Using FAUST with ROS
(version 0.0.01)

Grame
Centre National de Création Musicale

October 2014

Contents

Chapter 1

Introduction

FAUST (*Functional Audio Stream*) is a functional programming language specifically designed for real-time signal processing and synthesis. FAUST targets high-performance signal processing applications and audio plug-ins for a variety of platforms and standards.
ROS (*Robot Operating System*) is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms.

1.1 Faust

1.1.1 Design Principles

Various principles have guided the design of FAUST:

- FAUST is a *specification language*. It aims at providing an adequate notation to describe *signal processors* from a mathematical point of view. FAUST is, as much as possible, free from implementation details.
- FAUST programs are fully compiled, not interpreted. The compiler translates FAUST programs into equivalent C++ programs taking care of generating the most efficient code. The result can generally compete with, and sometimes even outperform, C++ code written by seasoned programmers.
- The generated code works at the sample level. It is therefore suited to implement low-level DSP functions like recursive filters. Moreover the code can be easily embedded. It is self-contained and doesn't depend of any DSP library or runtime system. It has a very deterministic behavior and a constant memory footprint.
- The semantic of FAUST is simple and well defined. This is not just of academic interest. It allows the FAUST compiler to be *semantically driven*. Instead of compiling a program literally, it compiles the mathematical function it denotes. This feature is useful for example to promote components reuse while preserving optimal performance.

- FAUST is a textual language but nevertheless block-diagram oriented. It actually combines two approaches: *functional programming* and *algebraic block-diagrams*. The key idea is to view block-diagram construction as function composition. For that purpose, FAUST relies on a *block-diagram algebra* of five composition operations (`:` `,` `~` `<:` `:>`).

- Thanks to the notion of *architecture*, FAUST programs can be easily deployed on a large variety of audio platforms and plugin formats without any change to the FAUST code.

1.1.2 Signal Processor Semantic

A FAUST program describes a *signal processor*. The role of a *signal processor* is to transforms a group of (possibly empty) *input signals* in order to produce a group of (possibly empty) *output signals*. Most audio equipments can be modeled as *signal processors*. They have audio inputs, audio outputs as well as control signals interfaced with sliders, knobs, vu-meters, etc.

More precisely :

FAUST considers two type of signals: *integer signals* ($s : \mathbb{N} \rightarrow \mathbb{Z}$) and *floating point signals* ($s : \mathbb{N} \rightarrow \mathbb{Q}$). Exchanges with the outside world are, by convention, made using floating point signals. The full range is represented by sample values between -1.0 and +1.0.

- A *signal* s is a discrete function of time $s : \mathbb{N} \rightarrow \mathbb{R}$. The value of signal s at time t is written $s(t)$. The set $\mathbb{S} = \mathbb{N} \rightarrow \mathbb{R}$ is the set of all possible signals.

- A group of n signals (a n-tuple of signals) is written $(s_1, \ldots, s_n) \in \mathbb{S}^n$. The *empty tuple*, single element of $\mathbb{S}^0$ is notated $()$.

- A *signal processors* p, is a function from n-tuples of signals to m-tuples of signals $p : \mathbb{S}^n \rightarrow \mathbb{S}^m$. The set $\mathbb{P} = \bigcup_{n,m} \mathbb{S}^n \rightarrow \mathbb{S}^m$ is the set of all possible signal processors.

As an example, let's express the semantic of the FAUST primitive `+`. Like any FAUST expression, it is a signal processor. Its signature is $\mathbb{S}^2 \rightarrow \mathbb{S}$. It takes two input signals X_0 and X_1 and produce an output signal Y such that $Y(t) = X_0(t) + X_1(t)$.

Numbers are signal processors too. For example the number 3 has signature $\mathbb{S}^0 \rightarrow \mathbb{S}$. It takes no input signals and produce an output signal Y such that $Y(t) = 3$.

1.2 ROS

1.2.1 What is it ?

The following content is taken from ROS documentation. It can be found on ROS official website and ROS wiki.

Creating truly robust, general-purpose robot software is *hard*. From the robot's perspective, problems that seem trivial to humans often vary wildly between instances of tasks and environments. Dealing with these variations is so hard that no single individual, laboratory, or institution can hope to do it on their own.

ROS is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers.

As a result, ROS was built from the ground up to encourage *collaborative* robotics software development. For example, one laboratory might have experts in mapping indoor environments, and could contribute a world-class system for producing maps. Another group might have experts at using maps to navigate, and yet another group might have discovered a computer vision approach that works well for recognizing small objects in clutter. ROS was designed specifically for groups like these to collaborate and build upon each other's work, as is described throughout this site.

1.2.2 Concepts

Filesystem level

The filesystem level concepts mainly cover ROS resources that you encounter on disk, such as:

- **Packages** are the main unit for organizing software in ROS. A package may contain ROS runtime processes (nodes), a ROS-dependent library, datasets, configuration files, or anything else that is usefully organized together. Packages are the most atomic build item and release item in ROS. Meaning that the most granular thing you can build and release is a package.
- **Metapackages** are specialized Packages which only serve to represent a group of related other packages.
- **Services :** Service descriptions, stored in my_package/srv/MyServiceType.srv, define the request and response data structures for services in ROS.
- **Messages :** Message descriptions, stored in my_package/msg/MyMessageType.msg, define the data structures for messages sent in ROS.

Computation Graph level

The Computation Graph is the peer-to-peer network of ROS processes that are processing data together. The basic Computation Graph concepts of ROS are nodes, Master, Parameter Server, messages, services, topics, and bags, all of which provide data to the Graph in different ways.

