USE CASE: INTEGRATION OF A FAUST SIGNAL PROCESSING APPLICATION IN A LIVESTREAM WEBSERVICE

Christoph Kuhr
Anhalt University of Applied Sciences
Köthen, Germany
christoph.kuhr@hs-anhalt.de

Thomas Hofmann
Anhalt University of Applied Sciences
Köthen, Germany
thomas.hofmann@hs-anhalt.de

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

ABSTRACT

In this paper we present the deployment of a signal processing application that was developed with the FAUST programming language. The Soundjack realtime communication application is extended by a server cloud to handle up to 60 musicians of an orchestra. Each musician is connected to a Soundjack UDP client. An individual stereo mix of the multiple audio streams originating from the multiple Soundjack clients has to be provided to each listening musician. The bandwidth required to receive 60 parallel audio streams may exceed the bandwidth available on the client side. Also it is neither necessary nor efficient to transmit 60 x 60 streams from the server to the clients. Therefore the individual mixes are computed on the server side and then distributed to the clients. Multiple signal processing servers are deployed in the server cloud and handle audio streams with the JACK sound server. We developed an audio mixing application to mix the audio data that is transmitted by the UDP streams. The application is integrated in the community web page, which is part of the Soundjack server cloud. The mixing application was successfully deployed and integrated within the community web service and the JACK audio environment.

1. INTRODUCTION

1.1. Soundjack and fast-music

Soundjack [1] is a realtime communication software that establishes up to five peer to peer connections. This software was designed from a musical point of view and first published in 2009 [2]. Playing live music via the public internet is very sensitive in terms of latencies. Thus, the main goal of this application is the minimization of latencies and jitter. An inherent property of such a peer to peer network is that each signal source has to concurrently transmit its own stream to each client (figure 1).

The goal of the research project fast-music, in cooperation with the two companies GENUIN [3] and Symonics [4], is to develop a rehearsal environment for conducted orchestras via the public internet. 60 musicians and one conductor shall play together live while being distributed inside Germany with the topology shown in figure 2.

Because a peer to peer network topology does not scale well along a growing number of nodes, a star network topology is chosen to allow live communication for 60 participating musicians, as shown in figure 1. Further field of research is the transmission of low delay live video streams and motion capturing of the conductor.

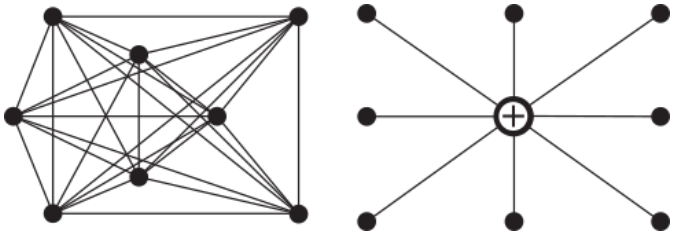

Figure 1: *Left: Peer to Peer Network Topology, Right: Star Network Topology*

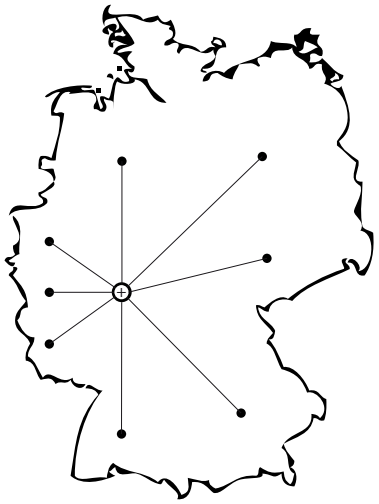

Figure 2: *Logical Center for Centralized Server-based Network Music Performances in Frankfurt on the Main, Germany*

1.2. Motivation

Inside the Soundjack cloud, digital signal processing is applied to audio and video streams. In most of the cases such signal processing is computationally expensive, which means that unwanted latencies may emerge, thus a graphics card based solution will be investigated. Preliminary work, which investigates the feasibility of the graphics card as an audio co-processor is presented in [5]. Required digital signal processing contains for example audio error concealment due to UDP packetloss. Our error concealment approach deals with the UDP packetloss by transforming the audio stream into the Wavelet [6] domain and decode it with the Viterbi algorithm [7], to predict the most likly sequence of audio samples from the Wavelet coeffitients. The Viterbi decoder relies on a Hid-

