

GStreamer Application Development Manual

Wim Taymans

GStreamer Application Development Manual

by Wim Taymans

This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v1.0 or later (the latest version is presently available at <http://www.opencontent.org/openpub/> (<http://www.opencontent.org/openpub/>))

Table of Contents

I. Overview.....	5
1. Introduction.....	6
1.1. What is GStreamer?	6
2. Motivation	7
2.1. Current problems	7
2.1.1. Multitude of duplicate code	7
2.1.2. 'One goal' media players	7
2.1.3. Non unified plugin mechanisms.....	7
2.1.4. Provision for network transparency	8
2.1.5. Catch up with the Windows(tm) world	8
3. Goals	9
3.1. The design goals	9
3.1.1. Clean and powerfull	9
3.1.2. Object oriented.....	9
3.1.3. Extensible.....	10
3.1.4. Allow binary only plugins	10
3.1.5. High performance	10
II. Basic concepts.....	12
4. GstElement.....	13
4.1. What is a GstElement.....	13
4.1.1. GStreamer source elements.....	13
4.1.2. GStreamer filter elements	14
4.1.3. GStreamer sink elements	15
4.2. Creating a GstElement	15
5. What are Plugins	17
6. GstPad	19
6.1. Getting pads from an element	19
6.1.1.	19

List of Figures

4-1. Visualisation of a source element	13
4-2. Visualisation of a filter element	14
4-3. Visualisation of a filter element with more than one output pad	14
4-4. Visualisation of a sink element	15

I. Overview

Table of Contents

1. Introduction.....	6
2. Motivation.....	7
3. Goals.....	9

The first chapter of the book gives you an overview of GStreamer design goals. Chapter 2 rapidly covers the basics of GStreamer programming. In chapter 3 we will move on to the examples. Since GStreamer adheres to the GTK+ programming model, the reader is assumed to understand the basics of GTK+. For a gentle introduction to GTK+, you may wish to read the *GTK+ Tutorial* or Eric Harlow's book *Developing Linux Applications with GTK+ and GDK*.

Chapter 1. Introduction

This chapter gives you an overview of the technologies described in this book.

1.1. What is GStreamer?

GStreamer is a framework for creating streaming media applications. The fundamental design comes from the video pipeline at Oregon Graduate Institute, as well as some ideas from DirectShow.

GStreamer's development framework makes it possible to write any streaming multimedia application. The framework includes several components to build a full featured media player capable of playing MPEG1, MPEG2, AVI, MP3, WAV, AU, ...

GStreamer, however, is much more than just another media player. Its main advantages are that the pluggable components also make it possible to write a full fledged video or audio editing application.

The framework is based on plug-ins that will provide the various codec and other functionality. The plugins can be connected and arranged in a pipeline. This pipeline defines the flow of the data. Pipelines can also be edited with a GUI editor and saved as XML so that pipeline libraries can be made with a minimum of effort.

This book is about GStreamer from a developer's point of view; it describes how to write a GStreamer application using the GStreamer libraries and tools.

Chapter 2. Motivation

Linux has historically lagged behind other operating systems in the multimedia arena. Microsoft's Windows[tm] and Apple's MacOS[tm] both have strong support for multimedia devices, multimedia content creation, playback, and realtime processing. Linux, on the other hand, has a poorly integrated collection of multimedia utilities and applications available, which can hardly compete with the professional level of software available for MS Windows and MacOS.

2.1. Current problems

We describe the typical problems in today's media handling on Linux.

2.1.1. Multitude of duplicate code

The Linux user who wishes to hear a sound file must hunt through their collection of sound file players in order to play the tens of sound file formats in wide use today. Most of these players basically reimplement the same code over and over again.

The Linux developer who wishes to embed a video clip in their application must use crude hacks to run an external video player. There is no library available that a developer can use to create a custom media player.

2.1.2. 'One goal' media players

Your typical MPEG player was designed to play MPEG video and audio. Most of these players have implemented a complete infrastructure focused on achieving their only goal: playback. No provisions were made to add filters or special effects to the video or audio data.

If I wanted to convert an MPEG2 video stream into an AVI file, my best option would be to take all of the MPEG2 decoding algorithms out of the player and duplicate them

into my own AVI encoder. These algorithms cannot easily be shared accross applications.

