Ingen: A Meta-Modular Plugin Environment

David E. Robillard
School of Computer Science, Carleton University
1125 Colonel By Drive
Ottawa ON K1S 5B6
Canada
d@drobilla.net

Abstract

This paper introduces Ingen, a polyphonic modular host for LV2 plugins that is itself an LV2 plugin. Ingen is a client/server system with strict separation between client(s) and the audio engine. This allows for many different configurations, such as a monolithic JACK application, a plugin in another host, or a remote-controlled network service. Unlike systems which compile or export plugins, Ingen itself runs in other hosts with all editing facilities available. This allows users to place a dynamic patching environment anywhere a host supports LV2 plugins. Graphs are natively saved in LV2 format, so users can develop and share plugins with others, without any programming skills.

Keywords

LV2, JACK, plugin, modular, synthesizer

1 Introduction

The Free Software world has long had powerful visual programming environments like Pure Data [Puckette, 1996], and higher level modular synthesizers like Alsa Modular Synth [Nagorni, 2003]. However, most existing software modular environments (or simply *modulars*) do not integrate as well as possible with other projects. Due to the limitations of popular plugin APIs, most existing modulars are primarily designed around built-in *internals*, and use a different interface themselves (e.g. running only as an application). This situation results in much effort spent building components that are not widely useful across applications.

There are several typical forms for audio processors: a stand-alone software application, a software plugin, or a remote device. Each has advantages depending on the situation. Remote control is necessary for hardware to integrate with a software environment, and increasingly popular for software due to the pervasiveness of tablets, powerful controllers, and networks. An ideal system must be controllable from any location to fit well in all these scenarios. Consequently, the same must be true of the plugins hosted within it.

This leads to the elegant conclusion that the ideal form of a modular plugin host, and the ideal form of a plugin within it, are one and the same. Ingen is an exercise in chasing this ideal: a modular host that has exactly the same external form as the plugins used within it. The practical benefit of such a design is that the user can build a device anywhere in the system where plugins are supported. This makes it possible to work around limitations in programs or the lack of an available plugin to solve the necessary problem. By making it simple for users to share their creations, the community at large can benefit from the pool of plugins created by users who would not have done so if writing code was required.

Ingen takes advantage of the LV2 plugin API's extensibility to achieve these goals. The two have a symbiotic relationship: when Ingen itself requires an API advancement, the improvement ideally becomes standardised in LV2. Other plugins may then use this API, resulting in more powerful plugins for use in Ingen, or other hosts. Likewise, Ingen benefits from advancements originally designed for other plugins.

This paper introduces Ingen as a useful tool for users, and shares the general conclusions reached over the years that led to its design, and consequently the design of many aspects of LV2.

2 Features and Philosophy

2.1 Internals Considered Harmful

Ingen is designed around the principal that generic plugins should be used wherever possible: internals are a symptom of an inadequate plugin API. With an open and extensible specification like LV2, these limitations can be addressed so progress needn't be stalled. This avoids a walled garden effect where a large amount of effort is spent on internals that only work in one program.

The minimalist ideal is for Ingen to have no internals whatsoever, but currently a few are required for tasks that are beyond the capabilities of generic plugins. In particular, LV2 currently lacks

polyphonic voice control, so the *Note* internal performs voice allocation, sending controls to particular voices based on MIDI input. Perhaps in the future there will be sufficient developer interest in (visibly) polyphonic plugins to develop an LV2 extension which will eliminate the need for a special voice allocation internal.

2.2 Messages Considered Wonderful

The common combination of GUI-centric design and direct memory access between plugins and their GUIs results in several problems. In particular, a large subset of many plugins' functionality is inaccessible except via the custom GUI. This severely limits the power of the system to intelligently automate or otherwise control plugins. In addition, shared access to mutable data from multiple threads is an infamously error-prone situation, made even more difficult with the addition of real-time requirements. It is all too common for a poorly written GUI to cause audio drop-outs, or crash the plugin entirely. Direct access to plugin internals is occasionally necessary (for visualisation in particular), but is an inherently flawed approach to plugin control in general.

