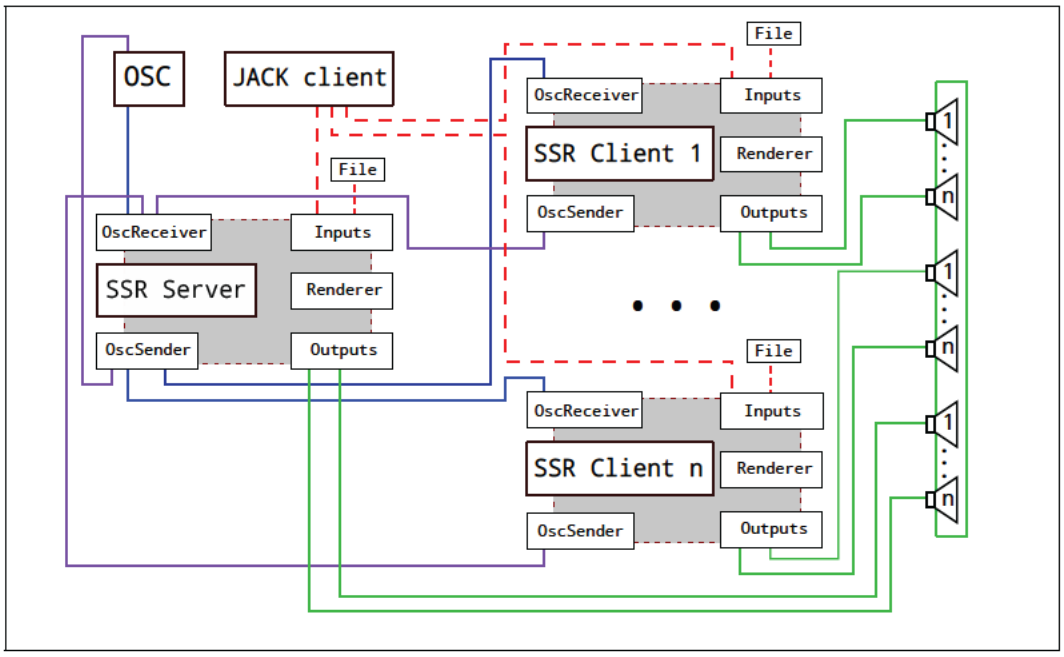

Fig. 4: A diagram displaying an SSR client/server setup, in which the server and the clients render audio collectively (e.g. WFS). The server instance is controlled by an OSC capable application (acting as another client) and controls its clients through OSC as well.
– OSC input – OSC output – Audio input – Audio output

3.4.4.2 Client-Server, Separate Rendering As shown in Figure 5, it is possible to have a setup, in which, similar to the one described in Figure 3, server and n clients render the same sources, but on separate output systems (e.g. several BS/BRS renderers or even a mixture of a WFS/HOA system and several BS/BRS renderers).

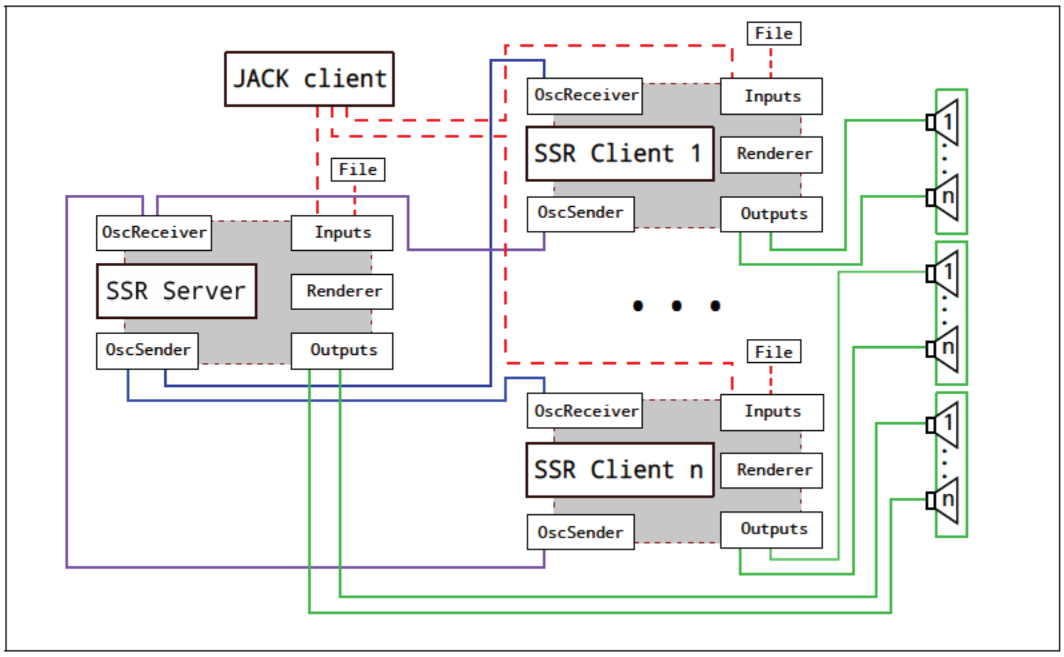

Fig. 5: A diagram displaying an SSR client/server setup, in which the server and the clients render audio to separate outputs (e.g. multiple BS renderers). The server instance is not controlled via OSC, but controls its clients through it.
– OSC input – OSC output – Audio input – Audio output

Figure 6 is an example of a similar setup, but again using an external OSC capable application to control the server instance.

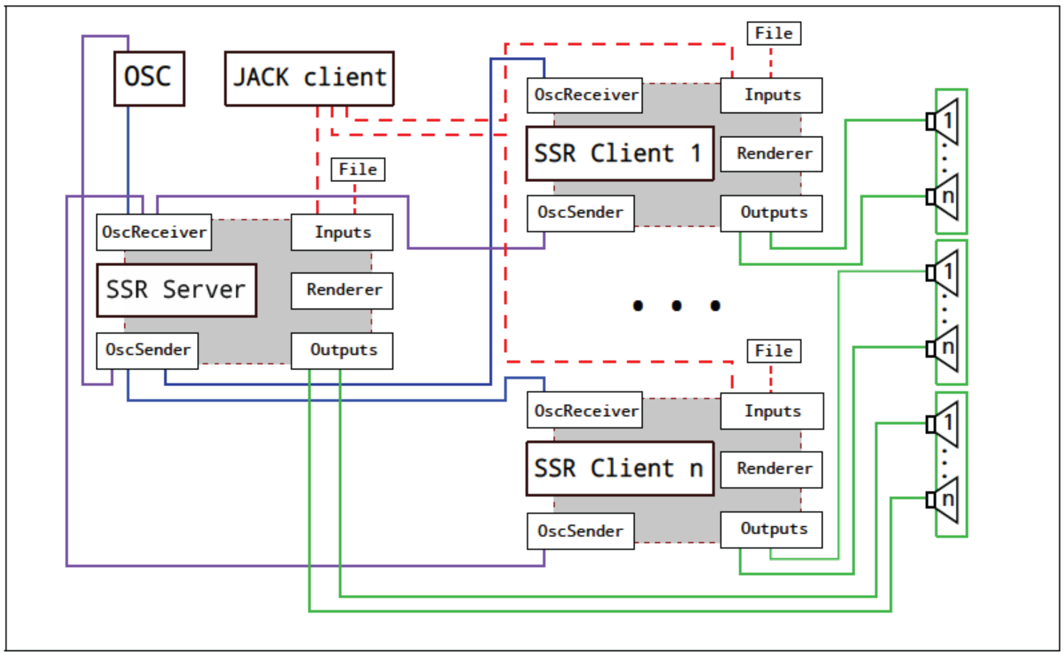

Fig. 6: A diagram displaying an SSR client/server setup, in which the server and the clients render audio separately (e.g. multiple BS renderers). The server instance is controlled by an OSC capable application (acting as another client) and controls its clients through OSC as well.
– OSC input – OSC output – Audio input – Audio output

3.4.4.3 Clients Only Using the new OSC interface, it is also possible to have client-only setups, in which an OSC capable application mimics an SSR server. This way a set of n clients can collectively (see Figure 7) or separately (see Figure 8) render audio, without the specific need of an SSR server instance controlling them. The clients send their update information back to the controlling application.
Much of the functionality implemented in the server-side of the OSC interface however has to be reapplied to the controlling software and its behavior, when dealing with SSR clients.

