Gstreamer Application Development

for GU4DEC (June 16th-18th, 2003, Dublin)

by Ronald Bultje <rbultje@ronald.bitfreak.net>
May 12th, 2003

Contents

Introduction

GStreamer is a media framework that allows developers to easily write media applications. Examples of these include video/audio players (Totem & Rhythmbox), recorders (gnome-sound-recorder, part of Gnome-Media), CD rippers (Sound-Juicer), etcetera. Other, currently not yet used, applications include button-sounds in Gnome (for accessibility (a11y) purposes), audio playback in games, cd playback, metadata editing, media transcoding (like Virtualdub) and much more. Basically, anything having to do with media can be built using GStreamer.

Most questions directed at our team don't have to do with missing features or anything. They are mostly questions like “how do I build a $random_application using GStreamer” and those kind of questions. The GStreamer team supports the view that building applications using GStreamer can be hard at first sight, since there is a lot to learn in a totally new library. In order to support GStreamer-based application development, this paper is supposed to give a general overview of what interfaces are available to build GStreamer-based applications, what features they provide (and miss), so people can use GStreamer in their own applications too.

The next chapters will describe four interfaces for accessing GStreamer's features. Most of it will focus on building command-line applications (for the case of the example, this is the easiest way to explain), but there will be hints for Gnome-applications, too. We will start with the native C interface, we will then step over to a graphical editor and its XML files. Lastly, we will support two higher-level libraries, monkey-media and libgstplay.

General concept – C examples

The most basic form of writing applications with the GStreamer library is by using the C interface. Beware, this interface is definitely not for the faint-hearted. GStreamer is not like most media-interfaces, which can be used with calls like:

```
library_play_file (const char *filename);
```

GStreamer is not aimed at playback specifically, and does not aim to provide such an interface in its own core. Application developers will have to think in terms of elements that have one specific function, for example reading a file (filesrc), decoding audio (vorbisfile) or outputting audio data to the sound card (e.g. osssink). For the higher-level interfaces, libgstplay is probably a good place to start (which has its own chapter later on in this paper).

Elements and pads

As said before, elements have one specific function. In order to play a file (or whatever one would want to do using GStreamer), developers have to create the correct elements, link them together (which means that they stream data to each other), and iterate the resulting set of elements.

Pads are ways of inputting or outputting data in GStreamer. A file source, for example, would have one pad through which the file data is pushed over to the next element that is linked to this pad. An audio decoder would then have two pads: one through which data is coming in from the element earlier on (the file source, for example), and one on which the decoded audio data is moved over to the next element. The audio output would then have one pad, through which data is pushed in which has to be played back using the sound card. See figure 1 for details.

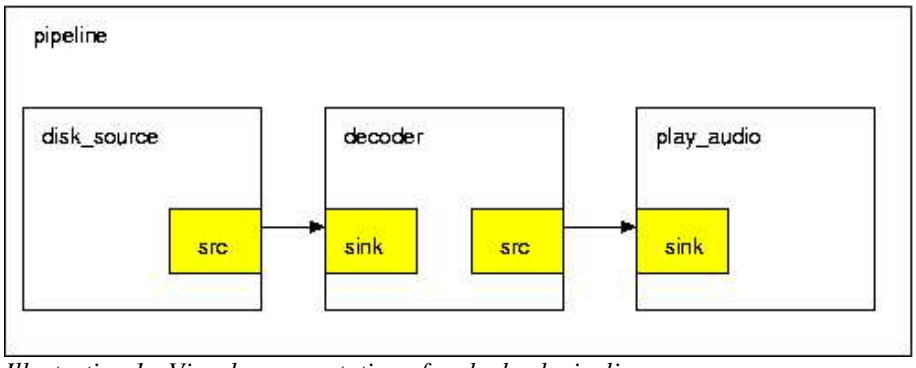

Illustration 1 - Visual representation of a playback pipeline

There are two types of pads, sink and source pads. Source pads are pads through which data is pushed out to other elements. Sink pads are pads through which data comes in to the element. Each element can have an indefinate number of sink or source pads. Similarly, there are source and sink *elements*. Source elements are elements that only output data (such as file readers or video/audio device plugins). Sink elements are elements that only receive data (such as sound card output plugins).