2.1.3. Non unified plugin mechanisms

Your typical media player might have a plugin for different media types. Two media players will typically implement their own plugin mechanism so that the codecs cannot be easily exchanged.

The lack of a unified plugin mechanism also seriously hinders the creation of binary only codecs. No company is willing to port their code to all the different plugin mechanisms.

While GStreamer also uses it own plugin system it offers a very rich framework for the plugin.

2.1.4. Provision for network transparency

No infrastructure is present to allow network transparent media handling. A distributed MPEG encoder will typically duplicate the same encoder algorithms found in a non-distributed encoder.

No provisions have been made for emerging technologies such as the GNOME object embedding using BONOBO.

2.1.5. Catch up with the Windows(tm) world

We need solid media handling if we want to see Linux succeed on the desktop.

We must clear the road for commercially backed codecs and multimedia applications so that Linux can become an option for doing multimedia.

Chapter 3. Goals

GStreamer was designed to provide a solution to the current Linux media problems.

3.1. The design goals

We describe what we try to achieve with GStreamer.

3.1.1. Clean and powerfull

GStreamer wants to provide a clean interface to:

-

The application programmer who wants to build a media pipeline. The programmer can use an extensive set of powerfull tools to create media pipelines without writing a single line of code. Performing complex media manipulations becomes very easy.

-

The plugin programmer. Plugin programmers are provided a clean and simple API to create self contained plugins. An extensive debugging and tracing mechanism has been integrated. GStreamer also comes with an extensive set of real-life plugins that serve as an example too.

3.1.2. Object oriented

Adhere as much as possible to the GTK+ object model. A programmer familiar with GTK+ will be confortable with GStreamer.

GStreamer uses the mechanism of signals and object arguments.

All objects can be queried at runtime for their various properties and capabilities.

3.1.3. Extensible

All GStreamer Objects can be extended using the GTK+ inheritance methods.

All plugins are loaded dynamically and can be extended and upgraded independently.

3.1.4. Allow binary only plugins

plugins are shared libraries that are loaded at runtime. since all the properties of the plugin can be set using the GObject arguments, there is no need to have any header files installed for the plugins.

Special care has been taken into making the plugin completely self contained. This is in the operations, specification of the capabilities of the plugin and properties.

3.1.5. High performance

High performance is obtained by:

- Using glib g_mem_chunk where possible to minimize dynamic memory allocation.
- Connections between plugins are extremely light-weight. Data can travel the pipeline with minimal overhead.
- Provide a mechanism to directly work on the target memory. A plugin can for example directly write to the X servers shared mem. Buffers can also point to arbitrary memory like kernel memory.
-

RefCounting and copy on write to minimize the amount of memcpy. Subbuffers to efficiently split the data in a buffer.

-

Pipelines can be constructed using cothreads to minimize the threading overhead. Cothreads are a simple user-space method for switching between subtasks.

-

HW acceleration is possible by writing a specialized plugin.

-

Uses a plugin registry with the specifications of the plugins so that the plugin loading can be delayed until the plugin is actually used.

II. Basic concepts

Table of Contents

4. GstElement	13
5. What are Plugins.....	17
6. GstPad.....	19

We will first describe the basics of the GStreamer programming by introducing the different objects needed to create a media pipeline.

We will use a visual representation of these objects so that we can visualize the more complex pipelines you will learn to build later on.

Chapter 4. GstElement

The most important object in GStreamer for the application programmer is the GstElement object.

4.1. What is a GstElement

The GstElement is the basic building block for the media pipeline. All the different components you are going to use are derived from this GstElement. This means that a lot of functions you are going to use operate on this object.

You will see that those elements have pads. These are the elements connections with the 'outside' world. Depending on the number and direction of the pads, we can see three types of elements: source, filter and sink element.

These three types are all the same GstElement object, they just differ in how the pads are.

4.1.1. GStreamer source elements

This element will generate data that will be used by the pipeline. It is typically a file or an audio source.

Below you see how we will visualize the element. We always draw a src pad to the right of the element.

Figure 4-1. Visualisation of a source element

Source elements do not accept data, they only generate data. You can see this in the figure because it only has a src pad. A src pad can only generate buffers.