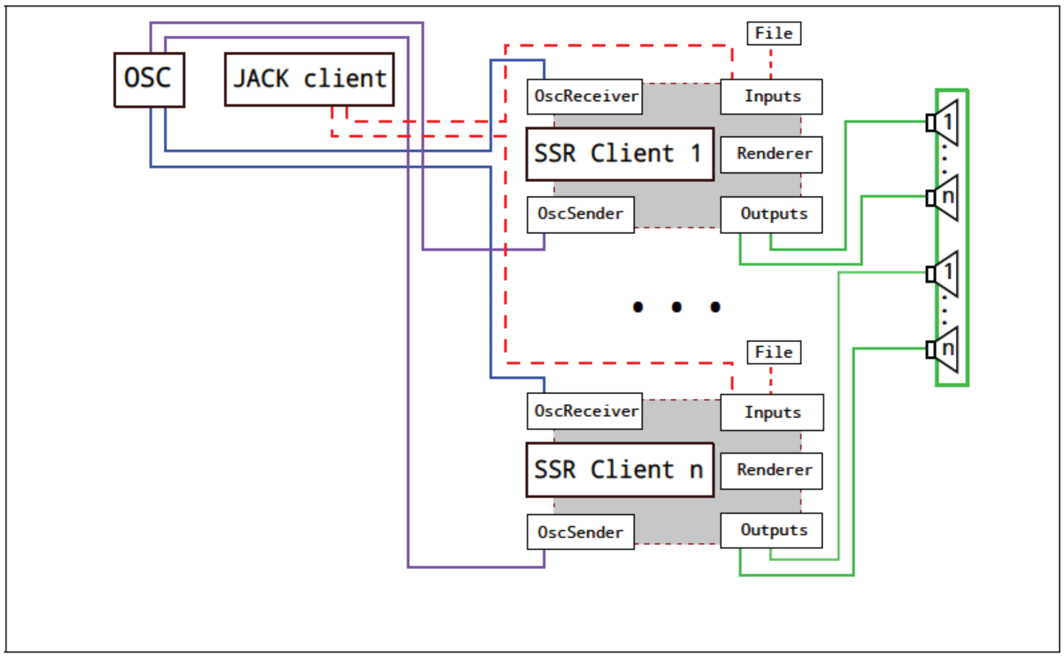

Fig. 7: A diagram displaying an SSR client cluster setup, in which a set of clients render audio collectively (e.g. medium or large-scale WFS setup). An OSC capable application acts as an SSR server instance and controls the clients.
– OSC input – OSC output – Audio input – Audio output

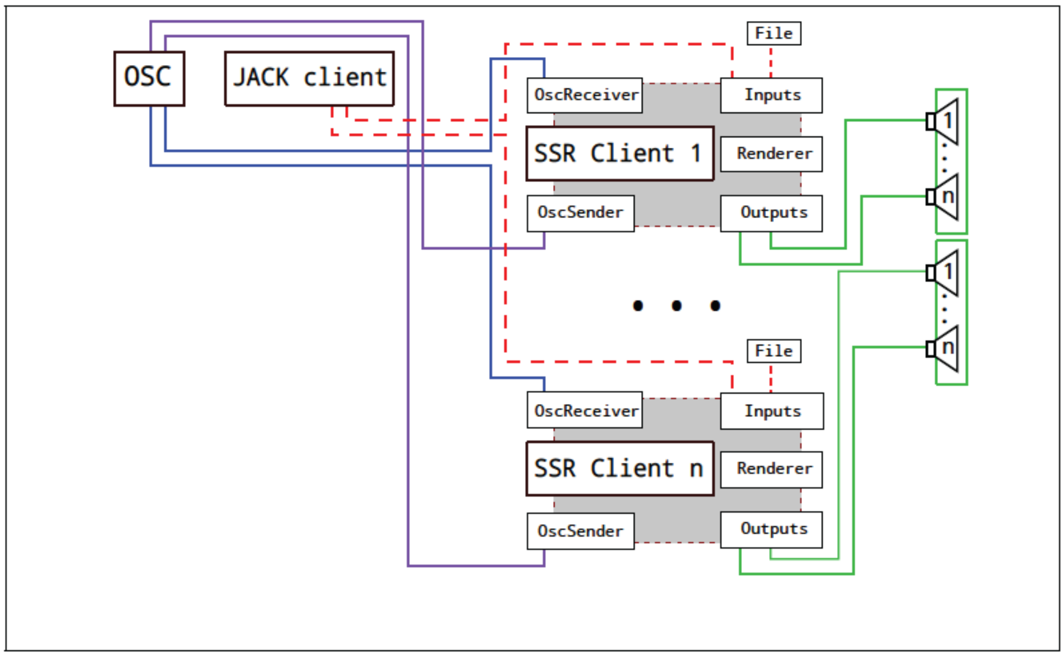

Fig. 8: A diagram displaying an SSR client cluster setup, in which a set of clients render audio separately (e.g. multiple BS renderers). An OSC capable application acts as an SSR server instance and controls the clients.
– OSC input – OSC output – Audio input – Audio output

3.4.5 Message Levels

To be able to distinguish between types of clients and servers, several message levels were implemented for the OSC interface conceived in the course of this work.
The `enumeration class MessageLevel` (see Listing 3) defines the four types `CLIENT`, `GUI_CLIENT`, `SERVER`, `GUI_SERVER`, which are represented as non-negative integers (in ascending order), starting from 0.

```
enum class MessageLevel : id_t
{
  CLIENT = 0,
  GUI_CLIENT,
  SERVER,
  GUI_SERVER,
  MAX_VALUE = GUI_SERVER
};
```

Listing 3: src/ssr_global.h: `enum class MessageLevel`

SSR client instances subscribe to SSR server instances with the `MessageLevel` `CLIENT` by default. Server instances get the `MessageLevel` `SERVER` assigned to by each client on subscribing to it.
In the OSC interface it is implemented as follows: A (recycable and reconfigurable) list of clients is held by a server instance, which enables for the `MessageLevel` to change for each client. Every client instance holds a (reconfigurable) server representation, that enables for the `MessageLevel` to change for each client towards its server.
Several messages, such as information related to Central Processing Unit (CPU) load or master signal level are not useful for a rendering client (additionally they are issued in very short intervals, which can lead to performance issues), which is why they are only sent to clients with a `MessageLevel` `GUI_CLIENT` or servers with a `MessageLevel` `GUI_SERVER`.
Lightweight OSC capable applications used to control an SSR server instance are clients to said server instance. An elevated `MessageLevel` of `SERVER` (instead of `CLIENT`) enables them to send messages to the server and have them evaluated.
Analogous to a server instance holding a `MessageLevel` of `GUI_SERVER` towards its clients, a client instance can hold the same `MessageLevel` towards a server instance to receive the above mentioned performance heavy OSC messages. How the setting up of message levels is achieved, is further elaborated

in the following section.

3.4.6 Message Interface

OSC offers the possibility of a hierarchical path tree that can be used to group messages by type (i.e. context). In conjunction with messages only understood by client or server (or a context dependant meaning), most of the messages understood by the IP interface (see 3.3) are implemented. Additional features, related to server-client and client-only functionality, were integrated as well.
In general, it can be distinguished between *direct* messages — sent from a server (or an application mimicking one) to a client or a server to trigger processing (see Table 4), reference (see Table 5), scene (see Table 6), source (see Table 7), tracker (see Table 4) or transport (see Table 4) related operations in the SSR and *update* messages (see Table 8) — sent from a client to a server upon successful processing an operation related to a *direct* message.

Path	Types	Description	Example
`/alive`		Alive notification from client (in response to a /poll)	`[/alive]`
`/message_level`	i	Set message level of sender	`[/message_level, 1]`
`/message_level`	ssi	Set message level of a specific client	`[/message_level, “127.0.0.1”, “50002”, 1]`
`/subscribe`	F	Unsubscribe sender	`[/subscribe, false]`
`/subscribe`	Fss	Unsubscribe specific client	`[/subscribe, false, “127.0.0.1”, “50002”]`
`/subscribe`	T	Subscribe sender	`[/subscribe, true]`
`/subscribe`	Ti	Subscribe sender with specific message level	`[/subscribe, true, 1]`
`/subscribe`	Tssi	Subscribe specific client with specific message level	`[/subscribe, true, “127.0.0.1”, “50002”, 1]`

Tab. 2: OSC messages relevant for subscribing and setting of message levels for clients.

Understood by server.
Data types and their acronyms are listed in Table 1.

A special set of *direct* message are the *subscribe* and *message level* (see Table 2) and *poll* and *message level* (see Table 3) messages. The former — understood only by SSR server instances — enable clients to subscribe (with a certain message level) or subscribe other clients (with a predefined message level) and set their own message level or that of another client. The latter set — only understood by clients — enables servers (or applications

Path	Types	Description	Example
`/message_level`	i	Set message level of sender (the server)	`[/message_level, 1]`
`/poll`		Poll client (continously sent), triggering subscribe (see Table 2)	`[/poll]`

Tab. 3: OSC messages relevant for polling and setting of message levels for servers subscribed to.

Understood by client.
Data types and their acronyms are listed in Table 1.

mimicking one) to poll yet unsubscribed clients to have them subscribe and subscribed clients to reply with an alive message. Similar to the *message level* message understood by server instances, the one understood by clients sets the message level (of the server representation in the client).