C example

Let us now try to build a simple media player using the C interface. As said before, playback of a file requires (at least) three elements: a file reader (filesrc), a decoder (vorbisfile) and a soundcard output (osssink). They will be linked together, and the resulting pipeline will be iterated until playback is finished.

```
#include <gst/gst.h>

int
main (int argc, char *argv[])
{
  GstElement *pipeline, *filesrc, *decoder, *audiosink;

  gst_init(&argc, &argv);

  if (argc != 2) {
    g_print ("usage: %s <vorbis filename>\n", argv[0]);
    exit (-1);
  }

  /* create a new pipeline to hold the elements */
  pipeline = gst_pipeline_new ("pipeline");

  /* create a disk reader */
  filesrc = gst_element_factory_make ("filesrc", "disk_source");
  g_object_set (G_OBJECT (filesrc), "location", argv[1], NULL);

  /* now it's time to get the decoder */
  decoder = gst_element_factory_make ("vorbisfile", "decoder");

  /* and an audio sink */
  audiosink = gst_element_factory_make ("osssink", "play_audio");

  /* add objects to the main pipeline */
  gst_bin_add_many (GST_BIN (pipeline), filesrc, decoder, audiosink,
NULL);

  /* link src to sink */
  gst_element_link_many (filesrc, decoder, audiosink, NULL);

  /* start playing */
  gst_element_set_state (pipeline, GST_STATE_PLAYING);

  while (gst_bin_iterate (GST_BIN (pipeline)));
```

```
  /* stop the pipeline */
  gst_element_set_state (pipeline, GST_STATE_NULL);

  /* we don't need a reference to these objects anymore */
  gst_object_unref (GST_OBJECT (pipeline));

  exit (0);
}
```

The above example does a few things. Firstly, it creates a pipeline, which is a container of elements that can be iterated. Then, it creates three elements (filesrc – a file reader, vorbisfile – an Ogg/Vorbis decoder, osssink – an OSS output plugin), links them together and adds them to the pipeline. Next, it starts the pipeline and iterates it. Lastly, all references are removed and the memory is thus free'ed. Note that we currently use a `while ()` loop with `gst_bin_iterate ()` to iterate the pipeline. This keeps the application busy, so it isn't useful for Gnome-based applications (because they would be unresponsive while playing a file). Gnome-based applications would use `g_idle_add ()` to set an idle-handler. The idle-function would then call the `gst_bin_iterate ()` function:

```
static gboolean
idle_handler (gpointer data)
{
  GstElement *pipeline = GST_PIPELINE (data);

  return gst_bin_iterate (GST_BIN (pipeline));
}

int
main (int argc, char *argv[])
{
[..]
  g_idle_add ((GsourceFunc *) idle_handler, pipeline);

  gtk_main ();
[..]
}
```

Another possibility is to use threads. The main pipeline can be a thread; we do not need to iterate the thread, it will run out of itself after having been brought into the playing state. monkey-media uses this method.

Events, queries and more

For real graphical playback, there is much more on it than only this. Users want to know the progress of a file (usually a slider plus time value), want to be able to search through the stream, and metadata is important too. GStreamer does provide all this, too.

Elements that support metadata use the `metadata` property for this. Applications can then

use the following code to read metadata:

```
GstCaps *metadata;
g_object_get (G_OBJECT (element), "metadata", &metadata, NULL);
```

The metadata object contains properties such as "title" that describe the media.

Querying is used to request information about the status of the stream, such as the current position of the stream or the total length of the stream. The following example explains how to use that to give information about the progress of the playback (as a percentage of the whole file):

```
guint64 position, total;
GstPad *pad  = gst_element_get_pad (element, "src");
gst_pad_query (pad, GST_QUERY_TOTAL, GST_FORMAT_TIME, &total);
gst_pad_query (pad, GST_QUERY_POSITION, GST_FORMAT_TIME, &position);
g_print ("Current position: %d%%\n", 100 * position / total);
```

For seeking in a stream, GStreamer uses events. Events basically work exactly like queries. The following example should give a generic impression of how it works:

```
GstPad *pad = gst_element_get_pad (element, "src");
GstEvent *event = gst_event_new_seek (GST_SEEK_METHOD_SET, 0);
gst_pad_send_event (pad, event);
```

This will make the stream search back to the beginning of a file.

