GStreamer Application Development Manual (0.8.9.2)

Wim Taymans

Steve Baker

Andy Wingo

Ronald S. Bultje

GStreamer Application Development Manual (0.8.9.2)

by Wim Taymans, Steve Baker, Andy Wingo, and Ronald S. Bultje

Table of Contents

Chapter 1. Introduction

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

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 type of streaming multimedia application. The `GStreamer` framework is designed to make it easy to write applications that handle audio or video or both. It isn't restricted to audio and video, and can process any kind of data flow. The pipeline design is made to have little overhead above what the applied filters induce. This makes `GStreamer` a good framework for designing even high-end audio applications which put high demands on latency.

One of the the most obvious uses of `GStreamer` is using it to build a media player. `GStreamer` already includes components for building a media player that can support a very wide variety of formats, including MP3, Ogg/Vorbis, MPEG-1/2, AVI, Quicktime, mod, and more. `GStreamer` , however, is much more than just another media player. Its main advantages are that the pluggable components can be mixed and matched into arbitrary pipelines so that it's possible to write a full-fledged video or audio editing application.

The framework is based on plugins that will provide the various codec and other functionality. The plugins can be linked 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.

The `GStreamer` core function is to provide a framework for plugins, data flow and media type handling/negotiation. It also provides an API to write applications using the various plugins.

Structure of this Manual

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. For an explanation about writing plugins, we suggest the Plugin Writers Guide[1].

Part I in *`GStreamer` Application Development Manual (0.8.9.2)* gives you an overview of `GStreamer` 's motivation design goals.

Part II in *`GStreamer` Application Development Manual (0.8.9.2)* rapidly covers the basics of `GStreamer` application programming. At the end of that chapter, you should be able to build your own audio player using `GStreamer`

In Part III in *`GStreamer` Application Development Manual (0.8.9.2)*, we will move on to complicated subjects which make `GStreamer` stand out of its competitors. We will discuss application-pipeline interaction using dynamic parameters and interfaces, we will discuss threading and threaded pipelines, scheduling and clocks (and synchronization). Most of those topics are not just there to introduce you to their API, but primarily to give a deeper insight in solving application programming problems with `GStreamer` and understanding their concepts.

Next, in Part IV in *`GStreamer` Application Development Manual (0.8.9.2)*, we will go into higher-level programming APIs for `GStreamer` . You don't exactly need to know all the details from the previous parts to understand this, but you will need to understand basic `GStreamer` concepts nevertheless. We will, amongst others, discuss XML, playbin and autopluggers.

In Part V in `GStreamer` *Application Development Manual (0.8.9.2)*, you will find some random information on integrating with GNOME, KDE, OS X or Windows, some debugging help and general tips to improve and simplify `GStreamer` programming.

In order to understand this manual, you will need to have a basic understanding of the C language. Since `GStreamer` uses GLib 2.0[2], the reader is assumed to understand the basics of the GObject object model[3]. It is recommended to have skimmed through the introduction of the GObject tutorial[4] before reading this. You may also want to have a look at Eric Harlow's book *Developing Linux Applications with GTK+ and GDK*.

Notes

1. http://gstreamer.freedesktop.org/data/doc/gstreamer/head/pwg/html/index.html
2. http://developer.gnome.org/arch/gtk/glib.html
3. http://developer.gnome.org/doc/API/2.0/gobject/index.html
4. http://www.le-hacker.org/papers/gobject/index.html

Chapter 2. Motivation & Goals

Linux has historically lagged behind other operating systems in the multimedia arena. Microsoft's Windows™ and Apple's MacOS™ 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.

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

Current problems

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

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.

'One goal' media players/libraries

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 you want to convert an MPEG-2 video stream into an AVI file, your best option would be to take all of the MPEG-2 decoding algorithms out of the player and duplicate them into your own AVI encoder. These algorithms cannot easily be shared across applications.

Attempts have been made to create libraries for handling various media types. Because they focus on a very specific media type (avifile, libmpeg2, ...), significant work is needed to integrate them due to a lack of a common API. `GStreamer` allows you to wrap these libraries with a common API, which significantly simplifies integration and reuse.

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 plugin system of the typical media player is also very tailored to the specific needs of the application.

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 developer and ensures the plugin can be used in a wide range of applications, transparently interacting with other plugins. The framework that `GStreamer` provides for the plugins is flexible enough to host even the most demanding plugins.

Poor user experience

Because of the problems mentioned above, application authors have so far often been urged to spend a considerable amount of time in writing their own backends, plugin mechanisms and so on. The result has often been, unfortunately, that both the backend as well as the user interface were only half-finished. Demotivated, the application authors would start rewriting the whole thing and complete the circle. This leads to a *poor end user experience*.

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 technologies such as the GNOME object embedding using Bonobo[1].

The `GStreamer` core does not use network transparent technologies at the lowest level as it only adds overhead for the local case. That said, it shouldn't be hard to create a wrapper around the core components. There are tcp plugins now that implement a `GStreamer` Data Protocol that allows pipelines to be slit over TCP. These are located in the gst-plugins module directory gst/tcp.

Catch up with the Windows™ 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.

The design goals

We describe what we try to achieve with `GStreamer`.

Clean and powerful

`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 powerful 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 examples too.

Object oriented

`GStreamer` adheres to the GLib 2.0 object model. A programmer familiar with GLib 2.0 or older versions of GTK+ will be comfortable with `GStreamer`.

`GStreamer` uses the mechanism of signals and object properties.

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

GStreamer intends to be similar in programming methodology to GTK+. This applies to the object model, ownership of objects, reference counting, ...

Extensible

All GStreamer Objects can be extended using the GObject inheritance methods.

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

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 properties, there is no need (and in fact no way) to have any header files installed for the plugins.

Special care has been taken to make plugins completely self-contained. All relevant aspects of plugins can be queried at run-time.

High performance

High performance is obtained by:

- using GLib's g_mem_chunk and fast non-blocking allocation algorithms where possible to minimize dynamic memory allocation.
- extremely light-weight links between plugins. Data can travel the pipeline with minimal overhead. Data passing between plugins only involves a pointer dereference in a typical pipeline.
- providing a mechanism to directly work on the target memory. A plugin can for example directly write to the X server's shared memory space. Buffers can also point to arbitrary memory, such as a sound card's internal hardware buffer.
- refcounting and copy on write minimize usage of memcpy. Sub-buffers efficiently split buffers into manageable pieces.
- the use of cothreads to minimize the threading overhead. Cothreads are a simple and fast user-space method for switching between subtasks. Cothreads were measured to consume as little as 600 cpu cycles.
- allowing hardware acceleration by using specialized plugins.
- using a plugin registry with the specifications of the plugins so that the plugin loading can be delayed until the plugin is actually used.
- all critical data passing is free of locks and mutexes.

Clean core/plugins separation

The core of GStreamer is essentially media-agnostic. It only knows about bytes and blocks, and only contains basic elements. The core of GStreamer is functional enough to even implement low-level system tools, like cp.

All of the media handling functionality is provided by plugins external to the core. These tell the core how to handle specific types of media.

Provide a framework for codec experimentation

`GStreamer` also wants to be an easy framework where codec developers can experiment with different algorithms, speeding up the development of open and free multimedia codecs like Theora and Vorbis[2].

Notes

1. http://developer.gnome.org/arch/component/bonobo.html
2. http://www.xiph.org/ogg/index.html

Chapter 3. Foundations

This chapter of the guide introduces the basic concepts of `GStreamer`. Understanding these concepts will be important in reading any of the rest of this guide, all of them assume understanding of these basic concepts.

Elements

An *element* is the most important class of objects in `GStreamer`. You will usually create a chain of elements linked together and let data flow through this chain of elements. An element has one specific function, which can be the reading of data from a file, decoding of this data or outputting this data to your sound card (or anything else). By chaining together several such elements, you create a *pipeline* that can do a specific task, for example media playback or capture. `GStreamer` ships with a large collection of elements by default, making the development of a large variety of media applications possible. If needed, you can also write new elements. That topic is explained in great deal in the Plugin Writer's Guide.

Bins and pipelines

A *bin* is a container for a collection of elements. A pipeline is a special subtype of a bin that allows execution of all of its contained child elements. Since bins are subclasses of elements themselves, you can mostly control a bin as if it where an element, thereby abstracting away a lot of complexity for your application. You can, for example change state on all elements in a bin by changing the state of that bin itself. Bins also forward some signals from their contained childs (such as errors and tags).

A pipeline is a bin that allows to *run* (technically referred to as "iterating") its contained childs. By iterating a pipeline, data flow will start and media processing will take place. A pipeline requires iterating for anything to happen. you can also use threads, which automatically iterate the contained childs in a newly created threads. We will go into this in detail later on.

Pads

Pads are used to negotiate links and data flow between elements in `GStreamer`. A pad can be viewed as a "plug" or "port" on an element where links may be made with other elements, and through which data can flow to or from those elements. Pads have specific data handling capabilities: A pad can restrict the type of data that flows through it. Links are only allowed between two pads when the allowed data types of the two pads are compatible. Data types are negotiated between pads using a process called *caps negotiation*. Data types are described as a `GstCaps`.

