GETTING OSC TO WORK BETTER WITH FAUST – A PROPOSAL

Albert Gräf

IKM, Musicology, Computer Music Working Group
Johannes Gutenberg University (JGU) Mainz, Germany
aggraef@gmail.com

ABSTRACT

Faust offers very comprehensive OSC[1] support through its own OSC library which gets activated with the `-osc` option of the tool scripts. However, in the current implementation each Faust module needs its own UDP port for incoming OSC, which eats up valuable UDP address space and becomes inconvenient if a lot of Faust modules are to be run simultaneously. In this short note we sketch out an improved system based on a client-server architecture (yet to be implemented!) which we think will improve the OSC handling considerably.

1. INTRODUCTION

A Faust program typically contains one or more control variables for the purpose of changing certain parameters of the signal processing algorithm implemented by the program. E.g., the following little program realizes a simple linear gain control (after mixing down the incoming stereo signal to mono):

```
gain = nentry("gain", 0.3, 0, 10, 0.01);
process = + : *(gain);
```

Let's say that this program is in the monogain.dsp source file. Without any further ado, the program can be equipped with an OSC interface to control the gain parameter (with the external name `gain` specified as the UI label of the control). If the target architecture supports it, you do this by simply specifying the `-osc` option when compiling the program, e.g. (using the Jack architecture):

```
faust2jack -osc monogain.dsp
```

When run from the command line, the resulting program will print something like this, informing the user about the UDP ports to be used for OSC communication:

```
Faust OSC version 0.96 application 'monogain'
is running on UDP ports 5510, 5511, 5512
```

Unless the UDP ports are chosen explicitly with corresponding command line options, the program will use some default port numbers automatically assigned by Faust's OSC library, see [1, Chapter 6] for details. There are always three port numbers for OSC input, OSC output, and error notifications, respectively. In the above example, port 5510 would be used on the OSC device for outgoing messages, and messages with the OSC address `/monogain/gain` would then change the `gain` control in the `monogain` dsp from the device. The address `/monogain/gain` is generated automatically by the OSC library, but it is also possible to use custom OSC addresses by specifying corresponding meta-data in the Faust program. In addition, Faust's OSC library provides OSC address matching using wildcards, and pre-defined messages for discovering the OSC controls offered by a dsp module.

This kind of setup will work very well with any kind of OSC device connected to the (local) network, apps on smartphones and tablets, real-time applications such as INScore, Max and Pd, and MIDI/OSC bridges like Osculator and osc2midi. For instance, Fig. 1 shows TouchOSC, an Android and iOS application often used for that purpose. It allows you to configure the OSC address of each GUI element in a separate editor application (the UDP connection is set in the app itself).

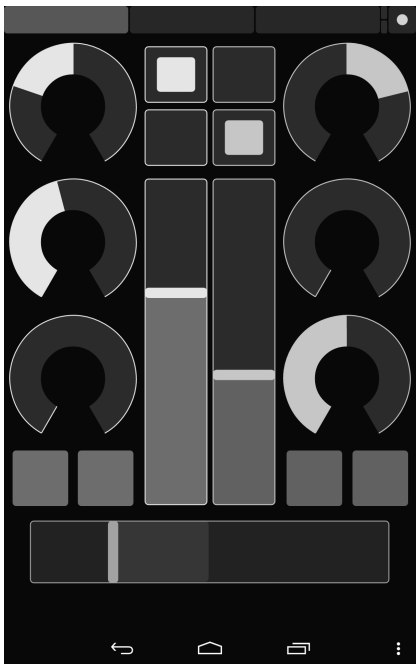

Figure 1: TouchOSC app (Android/iOS).