Conclusion

With all that has just been shown, one can already build a simple audio player. However, this means that each project that wants to use GStreamer for audio playback, will have to use a lot of similar calls, and for most purposes, the end code will be quite long. There are simpler ways to achieve this, which we will look at next. The above can be used as a generic introduction to the GStreamer framework.

Using graphical editors – XML examples

Just like with building graphical applications, quite a lot of people prefer graphical overviews of what they are doing. The biggest advantage of this is that it actually gives a better overview of the whole picture. Just like with building Gtk+-apps, it is easy to lose track. That is why GStreamer, just like Gtk+ does with Glade, has a graphical editor to represent pipelines. Pipelines can be exported to XML files, which can be loaded in any application. Let us first look at the graphical editor. An example of how to use XML files in applications will come after that.

The GStreamer Pipeline Editor

gst-editor allows for quick pipeline building. Its interface is similar to the Glade-way of working, so developers familiar with Glade will quickly get used to gst-editor as well. Building a pipeline as given in figure 2 is a matter of a few clicks and takes more or less a minute to build.

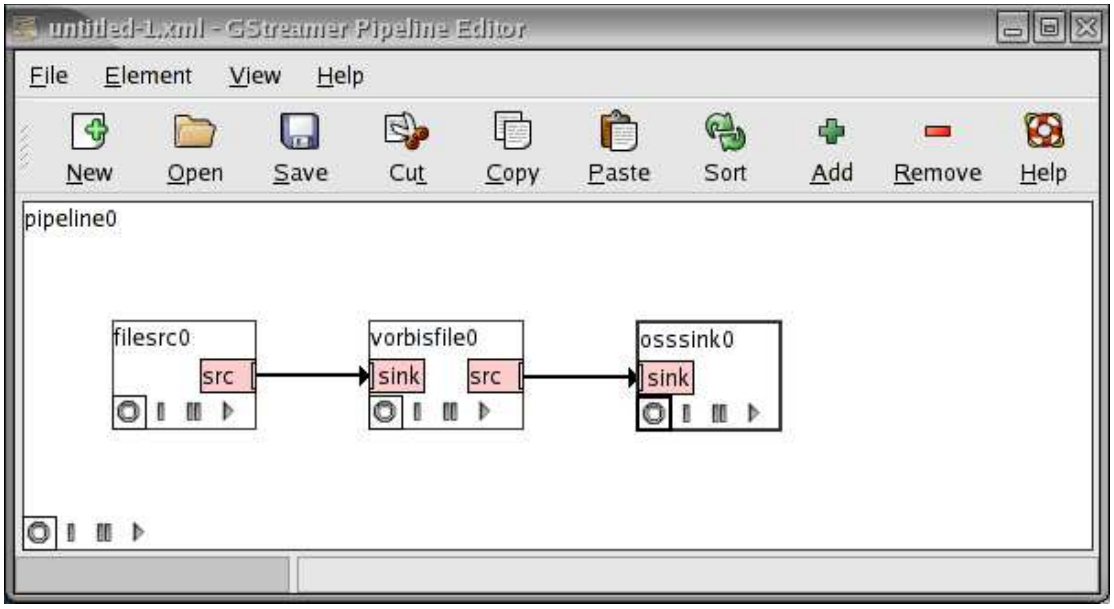

Illustration 2 - A pipeline built using the GStreamer Pipeline Editor

The above pipeline is very similar to the C pipeline that has been discussed in the previous chapter. More advanced pipelines are also possible, they will pop up later in this chapter. First, a simple start. The pipeline can be tested by pressing the "play" icon for pipeline0. Using `File -> Save`, the pipeline can be saved to a (XML) file, which can be loaded in applications.