An analogy may be helpful here. A pad is similar to a plug or jack on a physical device. Consider, for example, a home theater system consisting of an amplifier, a DVD player, and a (silent) video projector. Linking the DVD player to the amplifier is allowed because both devices have audio jacks, and linking the projector to the DVD player is allowed because both devices have compatible video jacks. Links between the projector and the amplifier may not be made because the projector and amplifier have different types of jacks. Pads in `GStreamer` serve the same purpose as the jacks in the home theater system.

For the most part, all data in `GStreamer` flows one way through a link between elements. Data flows out of one element through one or more *source pads*, and elements accept incoming data through one or more *sink pads*. Source and sink elements have only source and sink pads, respectively. Data is embodied in a `GstData` structure.

Chapter 4. Initializing `GStreamer`

When writing a `GStreamer` application, you can simply include `gst/gst.h` to get access to the library functions. Besides that, you will also need to intialize the `GStreamer` library.

Simple initialization

Before the `GStreamer` libraries can be used, `gst_init` has to be called from the main application. This call will perform the necessary initialization of the library as well as parse the `GStreamer` -specific command line options.

A typical program [1] would have code to initialize `GStreamer` that looks like this:

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

int
main (int   argc,
      char *argv[])
{
  guint major, minor, micro;

  gst_init (&argc, &argv);

  gst_version (&major, &minor, &micro);
  printf ("This program is linked against GStreamer %d.%d.%d\n",
          major, minor, micro);

  return 0;
}
```

Use the GST_VERSION_MAJOR, GST_VERSION_MINOR and GST_VERSION_MICRO macros to get the `GStreamer` version you are building against, or use the function `gst_version` to get the version your application is linked against. `GStreamer` currently uses a scheme where versions with the same major and minor versions are API-/ and ABI-compatible.

It is also possible to call the `gst_init` function with two NULL arguments, in which case no command line options will be parsed by `GStreamer` .

The popt interface

You can also use a popt table to initialize your own parameters as shown in the next example:

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

int
main (int   argc,
      char *argv[])
{
  gboolean silent = FALSE;
  gchar *savefile = NULL;
  struct poptOption options[] = {
    {"silent",  's',  POPT_ARG_NONE|POPT_ARGFLAG_STRIP,    &silent,   0,
     "do not output status information", NULL},
    {"output",  'o',  POPT_ARG_STRING|POPT_ARGFLAG_STRIP,  &savefile, 0,
     "save xml representation of pipeline to FILE and exit", "FILE"},
    POPT_TABLEEND
  };

  gst_init_with_popt_table (&argc, &argv, options);

  printf ("Run me with --help to see the Application options appended.\n");

  return 0;
}
```

As shown in this fragment, you can use a popt[2] table to define your application-specific command line options, and pass this table to the function `gst_init_with_popt_table`. Your application options will be parsed in addition to the standard `GStreamer` options.

Notes

1. The code for this example is automatically extracted from the documentation and built under `examples/manual` in the GStreamer tarball.
2. http://developer.gnome.org/doc/guides/popt/

Chapter 5. Elements

The most important object in `GStreamer` for the application programmer is the `GstElement`[1] object. An element is the basic building block for a media pipeline. All the different high-level components you will use are derived from `GstElement`. Every decoder, encoder, demuxer, video or audio output is in fact a `GstElement`

What are elements?

For the application programmer, elements are best visualized as black boxes. On the one end, you might put something in, the element does something with it and something else comes out at the other side. For a decoder element, ifor example, you'd put in encoded data, and the element would output decoded data. In the next chapter (see Pads and capabilities), you will learn more about data input and output in elements, and how you can set that up in your application.

Source elements

Source elements generate data for use by a pipeline, for example reading from disk or from a sound card. Figure 5-1 shows how we will visualise a source element. We always draw a source pad to the right of the element.

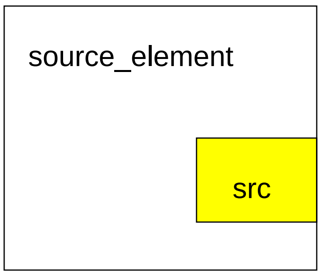

Figure 5-1. Visualisation of a source element

Source elements do not accept data, they only generate data. You can see this in the figure because it only has a source pad (on the right). A source pad can only generate data.

Filters, convertors, demuxers, muxers and codecs

Filters and filter-like elements have both input and outputs pads. They operate on data that they receive on their input (sink) pads, and will provide data on their output (source) pads. Examples of such elements are a volume element (filter), a video scaler (convertor), an Ogg demuxer or a Vorbis decoder.

Filter-like elements can have any number of source or sink pads. A video demuxer, for example, would have one sink pad and several (1-N) source pads, one for each elementary stream contained in the container format. Decoders, on the other hand, will only have one source and sink pads.

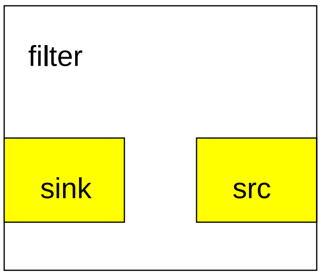

Figure 5-2. Visualisation of a filter element

Figure 5-2 shows how we will visualise a filter-like element. This specific element has one source and one sink element. Sink pads, receiving input data, are depicted at the left of the element; source pads are still on the right.

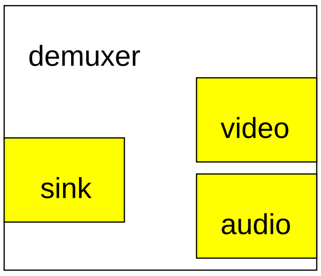

Figure 5-3. Visualisation of a filter element with more than one output pad

Figure 5-3 shows another filter-like element, this one having more than one output (source) pad. An example of one such element could, for example, be an Ogg demuxer for an Ogg stream containing both audio and video. One source pad will contain the elementary video stream, another will contain the elementary audio stream. Demuxers will generally fire signals when a new pad is created. The application programmer can then handle the new elementary stream in the signal handler.

Sink elements

Sink elements are end points in a media pipeline. They accept data but do not produce anything. Disk writing, soundcard playback, and video output would all be implemented by sink elements. Figure 5-4 shows a sink element.

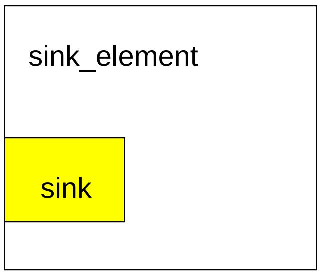

Figure 5-4. Visualisation of a sink element

Creating a `GstElement`

The simplest way to create an element is to use `gst_element_factory_make ()`[2]. This function takes a factory name and an element name for the newly created element. The name of the element is something you can use later on to look up the element in a bin, for example. The name will also be used in debug output. You can pass NULL as the name argument to get a unique, default name.

When you don't need the element anymore, you need to unref it using `gst_object_unref ()`[3]. This decreases the reference count for the element by 1. An element has a refcount of 1 when it gets created. An element gets destroyed completely when the refcount is decreased to 0.

The following example [4] shows how to create an element named *source* from the element factory named *fakesrc*. It checks if the creation succeeded. After checking, it unrefs the element.

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

int
main (int    argc,
      char *argv[])
{
  GstElement *element;

  /* init GStreamer */
  gst_init (&argc, &argv);

  /* create element */
  element = gst_element_factory_make ("fakesrc", "source");
  if (!element) {
    g_print ("Failed to create element of type 'fakesrc'\n");
    return -1;
  }

  gst_object_unref (GST_OBJECT (element));

  return 0;
}
```

`gst_element_factory_make` is actually a shorthand for a combination of two functions. A `GstElement` [5] object is created from a factory. To create the element, you have to get access to a `GstElementFactory` [6] object using a unique factory name. This is done with `gst_element_factory_find ()`[7].

The following code fragment is used to get a factory that can be used to create the *fakesrc* element, a fake data source. The function `gst_element_factory_create ()`[8] will use the element factory to create an element with the given name.

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

int
main (int    argc,
      char *argv[])
{
  GstElementFactory *factory;
  GstElement * element;

  /* init GStreamer */
  gst_init (&argc, &argv);

  /* create element, method #2 */
  factory = gst_element_factory_find ("fakesrc");
  if (!factory) {
    g_print ("Failed to find factory of type 'fakesrc'\n");
    return -1;
  }
  element = gst_element_factory_create (factory, "source");
  if (!element) {
    g_print ("Failed to create element, even though its factory exists!\n");
    return -1;
  }

  gst_object_unref (GST_OBJECT (element));

  return 0;
}
```

Using an element as a `GObject`

A `GstElement` [9] can have several properties which are implemented using standard `GObject` properties. The usual `GObject` methods to query, set and get property values and `GParamSpecs` are therefore supported.

Every `GstElement` inherits at least one property from its parent `GstObject` : the "name" property. This is the name you provide to the functions `gst_element_factory_make ()` or `gst_element_factory_create ()`. You can get and set this property using the functions `gst_object_set_name` and `gst_object_get_name` or use the `GObject` property mechanism as shown below.

