Audio Rendering/Processing and Control Ubiquity ? a Solution Built Using the Faust Dynamic Compiler and JACK/NetJack

Stephane Letz, Sarah Denoux, Yann Orlarey

Audio Rendering/Processing and Control Ubiquity ? a Solution Built Using the Faust Dynamic Compiler and JACK/NetJack

Stephane Letz
GRAME
letz@grame.fr

Sarah Denoux
GRAME
sdenoux@grame.fr

Yann Orlarey
GRAME
orlarey@grame.fr

ABSTRACT

We usually think of an audio application as a self-contained executable that will compute audio, allow user interface control, and render sound in a single process, on a unique machine.

With the appearance of fast network and sophisticated, light and wireless control devices (such as tablets, smartphones...) the three different parts (that are *audio computation*, *interface control* and *sound rendering*) can naturally be decoupled to run on different processes on a given machine, or even on different machines (on a LAN or WAN network).

We describe a solution to run and control audio DSP on different machines based on:

- the FAUST audio DSP language which permits local and remote dynamic compilation, code migration and deployment (using *libfaust*, *libfaustremote* and LLVM)
- local and remote control capabilities (via *OSC* and *HTTP* based control interfaces)
- JACK/NetJack network audio real-time layer to handle remote audio processing and rendering.

1. INTRODUCTION

Audio applications are usually self-contained executables running on a single machine. Most of them can be extended with additional audio processing capabilities using plug-ins following different standards (VST, Audio Unit, LV2, etc.). Control is usually either done directly by interacting with the application user interface, using the keyboard, mouse, or multi-touch devices that liberate and extend control access.

Several control protocols have been designed over the years, going from the old MIDI protocol to the more general and flexible Open Sound Control [1] one, and several more specialized ones (Mackie...).

Audio architectures themselves have evolved to offer to the developer and the user a more flexible infrastructure to build custom working environments. The Jack Audio Connection Kit [2] (JACK) for instance not only allows applications to share audio and MIDI devices, but also do inter-application audio and MIDI routing. With its NetJack extension, networks of connected machines exchanging audio and MIDI streams in real-time on the LAN can be built. Alternative audio over network solutions, like JackTrip, an infrastructure over TCP/IP Wide Area Networks, have been developed and continuously extended [3], [4]. Concerning remote DSP processing, proprietary and commercial solutions like Apple Logic Pro distributed network audio system [1] have existed for years.

1.1 Control, Compute, Communicate

Until now, very few solutions have existed which easily compile, migrate and deploy *arbitrary* audio DSP code on different machines, only using open-source components. This can be done if the different parts of the audio application are clearly defined and separated, thus more clearly expressing the *ubiquitous* nature of the computation the application is going to achieve.

We can roughly describe any audio application as having three principal parts (see Figure 1), which are named with the *Control-Compute-Communicate* terminology:

- *control*: the control part changes the parameters in real-time
- *compute*: the audio DSP computation processes audio inputs and produces audio outputs
- *communicate* (with the audio card): the audio rendering part triggers the audio computation.

This paper presents a practical solution using the FAUST audio DSP language and the JACK/NetJack audio system where:

- control can be done locally or externally using the flexible *architecture* concept used to wrap FAUST compiled code
- DSP code can be compiled and deployed locally or externally by "migrating" the DSP source itself, using *libfaust* and *libfaustremote* libraries
- distributed audio rendering and processing can be implemented using the JACK/NetJack audio/MIDI infrastructure.

[1] see https://documentation.apple.com/en/logicpro/usermanual/

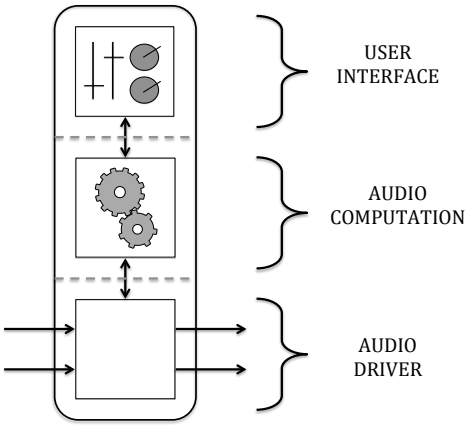

Figure 1. Audio application structure

2. FAUST AUDIO DSP LANGUAGE

FAUST [Functional Audio Stream] [5] [6] [7] is a functional, synchronous, domain-specific programming language specifically designed for real-time signal processing and synthesis. A unique feature of FAUST, compared to other existing music languages like Max [2], PureData, Supercollider, etc., is that programs are not interpreted, but fully compiled. FAUST provides a high-level alternative to C/C++ to implement efficient sample-level DSP algorithms.