Loading XML files into applications

GStreamer contains full XML load/save support. The pipeline XML files contain descriptions of each element that's being used, including properties, links between pads, etcetera. Just like with Glade, it is also possible to get references to specific elements inside the pipeline. So for the above pipeline, a developer could set the filename in the editor and save it as part of the XML filename. However, one could also leave that up to the application that loads the XML file. The latter is much more interesting for applications. However, for the case of the example, we will now assume that the XML file contains the filename information. Let us try to build an application that can load XML files:

```
#include <gst/gst.h>

int
main (int argc, char *argv[])
{
  GstElement *pipeline;
  GstXML *xml;

  gst_init(&argc, &argv);

  if (argc != 2) {
    g_print ("usage: %s <xml filename>\n", argv[0]);
    exit (-1);
  }

  /* Create XML object */
  xml = gst_xml_new ();

  /* parse XML document */
  if (!gst_xml_parse_file (xml, argv[1], "/")) {
    g_print ("Error parsing pipeline\n");
    exit (-1);
  }

  /* create a new pipeline to hold the elements */
  pipeline = gst_xml_get_element (xml, "pipeline0");

  /* start playing */
  gst_element_set_state (pipeline, GST_STATE_PLAYING);

  while (gst_bin_iterate (GST_BIN (pipeline)));

  /* stop the pipeline */
  gst_element_set_state (pipeline, GST_STATE_NULL);

  /* we don't need a reference to these objects anymore */
  gst_object_unref (GST_OBJECT (pipeline));
  gst_object_unref (GST_OBJECT (xml));

  exit (0);
}
```

As can be seen, the application is largely the same as our first C-application. The only

difference is that elements are not created individually. Rather, the XML file contains all information. Apart from that, there is no real difference. Specific elements can be retrieved using `gst_xml_get_element ()`, which is important for setting properties (such as filename for filesrc). The same call is also used to retrieve the pipeline that will be iterated. As said in the previous chapter, we use `while (...)` for the sake of the example. Most (Gnome) apps would use a `g_idle_add (...)` handler instead.

On to more complicated things

The simplicity of this pipeline allows us to do various things without having to change C-code that loads the XML file. Let us try to build a simple video player. Two things will be different with respect to the previous example. First of all, we will have to use a video output plugin too. Secondly, we will use an autoplugger. The autoplugger (spider) is able to autodetect file types of input streams. Consequently, a user can take whatever input, and spider will select the right parser/decoder plugins.