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

int
main (int    argc,
      char *argv[])
{
  GstElement *element;
  const gchar *name;

  /* init GStreamer */
  gst_init (&argc, &argv);

  /* create element */
  element = gst_element_factory_make ("fakesrc", "source");

  /* get name */
  g_object_get (G_OBJECT (element), "name", &name, NULL);
  g_print ("The name of the element is '%s'.\n", name);

  gst_object_unref (GST_OBJECT (element));

  return 0;
}
```

Most plugins provide additional properties to provide more information about their configuration or to configure the element. **gst-inspect** is a useful tool to query the properties of a particular element, it will also use property introspection to give a short explanation about the function of the property and about the parameter types and ranges it supports. See the appendix for details about **gst-inspect**.

For more information about `GObject` properties we recommend you read the GObject manual[10] and an introduction to The Glib Object system[11].

A `GstElement` [12] also provides various `GObject` signals that can be used as a flexible callback mechanism. Here, too, you can use **gst-inspect** to see which signals a specific elements supports. Together, signals and properties are the most basic way in which elements and applications interact.

More about element factories

In the previous section, we briefly introduced the `GstElementFactory`[13] object already as a way to create instances of an element. Element factories, however, are much more than just that. Element factories are the basic types retrieved from the `GStreamer` registry, they describe all plugins and elements that `GStreamer` can create. This means that element factories are useful for automated element instancing, such as what autopluggers do, and for creating lists of available elements, such as what pipeline editing applications (e.g. `GStreamer` Editor[14]) do.

Getting information about an element using a factory

Tools like **gst-inspect** will provide some generic information about an element, such as the person that wrote the plugin, a descriptive name (and a shortname), a rank and a category. The category can be used to get the type of the element that can be created using this element factory. Examples of categories include `Codec/Decoder/Video` (video decoder), `Codec/Encoder/Video` (video encoder), `Source/Video` (a video generator), `Sink/Video` (a video output), and all these exist for audio as well, of course. Then, there's also `Codec/Demuxer` and `Codec/Muxer` and a whole lot more. **gst-inspect** will give a list of all factories, and **gst-inspect <factory-name>** will list all of the above information, and a lot more.

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

int
main (int    argc,
      char *argv[])
{
  GstElementFactory *factory;

  /* init GStreamer */
  gst_init (&argc, &argv);

  /* get factory */
  factory = gst_element_factory_find ("sinesrc");
  if (!factory) {
    g_print ("You don't have the 'sinesrc' element installed, go get it!\n");
    return -1;
  }

  /* display information */
  g_print ("The '%s' element is a member of the category %s.\n"
           "Description: %s\n",
           gst_plugin_feature_get_name (GST_PLUGIN_FEATURE (factory)),
           gst_element_factory_get_klass (factory),
           gst_element_factory_get_description (factory));

  return 0;
}
```

You can use `gst_registry_pool_feature_list (GST_TYPE_ELEMENT_FACTORY)` to get a list of all the element factories that `GStreamer` knows about.

Finding out what pads an element can contain

Perhaps the most powerful feature of element factories is that they contain a full description of the pads that the element can generate, and the capabilities of those pads (in layman words: what types of media can stream over those pads), without actually having to load those plugins into memory. This can be used to provide a codec selection list for encoders, or it can be used for autoplugging purposes for media players. All current `GStreamer`-based media players and autopluggers work this way. We'll look closer at these features as we learn about `GstPad` and `GstCaps` in the next chapter: Pads and capabilities

Linking elements

By linking a source element with zero or more filter-like elements and finally a sink element, you set up a media pipeline. Data will flow through the elements. This is the basic concept of media handling in `GStreamer`.

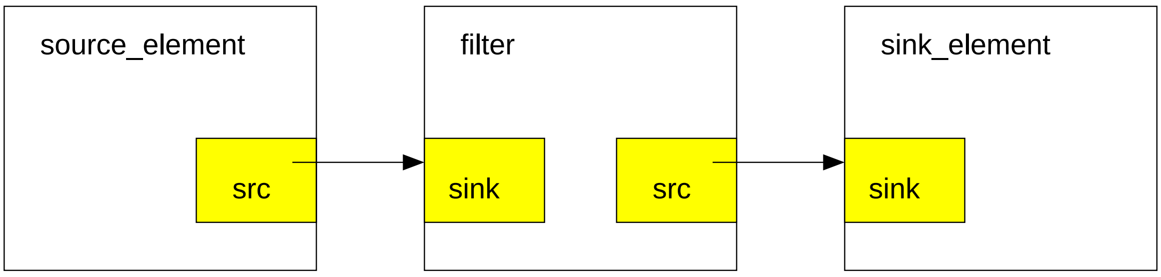

Figure 5-5. Visualisation of three linked elements

By linking these three elements, we have created a very simple chain of elements. The effect of this will be that the output of the source element ("element1") will be used as input for the filter-like element ("element2"). The filter-like element will do something with the data and send the result to the final sink element ("element3").

Imagine the above graph as a simple Ogg/Vorbis audio decoder. The source is a disk source which reads the file from disc. The second element is a Ogg/Vorbis audio decoder. The sink element is your soundcard, playing back the decoded audio data. We will use this simple graph to construct an Ogg/Vorbis player later in this manual.

In code, the above graph is written like this:

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

int
main (int   argc,
      char *argv[])
{
  GstElement *source, *filter, *sink;

  /* init */
  gst_init (&argc, &argv);

  /* create elements */
  source = gst_element_factory_make ("fakesrc", "source");
  filter = gst_element_factory_make ("identity", "filter");
  sink = gst_element_factory_make ("fakesink", "sink");

  /* link */
  gst_element_link_many (source, filter, sink, NULL);
[..]

}
```

For more specific behaviour, there are also the functions `gst_element_link ()` and `gst_element_link_pads ()`. You can also obtain references to individual pads and link those using various `gst_pad_link_* ()` functions. See the API references for more details.

Element States

After being created, an element will not actually perform any actions yet. You need to change elements state to make it do something. `GStreamer` knows four element states, each with a very specific meaning. Those four states are:

- `GST_STATE_NULL` : this is the default state. This state will deallocate all resources held by the element.
- `GST_STATE_READY` : in the ready state, an element has allocated all of its global resources, that is, resources that can be kept within streams. You can think about opening devices, allocating buffers and so on. However, the stream is not opened in this state, so the stream positions is automatically zero. If a stream was previously opened, it should be closed in this state, and position, properties and such should be reset.
- `GST_STATE_PAUSED` : in this state, an element has opened the stream, but is not actively processing it. An element should not modify the stream's position, data or anything else in this state. When set back to PLAYING, it should continue processing at the point where it left off as soon as possible.
- `GST_STATE_PLAYING` : in the PLAYING state, an element does exactly the same as in the PAUSED state, except that it actually processes data.

You can change the state of an element using the function `gst_element_set_state ()`. If you set an element to another state, `GStreamer` will internally traverse all intermediate states. So if you set an element from NULL to PLAYING, `GStreamer` will internally set the element to READY and PAUSED in between.

Even though an element in `GST_STATE_PLAYING` is ready for data processing, it will not necessarily do that. If the element is placed in a thread (see Chapter 15), it will process data automatically. In other cases, however, you will need to *iterate* the element's container.

Notes

1. ../../gstreamer/html/GstElement.html
2. http://gstreamer.freedesktop.org/data/doc/gstreamer/stable/gstreamer/html/GstElementFacto element-factory-make
3. http://gstreamer.freedesktop.org/data/doc/gstreamer/stable/gstreamer/html/GstObject.html#g object-unref
4. The code for this example is automatically extracted from the documentation and built under `examples/manual` in the GStreamer tarball.
5. http://gstreamer.freedesktop.org/data/doc/gstreamer/stable/gstreamer/html/GstElement.html
6. http://gstreamer.freedesktop.org/data/doc/gstreamer/stable/gstreamer/html/GstElementFacto
7. http://gstreamer.freedesktop.org/data/doc/gstreamer/stable/gstreamer/html/GstElementFacto element-factory-find

8. http://gstreamer.freedesktop.org/data/doc/gstreamer/stable/gstreamer/html/GstElementFacto element-factory-create
9. http://gstreamer.freedesktop.org/data/doc/gstreamer/stable/gstreamer/html/GstElement.html
10. http://developer.gnome.org/doc/API/2.0/gobject/index.html
11. http://le-hacker.org/papers/gobject/index.html
12. http://gstreamer.freedesktop.org/data/doc/gstreamer/stable/gstreamer/html/gstreamer/html/
13. http://gstreamer.freedesktop.org/data/doc/gstreamer/stable/gstreamer/html/GstElement.html
14. http://gstreamer.freedesktop.org/modules/gst-editor.html

Chapter 6. Bins

A bin is a container element. You can add elements to a bin. Since a bin is an element itself, a bin can be handled in the same way as any other element. Therefore, the whole previous chapter (Elements) applies to bins as well.

What are bins

Bins allow you to combine a group of linked elements into one logical element. You do not deal with the individual elements anymore but with just one element, the bin. We will see that this is extremely powerful when you are going to construct complex pipelines since it allows you to break up the pipeline in smaller chunks.

The bin will also manage the elements contained in it. It will figure out how the data will flow in the bin and generate an optimal plan for that data flow. Plan generation is one of the most complicated procedures in `GStreamer` . You will learn more about this process, called scheduling, in Chapter 16.

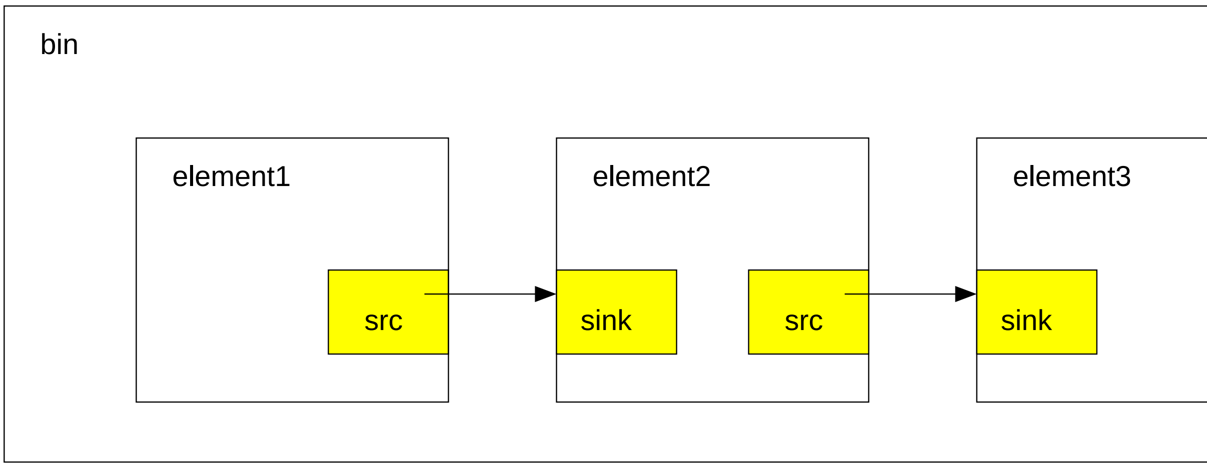

Figure 6-1. Visualisation of a bin with some elements in it

There are two specialized types of bins available to the `GStreamer` programmer:

- A pipeline: a generic container that allows scheduling of the containing elements. The toplevel bin has to be a pipeline. Every application thus needs at least one of these. Applications can iterate pipelines using `gst_bin_iterate ()` to make it process data while in the playing state.
- A thread: a bin that will be run in a separate execution thread. You will have to use this bin if you have to carefully synchronize audio and video, or for buffering. You will learn more about threads in Chapter 15.

Creating a bin

Bins are created in the same way that other elements are created, i.e. using an element factory. There are also convenience functions available (`gst_bin_new ()`, `gst_thread_new ()` and `gst_pipeline_new ()`). To add elements to a bin or remove elements from a bin, you can use `gst_bin_add ()` and `gst_bin_remove ()`. Note that the bin that you add an element to will take ownership of that element. If you destroy the bin, the element will be dereferenced with it. If you remove an element from a bin, it will be dereferenced automatically.

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

int
main (int argc,
      char *argv[])
{
  GstElement *bin, *pipeline, *source, *sink;

  /* init */
  gst_init (&argc, &argv);

  /* create */
  pipeline = gst_pipeline_new ("my_pipeline");
  bin = gst_pipeline_new ("my_bin");
  source = gst_element_factory_make ("fakesrc", "source");
  sink = gst_element_factory_make ("fakesink", "sink");

  /* set up pipeline */
  gst_bin_add_many (GST_BIN (bin), source, sink, NULL);
  gst_bin_add (GST_BIN (pipeline), bin);
  gst_element_link (source, sink);

[..]

}
```