The solution to this problem is a classic one: separate the plugin and its user interface, and have the two communicate via messages. If these messages are meaningful (i.e. not opaque), the plugin can be controlled in the same way from any source: the GUI, the host, other plugins, scripts, and so on. Traditionally, MIDI is used for this purpose, but MIDI has significant limitations. Ingen supports sending arbitrary messages between hosted plugins and their UIs (including MIDI), and is controlled entirely via messages itself.

Control via standard and portable messages is the key to building audio components that can be deployed in many different scenarios. Designing the system fundamentally around this principle (rather than "bolting on" partial support for remote control) ensures that all interfaces to the system enjoy the same power.

2.3 Polyphony

Though inspired by modular synthesizers, Ingen does not seek to emulate the limitations of hardware. Polyphony in particular is an important feature where software has a distinct advantage. This is an area where extreme minimalism is counterproductive: though it is possible to build a polyphonic synth manually in a monophonic modular by replicating voices, this is a burden on the user. Instead, Ingen implements polyphony internally. Nodes can simply be flagged as polyphonic, and they will be replicated as necessary. Polyphony[1] is a property of the containing graph, i.e. if a graph has polyphony p, all nodes in that graph have either 1 or p voices. Any connection between polyphonic ports is a polyphonic connection, and any connection from a polyphonic port to a monophonic port mixes down all voices.

2.4 Data Types

Ingen supports many data types, including audio, "control voltage" (CV, audio-rate numeric controls), and events in any format such as MIDI. A port transmits either signals or sequences: audio and CV are the only signal types, everything else is a sequence. Sequences are a series of "events" or "messages" transmitted in-band with audio. LADSPA-style control ports are control-rate signals from the point of view of the plugin, but in Ingen are exposed as sequences of floating point numbers to allow control changes to be transmitted with sample accuracy.

The ability to work with many data types is powerful, but requires the user to understand the types of different ports. Ingen distinguishes port data types by colour, and also adds hint symbols as shown in Table 1. Signal and sequence ports are distinguished by shape: signal ports have rounded borders (suggesting *continuous*), and sequence ports have bevelled borders (suggesting *discrete*). A symbol is also composed on the type hint, for example, a real number signal (CV) is tagged '$\tilde{\mathbb{R}}$', and real number messages are tagged '$\ddot{\mathbb{R}}$'.

Symbol(s)	Data type
$\sim$	Audio (floating point)
$\mathbb{R}$	Real (floating point)
$\mathbb{Z}$	Integer
$\mathbb{M}$	MIDI
$\square, \boxtimes$	Boolean
$=$	Patch message

Table 1: Type hint symbols for ports.

Despite the many different data types, Ingen attempts to preserve the "anything to anywhere" ability of classic modular synthesizers wherever possible. For example, a float message output can be connected to a CV input; Ingen will automatically write the CV buffer as if the signal were continuous.

2.5 Inter-Plugin Communication

Many plugins must communicate both audio and messages, a typical example being MIDI synthesizers. Both are transmitted in Ingen in the same context to allow sample-accurate real-time message

[1] As opposed to the boolean polyphon*ic*.

handling, and avoid threading issues. This is distinct from some systems, such as Pd, where message transmission follows different rules than signal transmission. In other words, messages in Ingen are in-band with audio signals.

The benefit of this approach is a single consistent concept of real-time: plugins have one `run()` method which processes all inputs and emits all outputs synchronously. However, some plugins must perform non-real-time operations in response to messages. For example, a sampler plugin may need to load samples from disk.

The LV2 worker extension solves this problem. The worker extension provides a simple API for plugins to schedule a callback to be called "soon" in a non-real-time thread, and a mechanism for replying back to the audio thread in a later cycle. This makes it possible for plugins to perform non real-time operations, but the API is designed such that its use is inherently real-time safe, and plugins do not need to use any non-portable threading libraries. Having this mechanism implemented by the host has performance benefits as well, for example, the host can share one ring buffer and worker thread for all plugins. This can dramatically reduce the memory consumption when many plugins are loaded.