Path	Types	Description	Example
`/processing/state`	F	Unset processing state	`[/processing/state, false]`
`/processing/state`	T	Set processing state	`[/processing/state, true]`
`/tracker/reset`		Reset tracker	`[/tracker/reset]`
`/transport/rewind`		Rewind the JACK transport	`[/transport/rewind]`
`/transport/seek`	s	Seek to time code in JACK transport	`[/transport/seek, ‘‘42:00:00’’]`
`/transport/state`	F	Unset JACK transport state	`[/transport/state, false]`
`/transport/state`	T	Set JACK transport state	`[/transport/state, true]`

Tab. 4: OSC messages relevant for processing, tracker and (JACK) transport related settings.

Understood by server and client.
Data types and their acronyms are listed in Table 1.

Path	Types	Description	Example
`/reference/orientation`	f	Set azimuth of reference point	`[/reference/orientation, -90.0]`
`/reference/position`	ff	Set position of reference	`[/reference/position, 1.5, 2.0]`
`/reference_offset/orientation`	f	Set azimuth of reference offset position	`[/reference_offset/orientation, -90.0]`
`/reference_offset/position`	ff	Set position of reference offset	`[/reference_offset/position, 1.5, 2.0]`

Tab. 5: OSC messages relevant for reference management.

Understood by server and client.
Data types and their acronyms are listed in Table 1.

Path	Types	Description	Example
`/scene/amplitude_reference _distance`	f	Set amplitude reference distance.	`[/scene/amplitude_reference _distance, 6.0]`
`/scene/auto_rotate_sources`	F	Disable automatic rotation of sources.	`[/scene/auto_rotate_sources, false]`
`/scene/auto_rotate_sources`	T	Enable automatic rotation of sources.	`[/scene/auto_rotate_sources, true]`
`/scene/clear`		Delete all sources	`[/scene/clear]`
`/scene/decay_exponent`	f	Set amplitude decay exponent in virtual space ($1/r^{exp}$).	`[/scene/decay_exponent, 2.0]`
`/scene/load`	s	Load scene from ASDF file.	`[/scene/load, ‘‘example.asd’’]`
`/scene/master_signal_level`	f	Set the renderers signal level.	`[/scene/master_signal_level, -20]`
`/scene/save`	s	Save scene to ASDF file.	`[/scene/save, ‘‘example.asd’’]`
`/scene/volume`	f	Set scene master volume.	`[/scene/volume, 0.23]`

Tab. 6: OSC messages relevant for scene management.

Understood by server and client.
Data types and their acronyms are listed in Table 1.

When starting an SSR server instance (see 3.4.3.2), it responds to the messages shown in Table 2 , 6 , 7 , 4 , 5 and 8.
A client instance (see 3.4.3.1) will only respond to the *direct* messages listed in Table 3 , 6 , 7 , 4 and 5, but is able to send *update* messages.

There is one significant difference between the *direct* messages understood by the OSC interface and the functionality of the IP interface. The latter expects source gain to be transmitted on a logarithmic scale, ranging from *-inf* to *inf*. However, the SSR is internally calculating on a linear scale and a linear gain level of *0* is therefore hard to be reached[10]. For a more intuitive use, a linear scale was chosen for the OSC interface, ranging from *0.0* to *inf* (see gain related messages in Table 7), where *1.0* signifies 100% source level.

[10] https://github.com/SoundScapeRenderer/ssr/issues/28

Path	Types	Description	Example
`/source/delete`	i	Delete source with given id	`[/source/delete, 1]`
`/source/file_channel`	ii	Set a source's file channel	`[/source/file_channel, 1, 2]`
`/source/file_name_ or_port_number`	is	Set a source's file name or port number	`[/source/file_name_ or_port_number, 1, ‘‘1’’]`
`/source/port_name`	is	Set a source's JACK input port name	`[/source/port_name, 1, ‘‘system:capture_2’’]`
`/source/gain`	if	Set a source's gain on a linear scale (0.0 - inf)	`[/source/gain, 1, 0.2]`
`/source/model`	is	Set a source's model	`[/source/model, 1, ‘‘point’’]`
`/source/mute`	iF	Unmute a source	`[/source/mute, 1, false]`
`/source/mute`	iT	Mute a source	`[/source/mute, 1, true]`
`/source/name`	is	Set a source's name	`[/source/name, 1, ‘‘Daisy’’]`
`/source/new`	i	Create a new source stub using id	`[/source/new, 1]`
`/source/new`	sssffffTFF	Create a new source (auto-generated id) with name, model, port number, X-coordinate, Y-coordinate, orientation, gain, movability, orientation movability and mute status	`[/source/new, ‘‘Daisy’’, ‘‘point’’, ‘‘1’’, 1.0, 2.5, 90.0, 0.2, true, false, false]`
`/source/new`	sssffffisTFF	Create a new source (auto-generated id) with name, model, port number, X-coordinate, Y-coordinate, orientation, gain, file channel, properties file, movability, orientation movability and mute status	`[/source/new, ‘‘Daisy’’, ‘‘point’’, ‘‘1’’, 1.0, 2.5, 90.0, 0.2, 2, ‘‘properties.xml’’, true, false, false]`
`/source/orientation`	if	Set a source's orientation	`[/source/orientation, 1, -90.0]`
`/source/position`	iff	Set a source's position	`[/source/position, 1, 1.5, 2.0]`
`/source/position_fixed`	iF	Set a source movable	`[/source/position_fixed, 1, false]`
`/source/position_fixed`	iT	Set a source immovable	`[/source/position_fixed, 1, true]`
`/source/properties_file`	is	Set a source's properties file	`[/source/properties_file, 1, ‘‘source-properties.xml’’]`

Tab. 7: OSC messages relevant for source management.

Understood by server and client.
Data types and their acronyms are listed in Table 1.

Path	Types	Description
`/update/cpu_load`	f	CPU load changes.
`/update/processing/state`	T	Processing state is set.
`/update/processing/state`	F	Processing state is unset.
`/update/reference/orientation`	f	Reference orientation changes.
`/update/reference/position`	ff	Reference position changes.
`/update/reference_offset/orientation`	f	Reference offset orientation changes.
`/update/reference_offset/position`	ff	Reference offset position changes.
`/update/scene/amplitude_reference_distance`	f	Amplitude reference distance changes.
`/update/scene/auto_rotate_sources`	T	Auto rotation of sources is set.
`/update/scene/auto_rotate_sources`	F	Auto rotation of sources is unset.
`/update/scene/decay_exponent`	f	The scene's decay exponent has changed.
`/update/scene/master_signal_level`	f	Master signal level has changed.
`/update/scene/sample_rate`	i	Sample rate of the scene changed.
`/update/scene/volume`	f	Volume of the scene has changed.
`/update/source/delete`	i	A source with given id was deleted.
`/update/source/file_channel`	ii	A source's file channel was set.
`/update/source/file_name_or_port_number`	is	A source's file name or port number was set.
`/update/source/gain`	if	A source's gain was set.
`/update/source/length`	ii	A source's length was set.
`/update/source/level`	if	A source's output level has changed.
`/update/source/model`	is	A source's model was set.
`/update/source/mute`	iF	A source was unmuted.
`/update/source/mute`	iT	A source was muted.
`/update/source/name`	is	A source's name was set.
`/update/source/orientation`	if	A source's orientation was set.
`/update/source/new`	i	A new source with given id was created.
`/update/source/port_name`	is	A source's JACK port_name was set.
`/update/source/position`	iff	A source's position was set.
`/update/source/position_fixed`	iF	A source was set to be movable.
`/update/source/position_fixed`	iT	A source was set to be immovable.
`/update/source/properties_file`	is	A source's properties_file was set.
`/update/transport/seek`	s	JACK transport seeked to a timecode position.
`/update/transport/state`	F	JACK transport was stopped.
`/update/transport/state`	T	JACK transport was started.

Tab. 8: OSC messages for updating information on CPU load, processing, reference, scene, source, and transport of clients on a server.
No examples are given, as they are mostly analogous to the ones in Table 4 , 6 and 7.

Understood by server.
Data types and their acronyms are listed in Table 1.