2.1 The compilation chain

The FAUST compiler translates a FAUST program into an equivalent imperative program (typically C, C++, Java, etc.), taking care of generating efficient code. The FAUST package also includes various architecture files, providing the glue between the generated code and the external world (audio drivers and user interfaces).

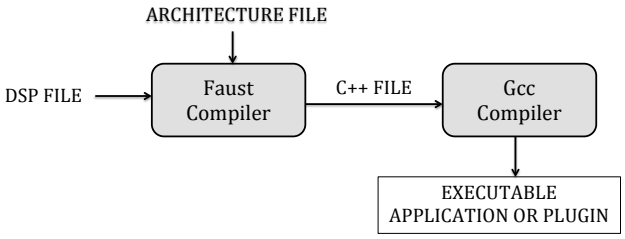

Figure 2. Steps of FAUST compilation chain

The current version of the FAUST compiler produces the resulting DSP code as a C++ class, to be inserted in the architecture file. The C++ file is finally compiled with a regular C++ compiler to produce the final executable program or plug-in (Figure 2).

The resulting application is structured as shown in Figure 1. The DSP becomes an audio computation module, linked to the user interface and the audio driver.

If compilers have the advantage of efficiency, they have their own drawbacks compared to interpreters. Compilers traditionally require a whole chain of tools to be installed (compiler, linker, development libraries, etc.). For non-programmers this task can be complex. The development cycle, from the edition of the source code to a running application, is much longer with a compiler than with an interpreter. This can be a problem in creative situations where quick experimentation is essential. Moreover, binary code is usually not compatible across platforms and operating systems.

2.2 FIR: Faust Imperative Representation

In order to generate alternative output (like pure C, Java, JavaScript, LLVM IR etc.), an intermediate language called FIR (Faust Imperative Representation) has been defined in the *faust2* development branch. This language allows the description of the calculations performed on the samples in a generic manner. It contains primitives to read and write variables and arrays, perform arithmetic operations, and define the necessary control structures (*for* and *while* loops, *if* structure etc.). The *language of signals* internal to the compiler is now compiled in this FIR intermediate language.

2.3 LLVM

LLVM (formerly Low Level Virtual Machine) is a compiler infrastructure, designed for compile-time, link-time, run-time optimization of programs written in arbitrary programming languages. Executable code is produced dynamically using a "Just In Time" compiler from a specific code representation, called LLVM IR [3]. Clang, the "LLVM native" C/C++/Objective-C compiler is a front-end for LLVM Compiler. It can, for instance, convert a C or C++ source file into LLVM IR code (Figure 3).

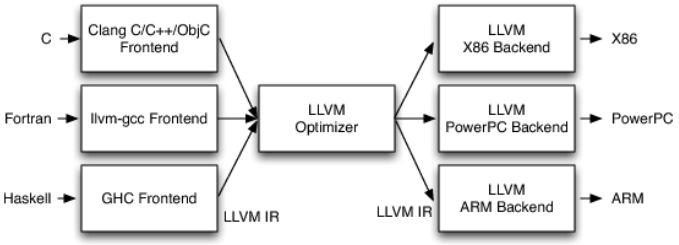

Figure 3. LLVM compiler structure

Domain-specific languages like FAUST can easily target the LLVM IR. This has been done by developing a special LLVM IR back-end in the FAUST compiler [8].

2.4 Dynamic compilation chain

The complete chain goes from the DSP source code, compiled in LLVM IR using the LLVM back-end, to finally produce the executable code using the LLVM JIT. All steps are done in memory. Pointers on executable functions can be retrieved in the resulting LLVM module, and their code directly called with the appropriate parameters (Figure 4).

[2] the *gen* object added in Max6 now creates compiled code from a patch-like representation, using the same LLVM based technology

[3] *The Intermediate Representation* is an intermediate SSA representation

In the *faust2* development branch, the FAUST compiler has been packaged as an embeddable library called *libfaust*, published with an associated API [8]. This API imitates the concept of oriented-object languages, like C++.