3 Architecture

3.1 Model

Ingen uses a simple data model to describe all components of a graph. Each object has a unique path (like `/fx/verb1`) and a set of key:value properties. Keys are URIs (making state meaningful), and values may have any type. Essentially, everything is a hierarchical tree of dictionaries.

The use of a consistent data model allows for a very simple protocol to perform a large number of operations. Rather than adding "commands" to the interface for every new feature, changes are implemented in terms of property changes. Only a few methods are required to allow arbitrary property changes, so this allows for a powerful yet stable protocol. There are no issues with breaking the number or order of arguments, since properties have no order. New information can be added freely without requiring any changes to old code.

3.2 Protocol

The Ingen protocol itself is very similar. Messages are built from LV2 Atoms [Robillard, 2014], particularly "Object"[2] which is a dictionary with URI keys and any type of value.

[2]This is an "object" in the JSON sense, not as in object-oriented programming.

The LV2 Patch extension defines several messages, similar to HTTP and DAV methods, which can be used to access and manipulate the graph. The simplest is a `Get`, which requests a description of the given subject:

```
[
  a patch:Get ;
  patch:subject </osc> ;
]
```

The response describes the subject in the same format, in this case the plugin instance, or *block*:

```
</osc>
    a ingen:Block ;
    lv2:prototype <urn:someplugin> ;
    ingen:canvasX 42.0 ;
    ingen:canvasY 24.0 .
```

Manipulation is similar. For example, a `Put` message can be used to create the above block:

```
[
  a patch:Put ;
  patch:subject </osc> ;
  patch:body [
    a ingen:Block ;
    lv2:prototype <urn:someplugin> ;
    ingen:canvasX 42.0 ;
    ingen:canvasY 24.0 ;
  ]
]
```

Syntactically, this says "I am a `Put` message, with subject `/osc`, and body `[ a ingen:Block ...]`". The definition of `patch:Put` and the associated properties gives us the meaning: "*put* this block at `/osc`".

The short names here are abbreviations of URIs, for example, `patch:Put` expands to `http://lv2plug.in/ns/ext/patch#Put`. URIs are used here to provide a global namespace, but when properly documented, also provide transparency. For example, the above URI leads to documentation which describes the meaning of a `patch:Put`. This documentation is also machine readable to support intelligent tools. For example, a `patch:Put` *must* have one `patch:subject` property, and the same tools used for LV2 plugin validation can ensure this restriction is obeyed. Note, however, that no Internet access is involved in handling messages; properly documenting URIs is simply a best practice for convenience and tool support.

There are similar messages to delete elements, set properties (including control values), and so on. All messages are defined in the LV2 Patch extension, which is also used by some plugins for control (for example, the LV2 example sampler uses this vocabulary to load samples).

3.2.1 Serialisation

Conceptually, Ingen uses the same protocol everywhere. However, the above text serialization would only be used over a network, or shown for debugging purposes. When running in the same process, messages are instead serialised as binary LV2 atoms for increased performance. These two encodings are conceptually identical and differ only in representation. Similarly, plugins inside Ingen which communicate with atoms are connected directly, with no serialisation.

The same textual serialisation used in the remote protocol is used when saving graphs. Conceptually, the Ingen protocol can be considered a stream of patches to the saved graph (hence the name of the LV2 Patch extension). For example, the description of the `/osc` block returned by the server in Section 3.2 could be a literal snippet of a saved graph file. Ports use the standard LV2 vocabulary, so Ingen graphs can be loaded by applications with LV2 support, with no special Ingen support required.

3.3 Event Handling

Building and manipulating a graph of plugins requires operations that are not real-time safe. To allow live editing without dropouts, Ingen must avoid all such operations (such as memory allocation or mutex locking) in the audio thread.