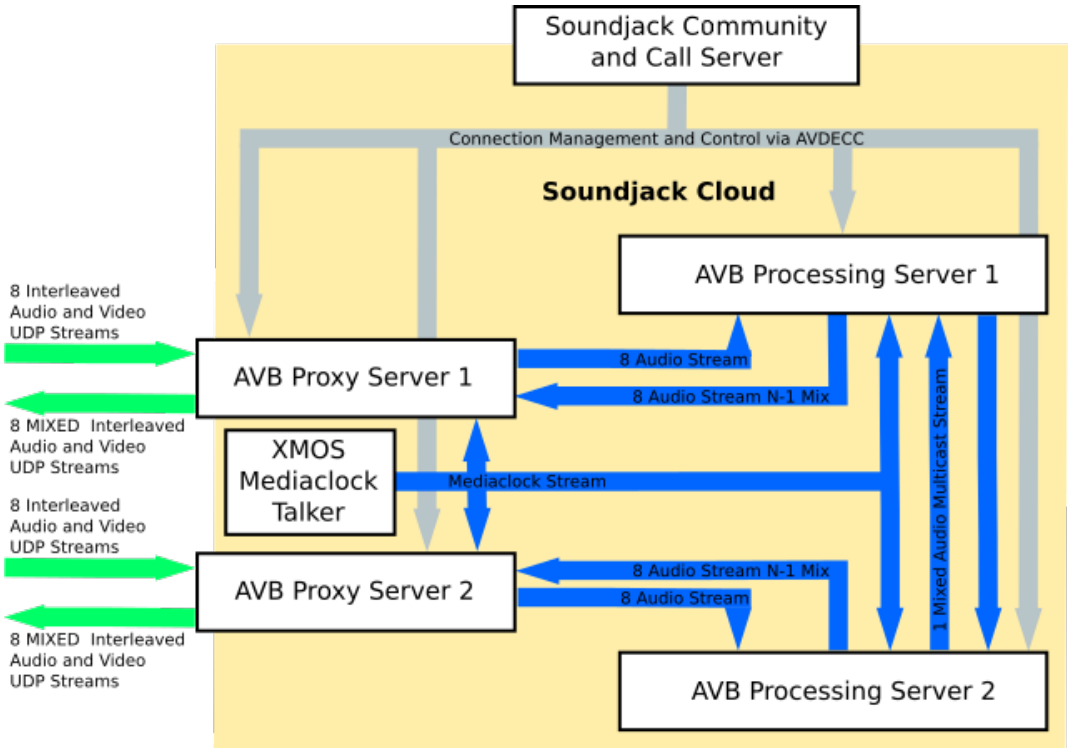

Figure 3: *Soundjack Realtime Processing Cloud*

den Markov Model (HMM) [8] that is trained by a recurrent neural network, that has yet to be defined.

The training phase in this machine learning approach is running offline with Wavelet coeffitients that are generated from audio samples that result from synthesized MIDI notes.

During the decoding (online) phase, realtime audio data is fed into the graphics card that is running the Viterbi decoder.

This is work in progress and it is yet to be evaluated, whether the Viterbi decoder and the Wavelet transforms can be implemented with FAUST.

Another signal processing application shall be a virtual room simulation, also processed on the graphics card [9], that places the Soundjack client's sound sources, i.e. the user's musical instruments, in a virtual soundscape. This virtual soundscape shall be rendered for each user and shall be implemented using finite impulse response (FIR) filters [10] and apply some binaural [11] encoding that optimizes the immersion of the virtual soundscape for headphones. The strategy to implement such a signal processing application is based on a graphics card based solution, since a heavy CPU utilization is expected.

Audio Video Bridging / Time-Sensitive Networking (AVB / TSN) is a technology which handles real time constrained audio and video streaming in computer networks. This technology is a set of IEEE 802.1 industry standards, which operate on OSI-Layer 2 [12] and is used as the underlying infrastructure for the Soundjack realtime processing cloud. The two AVB server types required for the Soundjack cloud are an AVB proxy server and an AVB processing server, which are described in detail in [13]. The AVB proxy and processing servers are each connected to the same AVB LAN segment. Additionally, each server is also connected to a non-AVB LAN Segment, as well as to the Soundjack-Session and Management servers. Since AVB connection management and control traffic is not necessarily time-sensitive, it is sufficient to use a non-AVB LAN segment for command and control purposes. The Soundjack session server provides the community services of Soundjack and also handles the connection management of public internet streams, peer to peer connections as well as client-server connections. To distribute the computational complexity of connecting 60 streams to each other, mix them and apply other signal processing algorithms, a scalable solution for a realtime processing cloud is required, figure 3 shows the network topology approach of the Soundjack realtime processing cloud.

In this paper we present the development, the evaluation and the integration of a FAUST [14] application, that had previously been developed in the C programming language. The C application did not reach a testable state until today. Thus, a different and presumably faster development approach is tested. The derived binaries achieve comparable or even better performance as programs by seasoned C++ programmers [15].

2. WEB APPLICATION INTEGRATION

The Soundjack user interface is provided as a web page. It allows to control the parameters used by the Soundjack streaming client, such as used audio interfaces, codec and bitrate, sizes of sample and network buffers, or level of the user's own signal (figure 4). In order to achieve an overall balanced mix the newly developed server side mixer is employed. The corresponding user interface is also a HTML user interface exposed via HTTP (figure 6). The