There are various functions to lookup elements in a bin. You can also get a list of all elements that a bin contains using the function `gst_bin_get_list ()`. See the API references of `GstBin` [1] for details.

Custom bins

The application programmer can create custom bins packed with elements to perform a specific task. This allows you, for example, to write an Ogg/Vorbis decoder with just the following lines of code:

```
int
main (int argc
      char *argv[])
{
  GstElement *player;

  /* init */
  gst_init (&argc, &argv);

  /* create player */
  player = gst_element_factory_make ("oggvorbisplayer", "player");

  /* set the source audio file */
  g_object_set (G_OBJECT (player), "location", "helloworld.ogg", NULL);

  /* start playback */
  gst_element_set_state (GST_ELEMENT (player), GST_STATE_PLAYING);
[..]
}
```

Custom bins can be created with a plugin or an XML description. You will find more information about creating custom bin in the Plugin Writers Guide[2].

Notes

1. http://gstreamer.freedesktop.org/data/doc/gstreamer/stable/gstreamer/html/GstBin.html
2. http://gstreamer.freedesktop.org/data/doc/gstreamer/head/pwg/html/index.html

Chapter 7. Pads and capabilities

As we have seen in Elements, the pads are the element's interface to the outside world. Data streams from one element's source pad to another element's sink pad. The specific type of media that the element can handle will be exposed by the pad's capabilities. We will talk more on capabilities later in this chapter (see the Section called *Capabilities of a pad*).

Pads

A pad type is defined by two properties: its direction and its availability. As we've mentioned before, `GStreamer` defines two pad directions: source pads and sink pads. This terminology is defined from the view of within the element: elements receive data on their sink pads and generate data on their source pads. Schematically, sink pads are drawn on the left side of an element, whereas source pads are drawn on the right side of an element. In such graphs, data flows from left to right. [1]

Pad directions are very simple compared to pad availability. A pad can have any of three availabilities: always, sometimes and on request. The meaning of those three types is exactly as it says: always pads always exist, sometimes pad exist only in certain cases (and can disappear randomly), and on-request pads appear only if explicitely requested by applications.

Dynamic (or sometimes) pads

Some elements might not have all of their pads when the element is created. This can happen, for example, with an Ogg demuxer element. The element will read the Ogg stream and create dynamic pads for each contained elementary stream (vorbis, theora) when it detects such a stream in the Ogg stream. Likewise, it will delete the pad when the stream ends. This principle is very useful for demuxer elements, for example.

Running `gst-inspect oggdemux` will show that the element has only one pad: a sink pad called 'sink'. The other pads are "dormant". You can see this in the pad template because there is an "Exists: Sometimes" property. Depending on the type of Ogg file you play, the pads will be created. We will see that this is very important when you are going to create dynamic pipelines. You can attach a signal handler to an element to inform you when the element has created a new pad from one of its "sometimes" pad templates. The following piece of code is an example of how to do this:

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

static void
cb_new_pad (GstElement *element,
     GstPad     *pad,
     gpointer    data)
{
  g_print ("A new pad %s was created\n", gst_pad_get_name (pad));

  /* here, you would setup a new pad link for the newly created pad */
[..]

}

int
main(int argc, char *argv[])
{
  GstElement *pipeline, *source, *demux;

  /* init */
  gst_init (&argc, &argv);
  /* create elements */
  pipeline = gst_pipeline_new ("my_pipeline");
  source = gst_element_factory_make ("filesrc", "source");
  g_object_set (source, "location", argv[1], NULL);
  demux = gst_element_factory_make ("oggdemux", "demuxer");

  /* you would normally check that the elements were created properly */

  /* put together a pipeline */
  gst_bin_add_many (GST_BIN (pipeline), source, demux, NULL);
  gst_element_link (source, demux);

  /* listen for newly created pads */
  g_signal_connect (demux, "new-pad", G_CALLBACK (cb_new_pad), NULL);

  /* start the pipeline */
  gst_element_set_state (GST_ELEMENT (pipeline), GST_STATE_PLAYING);
  while (gst_bin_iterate (GST_BIN (pipeline)));

[..]

}
```

Request pads

An element can also have request pads. These pads are not created automatically but are only created on demand. This is very useful for multiplexers, aggregators and tee elements. Aggregators are elements that merge the content of several input streams together into one output stream. Tee elements are the reverse: they are elements that have one input stream and copy this stream to each of their output pads, which are created on request. Whenever an application needs another copy of the stream, it can simply request a new output pad from the tee element.

The following piece of code shows how you can request a new output pad from a "tee" element:

```
static void
some_function (GstElement *tee)
{
  GstPad * pad;

  pad = gst_element_get_request_pad (tee, "src%d");
  g_print ("A new pad %s was created\n", gst_pad_get_name (pad));

  /* here, you would link the pad */
[..]
}
```

The `gst_element_get_request_pad ()` method can be used to get a pad from the element based on the name of the pad template. It is also possible to request a pad that is compatible with another pad template. This is very useful if you want to link an element to a multiplexer element and you need to request a pad that is compatible. The method `gst_element_get_compatible_pad ()` can be used to request a compatible pad, as shown in the next example. It will request a compatible pad from an Ogg multiplexer from any input.