3.4.7 Workflow Examples

Using any OSC capable programming language or application enables for communication with the SSR. The following examples illustrate simple workflows using sclang and should therefore be OS agnostic.

```
// sclang is a client, controlling a SSR server instance
(
  // set address of the client instance
  ~address = NetAddr("localhost", 50001);
  // print all OSC messages sent to sclang
  OSCFunc.trace(true, true);
  // subscribe to server with MessageLevel::SERVER
  ~address.sendMsg("/subscribe", $T, 2);
  // add new source with standard input at -1.0/1.0
  ~address.sendMsg("/source/new", "in_1", "point", "1",
    -1.0, 1.0, 0.1, 0.1, 0, "1", $F, $F, $T);
  // add new source with standard input at 1.0/1.0
  ~address.sendMsg("/source/new", "in_2", "point", "2",
    1.0, 1.0, 0.1, 0.1, 0, "1", $F, $F, $T);
  // unmute source 1
  ~address.sendMsg("/source/mute", 1, $F);
  // unmute source 2
  ~address.sendMsg("/source/mute", 2, $F);
  // move source 1 to -2.0/2.0
  ~address.sendMsg("/source/position", 1, -2.0, 2.0);
  // move source 2 to 2.0/2.0
  ~address.sendMsg("/source/position", 2, 2.0, 2.0);
  // remove all sources
  ~address.sendMsg("/scene/clear");
  // unsubscribe from server
  ~address.sendMsg("/subscribe", $F);
)
```

Listing 4: supercollider/workflows.scd: sclang as client controlling an SSR server instance

3.4.7.1 Controlling a Server

As shown in Listing 4, it is necessary to subscribe to the server instance with a `MessageLevel` of `SERVER` or higher. After doing so, also all *direct* OSC messages (i.e. Table 7, 6 , 5, 2) are evaluated when sent to the SSR.
The server instance will relay valid messages to all of its active clients.

```
// sclang is a server, controlling a SSR client instance
(
  // set address of the client instance
  ~address = NetAddr("localhost", 50001);
  // print all OSC messages sent to sclang
  OSCFunc.trace(true, true);
  // poll client instance to make it subscribe
  ~address.sendMsg("/poll");
  // subsequent poll makes client emit /alive message
  ~address.sendMsg("/poll");
  // add new source with standard input at -1.0/1.0
  ~address.sendMsg("/source/new", "in_1", "point", "1",
    -1.0, 1.0, 0.1, 0.1, 0, "1", $F, $F, $T);
  // add new source with standard input at 1.0/1.0
  ~address.sendMsg("/source/new", "in_2", "point", "2",
    1.0, 1.0, 0.1, 0.1, 0, "1", $F, $F, $T);
  // unmute source 1
  ~address.sendMsg("/source/mute", 1, $F);
  // unmute source 2
  ~address.sendMsg("/source/mute", 2, $F);
  // set message level to GUI_SERVER (a lot of messages!)
  ~address.sendMsg("/message_level", 3);
  // move source 1 to -2.0/2.0
  ~address.sendMsg("/source/position", 1, -2.0, 2.0);
  // move source 2 to 2.0/2.0
  ~address.sendMsg("/source/position", 2, 2.0, 2.0);
  // set message level back to SERVER
  ~address.sendMsg("/message_level", 1);
  // remove all sources
  ~address.sendMsg("/scene/clear");
)
```

Listing 5: supercollider/workflows.scd: sclang mimics server, controlling an SSR client instance

3.4.7.2 Server Mimicry When mimicking an SSR server instance in a client-only setup (e.g. Figure 7 or Figure 8), it is necessary to send a poll message to the client instance to make it subscribe (which sets the server's address and port up internally).

Afterwards — similar to the example in the subsection 3.4.4.3 — all *direct* OSC messages are accepted by the client instance, when coming from the server address and port.

An interesting concept here is to (temporarily) set a different `MessageLevel` for the application acting as a server (e.g. to `GUI_SERVER`), to receive GUI relevant messages, as explained in 3.4.6.

4 Discussion

The OSC based networking extension created for the SSR can be considered a valuable usability improvement. Its implemented features are further discussed in the following section, followed by an outlook on related future work. The extension is additionally extensively documented in the source code, to ensure the ease of further development.
Due to the versatility of how the SSR can be used in a networking context, it is likely, that some of its possibilities are not even accounted for.

4.1 Implemented Features

The OSC interface described in 3.4 can be seen as a full replacement (with one minor exception, detailed in 4.3.4) for the IP interface, already in place. Its additional features are what set it trully apart though, when not only regarding non-reliance on external software to enable OSC capabilities.

The implementation follows the internal PubSub interface, as described in 3.2 and extends it, where appropriate. Additionally, an open client-server architecture has been created, according to a message level system, further elaborated in 3.4.5. An attempt at giving extensive examples on the various setup possibilities, that are now available, is made in 3.4.4, some of which are still dependant on various missing features (see 4.3).
The OSC messaging system is adhering to the aforementioned client-server architecture by distinguishing between client-only, server-only and messages available to clients and servers alike (see 3.4.6). Examples for different workflows are given in 3.4.7 to illustrate simple use cases.
This puts the OSC interface in the unique position of providing a native messaging interface and a flexible architecture. It can be used from single local instances up to large scale networked setups (with the limitations discussed in 4.3.2 and 4.3.5).
While with the IP interface, multiple instances are only controllable by using an OSC capable application or one, that is able to send XML-formatted strings over TCP/IP, the OSC interface can deal with n clients natively, while only one instance has to be controlled using OSC. The behavior implies, that setups are possible, in which a large collection of different types of renderers can share the same scene, which is particularly useful for e.g. auditioning different rendering algorithms over the same system, or rather in the same room.
Message sending takes place over UDP, instead of TCP, which lowers the complexity of the network topology (UDP does not perform handshakes for

every packet sent, unlike TCP) and thus the size of each message sent.
The OSC interface therefore implements messaging, while using lower bandwidth and offering a greater feature set. In 4.2 a test environment is introduced, that further elaborates the overall functionality and feasibility of the message interface.

4.2 Automated Tests

The SSR was developed without the help of a test framework, which is responsible for testing its components, after they have been changed. This means, that internal (e.g. the PubSub interface) or external (e.g. the IP or OSC interface) functionality might or might not work as expected. To test the OSC interface's logical coherency and robustness automatically, a set of tests was written in sclang.
The tests are divided into those probing robustness of the OSC interface and others probing its functionality. The robustness tests further divide into server and client specific tests, where authorized and unauthorized access is tried. The functionality tests are grouped by tests for general operability, i.e. testing certain features or workflows once and long-running tests, where features are tried repeatedly.

4.2.1 Robustness

Listing 6 and 7 describe server-side tests for robustness. While the first test will not lead to any processed action by the server, the latter will. This is explained by sclang not being a subscribed client with a `MessageLevel` of `SERVER` or higher in the first case. However, in the second test sclang subscribes to the SSR server instance, which is why the OSC messages are evaluated in this case.

The tests described in Listing 8 and 9 are client-side tests for robustness, that work in a similar fashion to the aforementioned server-side tests. While the sent OSC messages are not evaluated in the first case, because sclang, mimicking a server instance (see 3.4.7.2), did not poll the SSR client instance up front, in the second case the messages are evaluated, because it did poll the client first.

In all tests for robustness the attempt is made to force errors in the implementation of the message interface (as defined in 3.4.6). This is achieved by purposely using ranges of data types for messages, that are not allowed or not defined in the SSR's internal implementation.

```
// sclang tries to control/send to server (not subscribed)
(
  // print all OSC messages sent to sclang
  OSCFunc.trace(true, true);
  ~messageLevelTestServerRandomAll.value;
  ~sourceTestRandomAll.value;
  ~updateTestRandomAll.value;
  ~subscribeTestOtherClient.value;
  ~processingTestRandomAll.value;
  ~transportTestRandomAll.value;
  ~trackerTestRandomAll.value;
  ~referenceTestRandomAll.value;
  ~sceneTestRandomAll.value;
)
```

Listing 6: supercollider/tests.scd: sclang (unsubscribed) tries to control an SSR server

```
// sclang controls server (subscribed)
(
  // print all OSC messages sent to sclang
  OSCFunc.trace(true, true);
  // set address of the server instance
  ~address = NetAddr("127.0.0.1", 50001);
  // subscribe to server with MessageLevel::SERVER
  ~address.sendMsg("/subscribe", $T, 2);
  // send alive message on subsequent poll
  ~responder_poll = OSCFunc(
    { |msg, time, addr, recvPort|
      ~address.sendMsg("/alive");
    }, '/poll'
    , ~address
  );
  ~messageLevelTestServerRandomAll.value;
  ~sourceTestRandomAll.value;
  ~updateTestRandomAll.value;
  ~processingTestRandomAll.value;
  ~transportTestRandomAll.value;
  ~trackerTestRandomAll.value;
  ~referenceTestRandomAll.value;
  ~sceneTestRandomAll.value;
)
```

Listing 7: supercollider/tests.scd: sclang (subscribed) tries to control an SSR server

Two examples for weak spot exploitations were the use of negative integers for IDs in source related messages (only non-zero, non-negative IDs are al-

```
// sclang tries to control client (not polled)
(
  // print all OSC messages sent to sclang
  OSCFunc.trace(true, true);
  ~messageLevelTestClientRandomAll.value;
  ~cpuLoadTestClient.value;
  ~sourceTestRandomAll.value;
  ~processingTestRandomAll.value;
  ~transportTestRandomAll.value;
  ~trackerTestRandomAll.value;
  ~referenceTestRandomAll.value;
  ~sceneTestRandomAll.value;
)
```

Listing 8: supercollider/tests.scd: sclang tries to control an SSR client (without polling it)

```
// sclang tries to control client (polled)
(
  // print all OSC messages sent to sclang
  OSCFunc.trace(true, true);
  // set address of the server instance
  ~address = NetAddr("127.0.0.1", 50001);
  ~pollTestClientRandomAll.value;
  ~messageLevelTestClientRandomAll.value;
  ~cpuLoadTestClient.value;
  ~sourceTestRandomAll.value;
  ~processingTestRandomAll.value;
  ~transportTestRandomAll.value;
  ~trackerTestRandomAll.value;
  ~referenceTestRandomAll.value;
  ~sceneTestRandomAll.value;
)
```

Listing 9: supercollider/tests.scd: sclang tries to control an SSR client (with previously polling it)

lowed internally) or supplying an empty string as hostname or port number for subscription messages.
The first example will lead to undefined behavior, if the range is not checked in the implementation, because a *static_ cast* is used internally to cast the value of the message data type (*unsigned int*) to the one expected by the SSR's Controller implementation (*signed int*) and the outcome of said operation is implementation dependant (depending on the OS in use).
The second example, if not checked for empty string, will lead to the OSC interface trying to create a possibly defective address and send poll messages

out to it.
While only some of the above mentioned scenarios could lead to a crash of the program under certain circumstances, left unhandled, all of them waste ressources, which is undesired. To circumvent possibly harmful input using the OSC interface, a set of sanity checks were implemented, that only allow for a received message to be processed, if all of its components fit the requirements.