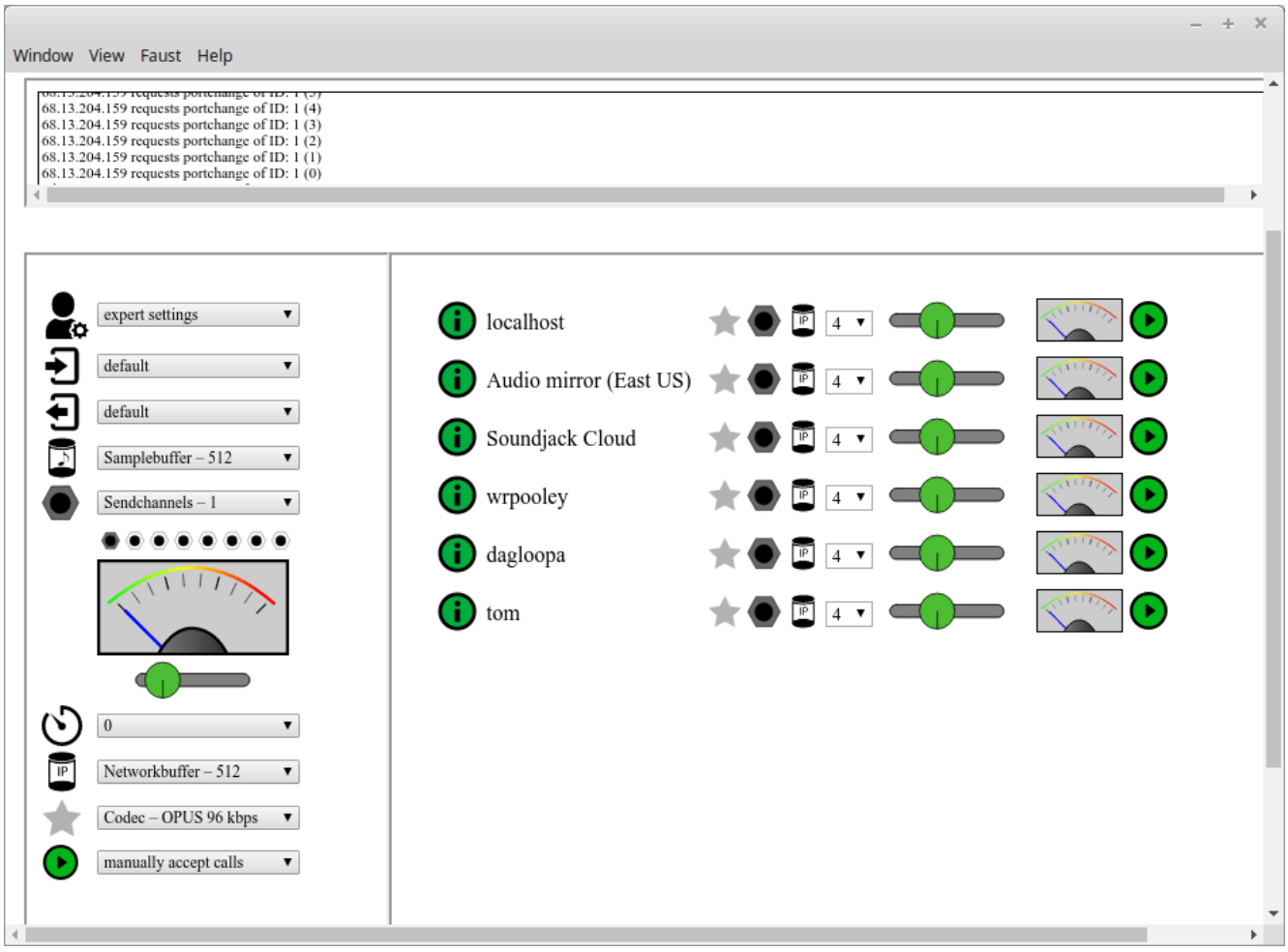

Figure 4: *Soundjack User Interface*

current integration into the existing user interface is achieved by an additional menu entry.

The Soundjack community and call server uses the Docker virtualization technology [16] for a webserver (Apache), database servers (MySql, Redis, Memcached), servers for traversal of NAT middleboxes (called nat1 and nat2), and one session service to handle individual Soundjack sessions. Each such service is running on a Linux kernel. The webserver offers a content management system (CMS) and delivers the user interface for configuring the multimedia client. For serverside scripting PHP is used, the delivered content consists of HTML, CSS, and Javascript. The used CMS is Joomla which itself uses MySql as its main data backend, additionally Redis as triple-store, and Memcached as in-memory database. The user database of Joomla resides in the MySql database and is also used by the session service. The services nat1, nat2, and session have been developed in the C programming language by the fast-music research group.

In order to facilitate deployment to the different environments for development, testing and production, a high degree of automation has been found useful. To achieve this each service is containerized using docker as container engine. A docker image contains the static dependencies of a service. A container is a running copy of an image, which then provides the actual service.

In order to create an image it is possible to start from scratch or build upon existing images. A source for images is a so called docker-registry, e. g. Docker Hub [17]. Image definitions are stored in a Dockerfile that describes how the image is built. The tool docker-compose [18] is used for the management of the infrastructure. Docker-compose introduces the concept of a service. The service definitions are stored in a docker-compose.yml file.

As Dockerfiles define images, docker-compose.yml files define the derived services. This nicely separates the build from the run time configuration. For running a service mainly the configuration for the network environment, service dependencies, and possibly volumes is needed. Also, for each service it references the underlying image and optionally its build configuration. The build configuration then defines the static content of an image.

The build of the images that provide C-programs as services uses a multistage build approach. The build stages use a compile container, which is based on the official gcc-container from the Docker Hub and customized to meet each program's build dependencies. After compilation, the resulting executable is copied into the image on which the service container is based. Original images found on Docker Hub serve as base image for the Redis and

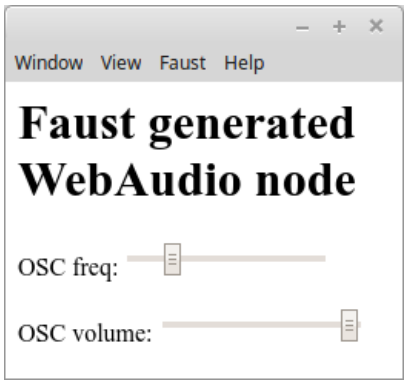

Figure 5: *Integration of a* FAUST *compiled WebAssembly into Electron*

Memcached services, in contrast to the Joomla and MySql services images that required some customization, e.g. no definition of volumes in the Dockerfile to keep own content. Instead we found it more robust to define volumes solely in the docker-compose.yml and deem them a runtime rather than build time configuration.