The compilation step, usually executed by GCC, is accessed through the function *createDSPFactory*. Given a FAUST source code (as a file or a string), the compilation chain (FAUST + LLVM JIT) generates the "prototype" of the class, as a *llvm-dsp-factory* pointer.

Next, the *createDSPInstance* function, corresponding to the *new className* of C++, instantiates a *llvm-dsp* pointer, to be used as any regular FAUST compiled DSP object, run and controlled through its interface.

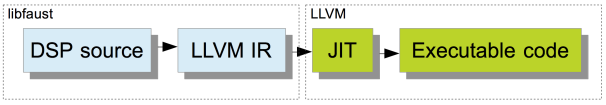

Figure 4. LLVM compiler structure

3. NETJACK NETWORK AUDIO LAYER

NetJack is a real-time Audio Transport over a generic IP Network, fully integrated into JACK [2]. Based on a master/slave model, NetJack synchronizes all clients to the master machine sound card, running all slaves with the same sampling rate and buffer size. When run directly in the JACK server, NetJack appears as two different parts (Figure 5):

- the master component (*netmanager* in JACK2 [4] implementation) is loaded in a server running and synchronized on an audio back-end
- the slave component is used as the back-end (*netjack* back-end in JACK2 implementation) of a slave JACK server running on the remote machine. This way the slave machine is synchronized with the master machine so that no re-sampling on the slave side is needed.

This way, two (or more...) separated machines running each an entire JACK infrastructure (that is a JACK server and several JACK applications) are connected and synchronized through the network.

But the NetJack protocol is also available in a library called *libjacknet* that embeds and offers the master and slave components as a C API, to be used in applications developed *outside of the JACK server context*, as explained in the following sections.

3.1 Remote processing

Since the entire FAUST DSP to executable code chain is now completely embeddable, it becomes quite easy and

[4] JACK2 is the C++ implementation of the JACK server and API running on major OS: Linux, OSX and Windows

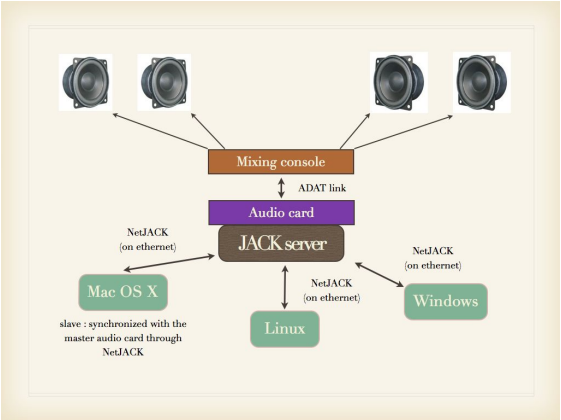

Figure 5. Typical NetJack use case

natural to extend this model on the network to enable remote processing. This way, compilation and DSP computation can easily be redirected on a remote machine.

3.1.1 Remote server

On the remote machine, the compilation/processing service appears as a specialized HTTP server waiting for requests. Remote processing service is detected on the local machine side using the *libfaustremote* library.

The first step (compilation) is carried out by the function *createRemoteDSPFactory*. The Faust DSP code is sent to the server, which compiles it and creates the "real" llvm-dsp-factory. The remote-dsp-factory returned to the user is a proxy for the "real" factory. Before sending the FAUST code, a FAUST to FAUST compilation step is locally executed, to solve all code dependencies, and thus send a completely self-contained expanded code version to the server (Figure 6).

All available machines on the network can be scanned with *getRemoteMachinesAvailable*, and for a given one, already compiled DSP factories can be retrieved with *getRemoteFactoriesAvailable*.

3.1.2 Local "proxy"

On the client side, a "proxy" API makes it transparent to create a *remote-dsp* pointer rather than a local *llvm-dsp*.

Using the *createRemoteDSPInstance* function, the remote-dsp-factory can then be instantiated to create remote-dsp instances, which can then run in the chosen audio/control architecture.

To be able to locally create the interface, the server returns a JSON encoded interface. This way, the function *buildUserInterface* can be recreated, giving the feeling that a remote-dsp works as a local llvm-dsp.