4.2.2 Functionality and Operability

```
// sclang controls a client (polled), adds sources and moves them
(
  // print all OSC messages sent to sclang
  OSCFunc.trace(true, true);
  // set address of the server instance
  ~address = NetAddr("127.0.0.1", 50001);
  // poll client instance to make it subscribe
  ~address.sendMsg("/poll");
  ~sourceTestMoving.value(amountOfSources: 20);
)
```

Listing 10: supercollider/tests.scd: sclang controls an SSR client (with previously polling it), creating several sources and moving them

The test described in Listing 10 is a test for functionality, which also serves as a long-running stress test for the SSR. It creates 20 sources, that are then moved around randomly, every 100ms, for 100s, which on a Lenovo W540, with an Intel i7-4700MQ and 16Gb RAM created less than 50% of CPU load.
Based on the above mentioned tests, the basic functionality of the OSC interface can be guaranteed and depending on the host's hardware also a maximum degree of capacity utilization can be estimated, when observing the SSR's workload towards the system, while using the long-running tests.
It has to be mentioned, that a higher load can be observed, when using higher levels of verbosity (especially above **-vv**). This is explained by the fact, that the SSR will print out every OSC message received and sent above the aforementioned verbosity level.

4.3 Future Work

Several features, interesting for different use cases, were out of scope for this work. They are however complementary to the OSC networking extension,

or can be implemented on top of it and will be discussed in the following subsections.
However, before any more changes can take place, the OSC interface first has to be merged into the main source code repository for the SSR. This will also entail an update to the user manual[11], to ensure extensive documentation of the various OSC messages now understood by the software and updated build instructions, that come with the usage of liblo (see 3.4.2). Especially the latter might prove as the defining time factor, as for seamless integration in the OSs a stable version of the OSC specification implementation will always be preferred over a development version. A request for a new stable release has already been directed towards the liblo maintainer[12].

4.3.1 Non-Renderer

The SSR features a GUI, that was in the process of being upgraded for Qt5 at the time of writing. Future versions of the software could be used to also display setups of networking instances, instead of only displaying the ones, that are locally running.
The implementation could be desirable for massive multi-channel setups and simply switching between several (local or network-attached) SSR instances alike. An additional identifier for the **-N** flag (see 3.4.3.2) could be used to start an instance in this mode.

The functionalities of the SSR's GUI, its several spatial audio renderers and OSC interface (amongst other parts) are determined by its PubSub. For the GUI part of the software to display information about a networked setup, or even switch between several of them, it is therefore not needed or even desirable for that instance to render audio at all. An instance with such features could be imagined as a GUI only frontend.
Figure 9 illustrates a scenario, in which a server instance is used to control a set of n clients, that collectively renders audio (e.g. on a large scale WFS or NFC-HOA system). In contrast to the client instances, the server does not render any audio (i.e. has no outputs) and might not even need any audio input.
The server in this example could also be a client, subscribed to a server instance in a cluster similar to the one in Figure 4 (i.e. as the OSC capable application controlling the server instance by using a `MessageLevel` of `SERVER` or higher).

[11] https://ssr.readthedocs.io/en/latest/
[12] https://sourceforge.net/p/liblo/bugs/42/

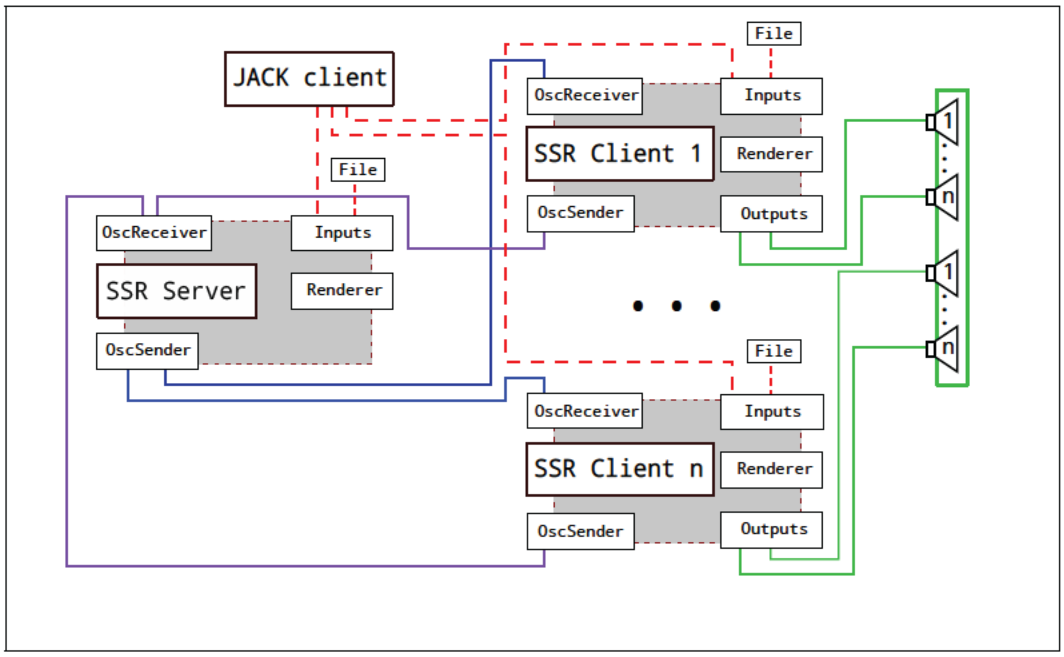

Fig. 9: A diagram displaying an SSR client/server setup, in which only the clients render audio collectively (e.g. medium or large-scale WFS). The server instance is not controlled via OSC, but controls its clients through it. Additionally, its rendering engine does not have any outputs.
– OSC input – OSC output – Audio input – Audio output

The example shown in Figure 10 is similar to the one before, with the difference, that its n clients render audio on separate systems.
Said client instances could be a cluster of headphone renderers, such as BRS renderers, or loudspeaker renderers, such as VBAP or WFS, or even a combination of both types.
Analogous to the example given before, the GUI only frontend could also act as a client with a `MessageLevel` of `SERVER` or higher (i.e. again being the OSC capable application), controlling the server in a setup, similar to the one shown in Figure 6.
For the aforementioned use cases to work, the GUI of the SSR has to be extended to show the networking specific information and be able to use the OSC interface through the PubSub interface (which itself would probably have to be extended as well). Additionally, a renderer should be conceived, that does not render audio, but still uses the PubSub, so the internal functionality of the SSR can be reused, leading to a relatively light-weight SSR GUI variant, only for controlling setups.
As there are many edge cases to networked setups, it seems still unclear, whether audio inputs would actually be needed for such a renderer.

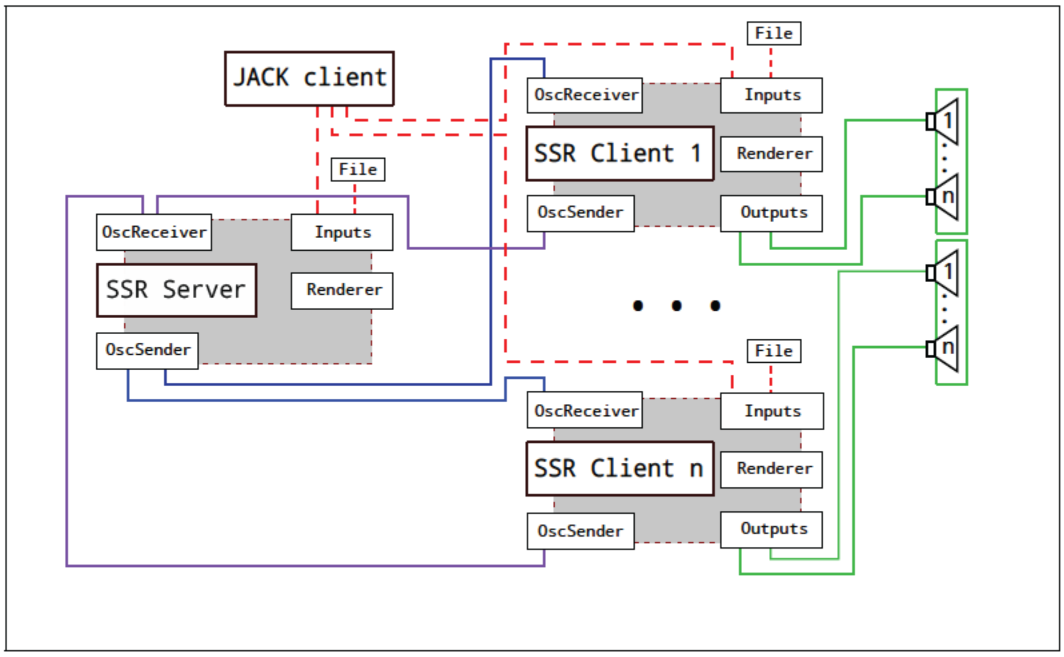

Fig. 10: A diagram displaying an SSR client/server setup, in which only the clients render audio to separate outputs (e.g. multiple BSs renderers). The server instance is not controlled via OSC, but controls its clients through it. Additionally, its rendering engine does not have any outputs.
– OSC input – OSC output – Audio input – Audio output