For the newly developed mixer a variation of the build process is currently implemented in a two step approach. First a container for the compilation of the mixing application is defined and then used for the compilation. The resulting binary is copied to the runtime environment, which could either be inside or outside of a container. A runtime container has been implemented and used for the interaction with the web interface of the developed mixing application. The compiler and runtime containers have been found useful for development purposes as they explicitly capture the requirements for compiling and running the mixer application. This facilitates repeated and consistent set up. The sizes of the resulting experimental images are currently 2 GB for the compile container and about 800 MB for the runtime one. They are not yet optimized for size.

An Electron application has been developed for Soundjack. That application bundles the native Soundjack Client into a browser with an integrated scripting environment (Electron [19] and node.js [20]). In order to extend the range of available audio processing capabilities on the client side, the integration of a FAUST application into that Electron application has been investigated. The building process for the integration of native binaries is automated, the distribution (compiling, upload of binaries, download and repackaging into the Electron Application) of the binaries however, is still cumbersome. Promising in regards of the maintainance process seemed the integration of a WebAssembly.

An integration of a WebAssembly as compiled by FAUST into an application bundle has been investigated. That application bundles the Soundjack Client into a browser with an integrated scripting environment (Electron [19], node.js [20]). For this purpose an Electron application has been developed in Typescript [21]. In order to circumvent cross origin resource sharing restrictions, the WebAssembly code is served from a web server internal to the developed Electron application (see the Typescript code in the listing below). This provides the possibility to also use FAUST programs on the client side for all platforms where the Electron application is available. Figure 5 shows the OSC example running inside the Electron application.

```
var promise = new Promise((resolve:any,
    reject:any) => {
  var webserver = Express();
  webserver.use(Express.static('
      webserver_root'));
  webserver.listen(8888);
});
promise.then((res:any) => {
  console.log('internal webserver started')
      ;
});
```

3. JACK AUDIO INTEGRATION

The mixing application is compiled with `faust2jackconsole` with the `-httpd` option set. The existing user interface is offered as a web user interface therefore the HTTP interface of the FAUST mixing application integrates smoothly. Obviously on the headless server no user interface is required. On server start up, the mixing application is forked by the main process that also starts the JACK [22] server. The server software also forks two processes for the reception and transmission of the audio streams. These two processes create the JACK clients "AVB_Talker" and "AVB_Listener". The incoming audio stream is originating from the "AVB_Listener" and for each stereo audio stream two audio channels are created. This also applies to the "AVB_Talker", two channels per audio stream. All audio channels of the "AVB_Listener" client are connected to the corresponding input channels of the mixing application with the JACK client name "soundjackMulticastMixer", the mixed stereo output channels of the "soundjackMulticastMixer" are connected to audio channels of the "AVB_Talker" client. Thus, a mixed stereo signal can be distributed to all streams that return to the Soundjack clients.

4. AUDIO MIXING APPLICATION

Up to 60 musicians shall participate in a Soundjack server session. A single proxy server is able to handle up to 8 Soundjack client streams. Thus, eight pairs of proxy and processing servers are required to provide enough resources for such a session. In a conventional mixing environment, all 60 audio streams would be connected to a single mixing stage, which would apply proper signal mixing and routing. The Soundjack cloud on the other hand, uses multicast streaming technologies of the underlying Ethernet network, which is not subject of this paper. One processing server receives eight stereo streams from its paired proxy server, mixes them and transmits the mixed stereo stream via multicast to the seven other processing servers. The volumes of each incoming stream in the mix is controlled by the master user (e.g. the producer or conductor). The second mixing stage gives control to the eight client users. Any client user is allowed to control the volume of their own send stream, mixed to his or her own return stream. Any remaining stream volume remains the same as after the first mixing stage. Thus, the Soundjack client users can have a so called "N-1" mix, i.e. a mix of N streams minus the users own stream. The audio signal originating from this client would suffer from round trip latency, which would be an obstacle for playing time aligned to the other musicians. Finally, each of the "N-1" mixes is mixed with the incoming multicast stereo streams of the other processing servers, i.e. the other Soundjack client streams. Two