Conveniently, message-based control lends itself to an event-oriented implementation, which makes for an elegant solution to this problem. All operations in Ingen are implemented as events which are triggered by the receipt of some message. An event has three phases:

1. Pre-Process: Upon receipt of the message, perform any non-real-time operations necessary before the change can be applied (e.g. instantiate a plugin). When finished, push the event into a queue for the audio thread.

2. Process: In the audio thread, apply the changes prepared in the pre-process stage (e.g. insert an instantiated plugin into a graph). After this, the change is effectively complete. When finished, push the event (including references to any resources that need to be freed) into a queue for post-processing.

3. Post-Process: Clean up any necessary resources, and broadcast the change to all clients.

4 Examples

The most straightforward use of a modular is to build chains of effects plugins. Though simple, even this provides an improvement over what is easily achievable in hosts with a strictly linear signal path. For example, processing the left and right channels separately in a DAW like Ardour can be achieved this way without complicating the session's bus routing.

More interesting is to custom-build instruments. Figure 1 shows an example of an extremely simple polyphonic synthesizer, with only one envelope, saw oscillator, and low pass filter.

Ingen allows graphs to be nested, and has no restrictions on the type or number of ports present. For example, Figre 2 demonstrates adding sidechain compression to a synthesizer graph.

It can be useful to combine existing high-level plugins with more low level components. For example, Figure 3 shows a graph which contains multiple instruments. A MIDI filter [Gareus, 2014] plugin is used to send automatic chords to an electric piano, while the input note is sent to a synthesizer.

5 Future Directions

Ingen is currently useful as an environment for hosting plugins with flexible routing. Its architecture allows it to function in a diverse range of environments, which has been the focus of development to date.

One goal for future development is to become a more powerful programming environment. Since plugins are free to communicate with arbitrary messages, the necessary infrastructure is already available, but an appropriate set of plugins is missing. Existing systems like Max/MSP and Pd are very mature in this respect, but use a different model than Ingen and LV2. In particular, it will be interesting to investigate how to exploit *meaningful* messages to provide a powerful modular programming environment.

References

Robin Gareus. 2014. midifilter.lv2. https://github.com/x42/midifilter.lv2.

Matthias Nagorni. 2003. Alsa Modular Synth. http://alsamodular.sourceforge.net/.

Miller Puckette. 1996. Pure Data: Another integrated computer music environment. *Proceedings of the Second Intercollege Computer Music Concerts*, pages 37–41.

David Robillard. 2014. LV2 Atoms: A data model for real-time audio plugins. In *Linux Audio Conference 2014*.