```
static void
link_to_multiplexer (GstPad *tolink_pad,
       GstElement *mux)
{
  GstPad *pad;
  pad = gst_element_get_compatible_pad (mux, tolink_pad);
  gst_pad_link (tolinkpad, pad);

  g_print ("A new pad %s was created and linked to %s\n",
    gst_pad_get_name (pad), gst_pad_get_name (tolink_pad));
}
```

Capabilities of a pad

Since the pads play a very important role in how the element is viewed by the outside world, a mechanism is implemented to describe the data that can flow or currently flows through the pad by using capabilities. Here,w e will briefly describe what capabilities are and how to use them, enough to get an understanding of the concept. For an in-depth look into capabilities and a list of all capabilities defined in `GStreamer` , see the Plugin Writers Guide[2].

Capabilities are attached to pad templates and to pads. For pad templates, it will describe the types of media that may stream over a pad created from this template. For pads, it can either be a list of possible caps (usually a copy of the pad template's capabilities), in which case the pad is not yet negotiated, or it is the type of media that currently streams over this pad, in which case the pad has been negotiated already.

Dissecting capabilities

A pads capabilities are described in a `GstCaps` object. Internally, a `GstCaps` [3] will contain one or more `GstStructure` [4] that will describe one media type. A negotiated pad will have capabilities set that contain exactly *one* structure. Also, this structure will contain only *fixed* values. These constraints are not true for unnegotiated pads or pad templates.

As an example, below is a dump of the capabilities of the "vorbisdec" element, which you will get by running **gst-inspect vorbisdec**. You will see two pads: a source and a sink pad. Both of these pads are always available, and both have capabilities attached to them. The sink pad will accept vorbis-encoded audio data, with the mime-type "audio/x-vorbis". The source pad will be used to send raw (decoded) audio samples to the next element, with a raw audio mime-type (either "audio/x-raw-int" or "audio/x-raw-float"). The source pad will also contain properties for the audio samplerate and the amount of channels, plus some more that you don't need to worry about for now.

```
Pad Templates:
  SRC template: 'src'
    Availability: Always
    Capabilities:
      audio/x-raw-float
                   rate: [ 8000, 50000 ]
               channels: [ 1, 2 ]
             endianness: 1234
                  width: 32
          buffer-frames: 0

  SINK template: 'sink'
    Availability: Always
    Capabilities:
      audio/x-vorbis
```

Properties and values

Properties are used to describe extra information for capabilities. A property consists of a key (a string) and a value. There are different possible value types that can be used:

- Basic types, this can be pretty much any `GType` registered with Glib. Those properties indicate a specific, non-dynamic value for this property. Examples include:
  - An integer value (`G_TYPE_INT`): the property has this exact value.
  - A boolean value (`G_TYPE_BOOLEAN`): the property is either TRUE or FALSE.
  - A float value (`G_TYPE_FLOAT`): the property has this exact floating point value.
  - A string value (`G_TYPE_STRING`): the property contains a UTF-8 string.
- Range types are `GType`s registered by `GStreamer` to indicate a range of possible values. They are used for indicating allowed audio samplerate values or supported video sizes. The two types defined in `GStreamer` are:
  - An integer range value (`GST_TYPE_INT_RANGE`): the property denotes a range of possible integers, with a lower and an upper boundary. The "vorbisdec" element, for example, has a rate property that can be between 8000 and 50000.
  - A float range value (`GST_TYPE_FLOAT_RANGE`): the property denotes a range of possible floating point values, with a lower and an upper boundary.
- A list value (`GST_TYPE_LIST`): the property can take any value from a list of basic values given in this list.

What capabilities are used for

Capabilities describe the type of data that is streamed between two pads, or that one pad (template) supports. This makes them very useful for various purposes:

- Autoplugging: automatically finding elements to link to a pad based on its capabilities. All autopluggers use this method.
- Compatibility detection: when two pads are linked, `GStreamer` can verify if the two pads are talking about the same media type. The process of linking two pads and checking if they are compatible is called "caps negotiation".
- Metadata: by reading the capabilities from a pad, applications can provide information about the type of media that is being streamed over the pad, which is information about the stream thatis currently being played back.
- Filtering: an application can use capabilities to limit the possible media types that can stream between two pads to a specific subset of their supported stream types. An application can, for example, use "filtered caps" to set a specific (non-fixed) video size that will stream between two pads.

Using capabilities for metadata

A pad can have a set (i.e. one or more) of capabilities attached to it. You can get values of properties in a set of capabilities by querying individual properties of one structure. You can get a structure from a caps using `gst_caps_get_structure ()`:

```
static void
read_video_props (GstCaps *caps)
```

```
{
  gint width, height;
  const GstStructure *str;

  str = gst_caps_get_structure (caps);
  if (!gst_structure_get_int (str, "width", &width) ||
      !gst_structure_get_int (str, "height", &height)) {
    g_print ("No width/height available\n");
    return;
  }

  g_print ("The video size of this set of capabilities is %dx%d\n",
    width, height);
}
```

Creating capabilities for filtering

While capabilities are mainly used inside a plugin to describe the media type of the pads, the application programmer also has to have basic understanding of capabilities in order to interface with the plugins, especially when using filtered caps. When you're using filtered caps or fixation, you're limiting the allowed types of media that can stream between two pads to a subset of their supported media types. You do this by filtering using your own set of capabilities. In order to do this, you need to create your own `GstCaps` . The simplest way to do this is by using the convenience function `gst_caps_new_simple ()`:

```
static void
link_pads_with_filter (GstPad *one,
          GstPad *other)
{
  GstCaps *caps;

  caps = gst_caps_new_simple ("video/x-raw-yuv",
          "width", G_TYPE_INT, 384,
          "height", G_TYPE_INT, 288,
          "framerate", G_TYPE_DOUBLE, 25.,
          NULL);
  gst_pad_link_filtered (one, other, caps);
}
```

In some cases, you will want to create a more elaborate set of capabilities to filter a link between two pads. Then, this function is too simplistic and you'll want to use the method `gst_caps_new_full ()`:

```
static void
link_pads_with_filter (GstPad *one,
                       GstPad *other)
{
  GstCaps *caps;

  caps = gst_caps_new_full (
      gst_structure_new ("video/x-raw-yuv",
    "width", G_TYPE_INT, 384,
    "height", G_TYPE_INT, 288,
    "framerate", G_TYPE_DOUBLE, 25.,
    NULL),
      gst_structure_new ("video/x-raw-rgb",
    "width", G_TYPE_INT, 384,
    "height", G_TYPE_INT, 288,
    "framerate", G_TYPE_DOUBLE, 25.,
    NULL),
            NULL);

    gst_pad_link_filtered      (one,   other,   caps);
}
```

See the API references for the full API of `GstStructure` and `GstCaps`.

Ghost pads

You can see from Figure 7-1 how a bin has no pads of its own. This is where "ghost pads" come into play.

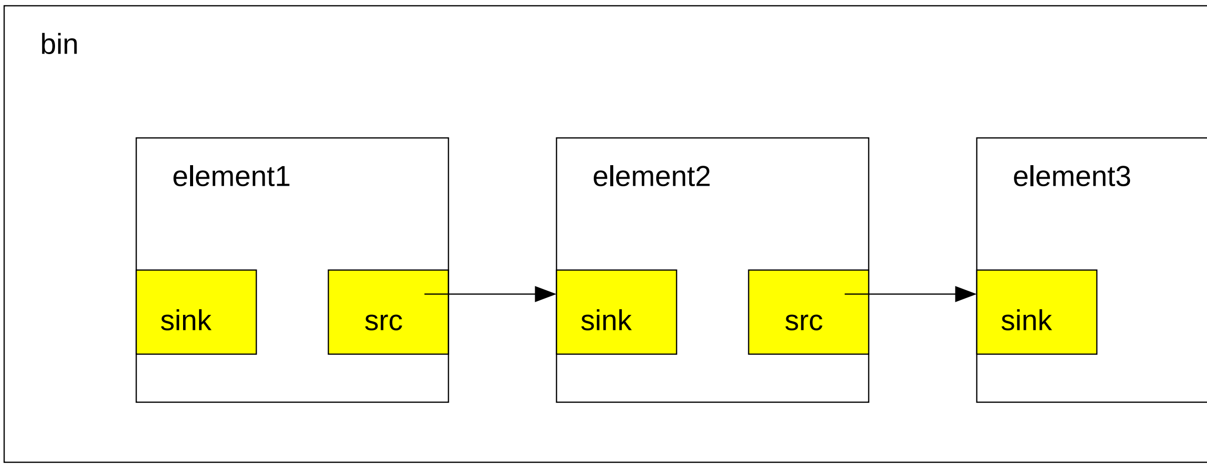

Figure 7-1. Visualisation of a `GstBin` [5] element without ghost pads

A ghost pad is a pad from some element in the bin that can be accessed directly from the bin as well. Compare it to a symbolic link in UNIX filesystems. Using ghost pads on bins, the bin also has a pad and can transparently be used as an element in other parts of your code.

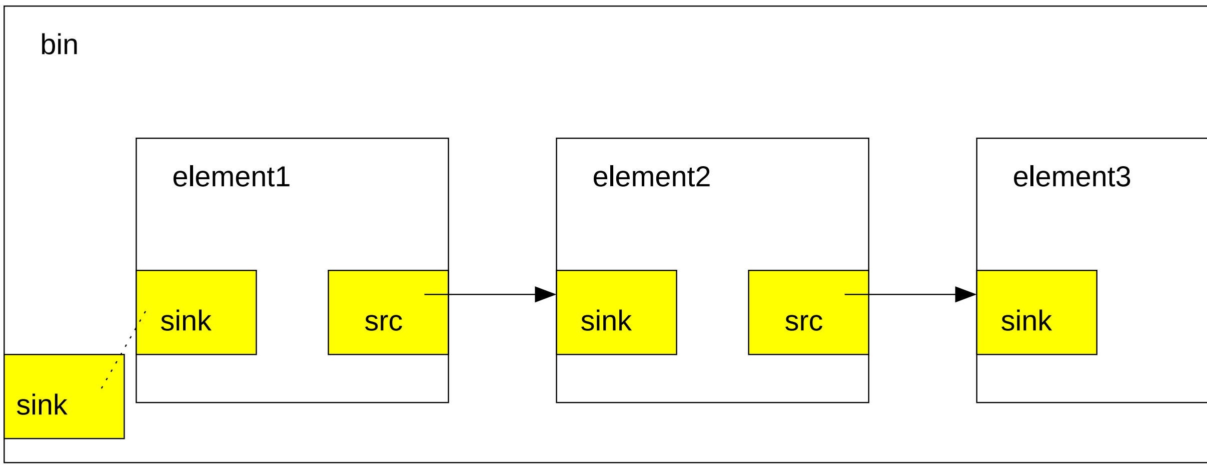

Figure 7-2. Visualisation of a `GstBin` [6] element with a ghost pad

Figure 7-2 is a representation of a ghost pad. The sink pad of element one is now also a pad of the bin. Obviously, ghost pads can be added to any type of elements, not just to a `GstBin`.

A ghostpad is created using the function `gst_element_add_ghost_pad ()`:

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

int
main (int    argc,
      char *argv[])
{
  GstElement *bin, *sink;

  /* init */
  gst_init (&argc, &argv);

  /* create element, add to bin, add ghostpad */
  sink = gst_element_factory_make ("fakesink", "sink");
  bin = gst_bin_new ("mybin");
  gst_bin_add (GST_BIN (bin), sink);
  gst_element_add_ghost_pad (bin,
      gst_element_get_pad (sink, "sink"), "sink");

[..]

}
```