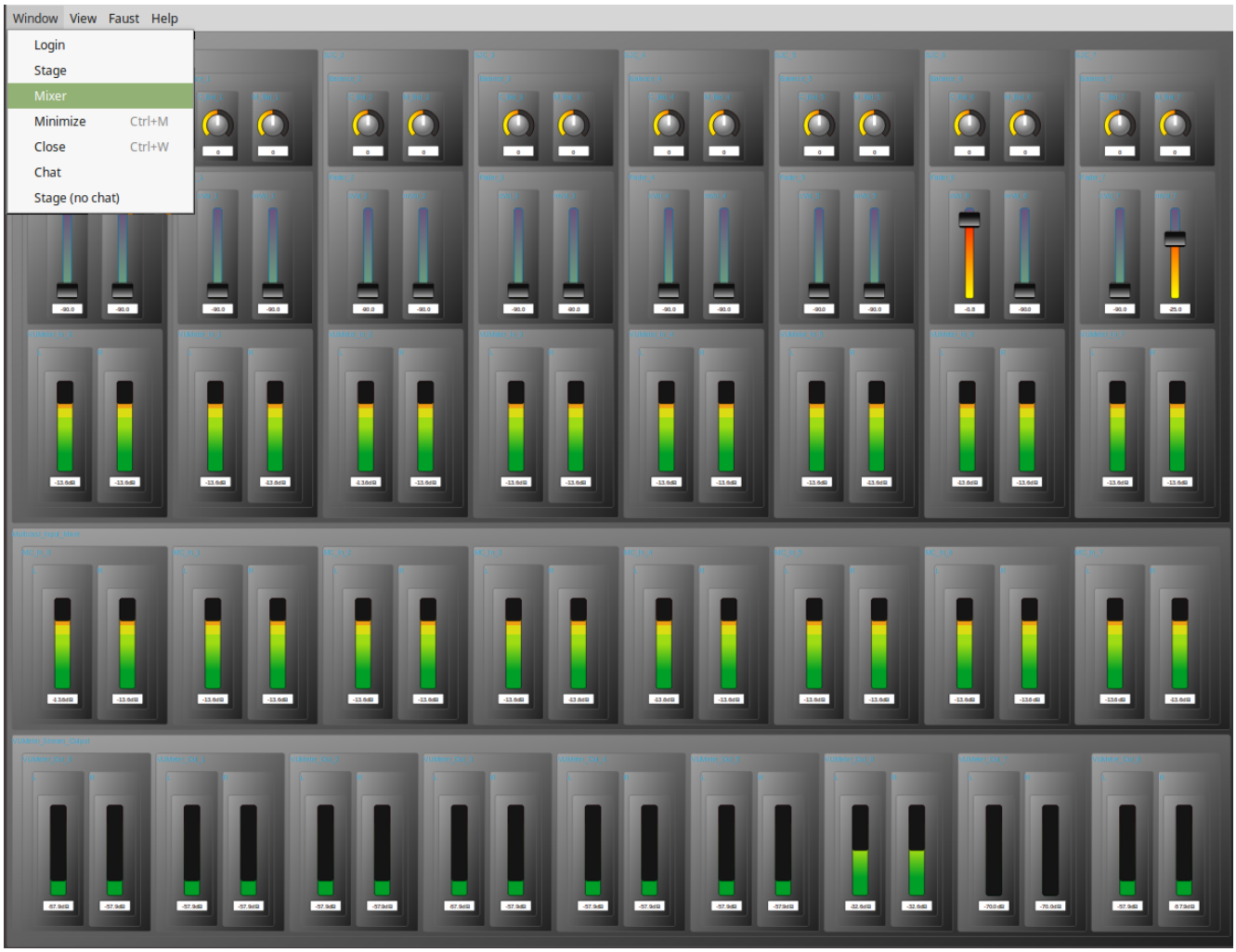

Figure 6: *Integration of the HTTP Interface of the Mixing Application into Electron*

different types of audio inputs and audio outputs are required to map these requirements:

- Inputs: Soundjack client stream, multicast stream
- Outpus: Soundjack client "N-1"-mix stream, mutlicast mix stream

FAUST provides simple expression syntax to build this complex routing structure. When we execute the application with the process identifier, the input multicast streams are immediately mixed down to a stereo channel, which then has to be connected to the "signalRouter" alongside the eight Soundjack client input streams. The resulting block diagram is shown in figure 7.

```
process =
        vgroup( "[3]",
          mcastInMix(N1) ,
          par(in, N2, (_,_))
          : signalRouter(N2,M)
         );
```

Consider the following values for the variables in the code listing above:

- $N1 = 7$ and represents the number of incoming multicast streams,
- $N2 = 8$ and represents the number of incoming Soundjack client streams,
- $M = 8$ and represents the number of outcoming Soundjack client streams.

For a better understanding of the code, the "mcStereoStream Output" identifier represents the multicast mix stream, as it is passed through the blocks. The input audio level for each of the incoming multicast streams is displayed with VU meters, eight of which are created with the "par" statement and mixed to mcStereoStreamOutput" with the ":>" operator.