3.1.3 NetJack audio/control connection

Using a NetJack low-latency audio connection the audio data is sent (using the *master* component of the *libjacknet* library) to the remote machine to be processed and sent back (as a *slave* component of the *libjacknet* library).

In addition to the standard audio flow, one MIDI port is used to transfer the controller values (Figure 6). The

benefit of this solution is to transmit synchronized audio and controller values in the same connection. Audio samples can be encoded using the different possible audio data types: float, integer, and compressed audio (using the OPUS [5] codec).

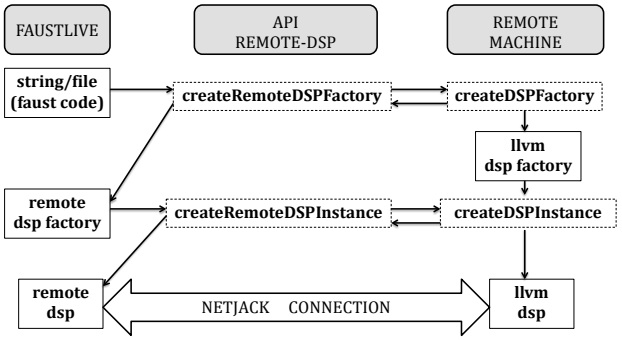

Figure 6. Remote compilation

The *libfaustremote* library uses *libcurl* to send HTTP requests to the remote server, handled with *libmicrohttpd*. The *ZeroConf* protocol is used to scan the remote machines presenting the service and export them as a list of available machines.

3.2 Remote rendering

NetJack layer can also be used to separate the audio processing and audio rendering parts. Instead of using it's own sound card, the machine will start a NetJack slave audio driver, receiving its audio inputs from a remote NetJack master and sending back its audio outputs.

4. USER INTERFACE CONTROL

A FAUST UI architecture is a glue between a host control layer and a FAUST module. It is responsible for associating a FAUST module parameter to a user interface element and to update the parameter value according to the user's actions.

This association is triggered by the *buildUserInterface* call, where the DSP object asks a UI object to build the module controllers. Moving UI elements later on changes parameter values which are "sampled" at each audio cycle and used by the DSP computation loop.

4.1 Local control

Local controllers are typically built using UI frameworks (like GTK or QT) that allow to create buttons, sliders, text entry zones or bargraphs. Those elements are then arranged on a complete window following an abstract layout description that is part of the Faust DSP source.

4.2 Remote control

Moving controls on a remote machine assumes that a control communication protocol has been defined between the local and remote machines.

[5] http://www.opus-codec.org

4.2.1 OSC control

The OSC [1] support opens the FAUST applications control to any OSC capable application or programming language. But it also transforms a full range of devices embedding sensors (wiimote, smartphones, tablets...) into physical interfaces for FAUST applications control, allowing their direct use as music instruments (Figure 8).

The UI with its layout and UI items hierarchy is encoded as a OSC address space, to be retrieved and used by OSC client applications. Several ports are defined:

- 5510 is the listening port number: control messages should be addressed to this port.
- 5511 is the output port number: answers to query messages are send to this port.
- 5512 is the error port number: used for asynchronous errors notifications.

4.2.2 HTTP control

The FAUST HTTP architecture provides an UI architecture to be controlled by standard browsers. The compiled application embeds a specialized HTTP server (developed using the *libmicrohttpd* library), that waits for connections on a identified port (like 5510), delivers the UI to clients as a JSON encoded string with some JavaScript code to decode and display it, and build a fully controllable client side user interface.

To ease the opening of the interface, a Qr Code is built from the HTTP address, thanks to *libqrencode*. Most smartphones and portable equipments have a QrCode decoder. By scanning the Qr Code, a browser gets connected to the interface page.

Control parameters are transferred in both directions, so that the browser can effectively display values produced by the DSP computation (like vumeters level for instance).

Several control machines can possibly be used, each one having its own opened browser. While the OSC control interface is designed to be used on a LAN network, the HTTP control model is easily usable on WAN, thus opening interesting possibilities (see Figure 7).

5. USE CASES

With the FAUST local and remote dynamic compiler and JACK/NetJack network audio layer in place, a wide variety of interesting use cases can now be put in practice.