- **Master :** The ROS Master provides name registration and lookup to the rest of the Computation Graph. Without the Master, nodes would not be able to find each other, exchange messages, or invoke services.
- **Nodes :** Nodes are processes that perform computation. ROS is designed to be modular at a fine-grained scale; a robot control system usually comprises many nodes. For example, one node controls a laser range-finder, one node controls the wheel motors, one node performs localization, one node performs path planning, one Node provides a graphical view of the system, and so on. A ROS node is written with the use of a ROS client library, such as roscpp or rospy.

- **Topics :** Messages are routed via a transport system with publish / subscribe semantics. A node sends out a message by publishing it to a given topic. The topic is a name that is used to identify the content of the message. A node that is interested in a certain kind of data will subscribe to the appropriate topic. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics. In general, publishers and subscribers are not aware of each others' existence. The idea is to decouple the production of information from its consumption. Logically, one can think of a topic as a strongly typed message bus. Each bus has a name, and anyone can connect to the bus to send or receive messages as long as they are the right type.

- **The Parameter Server :** The Parameter Server allows data to be stored by key in a central location. It is currently part of the Master.

- **Messages :** Nodes communicate with each other by passing messages. A message is simply a data structure, comprising typed fields. Standard primitive types (integer, floating point, boolean, etc.) are supported, as are arrays of primitive types. Messages can include arbitrarily nested structures and arrays (much like C structures).

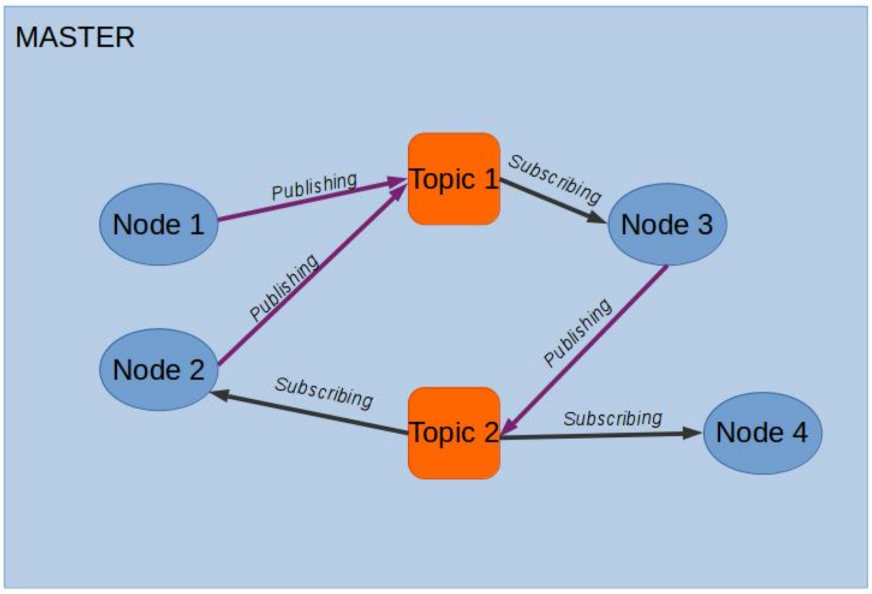

Figure 1.1: ROS Concepts in a diagram

Names

Names are really important in ROS. Valid names have these characteristics :

- first chararacter is an alpha character : [a-z][A-Z]
- subsequent characters can be alphanumeric : [a-z][A-Z][0-9], underscores : _ or forward slash : /
- there is at most one forward slash : /

For more informations on ROS and tutorials, please have a look to the website : www.wiki.ros.org.

Chapter 2

Compiling *FAUST* programm for ROS use

To compile a FAUST programm for a ROS use, you can use either the `faust2ros` command, or the `faust2rosgtk` one, which adds a gtk graphic user interface to the simple `faust2ros` command.

2.1 Compiling in a *FAUST* archive

In order to compile a DSP file into a FAUST archive, just type the command followed by your file :

```
    faust2ros my_dsp_file.dsp
```

It should output `faust.zip/my_dsp_file;` and the resulting `faust.zip` folder should contain the following elements:

`faust_msgs`	messages package to handle faust messages
`my_dsp_file`	package containing a .cpp file corresponding to the DSP file

If the DSP file is not in the current directory, make sure to type the right path. For instance :

```
    faust2ros ~/faust/examples/myfile.dsp
```

2.2 Compiling in a workspace

Thanks to the option `-install`, you have the possibility to create a package from your DSP file directly in the a workspace you chose. Just type :

```
    faust2ros -install faust_ws file.dsp
```

It should output :

```
    file.cpp;
```

and you should have a faust_ws repository looking like this :

```
faust_ws
├── build
├── devel
└── src
    ├── faust_msgs :  Messages Package
    │   │              Files to handle Faust messages.
    │   ├── include
    │   ├── msg
    │   │   └── ParamFaust.msg
    │   ├── src
    │   ├── CMakeLists.txt
    │   └── package.xml
    └── file :  File Package .
        ├── include
        ├── src
        │   └── file.cpp :  File generated with Faust compiler .
        ├── CMakeLists.txt
        └── package.xml
```

2.3 Renaming DSP file

If the dsp file does not fit you, you can rename it using the `-o` command. For instance, if you want the package generated from DSP file to have a different name that your DSP file name, you can type :

```
    faust2ros -o foobar file.dsp
```

The output is going to be :

```
faust.zip
├── faust_msgs
└── foobar
```

2.4 Examples

Here are some examples of files compilation.

Input :

```
    faust2rosgtk -install foo_ws -o foo1 file1.dsp
    -install foo_ws -o foo2 file2.dsp
    -install bar_ws -o bar file3.dsp
```

Output :

```
~
├── foo_ws
│   ├── faust_msgs
│   ├── foo1
│   └── foo2
└── bar_ws
    └── bar
```