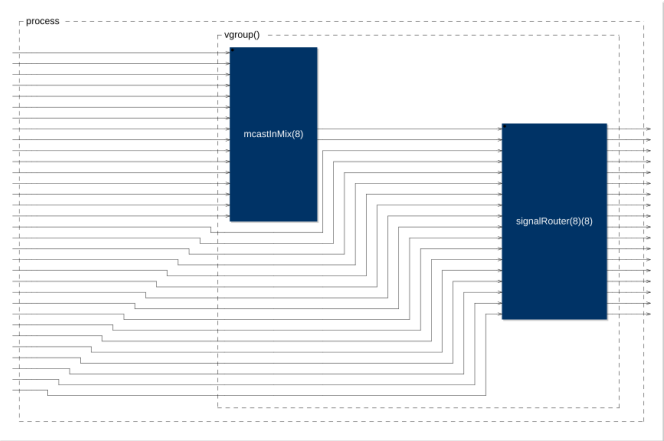

Figure 7: *Process Block Diagram*

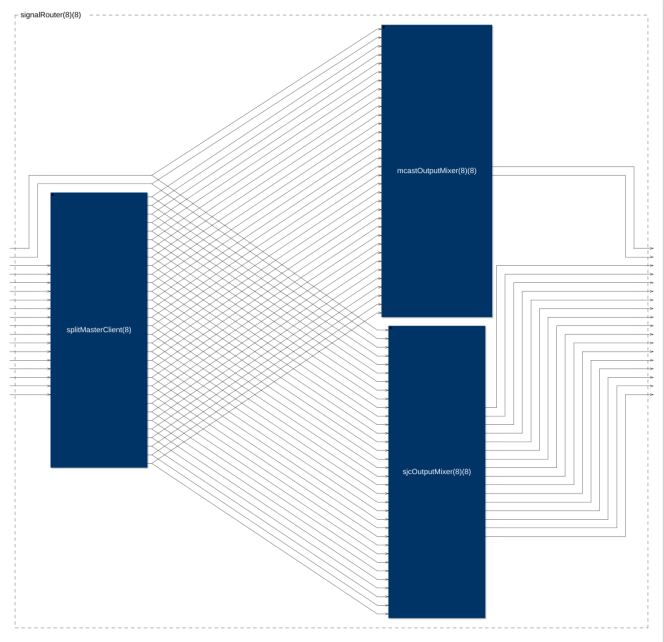

Figure 8: *"signalRouter" Block Diagram*

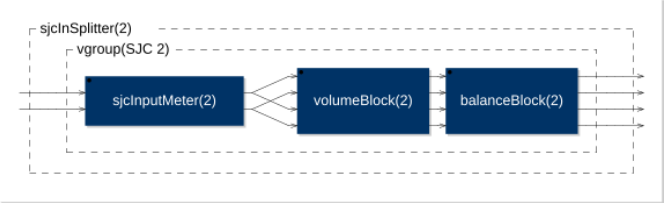

Figure 9: *"sjcInSplitter" Block Diagram*

```
mcStereoStreamOutput(l,r) = l,r;
mcInput(in) =
        hgroup("MCast In %in",
          vgroup("L", vumeter),
          vgroup("R", vumeter)
        );
mcastInMix(N1)  =
        hgroup("Mcast Input Mixer",
          par(in, N1, mcInput(in))
          :> mcStereoStreamOutput(_,_)
        );
```

Thus, the "signalRouter" has nine stereo input channels, as shown in figure 8, eight of which are connected to the "splitMasterClient" block in parallel to "mcStereoStreamOutput", which is presented in figure 10. All of the nine channels are split and distributed to "mcastOutputMixer(N2,M)" and "sjcOutputMixer(N2,M)".

```
signalRouter(N2,M) =
        mcStereoStreamOutput(_,_),
        splitMasterClient(N2)
     <: mcastOutputMixer(N2,M),
        sjcOutputMixer(N2,M);
```

The two mixing stages for master and client control are represented by two independent volume paths in "sjcInSplitter(in)". A stereo input channel is split into these two volume paths and both volume and stereo panning are applied to each path independently. Thus, the "sjcInSplitter(in)" block has two stereo output channels, as shown in figure 9. Eight parallel "sjcInSplitter" blocks are generated, resulting in 16 stereo output channels.

```
sjcInSplitter(in) =
    vgroup("SJC %in",
           sjcInputMeter(in)
        <: volumeBlock(in) : balanceBlock(
            in)
    );
```

```
splitMasterClient(N2) =
        hgroup("Inputs",
          par(in, N2, sjcInSplitter(in))
        );
```

The master volume is applied to the incoming multicast stereo channels, while the client volume is blocked by "(_,_,!,!)". The remaining eight stereo channels are mixed down to a stereo mix, which represents the first mixing stage, since the output is designated for the multicast output stream to the other processing servers.

```
mcastOutputMixer(N2,M) =
          mcStereoStreamOutput(!,!),
          par(in, N2, (_,_,!,!) )
       :> sjcOutputMeter(M) ;
```

With the use of a pattern matching function "selectChannel(out, in)", the correct channel is selected. If the "out" and "in" argument

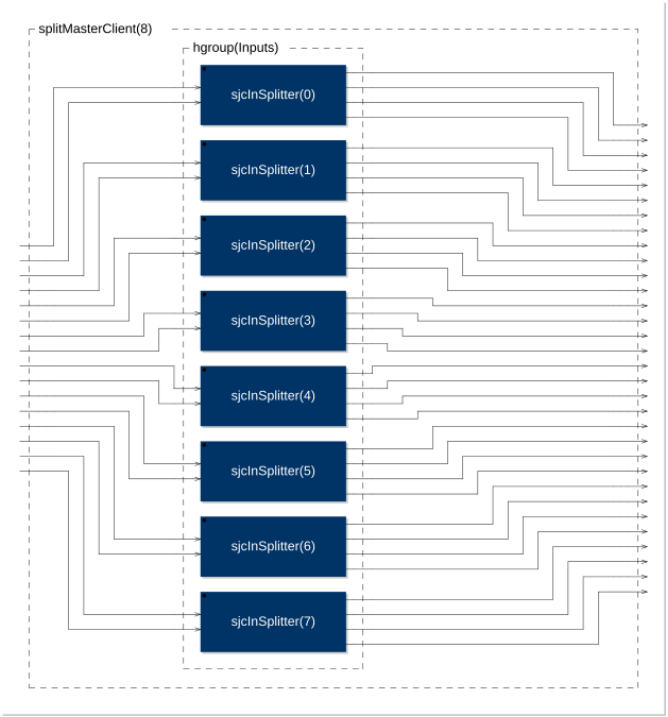

Figure 10: *"splitMasterClient" Block Diagram*

of the function call are equal, the client volume is applied and the master volume is applied in any other case. This represents the crucial part of the second mixing stage. There are other syntactical solutions to the same problem, but this code is much better suited to clarify the problem at hand.

```
selectChannel = case{
        (0,0)       => (!,!,_,_);
        (1,1)       => (!,!,_,_);
        (2,2)       => (!,!,_,_);
        (3,3)       => (!,!,_,_);
        (4,4)       => (!,!,_,_);
        (5,5)       => (!,!,_,_);
        (6,6)       => (!,!,_,_);
        (7,7)       => (!,!,_,_);
        (out,in)    => (_,_,!,!);
    };
```

