PdGst - GStreamer bindings for Pd

IOhannes m zmölnig
Institute of Electronic Music and Acoustics
University of Music and Dramatic Arts
Graz, Austria
zmoelnig@iem.at

ABSTRACT

In this paper we present *PdGst*, a language binding of the popular *GStreamer* framework to the Pd-world. On the Pd side, this enables simple handling of multi-threaded media-pipelines. On the GStreamer side, the benefits lay mainly in the ease of construction of new pipelines within the patcher paradigm and in the simplicity of control of and interaction with GStreamer (GST) elements within the Pure data environment.

Keywords

Pure data, GStreamer, frameworks, multimedia

1. MOTIVATION

Pure data (including it's various extensions) claims to be useable as a linear production tool. This claim holds true in Pd's core domain, live audio processing (with the main domain-specific interface to the real world being the sound card). To a certain extent, this claim holds true for different domains like live video processing as well, as long as the set of interfaces to the real world is confined to "live" interfaces, e.g. video capture cards and VGA output.

However, things start to become more complicated when the interfaces are wrapped into containers, as is usual with files and (worse) with streaming media.

Interfacing with files is - to a certain extent - made easy on proprietary platforms like Mac-OS or W32, where the operating system provides a unified API for accessing this data. Unfortunately on free operating systems like Linux no such unified API has established itself yet, and few of the existing APIs are supported by the various Pd extensions.

Interfacing with more complex media like network streams is barely supported at all, no matter which platform.

While there are solutions for pushing network streams, these seem to be unstable and expose inconsistent interfaces to the user. In practice, the authors became extremely frustrated when trying to create either multichannel audio streams (e.g. a 10-channel ogg/vorbis stream using the *pdogg*-library[3], which works well enough for stereo streams) or a simple ogg/theora video stream (using `[pdp_theonice~]` which is part of the *PiDiP*-library[2]).

Due to approaching deadlines, no time was invested into debugging and fixing the existing objects, but instead the streaming was re-written from scratch with the apparently stable GStreamer[5] framework.

Since GStreamer lacks an interface like Pd for real time interaction, the need for a bridge between these two (similar) worlds was recognised.

2. INTRODUCTION

"GStreamer is a framework for creating streaming media applications" [4]. It is a high-performance, heavily multi-threaded, cross-platform framework that is based on graphs of low-level media-handling components, allowing application developers to create applications that read media-streams from virtually any source (devices, files, network-streams, generators,...), apply transformations on them, demultiplex and multiplex them and finally output these streams to virtually any sink (devices, files, network-streams,...).

This makes it very similar to Pure data and other members of the Max-family. (However, Pd and friends have a strong focus on audio processing. With various extensions (e.g. Gem, pdp,...), Pd can also handle streams of other media types.) These streams have little to do with each other: for instance it is not possible, to connect a pdp packet stream to a `[dac ]` object (which makes sense). However, it is also not possible to multiplex several media streams together into a new stream. Instead, streams have to be separated at the source-objects, kept separately for processing and only at the sink object, streams can be made into a single stream again. For an example see Fig.1: the multi-modal stream contained in a movie-file has to be demultiplexed into it's components (#1: video (pdp) stream; #2 audio stream (left); #3 audio stream (right)) by the source object. The components are then transformed individually (e.g. delayed). Finally the separate streams are multiplexed into a network-stream in the sink object.

This of course offers great flexibility, as all streams can be transformed differently and independent of each other.

GStreamer on the other side, is totally media agnostic: It doesn't know (nor care), which streams flow from one graph-node to the next (see Fig.2).

Since connecting arbitrary stream sources to arbitrary stream sinks does not make sense most of the time, GStreamer provides a mechanism to (automagically, if desired) negotiate which sub-streams are eventually passed from one node

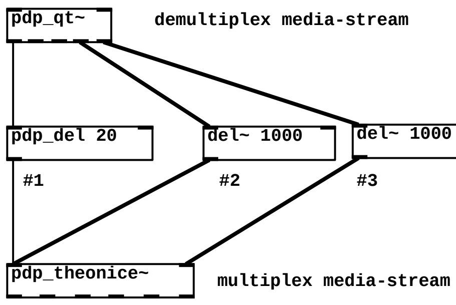

Figure 1: a multi-modal stream has to be split into separate streams within Pd

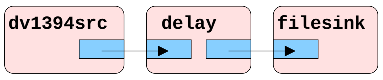

Figure 2: a multi-modal stream in GStreamer, containing 1 video stream and 2 audio streams

to another (e.g. a sink interfacing the sound card will only not be interested in the video portion of a multi-modal stream). An example is given in Fig.3

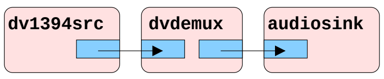

Figure 3: GStreamer auto-negotiation: `audiosink` will only receive the audio-substream

This comes at the cost of relatively low level interaction with media-streams. E.g. for reading and displaying an ogg/vorbis file, the pipeline has to deal with ogg-streams (as read from the file), vorbis-streams (as demultiplexed from the ogg-stream) and audio-streams (as decoded from the vorbis-stream). These various streams are not hidden from the user (unlike in Pd, where the user just sees one or several audio-streams coming out of the `[oggamp~]` object).