5.1 Remote control

Any Faust DSP program can be remotely controlled using either the OSC or the HTTP architecture. The DSP object is dynamically wrapped by the appropriate user interface C++ class. This way it becomes accessible on the network for any available control application.

We have tested OSC control from Max/MSP patches, and HTTP control using standard browsers on laptop, tablets or even smartphones. Sound installations are a typical use case where publishing the Qr Code built from the HTTP address allows visitors to interact with the system.

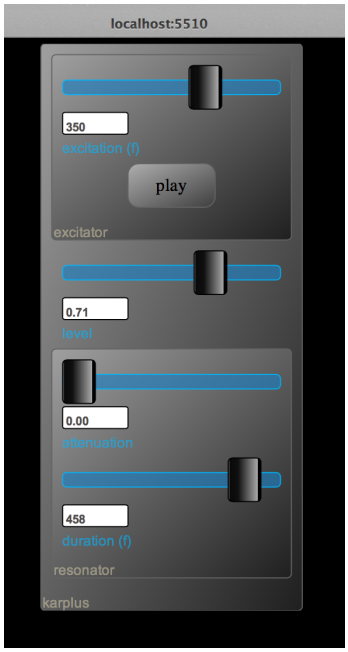

Figure 7. HTTP interface: *karplus* DSP running in a browser with a SVG based UI built from the JSON exported interface

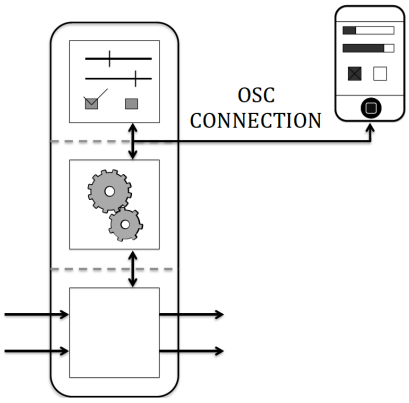

Figure 8. OSC interface

5.2 Remote audio rendering

By using JACK/NetJack on a LAN, several slave machines can access a master one, which would typically be connected on a high quality studio audio system.

Another typical use case is a class room, where a unique sound system is available: all pupils can possibly connect their machines and use it (Figure 9).

5.3 Remote audio processing

Migrating DSP code on a remote machine (Figure 10) makes sense when CPU heavy DSP cannot be computed on the user's machine. A typical case would be a composer coming with his/her laptop in the studio, and possibly using the more powerful available machines.

Another interesting use case we have experimented to facilitate the rapid development and experimentation of audio DSP programs on tablets and smartphones with the following steps:

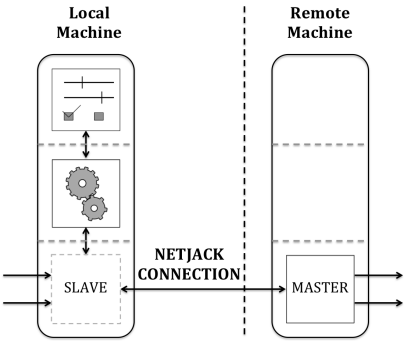

Figure 9. NetJack based audio rendering

- a generic application running on the tablet/smartphone scans all machines on the network with *getRemoteMachinesAvailable*
- for a given one, already compiled DSP factories can be retrieved with *getRemoteFactoriesAvailable*
- a remote DSP instance then can be created, connected to the tablet/smartphone audio system, and locally controlled

We have successfully tested this example on a MacBook Pro laptop, with audio rendering and control running in wireless mode on an iPad [6]. An audio effect can be developed and rapidly tested on the tablet, then later on fully compiled to native and self-contained version for final deployment.

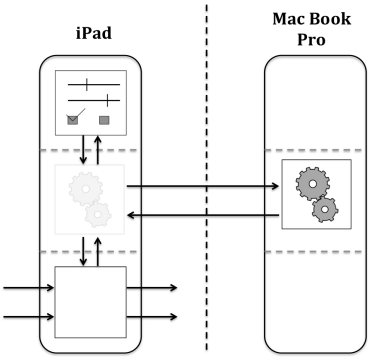

Figure 10. Remote processing

5.4 FaustLive as a demonstration platform

Most of the scenarios described above can be tested out with FaustLive, a QT based application available on Linux, OSX and Windows. FaustLive is a standalone just-in-time FAUST compiler, that allows to easily write, test, experiment and deploy DSP programs [9].