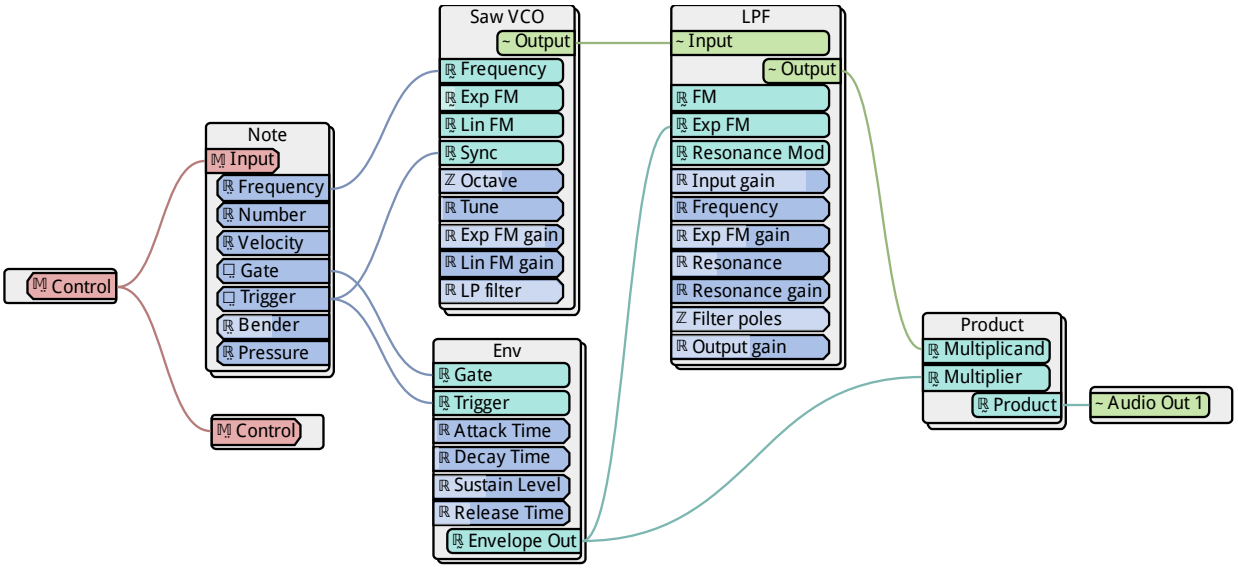

Figure 1: A simple polyphonic synthesizer.

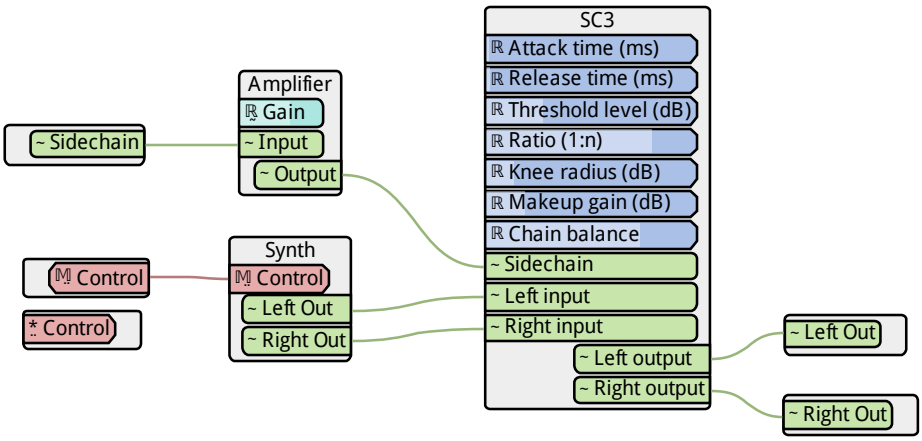

Figure 2: Adding sidechain compression to a synthesizer. The “Synth” block shown here is a nested Ingen graph (which can be edited by double-clicking in the interface).

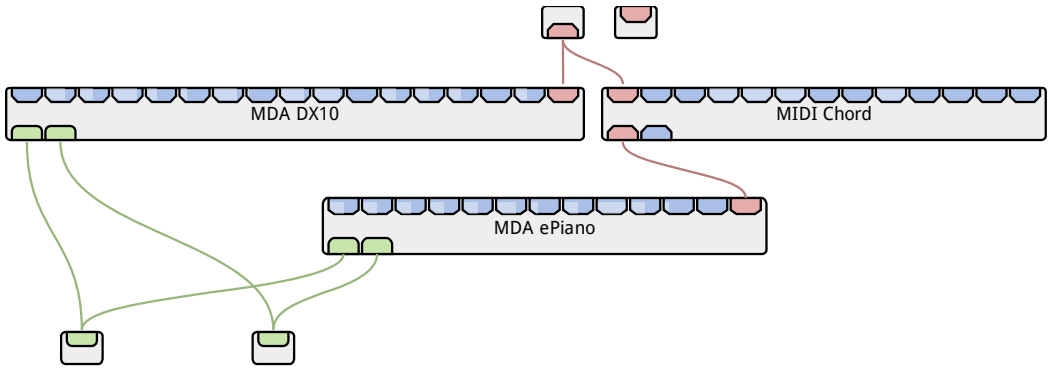

Figure 3: Simple MIDI processing to play the input root note on a DX10, and accompanying chords on an ePiano. Shown in vertical mode.