4.3.2 Alien Loudspeaker

For the examples given in Figure 3 , 9 , 7 and 4, which facilitate a set of n clients (server instances are counted as clients for the point made, where applicable), used for rendering in a medium or large scale loudspeaker based setup, an additional type of loudspeaker should be conceived and implemented in the different renderers.

```
enum model_t
{
  unknown = 0, ///< unknown loudspeaker type
  // TODO: find better names and better descriptions
  alien,  //< not belonging to this instance of ssr
  normal,      ///< normal loudspeaker
  subwoofer    ///< always on, regardless of source positions
};
```

Listing 11: src/loudspeaker.h: `enum model_t`

In the setups, where rendering takes place collectively, each client instance currently is reproducing the complete reproduction setup. This means, that for a reproduction setup with several hundred loudspeakers, each client creates exactly that amount of JACK outputs each, although it would only be responsible for discretely rendering audio on a subset of them.
To cope with this edge case, the current loudspeaker renderers would have to be extended to be able to distinguish between the loudspeakers of the current and that of other instances. This would enable them to create JACK outputs in the amount of the loudspeakers they are rendering audio for and reduce the overall processing usage.
In Listing 11 the `enum model_t`, defining the loudspeaker types available to the SSR, is extended to facilitate the new model type `alien`, which could be used internally by the renderers to identify loudspeakers, not to render on.

Already when defining a reproduction setup for the SSR, host-specific loudspeakers have to be taken into account. Listing 17 shows an attempt at providing a unique attribute for each part of the setup, that is referencing a loudspeaker — the hostname or IP address of the host — by extending the ASDF.
However, more work has to be put into implementing this feature, or rather improvement, as it also requires tests in medium and large scale setups, to ensure a discrete rendering, as if only using one host.

4.3.3 Status Messages

When reflecting about different use cases for networking setups involving the SSR, it became apparent, that in certain situations it would be desirable to be able to poll instances for information, involving sources, scenes and the like.
One example is the implementation of a light-weight, single-purpose GUI (e.g. non-interactive display of source positions) in another programming language, such as Python, Pd, or SuperCollider, while only relying on OSC for communication between the parts. Another example is the implementation of monitoring of certain aspects of a client or server instance (e.g. CPU usage). Both examples should allow a GUI (or any other monitoring) process to be subscribed, after the active SSR instances started rendering.
To be able to retrieve information from an SSR instance, its PubSub interface has to be extended and `get` functions implemented — where applicable — to return the desired information. A special case of this feature is described in 4.3.4.

4.3.4 Scene Transfer

The IP interface of the SSR implements a functionality to transfer all information related to a scene as an XML formatted string. This is useful, if the scene information should be stored on the machine requesting the information, instead of on the rendering machine.
Due to shortage of time to implement it and the original functionality heavily relying on the XML associated code, the OSC interface still lacks this feature, which in its context could also be used to transfer all scene information to another client, subscribed to a server.
It would prove particularly useful, if clients could for example request the scene currently held by their server instance, in the case where they are started after the server has been started, but its scene already being setup. The server would then send a set of instructions as OSC messages, needed for setting up the scene in question.
For this feature to work reliably, some edge cases have to be considered, such as gaps in the list of source IDs: Every source gets a unique non-zero, non-negative ID assigned on creation. When a source is deleted, its ID is not assigned to a source anymore and will not be reused, unless the whole scene is deleted and a new cycle of source creation reaches a number that high.
This means, if a scene with source ID gaps has to be transferred, the OSC messages have to be designed in such a way, that they can account for them, as every source message (apart from */source/new*) requires a valid source

ID and subsequent calls to the server would otherwise trigger incorrectly mapped operations on its clients sources.
Additionally, it would be useful to be able to transfer a scene to another client, by request, if the caller's `MessageLevel` is `SERVER` or higher.
The scene is a piece of redundant information, in a networked setup (whereas the reproduction setup is individual). Being able to transfer scenes, using OSC, would further improve usability, as the description files only have to be in one place.

4.3.5 Assigning In- and Outputs on the Fly

The SSR, being a JACK client, is able to add inputs for its sources and outputs for its renderers according to the configuration variables `INPUT_PREFIX` and `OUTPUT_PREFIX`, as shown in Listing 12.

```
########################## JACK settings
 ↪   ########################################

# alsa input port prefix
#INPUT_PREFIX = "alsa_pcm:capture_"

# alsa output port prefix
#OUTPUT_PREFIX = "alsa_pcm:playback_"
```

Listing 12: data/ssr.conf.example: JACK settings in the SSR configuration file

The approach however is somewhat static, as it only allows setting up one predefined client during startup. This can be the system's hardware in- and outputs or other JACK clients. The selected input name is added to the prefix to connect an SSR source to a JACK client output with that name. Following the configuration file example, given a source input name of `1`, the SSR would connect to the JACK client port named `alsa_pcm:capture_1`.
Dynamically reassigning source input or renderer output connections is only possible by using external tools, able to handle connections of a JACK session, such as QjackCtl (Capela, 2017), Patchage (Robillard, 2017) or aj-snapshot (Moors and Suominen, 2017).
Every JACK client is allowed to make connections to other clients on the same server on its own. This general feature should be harnessed in the case of the SSR to allow assigning and reassigning of source inputs and renderer outputs on the fly and exposing this functionality to the OSC interface (i.e. by modifying and extending the PubSub interface).

This would make the application more dynamic and allow easier scripting of scenes and reproduction setups alike. This becomes especially useful in the case of listening experiments, as the experiments will not have to rely on a mixture of different scripting languages anymore (e.g. see experimental setup in attachment of Grigoriev (2017)).

4.3.6 Interpolation of Moving Sources

Using the IP or OSC interface, it is possible to move sources in a scene to a new location. Unlike sWONDER (see 2.2), the SSR is not able to interpolate movement. When a new location for a source is requested, the movement is carried out instantly, whereas sWONDER is able to move a source (on a straight line) from one position to the next in a given time frame.
A series of movements can be requested in any desired time frame, which means, that spatial aliasing is very likely to occur, during sets of requests with a short time span between them.
To work around this, the SSR should implement a comparable feature to the one sWONDER facilitates and apply rate limiting on the source positioning request, depending on a dynamically settable threshold value in milliseconds. Applications, such as WFSCollider (see 2.5) or 3Dj (see 2.4) rely on their GUI or rather sclang to implement dynamic movements (e.g. circular or randomized) (Sauer and Snoei, 2017, pp. 56-62). While the creative process of source paths generation is clearly best placed in a visually interactive process, its communication with the rendering engine has to be high-performance and if scalable to large setups, ideally with low network throughput.
Therefore it has to be evaluated, if implementing a set of understood geometrical shapes, instead of sending a high frequency of source positioning messages could be a more feasible solution in the case of the SSR. In 4.3.7 a wider approach to this problem is discussed.

4.3.7 Dynamic Scene

When using the SSR in a more artistic context, such as musical scores, or in scientific experimental environments dealing with moving sources, this requires a dynamic scene. sWONDER (see 2.2) has a scoreplayer application, that can be synced with MIDI, which is used to record and play back scores (i.e. recorded source properties) from unvalidated XML files. WFSCollider has a fully integrated timeline, that can be used to place multiple (even concurrent) events, save and play them (Sauer and Snoei, 2017, p. 10).