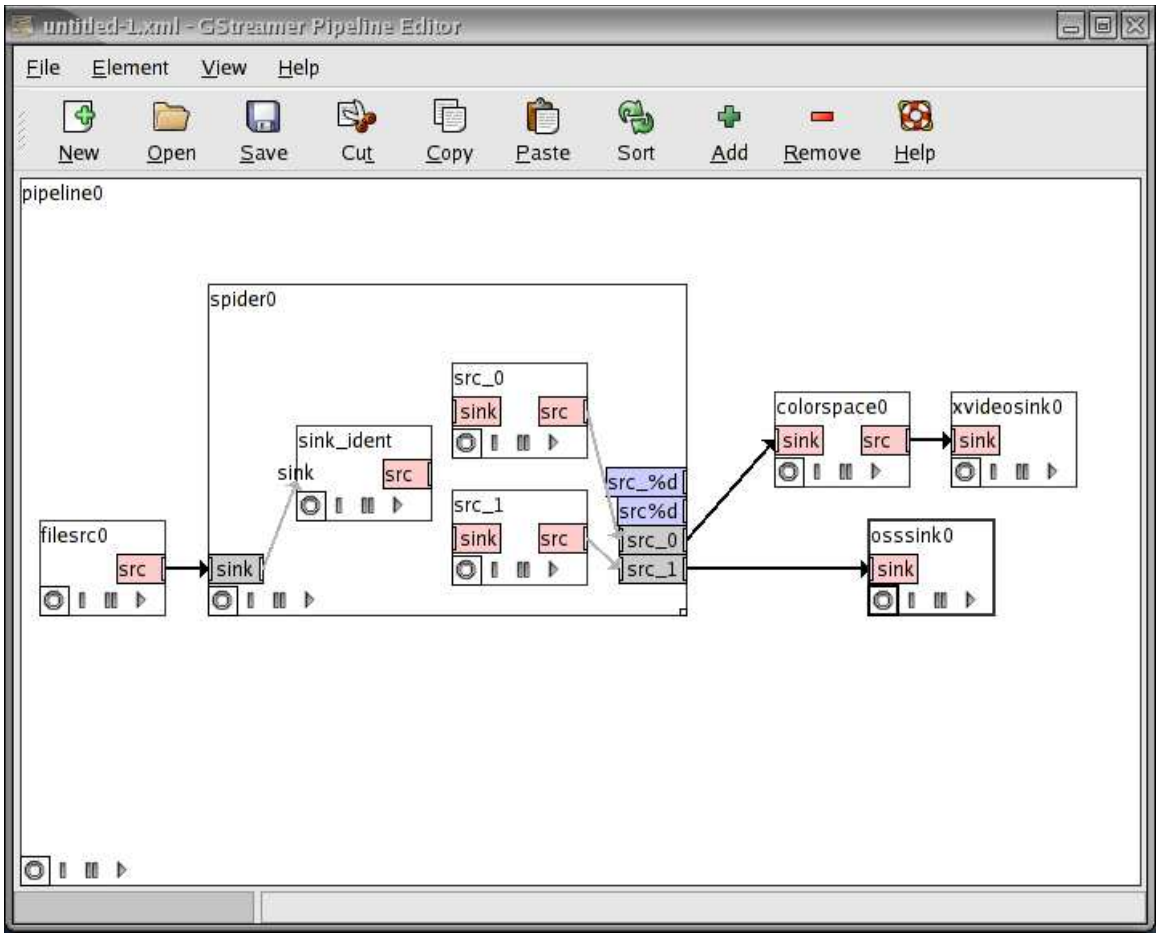

Illustration 3 - A more complicated pipeline loaded in the GStreamer Pipeline Editor

Figure 3 shows a more complicated pipeline. We see a video output plugin (xvideosink and colorspace), an audio output plugin (osssink), the already-discussed autoplugger spider, and a file source (filesrc). Pressing the "play" icon will now play a video, which can be selected in the Element Inspector (`View -> Element Inspector`). In a C application, one would use GObject properties for this.

Transcoding

We have currently mostly looked at playback of files. However, GStreamer can do much more than that. The GStreamer Pipeline Editor allows us to make a simple visual representation of audio transcoding from Ogg/Vorbis to FLAC, for example. For this, we replace the ossink element with two new elements: flacenc (Free Lossless Audio Codec encoder) and filesink (file writer). In the editor, this would look like this:

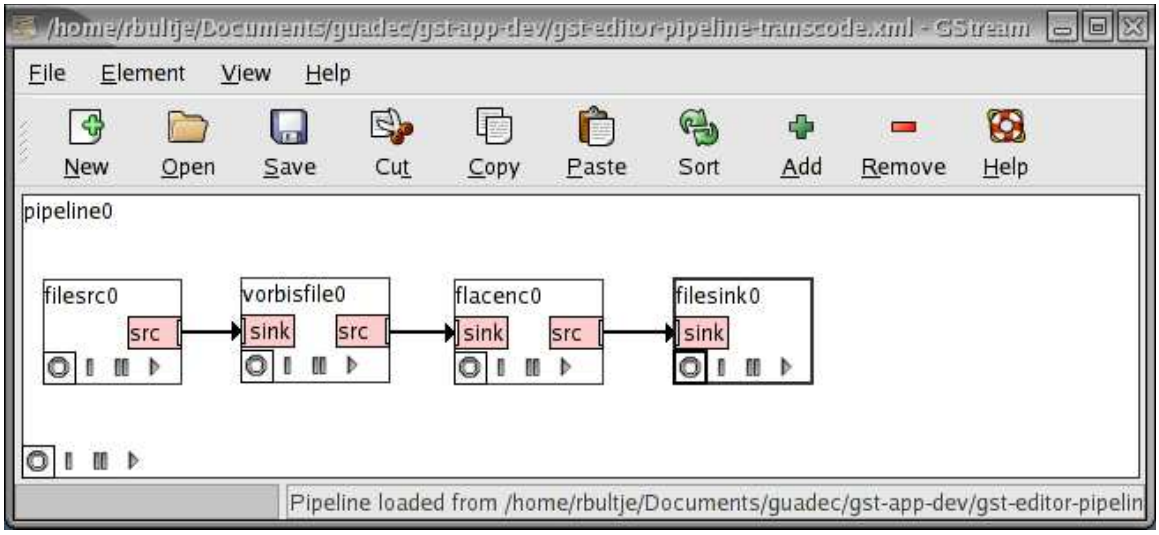

Illustration 4 - Example of a transcoding pipeline in the GStreamer Pipeline Editor