The actual mixing stage "sjcOutputMixer(N2,M)", which is presented in figure 11, takes the output of the multicast input mixer ("mcStereoStreamOutput") alongside the master and client volumes of all Soundjack client input stereo channels, splits and distributes them. Only this time it is split and distributed over the eight Soundjack client outputs in the form of a "N2+1 x M" matrix. The pattern matching function is now called by each of the split Soundjack client stereo input channel generation.

```
sjcOutputMixer(N2,M) =
    mcStereoStreamOutput(_,_),
    par(in, N2, (_,_,_,_))
```

```
<: par(out, M,
       (mcStereoStreamOutput(_,_),
       par(in, N2, (selectChannel(out, in))
           )
   :> sjcOutputMeter(out))
   );
```

The HTTP user interface of the successfully compiled FAUST mixing application is shown in 6. In this figure, the client volume of stream number 7 is set, the resulting level can be seen in the third right-most VU meter pair of the last row. The second right-most VU meter pair shows no level at all, which corresponds to client level set to zero for client stream 8. But the master volume of client stream number 8 is set, thus it is mixed - with a very low level - to all other channels except itself.

5. EVALUATION

The presented prototype was merely an evaluation of the workflow and development complexity of a FAUST application. The multicast mixing application has also been developed in the C programming language, but without being integrated nor tested until now. The reason is the complexity of the debugging process in a multiprocessing and multithreading application. Developing the mixing application from scratch under different design rules was very fast. It took less than a week to implement a deployable audio signal processing application with a JACK audio interface, an HTTP user interface and a non trivial signal processing application. Since the JACK audio interface was compiled on the fly without any customization, each debugging iteration could be verified by measuring the proper audio routing with common audio measurement tools. Faulty code could be debugged either by interpreting the error messages of the FAUST compiler, while design flaws could be tracked via the automatically generated ".svg" files. No memory management or handling of race condition needed to be taken care of. In contrast, the development and debugging of a C application requires preliminary memory management and proper synchronization between threads and processes. A user interface, that could be accessed via HTTP was not implemented in C yet.

The developed container environment greatly facilitated the repeated and consistent compilation and execution of the mixing application developed with FAUST. The setup of the two configurations took initially roughly three days of work which is now reduced to one cloning the repository and starting the services. On first start the necessary images are built. This initial build in the given environment (CPU, network) took roughly an hour of non interactive download, install, and compilation. Subsequent starts of the services do not suffer from that delay.

6. CONCLUSIONS

FAUST being a domain specific language for audio signal processing, allows to program inside that domain in a compact and convenient way. The clarity of the code for the presented mixing application underlines the fast development process and drove us to the conclusion to develop the mixing application with FAUST, rather than implementing our own from scratch.