[6] this is also an elegant way to deal with the current limitation of iOS, which does not allow to embed a native dynamic compiler infrastructure on the tablet itself...

Distributed control, processing and rendering can also be tested on multiple machines using the application and JACK/NetJack infrastructure.

6. PERFORMANCES AND BENCHMARKS

The following section gives some numbers concerning the performance and capabilities of the "control, compute, communicate" approach.

6.1 Compilation and startup

The first advantage of the fully dynamic and embedded compilation chain is to simplify the deployment of the Faust compiler technology itself. But it also gives major speedup in the compilation step when deploying FAUST DSP code. Here are three examples of simple to quite heavy DSP programs (Table 1).

Effect	C++ time	LLVM time
karplus32	5.3s	0.3s
cubic_interpolation	6.5s	1.6s
ethersonik	5.9s	0.7s

Table 1. Compilation and startup time for C++ and LLVM based chains, tested on a MacBook Pro 2,3 GHz

6.2 Remote processing and rendering

Running DSP code on the network using NetJack adds some latency. On a Gigabit LAN, roundtrip network latencies of 1 or 2 buffers can easily be obtained when using dozens of audio channels.

Wireless audio and control connections can be done, but more buffering packets have to be added in the NetJack connection. On a dedicated network we have tested an iPad to laptop connection, with 5 to 10 buffers of 1024 frames (compressed using the OPUS open-source codec) in the connection, thus adding an acceptable latency of 100 to 200 ms.

7. GETTING THE CODE

The various software components previously described are open-source projects available in the following sites.

- JACK server and NetJack can be found at: http://jackaudio.org. Latest describe NetJack version is part of the soon to be published 1.9.10 version of the JACK infrastructure.
- FAUST server can be found at: http://faust.grame.fr
- *libfaust* and *libfaustremote* libraries are part of the *faust2* branch and can be found at: http://sourceforge.net/projects/faudiostream/

8. CONCLUSION AND PERSPECTIVES

As a result of FAUST dynamic code compilation and migration capabilities based on *libfaust*, *libfaustremote*, and JACK/NetJack, audio DSP code can now be easily deployed and controlled on local and remote machines.

More precise benchmarks and analysis still need to be done on more complex audio DSP networks.

Moreover it could be interesting to have any FAUST audio DSP node embed its own DSP source (as a self-contained expanded string), so that a graph of connected audio DSP nodes could be analyzed and possibly be "rewritten" as an equivalent single FAUST DSP code, then possibly redeployed on another target.

Acknowledgments

One part of this work is made possible by a grant from the French National Research Agency (ANR) INEDIT Project (ANR-12-CORD-0009), and for the other part by a grant from the French National Research Agency (ANR) FEEVER project (ANR-13-BS02-0008).

9. REFERENCES

[1] M. Wright and A. Freed, "Open Sound Control: A New Protocol for Communicating with Sound Synthesizers", *International Computer Music Conference*, 1997, pp. 101–104.

[2] S. Letz, N. Arnaudov and R. Moret, "What's new in JACK2 ?", *Linux Audio Conference 2009*.

[3] J.P. Caceres, C. Chafe, "JackTrip: Under The Hood of an Engine For Network Audio", Journal of New Music Research, 39(3), 2010.

[4] J.P. Caceres, C. Chafe, "JackTrip/SoundWIRE meets server farm", Computer Music Journal, 34(3), 2010.

[5] Y. Orlarey, D. Fober, and S. Letz, "Syntactical and semantical aspects of Faust", *Soft Computing*, 8(9), 2004, pp. 623–632.

[6] Y. Orlarey, D. Fober, and S. Letz, " Adding Automatic Parallelization to Faust", *Linux Audio Conference*, 2009.

[7] S. Letz., Y. Orlarey and D. Fober, " Work Stealing Scheduler for Automatic Parallelization in Faust", *Linux Audio Conference*, 2010.

[8] S. Letz, Y. Orlarey and D. Fober, " Comment embarquer le compilateur Faust dans vos applications ?", *Journees d'Informatique Musicale*, 2013.

[9] S. Denoux, S. Letz, Y. Orlarey and D. Fober, " FAUSTLIVE Just-In-Time Faust Compiler... and much more", *Linux Audio Conference*, 2014.