4.1.2. GStreamer filter elements

Filter elements both have an input and an output pad. They operate on data they receive in the sink pad and send the result to the src pad.

Examples of a filter element might include: an MPEG decoder, volume filter,...

Filters may also contain any number of input pads and output pads. For example, a video mixer might have two input pads (the images of the two different video streams) and one output pad.

Figure 4-2. Visualisation of a filter element

The above figure shows the visualisation of a filter element. This element has one sink pad (input) and one src (output) pad. Sink pads are drawn on the left of the element.

Figure 4-3. Visualisation of a filter element with more than one output pad

The above figure shows the visualisation of a filter element with more than one output pad. An example of such a filter is the AVI splitter. This element will parse the input data and extracts the audio and video data. Most of these filters dynamically send out a signal when a new pad is created so that the application programmer can connect an arbitrary element to the newly created pad.

4.1.3. GStreamer sink elements

This element accepts data but will not generate any new data. A sink element is typically a file on disk, a soundcard, a display,... It is presented as below:

Figure 4-4. Visualisation of a sink element

4.2. Creating a GstElement

GstElements are created from factories. To create an element, one has to get access the a `GstElementFactory` using a unique factoryname.

The following code example is used to get a factory that can be used to create the mpg123 element, an mp3 decoder.

```
GstElementFactory *factory;  
  
factory = gst_elementfactory_find ("mpg123");
```

Once you have the handle to the elementfactory, you can create a real element with the following code fragment:

```
GstElement *element;  
  
element = gst_elementfactory_create (factory, "decoder");
```

`gst_elementfactory_create ()` will use the elementfactory to create an element with the given name. The name of the element is something you can use later on to lookup the element in a bin, for example.

A simple shortcut exists for creating an element from a factory. The following example creates an element, named "decoder" from the elementfactory named "mpg123". This convenient function is most widely used to create an element.

```
GstElement *element;  
  
element = gst_elementfactory_make ("mpg123", "decoder");
```

An element can be destroyed with:

```
GstElement *element;  
  
...  
gst_element_destroy (element);
```

Chapter 5. What are Plugins

A plugin is a shared library that contains at least one of the following items:

- one or more elementfactories
- one or more typedefinitions

The plugins have one simple method: `plugin_init ()` where all the elementfactories are created and the typedefinitions are registered.

the plugins are maintained in the plugin system. Optionally, the typedefinitions and the elementfactories can be saved into an XML representation so that the plugin system does not have to load all available plugins in order to know their definition.

The basic plugin structure has the following fields:

```
struct _GstPlugin {  
    gchar *name;                /* name of the plugin */  
    gchar *longname;            /* long name of plugin */  
    gchar *filename;            /* filename it came from */  
  
    GList *types;               /* list of types provided */  
    gint numtypes;  
    GList *elements;            /* list of elements provided */  
    gint numelements;  
  
    gboolean loaded;            /* if the plugin is in memory */  
};
```

You can query a GList of available plugins with:

```
GList *plugins;
```

```
plugins = gst_plugin_get_list ();

while (plugins) {
    GstPlugin *plugin = (GstPlugin *)plugins->data;

    g_print ("plugin: %s\n", gst_plugin_get_name (plugin));

    plugins = g_list_next (plugins);
}
```

Chapter 6. GstPad

As we have seen in the previous chapter (GstElement), the pads are the elements connections with the outside world.

The specific type of media that the element can handle will be exposed by the pads. The description of this media type is done with capabilities (GstCaps)

6.1. Getting pads from an element

Once you have created an element, you can get one of its pads with:

```
GstPad *srcpad;  
...  
srcpad = gst_element_get_pad (element, "src");  
...
```

This function will get the pad named "src" from the given element.

Alternatively, you can also request a GList of pads from the element. The following code example will print the names of all the pads of an element.

```
GList *pads;  
...  
pads = gst_element_get_pad_list (element);  
while (pads) {  
    GstPad *pad = GST_PAD (pads->data);  
  
    g_print ("pad name %s\n", gst_pad_get_name (pad));  
  
    pads = g_list_next (pads);  
}  
...
```

6.1.1.