Unfortunately the ASDF was never properly extended for these purposes

```xml
  <xs:element name="score">
    <xs:annotation><xs:documentation xml:lang="en">
        This holds the dynamic content of the scene, i.e. movement
         ↪  of
        sources/reference, start/end/length/loop information of
        soundfiles, ...
        Right now, it's not really defined and just to get an idea
         ↪  that
        such functionalities will be implemented sometime
         ↪  (hopefully).
    </xs:documentation></xs:annotation>
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element ref="event"/>
        <!-- There may be other elements here? -->
      </xs:choice>
    </xs:complexType>
  </xs:element>
```

Listing 13: data/asdf.xsd: Draft of the score element within the ASDF schema file

(see Listing 13), which is why the SSR is not able to deal with dynamic scenes in a comparable fashion yet.
Unlike sWONDER, the SSR uses schema validated XML only, whereas 3Dj and WFSCollider use a unique format or even SuperCollider code as score files.
Schema validated input is less error-prone and should generally be preferred over single-purpose or self-conceived formats. Therefore, it would be a good step to consolidate the ASDF schema part responsible for dynamic elements, while keeping in mind the overall message throughput as discussed in 4.3.6 and thus enable the SSR to deal with dynamic scenes efficiently.
Additionally, the GUI efforts made for WFSCollider could be combined with an SSR backend, as it lacks a tool for creation and controlling of dynamic content.

4.3.8 Network Enabled Head Tracking

Due to the higher availability of sensors, microcontrollers and embedded systems in recent years, it has become very affordable to build network enabled head tracking devices in small series. Many of the conceived devices, such as the GPL licensed *Hedrot* (Baskind, 2017), allow for OSC communication.
Using the OSC interface, such a head tracker can be added as a client to an SSR instance. This will probably require rate-limiting the sensor output,

but would enable a networked setup, that could prove to be cheaper, more reliable and flexible, than the compile-time opt-ins (i.e. VRPN, Polhemus Fastrak/ Patriot, InterSense InertiaCube3).
In the specific case of setting up a large array of independent BRS or BS renderers, connected to one server instance or application, it might be required to extend the messaging system to allow passing on of messages from one client to a server only towards one specific other client (a type of proxy messaging). This would ensure, that every renderer can be supplied with a specific stream of OSC messages from its assigned head tracker. Additionally, single (and local) renderers can be started as a server instance and clients can be assigned to them flexibly.

5 References

Ackermann, David and Maximilian Ilse (2015): *The Simulation of Monaural and Binaural Transfer Functions for a Ground Truth for Room Acoustical Analysis and Perception (GRAP).* Master's thesis, Technische Universität Berlin.

Audiokommunikation, Fachgebiet (2017): "Wellenfeldsynthese an der TU Berlin." URL `https://www.ak.tu-berlin.de/menue/forschung/wellenfeldsynthese/`.

Baalman, Marije A.J. (2007): *On wave field synthesis and electro-acoustic music, with a particular focus on the reproduction of arbitrarily shaped sound sources.* Ph.D. thesis, Technische Universität Berlin.

Baalman, Marije A.J.; Torben Hohn; Simon Schampijer; and Thilo Koch (2007): "Renewed architecture of the sWONDER software for Wave Field Synthesis on large scale systems." In: *Linux Audio Conference 2007.* Linux Audio Conference. URL `http://lac.linuxaudio.org/2007/papers/lac07_baalman_hohn_schampijer_koch.pdf`.

Baskind, Alexis (2017): "Hedrot." URL `https://abaskind.github.io/hedrot/`.

Böhm, Christoph (2015): *Entwicklung einer Versuchsumgebung zur Auralisation von virtuellen Konzerträumen für Musiker.* Master's thesis, Technische Universität Berlin.

Capela, Rui Nuno (2017): "QjackCtl." URL `https://qjackctl.sourceforge.io/`.

Cotton, M.; et al. (2011): "Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry." URL `https://tools.ietf.org/html/rfc6335`.

Cycling'74 (2017): "Max/MSP." URL `https://cycling74.com/`.

Davis, Paul (2016): "JACK Audio Connection Kit." URL `http://jackaudio.org/`.

Davis, Paul (2017): "Ardour." URL `https://ardour.org`.

Ellison, Lee (2017): "Audinate." URL `https://audinate.com`.

Fohl, Wolfgang (2013): *Sound - Perception - Performance*, chap. The Wave Field Synthesis Lab at the HAW Hamburg. Heidelberg: Springer International Publishing, pp. 243–255. doi:10.1007/978-3-319-00107-4_10. URL `http://dx.doi.org/10.1007/978-3-319-00107-4_10`.

Foundation, Game Of Life (2016): “Game Of Life Foundation.” URL `http://gameoflife.nl/en`.

Geier, Matthias; Jens Ahrens; and Sascha Spors (2008): “The SoundScape Renderer: A Unified Spatial Audio Reproduction Framework for Arbitrary Rendering Methods.” In: *Audio Engineering Society Convention 124*. Audio Engineering Society Inc. URL `http://www.aes.org/e-lib/browse.cfm?elib=14460`.

Geier, Matthias; Torben Hohn; and Sascha Spors (2012): “An Open-Source C++ Framework for Multithreaded Realtime Multichannel Audio Applications.” In: *Linux Audio Conference*. URL `http://lac.linuxaudio.org/2012/papers/19.pdf`.

Geier, Matthias and Sascha Spors (2010): “Conducting Psychoacoustic Experiments with the SoundScape Renderer.” In: *Sprachkommunikation 2010 - 9. ITG-Fachtagung 10/06/2010 - 10/08/2010 at Bochum, Deutschland*. VDE Verlag. URL `http://www.int.uni-rostock.de/fileadmin/user_upload/publications/spors/2010/Geier_ITGspeech2010_SSR_experiments.pdf`.

Geier, Matthias and Sascha Spors (2012): “Spatial Audio with the SoundScape Renderer.” In: *27th Tonmeistertagung - VDT International Convention*. URL `http://www.int.uni-rostock.de/fileadmin/user_upload/publications/spors/2012/Geier_TMT2012_SSR.pdf`.

Grigoriev, Dmitry (2017): *Synthesis of binaural stimuli for a listening test on room acoustic perception*. Master’s thesis, Technische Universität Berlin.

Guillot, Pierre; Eliott Paris; and Thomas Le Meur (2017a): “HoaLibrary.” URL `http://www.mshparisnord.fr/hoalibrary/en`.

Guillot, Pierre; Eliott Paris; and Thomas Le Meur (2017b): “HoaLibrary for PureData.” URL `https://github.com/cicm/hoalibrary-pd`.

Haller, Hans Peter (1995): *Das Experimentalstudio der Heinrich-Strobel-Stiftung des Südwestfunks Freiburg 1971-1989: Die Erforschung der Elektronischen Klangumformung und ihre Geschichte*, vol. 1 of *Südwestfunk-Schriftenreihe: Rundfunkgeschichte*. 1. Baden-Baden: Nomos Verlagsgesellschaft.

Koslowski, Konstantin (2013): *Presentation of Virtual Sources with an Increased Auditory Source Width in Wave Field Synthesis.* Bachelor's thesis, Technische Universität Berlin.

Lindau, Alexander (2014): *Binaural Resynthesis of Acoustic Environments. Technology and Perceptual Evaluation.* Ph.D. thesis, Technische Universität Berlin.

McCartney, James (2017): "SuperCollider." URL `https://supercollider.github.io/`.

Moors, Lieven and Jari Suominen (2017): "aj-snapshot." URL `http://aj-snapshot.sourceforge.net/`.

Pérez-López, Andrés (2014): *Real-Time 3D Audio Spatialization Tools for Interactive Performance.* Master's thesis, Universitat Pompeu Fabra, Barcelona.

Puckette, Miller (1997): "Pure Data: another integrated computer music environment." In: *Second Intercollege Computer Music Concerts, Tachikawa.* Kunitachi College of Music, pp. 37–41. URL `https://puredata.info/docs/articles/puredata1997`.

Puckette, Miller (2016): "PureData." URL `http://puredata.info`.

Pulkki, Ville (1997): "Virtual Sound Source Positioning Using Vector Base Amplitude Panning." In: *JAES*, vol. 45. Audio Engineering Society Inc., pp. 456–466. URL `http://www.aes.org/e-lib/browse.cfm?elib=7853`.

Quality & Usability Lab, TU Berlin, Telekom Innovation Laboratories; Universität Rostock Institut für Nachrichtentechnik; and Chalmers University of Technology Division of Applied Acoustics (2016): "SoundScape Renderer." URL `http://spatialaudio.net/ssr/`.

Robillard, David (2017): "Patchage." URL `https://drobilla.net/software/patchage`.

Sauer, Arthur and Wouter Snoei (2017): *Working with WFSCollider v2.2.4b.* The Game Of Life Foundation. URL `https://sourceforge.net/projects/wfscollider/files/WFSCollider%20Manual/Working%20with%20WFSCollider%20v2.2.4.pdf/download`.

Shuttleworth, Marc (2017): "Ubuntu Linux." URL `https://ubuntu.com`.

Sinclair, Stephen (2017): "liblo." URL `http://liblo.sourceforge.net/`.

Snoei, Wouter; Miguel Negrao; R. Ganchrow; and J. Truetzler (2016): “WFSCollider on Github.” URL `https://github.com/GameOfLife/WFSCollider`.

Spors, Sascha and Jens Ahrens (2010): “Analysis and Improvement of Pre-equalization in 2.5-Dimensional Wave Field Synthesis.” In: *Audio Engineering Society Convention 128*. Audio Engineering Society Inc. URL `http://www.aes.org/e-lib/browse.cfm?elib=15418`.