3. GSTREAMER

3.1 Elements & Pads

A basic GST pipeline consists of so called *element*s, which is analogous to a Pd *object*.

An element can have any number of *pad*s to send and receive streams from other elements. A pad always has a direction: *source pad*s are used to send data from the element (therefore being an equivalent of an *outlet* in Pd), and *sink pad*s receive data (an *inlet* in Pd lingo).

In Pd, an object's in- and outlets are defined at instantiation time. In GStreamer, this need not be the case: a *dynamic* pad can appear and vanish at runtime. For instance, a movie file might contain only a video track, or a video track and an audio track. How many streams are present will only be known after a certain file has been read by a previously instantiated element. The element will therefore create (and destroy) available source pads as the respective streams become available.

3.2 Bins

A number of (interconnected) elements can be contained within a so-called *bin*. Bins can then be re-used like ordinarily elements. This corresponds to Pd's concept of abstractions.

3.3 Communication

In order to communicate with an element, two mechanisms are used by GStreamer: properties and signals.

Properties are quasi-static states of the element. They can be used e.g. to tell a file-reading element *filesrc*, which file it should read. All properties are a key/value-pair, where the key is a symbolic name. The value can be an arbitrary complex structure, the type of which can be queried at runtime.

Properties can be queried or set or both, depending on their nature, at any time in the live of an element. For instance, the number of (dynamic) pads an element currently has, is a read-only property, which may have different values depending on the currently processed stream.

Since GST elements are heavily multi-threaded, a mechanism is needed in order to communicate with the parent application (which is potentially thread-unaware - like Pd). GStreamer implements this by using *signals* which are sent to a *bus*. The parent application can then poll this bus to see, whether an element has emitted a signal, and distribute any available messages within it's own context.

Signals are used for instance to tell the parent application that the End-Of-Stream has been reached, or to dynamically inform the application about embedded meta-data (e.g.: author of a song).

4. LANGUAGE BINDINGS FOR PD

Since Pd and GStreamer are conceptually very similar, creating Pd bindings for GStreamer is rather straightforward.

4.1 Elements are objects

Each GST element gets mapped to a corresponding Pd object Due to the dynamic availability of GST elements, this is done by means of a *sys_ loader*. Since pads in GStreamer can be dynamic and appear/disappear randomly, they do not map to Pd's in-/outlets so well. It was therefore decided to create a single outlet for *all* source pads of an element (and vice-versa for inlets and sink pads).

Which pads will actually be connected between two elements is negotiated at runtime, by a mechanism GStreamer calls *caps negotiation*. *caps* is an abbreviation for *capabilities*, and is used to describe the streams a certain pad can generate (if it is a source pad) or accept (if it is a sink pad). Since this is sometimes ambiguous, it is possible to restrict the possibilities by using so called *capsfilters*.[1]

GST bins have their natural equivalent in Pd's subpatches and abstractions.

4.2 Controlling GStreamer

[1]capsfilters are built into GStreamer. PdGst uses them in order to circumvent the problem of dynamic pads.

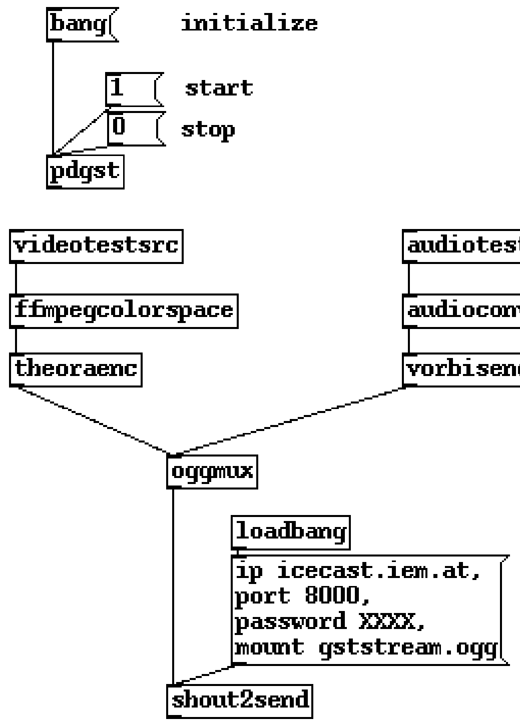

Figure 4: a simple PdGst patch that sends an ogg/theora stream to an icecast2 server

In order to control the entire GST graph within a bin, the object [pdgst] is introduced. The main purpose from the users point of view of this object is to initiate the building of the graph and to turn on/off it's execution.