Pressing "play" will make the CPU busy for a while. In end user applications, people would likely want to see a progress indicator, which can simply be built using query-functions that have been explained in the previous chapter. The resulting file can be played back by media players such as Rhythmbox or XMMS.

Debugging

One more interesting appliction of XML files is for pipeline debugging. Often, larger pipelines tend to be quite confusing. The function `gst_xml_write_file ()` writes a pipeline to an XML file/stream. This file can then be opened using gst-editor. In case of normal-operating pipelines, it can even run inside the editor.

Conclusion

We have now seen how to use the GStreamer Pipeline Editor for creating pipelines, and we have seen how to build a C application that can load XML files containing pipeline layouts (which can be written by gst-editor) and iterate the resulting pipeline. Just as with the C interface, virtually anything can be done using the XML pipeline descriptions. However, there is still a lot of manual code to be written.

In the next chapter, we will limit ourselves to playback of video and audio, where we will look at higher-level interfaces that allow for simple file playback using the techniques that have been discussed in the past two chapters.

Higher-level interfaces – monkey-media

Monkey-media is a high-level interface for building music-player-applications. It was created by Jorn Baayen and is currently worked on by several people. It is also used by the music organizer Rhythmbox, for which it was originally written. monkey-media uses GStreamer as its backend.

The purpose of monkey-media is to be a high-level wrapper for the GStreamer playback functionalities (music files, audio CD and internet radio), plus some more non-GStreamer functions.

Basic C example

```
#include <monkey-media.h>

static void
monkey_media_mixer_eos_cb (MonkeyMediaPlayer *player,
                           gpointer unused)
{
  monkey_media_main_quit ();
}

int
main (int argc, char **argv)
{
  MonkeyMediaPlayer *player;
  GError *error = NULL;

  /* some basic sanity checking */
  if (argc < 2) {
    fprintf (stderr, "Usage: %s filename\n", argv[0]);
    exit (-1);
  }

  /* initialize */
  monkey_media_init (&argc, &argv);

  player = monkey_media_player_new (&error);
  if (error != NULL) {
    /* an error occured */
    fprintf (stderr, "Failed to create mixer: %s\n", error->message);
    g_error_free (error);
    exit (-1);
  }

  /* load file from commandline */
  monkey_media_player_open (player, argv[1], &error);
  if (error != NULL) {
    /* an error occured */
    fprintf (stderr, "Failed to create stream: %s\n", error->message);
    g_error_free (error);
    exit (-1);
  }
```

```
  /* hook up end of queue signal */
  g_signal_connect (G_OBJECT (player), "eos",
                    G_CALLBACK (monkey_media_mixer_eos_cb), NULL);

  /* lego from now on */
  monkey_media_player_play (player);

  /* and off we go */
  monkey_media_main ();

  /* free memory */
  g_object_unref (G_OBJECT (player));

  exit (0);
}
```

Monkey-media uses a thread rather than a pipeline for the top-level element. Consequently, Gnome apps (or any application using another main loop already) do not need to call `monkey_media_main ()` if they do not want to. It can be omitted.

Apart from that, it should be noticed that monkey-media exports a similar API to competitors such as Xine. Developers can simply create a player object, set a location, set it to playing and wait for it to be finished. This is probably the kind of API that most developers are interested in.