In the above example, the bin now also has a pad: the pad called "sink" of the given element. The bin can, from here on, be used as a substitute for the sink element. You could, for example, link another element to the bin.

Notes

1. In reality, there is no objection to data flowing from a source pad to the sink pad of an element upstream (to the left of this element in drawings). Data will, however, always flow from a source pad of one element to the sink pad of another.
2. http://gstreamer.freedesktop.org/data/doc/gstreamer/head/pwg/html/index.html
3. ../../gstreamer/html/gstreamer-GstCaps.html
4. ../../gstreamer/html/gstreamer-GstStructure.html
5. ../../gstreamer/html/GstBin.html
6. ../../gstreamer/html/GstBin.html

Chapter 8. Buffers and Events

The data flowing through a pipeline consists of a combination of buffers and events. Buffers contain the actual pipeline data. Events contain control information, such as seeking information and end-of-stream notifiers. All this will flow through the pipeline automatically when it's running. This chapter is mostly meant to explain the concept to you; you don't need to do anything for this.

Buffers

Buffers contain the data that will flow through the pipeline you have created. A source element will typically create a new buffer and pass it through a pad to the next element in the chain. When using the GStreamer infrastructure to create a media pipeline you will not have to deal with buffers yourself; the elements will do that for you.

A buffer consists, amongst others, of:

- A pointer to a piece of memory.
- The size of the memory.
- A timestamp for the buffer.
- A refcount that indicates how many elements are using this buffer. This refcount will be used to destroy the buffer when no element has a reference to it.

The simple case is that a buffer is created, memory allocated, data put in it, and passed to the next element. That element reads the data, does something (like creating a new buffer and decoding into it), and unreferences the buffer. This causes the data to be free'ed and the buffer to be destroyed. A typical video or audio decoder works like this.

There are more complex scenarios, though. Elements can modify buffers in-place, i.e. without allocating a new one. Elements can also write to hardware memory (such as from video-capture sources) or memory allocated from the X-server using XShm). Buffers can be read-only, and so on.

Events

Events are control particles that are sent both up- and downstream in a pipeline along with buffers. Downstream events notify fellow elements of stream states. Possible events include discontinuities, flushes, end-of-stream notifications and so on. Upstream events are used both in application-element interaction as well as event-event interaction to request changes in stream state, such as seeks. For applications, only upstream events are important. Downstream events are just explained to get a more complete picture of the data concept.

Since most applications seek in time units, our example below does so too:

```
static void
seek_to_time (GstElement *element,
       guint64     time_ns)
{
  GstEvent *event;

  event = gst_event_new_seek (GST_SEEK_METHOD_SET |
              GST_FORMAT_TIME,
              time_ns);
  gst_element_send_event (element, event);
}
```

The function `gst_element_seek ()` is a shortcut for this. This is mostly just to show how it all works.

Chapter 9. Your first application

This chapter will summarize everything you've learned in the previous chapters. It describes all aspects of a simple `GStreamer` application, including initializing libraries, creating elements, packing elements together in a pipeline and playing this pipeline. By doing all this, you will be able to build a simple Ogg/Vorbis audio player.

Hello world

We're going to create a simple first application, a simple Ogg/Vorbis command-line audio player. For this, we will use only standard `GStreamer` components. The player will read a file specified on the command-line. Let's get started!

We've learned, in Chapter 4, that the first thing to do in your application is to initialize `GStreamer` by calling `gst_init ()`. Also, make sure that the application includes `gst/gst.h` so all function names and objects are properly defined. Use `#include <gst/gst.h>` to do that.

Next, you'll want to create the different elements using `gst_element_factory_make ()`. For an Ogg/Vorbis audio player, we'll need a source element that reads files from a disk. `GStreamer` includes this element under the name "filesrc". Next, we'll need something to parse the file and decoder it into raw audio. `GStreamer` has two elements for this: the first parses Ogg streams into elementary streams (video, audio) and is called "oggdemux". The second is a Vorbis audio decoder, it's conveniently called "vorbisdec". Since "oggdemux" creates dynamic pads for each elementary stream, you'll need to set a "new-pad" event handler on the "oggdemux" element, like you've learned in the Section called *Dynamic (or sometimes) pads* in Chapter 7, to link the Ogg parser and the Vorbis decoder elements together. At last, we'll also need an audio output element, we will use "alsasink", which outputs sound to an ALSA audio device.

The last thing left to do is to add all elements into a container element, a `GstPipeline`, and iterate this pipeline until we've played the whole song. We've previously learned how to add elements to a container bin in Chapter 6, and we've learned about element states in the Section called *Element States* in Chapter 5. We will use the function `gst_bin_sync_children_state ()` to synchronize the state of a bin on all of its contained children.

Let's now add all the code together to get our very first audio player:

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

/*
 * Global objects are usually a bad thing. For the purpose of this
 * example, we will use them, however.
 */

GstElement *pipeline, *source, *parser, *decoder, *sink;

static void
new_pad (GstElement *element,
  GstPad      *pad,
  gpointer     data)
{
  /* We can now link this pad with the audio decoder and
   * add both decoder and audio output to the pipeline. */
  gst_pad_link (pad, gst_element_get_pad (decoder, "sink"));
  gst_bin_add_many (GST_BIN (pipeline), decoder, sink, NULL);

  /* This function synchronizes a bins state on all of its
   * contained children. */
  gst_bin_sync_children_state (GST_BIN (pipeline));
}

int
main (int argc,
      char *argv[])
{
  /* initialize GStreamer */
  gst_init (&argc, &argv);

  /* check input arguments */
  if (argc != 2) {
    g_print ("Usage: %s <Ogg/Vorbis filename>\n", argv[0]);
    return -1;
  }

  /* create elements */
  pipeline = gst_pipeline_new ("audio-player");
  source = gst_element_factory_make ("filesrc", "file-source");
  parser = gst_element_factory_make ("oggdemux", "ogg-parser");
  decoder = gst_element_factory_make ("vorbisdec", "vorbis-decoder");
  sink = gst_element_factory_make ("alsasink", "alsa-output");

  /* set filename property on the file source */
  g_object_set (G_OBJECT (source), "location", argv[1], NULL);

  /* link together - note that we cannot link the parser and
   * decoder yet, becuse the parser uses dynamic pads. For that,
   * we set a new-pad signal handler. */
  gst_element_link (source, parser);
  gst_element_link (decoder, sink);
  g_signal_connect (parser, "new-pad", G_CALLBACK (new_pad), NULL);

  /* put all elements in a bin - or at least the ones we will use
   * instantly. */
  gst_bin_add_many (GST_BIN (pipeline), source, parser, NULL);

  /* Now set to playing and iterate. We will set the decoder and
   * audio output to ready so they initialize their memory already.
   * This will decrease the amount of time spent on linking these
   * elements when the Ogg parser emits the new-pad signal. */
  gst_element_set_state (decoder, GST_STATE_READY);
  gst_element_set_state (sink, GST_STATE_READY);
  gst_element_set_state (pipeline, GST_STATE_PLAYING);

  /* and now iterate - the rest will be automatic from here on.
   * When the file is finished, gst_bin_iterate () will return
   * FALSE, thereby terminating this loop. */
  while (gst_bin_iterate (GST_BIN (pipeline))) ;