2. THE PROBLEM

The interface described above is comprehensive and supports applications involving just a few Faust modules really well. But it quickly becomes unwieldy when working with a larger number of modules. Typical use cases of this kind are plug-ins in a DAW and modular synthesizers, which may well involve dozens of separate dsp modules. It also becomes a problem if you want to control multiple dsp modules from the same OSC device, because these

[1] Open Sound Control, cf. `http://opensoundcontrol.org/`

typically transmit OSC data only to a single port. Last but not least, the use of a separate UDP port for each dsp instance seems wasteful, given that OSC already has a much more advanced notion of hierarchical address space built right into it.

A solution offered by the Faust language itself is to combine existing dsp modules into larger ones, using the `component` construct (cf. [1, Section 3.4.4]) and the operations from Faust's block diagram algebra (BDA). But this requires you to create different modules for each needed dsp combo. When working in a plug-in host environment such as a DAW, it is usually preferable to keep the original Faust dsps separate and use the *host environment's* own capabilities to build synth-effect chains or signal graphs instead. This offers more flexibility to musicians and sound engineers who usually know their host environments very well and have their own ways of working with plug-ins within them.

3. A SOLUTION

The problem sketched out above can't really be avoided if Faust modules run as stand-alone, self-contained units. However, it could be solved by hooking into a dedicated Faust-OSC daemon running on each system or for each user session, which would be implemented as an OSC server sitting at a predictable address. This could be done in user space, on top of the existing OSC facilities. Faust's OSC library could automatically launch that server if it isn't running already, to avoid the complexities and cross-platform issues of a system service. This OSC server would then manage all the Faust-OSC traffic for that session and dispatch OSC messages using additional instance name prefixes to the OSC addresses, in order to distinguish between different instances of a Faust module. One might also throw in Zeroconf (Avahi/Bonjour) support[2] to facilitate setup of the network connections (OSC apps like TouchOSC readily support this protocol).

Fig. 2 depicts the kind of setup we have in mind here. Of course the downside of such an approach is added latency, but since the OSC server would dispatch to locally running Faust modules only, the overhead should be tolerable.

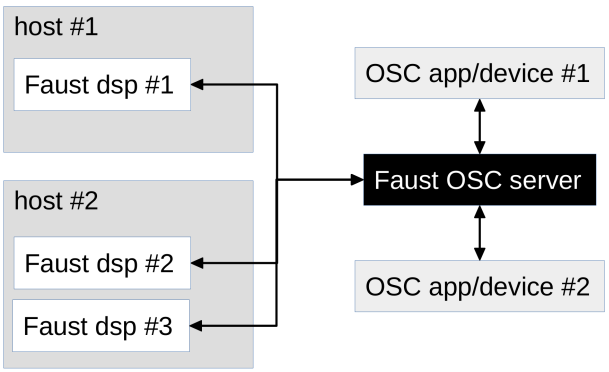

Figure 2: OSC devices controlling different Faust dsps via the Faust-OSC server.

Another issue is the naming of the module prefixes, which might require some cooperation from the host environments in which the Faust dsps are running, so that the user can distinguish different instances of the same Faust dsp. Automatic numbering of the different instances might be a solution to this (at least in host environments capable of displaying such instance names in their GUI). It might also be necessary to think about a protocol to enable OSC devices to switch between different instances of the same Faust module.

[2] `http://www.zeroconf.org/`

4. CONCLUSION

The author believes that the approach sketched out above would make operating Faust dsps via OSC much easier, besides the obvious practical benefits of saving precious UDP ports. Thus we think that it is worth the design and implementation effort. Please note that this has *not* been implemented yet. This short note is just there to kick off the discussion and solicit comments from the developer team and the wider Faust community.

We mention in passing that a similar approach is already being used with great success in the author's `faust~` external for Pd [2], which employs an OSC server running in Pd and allows the instance names to be chosen manually by entering them as arguments of the `faust~` objects. This is simpler than what we are proposing here, however, since it provides one OSC server per Pd instance, so the Faust modules and the OSC server all run in the same process.

5. REFERENCES

[1] GRAME, Centre National de Création Musicale, Lyon, *FAUST Quick Reference*, 2017.

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