The result of the monkey-media effort can best be described by showing Rhythmbox:

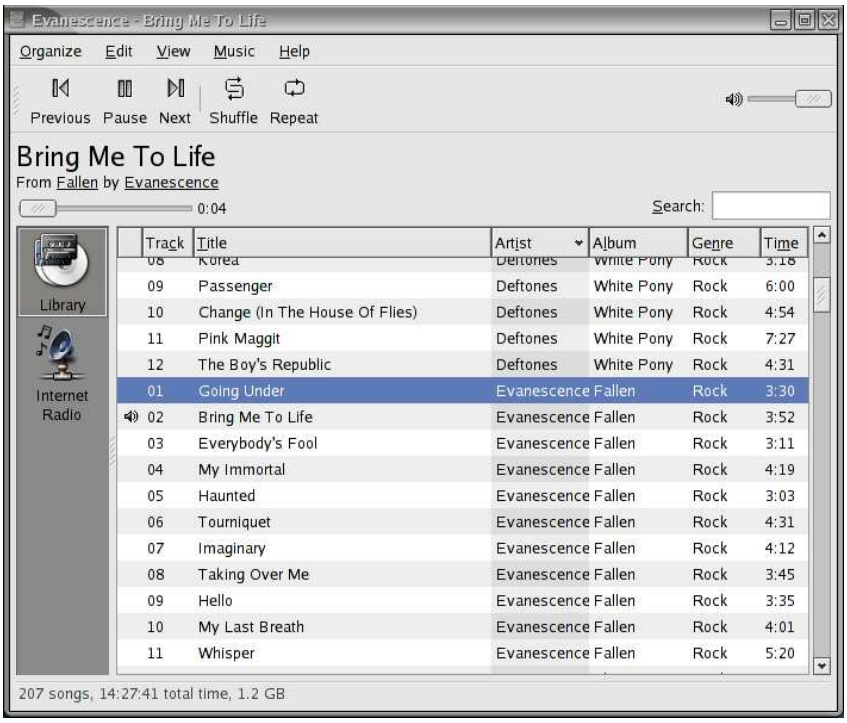

Illustration 5 - (net)Rhythmbox, an application based on monkey-media

Conclusion

Monkey-media provides a high-level API interface that is mostly aimed at music playback (audio CDs and Internet radio work, too). It provides an easy-to-use one-call playback API.

In the next chapter, we will have a look at video playback using GStreamer.

Video and visualization – libgstplay

The last example interface that will be discussed here, is libgstplay. libgstplay is internally under development in the GStreamer camp, development is currently lead by Julien Moutte.

The previous chapters have focused largely on audio functionalities. Admittedly, the audio part of GStreamer works much better than the video part, but video playback is one of the video-specific parts of GStreamer that does actually work fairly well. libgstplay does more than video playback only, though. It also supports audio; even better, audio visualizations work too! Especially the last one is a pretty nifty feature.

Let us now start with a simple C example to get into the world of libgstplay.

Simple C example

```
#include <gst/play/play.h>
#include <gst/gconf/gconf.h>

int
main (int argc, char *argv[])
{
  GstPlay *play;
  GstElement *video_sink, *audio_sink;

  /* Initing GStreamer library */
  gst_init (&argc, &argv);

  if (argc != 2) {
    g_print ("usage: %s <video filename>\n", argv[0]);
    exit (-1);
  }

  /* Creating the GstPlay object */
  play = gst_play_new (GST_PLAY_PIPE_VIDEO, NULL);

  /* Getting default audio and video plugins from GConf */
  audio_sink = gst_gconf_get_default_audio_sink ();
  video_sink = gst_gconf_get_default_video_sink ();

  /* Let's send them to GstPlay object */
  gst_play_set_audio_sink (play, audio_sink);
  gst_play_set_video_sink (play, video_sink);
  g_object_set (G_OBJECT (gst_play_get_sink_element (play, video_sink,
                                                     GST_PLAY_SINK_TYPE
_VIDEO)),
                "toplevel", TRUE, NULL);

  /* Setting location we want to play */
  gst_play_set_location (play, argv[1]);

  /* Change state to PLAYING */
```

```
  gst_play_set_state (play, GST_STATE_PLAYING);

  /* We now call the main procedure */
  while (gst_bin_iterate (GST_BIN (play->pipeline)));

  /* unref */
  g_object_unref (G_OBJECT (play));

  exit (0);
}
```

The above example sets up the basics of a video player. A few things are noteworthy compared to the previous examples. Firstly, the argument given to `gst_play_new`: `GST_PLAY_PIPE_VIDEO` means that we will playback video files. Other interesting values are `GST_PLAY_PIPE_AUDIO` and `GST_PLAY_PIPE_VIDEO_VISUALIZATION`. The last one is actually for audio playback, but sets up video visualization. We will come back to this later. The second noteworthy thing is that we use gconf keys to setup default video and audio output plugins.