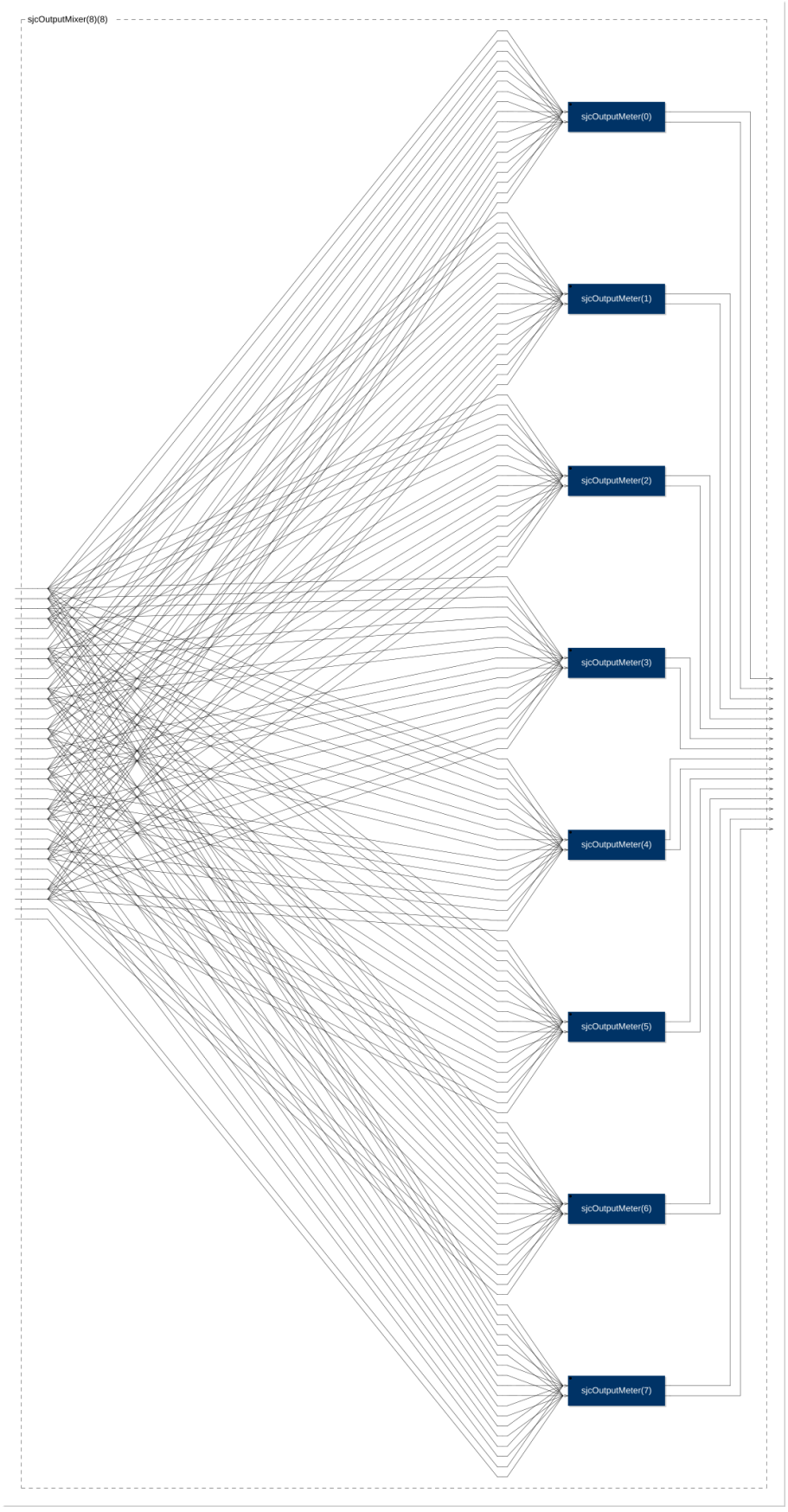

Figure 11: *"sjcOutputMixer" Block Diagram*

7. FUTURE WORK

One issue to be addressed in the future is the permission level of the mixing application, to give different volume control permissions to the conductor or producer and the musician. Thus, the conductor or producer will have volume control over the mix for the return streams as a whole and musicians will be able to control the volume of their own instrument in their own return stream, to generate an "N-1" mix. These permission will be enforced by the utilization of the JSON HTTP interface. The possibilities that arise from the "WebAssembly" compilation target seem promising and open the stage for further experiments.

Since the integration of a WebAssembly as compiled with FAUST into an Electron application is possible, it has to be evaluated whether the WebAssembly approach fullfills Soundjack's latency requirements. The possibilities already arising from the WebAssembly compilation target seem promising and open the stage for further experiments.

While the container for running the mixing application has been found useful for development purposes against its web interface, the actual mixing functionality has not been tested inside that container. This offers opportunities for further studies regarding audio processing and possible realtime requirements. From there also the size of the resulting runtime container may get optimized.

Furthermore, virtual room simulations developed with FAUST shall be evaluated. For this purpose the acceleration of those complex calculations by graphics cards shall be investigated in the future. The development of complex signal processing applications could be handed over to the Soundjack community, such that a producer or conductor would be able to load their own room acoustics application into the processing servers of the Soundjack cloud, without the necessity of an administrative integration.

8. ACKNOWLEDGMENTS

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

9. REFERENCES

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

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

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

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

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

[6] G. Luo, "Ultra low delay wavelet audio coding with low complexity for real time wireless transmission." Faculty of Technology, Buckinghamshire Chilterns University College, Queen Alexandra Road, High Wycombe, Buckinghamshire, UK, 2005.

[7] Z. Y. Zhihui Du and D. A. Bader, "A tile-based parallel viterbi algorithm for biological sequence alignment on gpu with cuda," in *IEEE 2010*. IEEE, 2010.

[8] L. R. Rabiner, "A tutorial on hidden markov models and selected applications in speech recognition," in *Proceedings of the IEEE, Vol. 77, No. 2*, Feb. 1989.

[9] M. Jedrzejewski and K. Marasek, "Computation of room acoustics using programmable video hardware," in *Computer Vision and Graphics, Springer-Verlag Netherlands*. PJWSTK Poland, 2006.

[10] F. Wefers and J. Berg, "High-performance real-time firfiltering using fast convolution on graphics hardware," in *Proc. of the 13th Int. Conference on Digital Audio Effects (DAFx-10)*. Graz, Austria: Institute of Technical Acoustics, RWTH Aachen University, Sep. 6–10, 2010, pp. DAFX–1 – DAFX–8.

[11] D. Wang and G. J. Brown, *Computational Auditory Scene Analysis*. John Wiley & Sons, Inc., 2005.

[12] H. Zimmermann, "Osi reference model -the iso model of architecture for open systems interconnection," in *IEEE Transactions on Communications, Vol. 28, No. 4*, Apr. 1980, pp. 425–432.

[13] C. Kuhr and A. Carôt, "Software architecture for a multiple avb listener and talker scenario," in *Proceedings of the Linux Audio Conference 2018*. Berlin, Germany: Linuxaudio.org, Jun. 7–10, 2018.

[14] Y. Orlarey, S. Letz, and D. Fober, *New Computational Paradigms for Computer Music*, G. Assayag, Ed. Paris, France: Delatour, 2009.

[15] C. N. d. C. M. GRAME, "Faust quick reference (version 0.9.100)," 2017.

[16] (2018, Mar.) Docker documentation. [Online]. Available: https://docs.docker.com/

[17] (2018, Mar.) Explore official repositories. [Online]. Available: https://hub.docker.com/explore/

[18] (2018, Mar.) Docker compose documentation. [Online]. Available: https://docs.docker.com/compose/

[19] (2018, Mar.) Electron documentation. [Online]. Available: https://electronjs.org/docs/tutorial/about

[20] (2018, Mar.) Node.js v9.8.0 documentation. [Online]. Available: https://nodejs.org/dist/latest-v9.x/docs/api/

[21] (2018, Mar.) Typecript documentation. [Online]. Available: http://www.typescriptlang.org/docs/home.html

[22] (2018, Apr. 23) Jack audio connection kit. [Online]. Available: https://jackaudio.org