It is therefore similar to Gem's [gemwin] object, which controls (apart from handling the the rendering window itself) the building and execution of Gem's render-graph.

It is also connected to the bin's bus in order to catch general signals that are not caught by specific elements/objects (see 4.3).

Finally, GStreamer allows some more operations on the entire bin, which is controlled by [pdgst] as well, e.g. exporting and importing the graph to/from XML-files.

4.3 Controlling elements

The main way to change the behaviour of an element, is setting it's properties. Therefore, each PdGst object will register methods for each property-key, which can be used to both set and query that value of a certain property. E.g. each element has a property "name". Sending a message [name((without arguments) to the object, will make the object emit a message [property name myname((if the value of "name" is currently "myname"). Setting a property can be done by specifying a new value in the message: [name newname(will set the value of "name" to "newname" (if this is possible).

Signals on the GST bus directed to a GST element, are forwarded to the the appropriate PdGst object. E.g. the GST element *id3demux* will parse an mp3 stream for ID3 tags. If it encounters such a tag, it will emit a signal named "tag", holding the contents of the tag. The corresponding PdGst object [id3demux] will catch this signal and output a message like `tag author Michael Jackson`. Note that strings will be converted into a single symbol, e.g. "Michael Jackson" will become a Pd symbol with a space!

4.4 Pd Streams

PdGst as described so far, only allows building and controlling GST pipelines within Pd. It does not offer any methods for adding Pd-native streams (e.g. audio-signals, pdp packets, Gem pixes) into the pipeline nor for extracting streams from the pipeline yet.

For this, we introduce several bridging objects into the various domains:

- [pdgst_in~], [pdgst_out~]
- [pix_gstin], [pix_gstout]
- [pdp_gstin], [pdp_gstout]

For an example patch that connects GST media-streams to Pd/Gem see Fig.5.

4.5 Internal communication

PdGst objects need to communicate with each other without interfering with user-generated messages. There two distinct ways to do this:

- use Pd's messaging system (e.g. connections) with a reserved selector
- create a shadow copy of the graph as expressed within Pd with a separate message-bus that does not interfere with Pd's

For the sake of simplicity and in order to be able to use Pd's message-routing system (e.g. [spigot]), we chose to use the (henceforth) reserved keyword "__gst" in order to send messages from one PdGst object to the other. This means that the user is able to interfere with the internals of PdGst, although the chances are rather low that this will happen accidentally.

5. STATUS

The implementation of PdGst has been broken into several phases, in order keep motivation of the developers high.

At the time of writing, *Phase 1* is in the bug-fixing stage and nears completion. We expect that at least *Phase 3* will be completed by the time of the PdCon09.[2]

5.1 Phase 1

Implementation of the basic GStreamer to Pd mapping. Creating and running simple pipelines is possible. Each GST element has a corresponding Pd objectclass of the same name.

5.2 Phase 2

More advanced features of pipelines, like creation of bins as (sub)patches.

[2]However, phases are not necessarily dependent on phases with a lower ordinal number. *Phase 3* can therefore be completed before *Phase 2* has even started.