  /* clean up nicely */
  gst_element_set_state (pipeline, GST_STATE_NULL);
  gst_object_unref (GST_OBJECT (pipeline));

  return 0;
}
```

We now have created a complete pipeline. We can visualise the pipeline as follows:

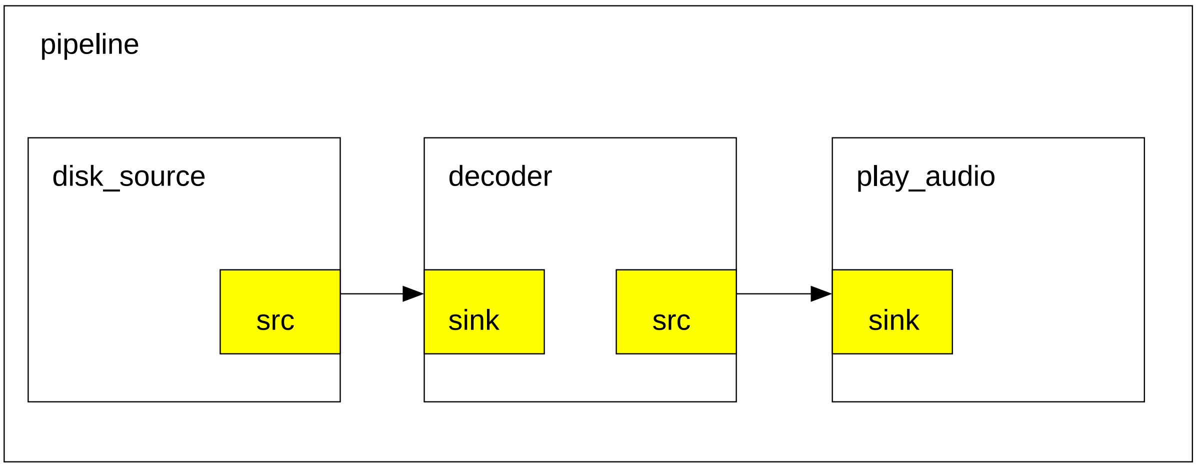

Figure 9-1. The "hello world" pipeline

Compiling and Running helloworld.c

To compile the helloworld example, use: **gcc -Wall $(pkg-config --cflags --libs gstreamer-0.8) helloworld.c -o helloworld**. `GStreamer` makes use of **pkg-config** to get compiler and linker flags needed to compile this application. If you're running a non-standard installation, make sure the `PKG_CONFIG_PATH` environment variable is set to the correct location (`$libdir/pkgconfig`). application against the uninstalled location.

You can run this example application with **./helloworld file.ogg**. Substitute `file.ogg` with your favourite Ogg/Vorbis file.

Conclusion

This concludes our first example. As you see, setting up a pipeline is very low-level but powerful. You will see later in this manual how you can create a more powerful media player with even less effort using higher-level interfaces. We will discuss all that in Part IV in *GStreamer Application Development Manual (0.8.9.2)*. We will first, however, go more in-depth into more advanced `GStreamer` internals.

It should be clear from the example that we can very easily replace the "filesrc" element with some other element that reads data from a network, or some other data source element that is better integrated with your desktop environment. Also, you can use other decoders and parsers to support other media types. You can use another audio sink if you're not running Linux, but Mac OS X, Windows or FreeBSD, or you can instead use a filesink to write audio files to disk instead of playing them back. By using an audio card source, you can even do audio capture instead of playback. All this shows the reusability of `GStreamer` elements, which is its greatest advantage.

Chapter 10. Position tracking and seeking

So far, we've looked at how to create a pipeline to do media processing and how to make it run ("iterate"). Most application developers will be interested in providing feedback to the user on media progress. Media players, for example, will want to show a slider showing the progress in the song, and usually also a label indicating stream length. Transcoding applications will want to show a progress bar on how much % of the task is done. `GStreamer` has built-in support for doing all this using a concept known as *querying*. Since seeking is very similar, it will be discussed here as well. Seeking is done using the concept of *events*.

Querying: getting the position or length of a stream

Querying is defined as requesting a specific stream-property related to progress tracking. This includes getting the length of a stream (if available) or getting the current position. Those stream properties can be retrieved in various formats such as time, audio samples, video frames or bytes. The functions used are `gst_element_query ()` and `gst_pad_query ()`.

Obviously, using either of the above-mentioned functions requires the application to know *which* element or pad to run the query on. This is tricky, but there are some good sides to the story. The good thing is that elements (or, rather, pads - since `gst_element_query ()` internally calls `gst_pad_query ()`) forward ("dispatch") events and queries to peer pads (or elements) if they don't handle it themselves. The bad side is that some elements (or pads) will handle events, but not the specific formats that you want, and therefore it still won't work.

Most queries will, fortunately, work fine. Queries are always dispatched backwards. This means, effectively, that it's easiest to run the query on your video or audio output element, and it will take care of dispatching the query to the element that knows the answer (such as the current position or the media length; usually the demuxer or decoder).

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

gint
main (gint   argc,
      gchar *argv[])
{
  GstElement *sink, *pipeline;

[..]

  /* run pipeline */
  do {
    gint64 len, pos;
    GstFormat fmt = GST_FORMAT_TIME;

    if (gst_element_query (sink, GST_QUERY_POSITION, &fmt, &pos) &&
        gst_element_query (sink, GST_QUERY_TOTAL, &fmt, &len)) {
      g_print ("Time: %" GST_TIME_FORMAT " / %" GST_TIME_FORMAT "\r",
        GST_TIME_ARGS (pos), GST_TIME_ARGS (len));
    }
  } while (gst_bin_iterate (GST_BIN (pipeline)));

[..]

}
```

If you are having problems with the dispatching behaviour, your best bet is to manually decide which element to start running the query on. You can get a list of supported formats and query-types with `gst_element_get_query_types ()` and `gst_element_get_formats ()`.

Events: seeking (and more)

Events work in a very similar way as queries. Dispatching, for example, works exactly the same for events (and also has the same limitations). Although there are more ways in which applications and elements can interact using events, we will only focus on seeking here. This is done using the seek-event. A seek-event contains a seeking offset, a seek method (which indicates relative to what the offset was given), a seek format (which is the unit of the offset, e.g. time, audio samples, video frames or bytes) and optionally a set of seeking-related flags (e.g. whether internal buffers should be flushed). The behaviour of a seek is also wrapped in the function `gst_element_seek ()`.

```
static void
seek_to_time (GstElement *audiosink,
              gint64      time_nanonseconds)
{
  gst_element_seek (audiosink,
      GST_SEEK_METHOD_SET | GST_FORMAT_TIME |
      GST_SEEK_FLAG_FLUSH, time_nanoseconds);
}
```

Chapter 11. Metadata

`GStreamer` makes a clear distinction between two types of metadata, and has support for both types. The first is stream tags, which describe the content of a stream in a non-technical way. Examples include the author of a song, the title of that very same song or the album it is a part of. The other type of metadata is stream-info, which is a somewhat technical description of the properties of a stream. This can include video size, audio samplerate, codecs used and so on. Tags are handled using the `GStreamer` tagging system. Stream-info can be retrieved from a `GstPad` .

Stream information

Stream information can most easily be read by reading them from a `GstPad` . This has already been discussed before in the Section called *Using capabilities for metadata* in Chapter 7. Therefore, we will skip it here.

Tag reading

Tag reading is remarkably simple in `GStreamer` Every element supports the “found-tag” signal, which will be fired each the time the element reads tags from the stream. A `GstBin` will conveniently forward tags found by its childs. Therefore, in most applications, you will only need to connect to the “found-tag” signal on the top-most bin in your pipeline, and you will automatically retrieve all tags from the stream.

Note, however, that the “found-tag” might be fired multiple times and by multiple elements in the pipeline. It is the application’s responsibility to put all those tags together and display them to the user in a nice, coherent way.

Tag writing

WRITEME

Chapter 12. Interfaces

In the Section called *Using an element as a `GObject`* in Chapter 5, you have learned how to use `GObject` properties as a simple way to do interaction between applications and elements. This method suffices for the simple'n'straight settings, but fails for anything more complicated than a getter and setter. For the more complicated use cases, `GStreamer` uses interfaces based on the Glib `GInterface` type.

Most of the interfaces handled here will not contain any example code. See the API references for details. Here, we will just describe the scope and purpose of each interface.

The Mixer interface

The mixer interface provides a uniform way to control the volume on a hardware (or software) mixer. The interface is primarily intended to be implemented by elements for audio inputs and outputs that talk directly to the hardware (e.g. OSS or ALSA plugins).

Using this interface, it is possible to control a list of tracks (such as Line-in, Microphone, etc.) from a mixer element. They can be muted, their volume can be changed and, for input tracks, their record flag can be set as well.

Example plugins implementing this interface include the OSS elements (osssrc, osssink, ossmixer) and the ALSA plugins (alsasrc, alsasink and alsamixer).

The Tuner interface

The tuner interface is a uniform way to control inputs and outputs on a multi-input selection device. This is primarily used for input selection on elements for TV- and capture-cards.

Using this interface, it is possible to select one track from a list of tracks supported by that tuner-element. The tuner will than select that track for media-processing internally. This can, for example, be used to switch inputs on a TV-card (e.g. from Composite to S-video).

This interface is currently only implemented by the Video4linux and Video4linux2 elements.

The Color Balance interface

The colorbalance interface is a way to control video-related properties on an element, such as brightness, contrast and so on. It's sole reason for existance is that, as far as its authors know, there's no way to dynamically register properties using `GObject`.

The colorbalance interface is implemented by several plugins, including xvimagesink and the Video4linux and Video4linux2 elements.

The Property Probe interface

The property probe is a way to autodetect allowed values for a `GObject` property. It's primary use (and the only thing that we currently use it for) is to autodetect devices in several elements. For example, the OSS elements use this interface to detect all OSS devices on a system. Applications can then "probe" this property and get a list of detected devices. Given the overlap between HAL and the practical implementations of this interface, this might in time be deprecated in favour of HAL.