Spors, Sascha; Rudolf Rabenstein; and Jens Ahrens (2008): “The Theory of Wave Field Synthesis Revisited.” In: *Audio Engineering Society Convention 124*. Audio Engineering Society Inc. URL `http://www.aes.org/e-lib/browse.cfm?elib=14488`.

Thaden, Dr.-Ing. Rainer (2017): “Four Audio.” URL `http://fouraudio.com`.

Vinet, Judd and Aaron Griffin (2017): “Arch Linux.” URL `https://archlinux.org`.

von Coler, Henrik and Andreas Pysiewicz (2017): “Electronic Studio.” URL `https://www.ak.tu-berlin.de/studio`.

Wierstorf, Hagen (2014): *Perceptual Assessment of Sound Field Synthesis*. Ph.D. thesis, Technische Universität Berlin.

Wierstorf, Hagen; Alexander Raake; and Sascha Spors (2012): “Localization of a virtual point source within the listening area for Wave Field Synthesis.” In: *Audio Engineering Society Convention 133*. Audio Engineering Society Inc. URL `http://www.aes.org/e-lib/browse.cfm?elib=16485`.

Wittek, Helmut (2007): *Perceptual differences between wavefield synthesis and stereophony*. Ph.D. thesis, University of Surrey.

Wright, Matthew (2002): “Open Sound Control 1.0 Specification.” URL `http://opensoundcontrol.org/spec-1_0`.

Appendices

A PDF Version

The PDF version of this work can be found on the Digital Ressource as the file `master-thesis/thesis/thesis.pdf`.

B LaTeX Sources

The LaTeX sources for this work can be found on the Digital Ressource in the file `master-thesis/thesis/thesis.tex`. The accompanying BibTeX file is located in `master-thesis/bib`.
All graphics used in this work can be found in `master-thesis/images`.

C Thesis Bibliography

The references used in this work, if not in the form of a website, can be found on the Digital Ressource in the folder `master-thesis/src`.

D OSC Interface Source Code

All C++ source code written for the OSC interface can be found on the Digital Ressource in the folder `ssr/src/networking`. However, there are more parts of the original SSR source code, that have been extended and modified, such as `ssr/src/controller.h` or `ssr/src/configuration.cpp`.
It is possible to get a better overview of the various changes, by using git's log features, as shown in Listing 14.

```
cd ssr
git log
```

Listing 14: The git log feature used in the ssr folder of the Digital Ressource.

To evaluate the differences between the original code base and the modified version, it is recommended to use Github's diff functionality for the dedicated branches:

- configuration-client-server[13]
- networking-with-osc[14]
- osc-tests[15]
- reproduction-with-hostnames[16]
- sclang-workflows[17]
- alien-loudspeaker[18]

The source code developed in the aforementioned branches was merged into a new, local branch called *testing* for the Digital Ressource. However, all of them are available separately in this local source code repository.
Therefore, git can also be used locally to checkout a specific branch of the source code, as shown in Listing 15.

```
cd ssr
git checkout networking-with-osc
```

Listing 15: The git checkout feature used in the ssr folder of the Digital Ressource to checkout the *networking-with-osc* branch.

Comparison between branches can also be done locally, as described in Listing 16.

The examples in Listing 15 and 16 can be applied analogous to the other branches.

[13] https://github.com/SoundScapeRenderer/ssr/compare/master...dvzrv:configuration-client-server
[14] https://github.com/SoundScapeRenderer/ssr/compare/master...dvzrv:networking-with-osc
[15] https://github.com/SoundScapeRenderer/ssr/compare/master...dvzrv:osc-tests
[16] https://github.com/SoundScapeRenderer/ssr/compare/master...dvzrv:reproduction-with-hostnames
[17] https://github.com/SoundScapeRenderer/ssr/compare/master...dvzrv:sclang-workflows
[18] https://github.com/SoundScapeRenderer/ssr/compare/master...dvzrv:alien-loudspeaker

```
    cd ssr
    git diff master...networking-with-osc
```

Listing 16: The git diff feature used in the ssr folder of the Digital Ressource to display the difference between the *networking-with-osc* and the *master* branch.

E SuperCollider Scripts

The SuperCollider code written for the tests (see 4.2) and workflows (see 3.4.7) are located on the Digital Ressource in the folder `ssr/supercollider`.
For using the scripts, SuperCollider version 3.7, or above is recommended.

F Reproduction Setup Changes

```
  <xs:element name="reproduction_setup">
    <xs:annotation><xs:documentation xml:lang="en">
        This section is for the setup of the reproduction system. This can
        be a loudspeaker setup or headphones or ...
    </xs:documentation></xs:annotation>
    <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element ref="loudspeaker"/>
        <xs:element ref="circular_array"/>
        <xs:element ref="linear_array"/>
        <xs:element name="skip">
          <xs:complexType>
            <xs:attribute name="number" type="xs:positiveInteger" default="1"/>
            <xs:attribute ref="hostname" use="optional"/>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
  </xs:element>

  <xs:element name="loudspeaker">
    <xs:complexType>
      <xs:all>
        <xs:element ref="position"/>    <!-- required -->
        <xs:element ref="orientation"/> <!-- required -->
      </xs:all>
      <xs:attribute ref="name"/>
      <xs:attribute ref="hostname"/>
      <xs:attribute ref="delay"/>
      <xs:attribute ref="weight"/>
      <xs:attribute name="model" type="loudspeaker_model_t"/>
      <!-- extensible by any attribute -->
      <xs:anyAttribute processContents="skip"/>
    </xs:complexType>
  </xs:element>

      <!-- an ID doesn't really make sense for a loudspeaker array, does it? -->
      <!-- <xs:attribute ref="id" use="optional"/> -->
      <xs:attribute name="number" type="more_than_one" use="required"/>
      <xs:attribute ref="name" use="optional"/>
      <xs:attribute ref="hostname"/>

      <xs:attribute name="number" type="more_than_one" use="required"/>
      <xs:attribute ref="name"/>
      <xs:attribute ref="hostname"/>
      <!-- extensible by any attribute -->

  <xs:attribute name="hostname" type="xs:token"/>
```

Listing 17: data/asdf.xsd: Reproduction setup, loudspeaker, circular array and linear array definition in ASDF, extended by a *hostname* attribute

Glossary

ASCII	American Standard Code for Information Interchange — a character encoding standard.
FAUST	Functional Audio Stream is a functional programming language specifically designed for realtime signal processing and synthesis.
ID	A name or number, that identifies an object.
Python	A multi-purpose, object-oriented programming language.
Qt4	Version 4 (legacy) of the cross-platform application framework for creating desktop applications.
Qt5	Version 5 of the cross-platform application framework for creating desktop applications.
Quark	Name for Classes extending the SuperCollider programming language, usually developed in a separate version controlled code repository.
sclang	Name of the SuperCollider programming language and the interpreter executable of the SuperCollider programming language.
stdout	The standard output is a stream where a program writes its output data to. This can be a log file or a terminal.
SuperCollider	A programming language, Integrated Development Environment (IDE) and synthesis server for realtime audio processing and synthesis.

Acronyms

AAP	Ambisonics Amplitude Panning.
ADAT	Alesis Digital Audio Tape.
ALSA	Advanced Linux Sound Architecture.
APF	Audio Processing Framework.
API	Application Programming Interface.

ASDF	Audio Scene Description Format.
BRIR	Binaural Room Impulse Response.
BRS	Binaural Room Synthesis.
BS	Binaural Synthesis.
CC	Creative Commons.
CICM	Centre de recherche Informatique et Création Musicale.
CNMAT	Center for New Music and Audio Technologies.
CPU	Central Processing Unit.
FDL	GNU Free Documentation License.
GPL	GNU General Public License.
GUI	Graphical User Interface.
HOA	Higher Order Ambisonics.
HRIR	Head Related Impulse Response.
HRTF	Head Related Transfer Function.
IANA	Internet Assigned Numbers Authority.
IP	Internet Protocol.
JACK	JACK Audio Connection Kit.
LGPL	GNU Lesser General Public License.
LTS	Long Term Support.
MADI	Multichannel Audio Digital Interface.
MIDI	Musical Instrument Digital Interface.
NFC-HOA	Near-Field-Compensated Higher Order Ambisonics.

OOP	Object-Oriented Programming.
OS	Operating System.
OSC	Open Sound Control.
Pd	PureData.
POSIX	Portable Operating System Interface.
PubSub	Publish-Subscribe message pattern.
RAII	Ressource Acquisition Is Initialization.
SSR	SoundScape Renderer.
TCP	Transmission Control Protocol.
TU Berlin	Technische Universität Berlin.
UDP	User Datagram Protocol.
VBAP	Vector Based Amplitude Panning.
WFS	Wave Field Synthesis.
XML	Extensible Markup Language.

List of Figures

List of Listings

List of Tables

Digital Ressource

This page holds a data disk.