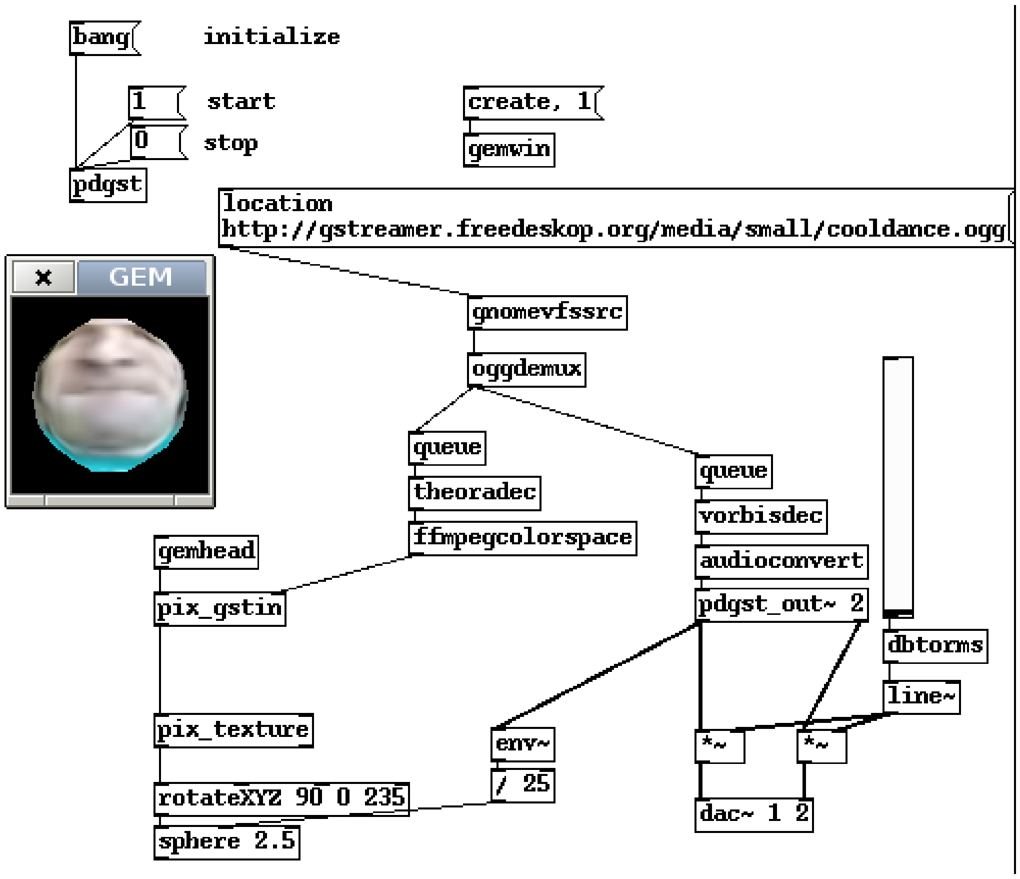

Figure 5: using GST streams within Pd/Gem (mockup)

5.3 Phase 3

Phase 3 adds objects that live in both the GStreamer and the Pd world media-wise. This allows to directly and conveniently add (e.g.) audio-streams generated in (plain) Pd into the GST pipeline. (Before, this was only possible using plumbing frameworks such as jack [1].)

5.4 Phase 4

GStreamer has built-in support for importing and exporting static pipelines via XML-files. There is also a syntax to express very simple pipelines as human-readable text (e.g. the pipeline in Fig.2 can be written as "dv1394src ! delay ! filesink"). While GStreamer can be used as-is for exporting a PdGst pipeline to an XML-interchange format, there might be a need to import text-based descriptions as a patch. A Pd2XML converter might have other merits as well.

5.5 Phase 5

Up till now, PdGst relies on the `[pdgst]` control object, for building and running a pipeline. From a user's point of view, it is desirable to be able to run a pipeline without such an extra housekeeping object, and instead run a pipeline by simply activating it's sources. The control object might still be needed for secondary tasks (like XML import/export).

6. CONCLUSIONS

We have introduced PdGst, a binding of the GStreamer framework to Pure data.

PdGst adds the power of a high-performance, multi-threaded media-agnostic framework to Pure data, while at the same time retaining the flexibility and interactivity of Pd.

From the GStreamer point-of-view, PdGst adds the possibility to easily create pipelines within the patcher paradigm.

7. ACKNOWLEDGEMENTS

We would like to thank Georg and Thomas Holzmann, who - unsuccessfully - tried to persuade us of the virtues of GStreamer during their *Google Summer of Code 2007* Project. We would also like to thank Yves Degoyon and Lluis Gomez i Bigorda, who brought back GStreamer into our lives in early 2009. Finally, a special thanks goes to the GStreamer community available at their IRC-channel[3], who has been very helpful in getting PdGst running.

8. REFERENCES

[1] P. Davis. The jack audio connection kit. In *Proc. of the Linux Audio Developer Conference*. ZKM Karlsruhe, 2003.

[2] Y. Degoyon, L. Gomez i Bigorda, and T. de la O. Pidip is definitely in pieces. `http://pure-data.svn.sourceforge.net/viewvc/pure-data/trunk/externals/pidip/`, 2002.

[3] `irc://freenode.org/gstreamer`

[3] O. Matthes. pdogg - a collection of ogg/vorbis externals for pd. `http://pure-data.svn.sourceforge.net/viewvc/pure-data/trunk/externals/pdogg/`, 2002.

[4] W. Taymans, S. Baker, A. Wingo, R. S. Bultje, and S. Kost. *GStreamer application development manual (0.10.22)*, 2009.

[5] G. team. Gstreamer: open source multimedia framework. `http://www.gstreamer.net/`, 2001.