The line after that can largely be considered as a cruel hack. libgstplay sets up video playback for embedded windows. This means that the Xwindow that is being created is set up to be embedded in another Xwindow (e.g. a Gnome application). For the sake of this example, we do not want that, we want a toplevel window. Normally, applications do not need this.

The rest is fairly straightforward. We enter a file for playback, set the thing to start playing and wait for that to be done. This is not very different from the monkey-media example or the native GStreamer interface example from the first chapter.

The result fo all this is a video player. By removing the toplevel property hack and calling some native Gnome functions, one would have a first Gnome Media Player. This is basically what applications such as Totem and gst-player try to do (see figure 6).

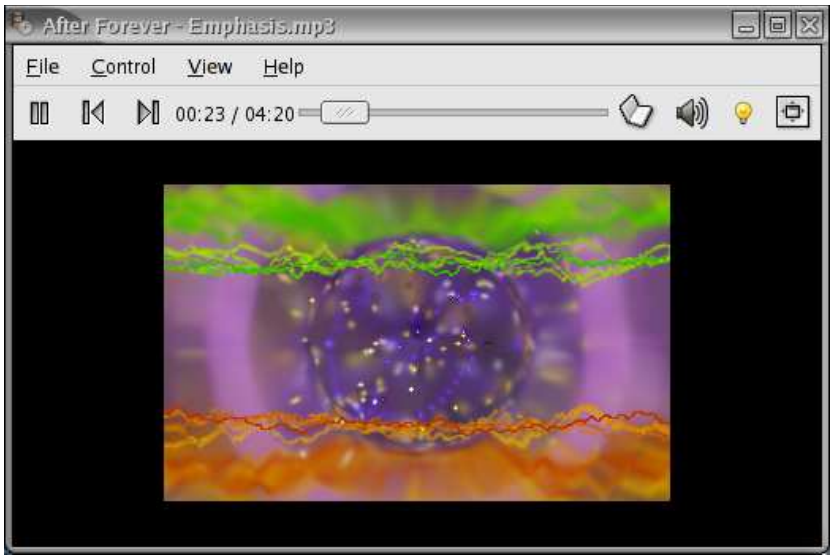

Illustration 6 - GStreamer player using visualizations

Conclusion

Libgstplay is, just like monkey-media (which has been discussed earlier on), a higher-level interface that tries to hide the hard parts of GStreamer and provide a simpler API interface that works like `play_file (const char *filename);` while also providing access to the GStreamer objects directly (this is not possible in monkey-media). Application developers that want to do something with video should seriously consider looking at libgstplay to provide double work.

Conclusion & Thanks

This far, four different ways of accessing the GStreamer functionalities have been discussed: the native interface, the XML interface, monkey-media and libgstplay.

The C interface is a low-level, fast way of getting things done. However, it should be noted that the C interface is hard to understand at first sight, and many things can also be done by using a simpler high-level interface. Virtually anything having to do with media can be achieved here.

The XML interface brings a graphical editor which allosws a visual (Glade-like) setup of pipelines, which can be loaded into an application afterwards. It still allows for virtual any application, but developers won't lose track as easy as with the C interface.

Monkey-media and libgstplay are two higher-level interfaces that target playback of video, audio, CDs and internet radio specifically. The setup of these is far easier, though, one does not have to worry about most of the GStreamer internals. Instead, the libraries will do most of the hard work for you. Monkey-media is a robust music package, while libgstplay is more targetted at video specifically.

Developers that want to use GStreamer in their applications should consider each of these interfaces.

My thanks goes to the GStreamer team for providing a wonderful interface. Moreover, thanks to the Rhythmbox team for monkey-media and Rhythmbox, and thanks to the gst-player hackers for creating libgstplay and gst-player.