This interface is currently implemented by many elements, including the ALSA, OSS, Video4linux and Video4linux2 elements.

The X Overlay interface

The X Overlay interface was created to solve the problem of embedding video streams in an application window. The application provides an X-window to the element implementing this interface to draw on, and the element will then use this X-window to draw on rather than creating a new toplevel window. This is useful to embed video in video players.

This interface is implemented by, amongst others, the Video4linux and Video4linux2 elements and by ximagesink, xvimagesink and sdlvideosink.

Chapter 13. Clocks in GStreamer

WRITEME

Chapter 14. Dynamic Parameters

Getting Started

The Dynamic Parameters subsystem is contained within the `gstcontrol` library. You need to include the header in your application's source file:

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

Your application should link to the shared library `gstcontrol`.

The `gstcontrol` library needs to be initialized when your application is run. This can be done after the the GStreamer library has been initialized.

```
  ...
  gst_init(&argc,&argv);
  gst_control_init(&argc,&argv);
  ...
```

Creating and Attaching Dynamic Parameters

Once you have created your elements you can create and attach dparams to them. First you need to get the element's dparams manager. If you know exactly what kind of element you have, you may be able to get the dparams manager directly. However if this is not possible, you can get the dparams manager by calling `gst_dpman_get_manager`.

Once you have the dparams manager, you must set the mode that the manager will run in. There is currently only one mode implemented called `"synchronous"` - this is used for real-time applications where the dparam value cannot be known ahead of time (such as a slider in a GUI). The mode is called `"synchronous"` because the dparams are polled by the element for changes before each buffer is processed. Another yet-to-be-implemented mode is `"asynchronous"`. This is used when parameter changes are known ahead of time - such as with a timelined editor. The mode is called `"asynchronous"` because parameter changes may happen in the middle of a buffer being processed.

```
  GstElement *sinesrc;
  GstDParamManager *dpman;
  ...
  sinesrc = gst_element_factory_make("sinesrc","sine-source");
  ...
  dpman = gst_dpman_get_manager (sinesrc);
  gst_dpman_set_mode(dpman, "synchronous");
```

If you don't know the names of the required dparams for your element you can call `gst_dpman_list_dparam_specs(dpman)` to get a NULL terminated array of param specs. This array should be freed after use. You can find the name of the required dparam by calling `g_param_spec_get_name` on each param spec in the array. In our example, `"volume"` will be the name of our required dparam.

Each type of dparam currently has its own `new` function. This may eventually be replaced by a factory method for creating new instances. A default dparam instance can be created with the `gst_dparam_new` function. Once it is created it can be attached to a required dparam in the element.

```
GstDParam *volume;
...
volume = gst_dparam_new(G_TYPE_DOUBLE);
if (gst_dpman_attach_dparam (dpman, "volume", volume)){
  /* the dparam was successfully attached */
  ...
}
```

Changing Dynamic Parameter Values

All interaction with dparams to actually set the dparam value is done through simple GObject properties. There is a property value for each type that dparams supports - these currently being `"value_double"`, `"value_float"`, `"value_int"` and `"value_int64"`. To set the value of a dparam, simply set the property which matches the type of your dparam instance.

```
#define ZERO(mem) memset(&mem, 0, sizeof(mem))
...

  gdouble set_to_value;
  GstDParam *volume;
  GValue set_val;
  ZERO(set_val);
  g_value_init(&set_val, G_TYPE_DOUBLE);
  ...
  g_value_set_double(&set_val, set_to_value);
  g_object_set_property(G_OBJECT(volume), "value_double", &set_val);
```

Or if you create an actual GValue instance:

```
  gdouble set_to_value;
  GstDParam *volume;
  GValue *set_val;
  set_val = g_new0(GValue,1);
  g_value_init(set_val, G_TYPE_DOUBLE);
  ...
  g_value_set_double(set_val, set_to_value);
  g_object_set_property(G_OBJECT(volume), "value_double", set_val);
```

Different Types of Dynamic Parameter

There are currently only two implementations of dparams so far. They are both for real-time use so should be run in the `"synchronous"` mode.

GstDParam - the base dparam type

All dparam implementations will subclass from this type. It provides a basic implementation which simply propagates any value changes as soon as it can. A new instance can be created with the function `GstDParam* gst_dparam_new (GType type)`. It has the following object properties:

- `"value_double"` - the property to set and get if it is a double dparam
- `"value_float"` - the property to set and get if it is a float dparam
- `"value_int"` - the property to set and get if it is an integer dparam

- `"value_int64"` - the property to set and get if it is a 64 bit integer dparam
- `"is_log"` - readonly boolean which is TRUE if the param should be displayed on a log scale
- `"is_rate"` - readonly boolean which is TRUE if the value is a proportion of the sample rate. For example with a sample rate of 44100, 0.5 would be 22050 Hz and 0.25 would be 11025 Hz.

GstDParamSmooth - smoothing real-time dparam

Some parameter changes can create audible artifacts if they change too rapidly. The GstDParamSmooth implementation can greatly reduce these artifacts by limiting the rate at which the value can change. This is currently only supported for double and float dparams - the other types fall back to the default implementation. A new instance can be created with the function `GstDParam* gst_dpsmooth_new (GType type)` . It has the following object properties:

- `"update_period"` - an int64 value specifying the number nanoseconds between updates. This will be ignored in `"synchronous"` mode since the buffer size dictates the update period.
- `"slope_time"` - an int64 value specifying the time period to use in the maximum slope calculation
- `"slope_delta_double"` - a double specifying the amount a double value can change in the given slope_time.
- `"slope_delta_float"` - a float specifying the amount a float value can change in the given slope_time.

Audible artifacts may not be completely eliminated by using this dparam. The only way to eliminate artifacts such as "zipper noise" would be for the element to implement its required dparams using the array method. This would allow dparams to change parameters at the sample rate which should eliminate any artifacts.

Timelined dparams

A yet-to-be-implemented subclass of GstDParam will add an API which allows the creation and manipulation of points on a timeline. This subclass will also provide a dparam implementation which uses linear interpolation between these points to find the dparam value at any given time. Further subclasses can extend this functionality to implement more exotic interpolation algorithms such as splines.

Chapter 15. Threads

GStreamer has support for multithreading through the use of the `GstThread` [1] object. This object is in fact a special `GstBin` [2] that will start a new thread (using Glib's `GThread` system) when started.

To create a new thread, you can simply use `gst_thread_new ()`. From then on, you can use it similar to how you would use a `GstBin`. You can add elements to it, change state and so on. The largest difference between a thread and other bins is that the thread does not require iteration. Once set to the `GST_STATE_PLAYING` state, it will iterate its contained children elements automatically.

Figure 15-1 shows how a thread can be visualised.

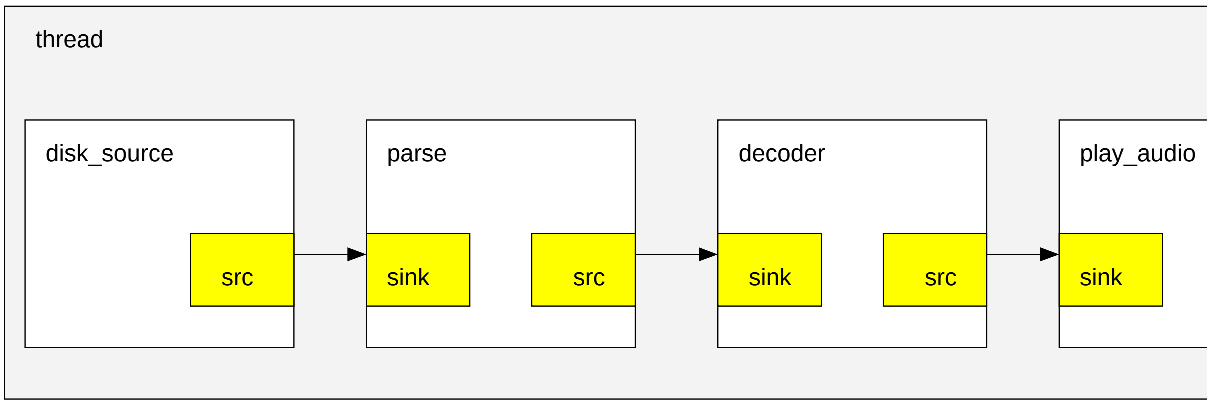

Figure 15-1. A thread

When would you want to use a thread?

There are several reasons to use threads. However, there's also some reasons to limit the use of threads as much as possible. We will go into the drawbacks of threading in `GStreamer` in the next section. Let's first list some situations where threads can be useful:

- Data buffering, for example when dealing with network streams or when recording data from a live stream such as a video or audio card. Short hickups elsewhere in the pipeline will not cause data loss. See Figure 15-2 for a visualization of this idea.
- Synchronizing output devices, e.g. when playing a stream containing both video and audio data. By using threads for both outputs, they will run independently and their synchronization will be better.
- Data pre-rolls. You can use threads and queues (thread boundaries) to cache a few seconds of data before playing. By using this approach, the whole pipeline will already be setup and data will already be decoded. When activating the rest of the pipeline, the switch from PAUSED to PLAYING will be instant.

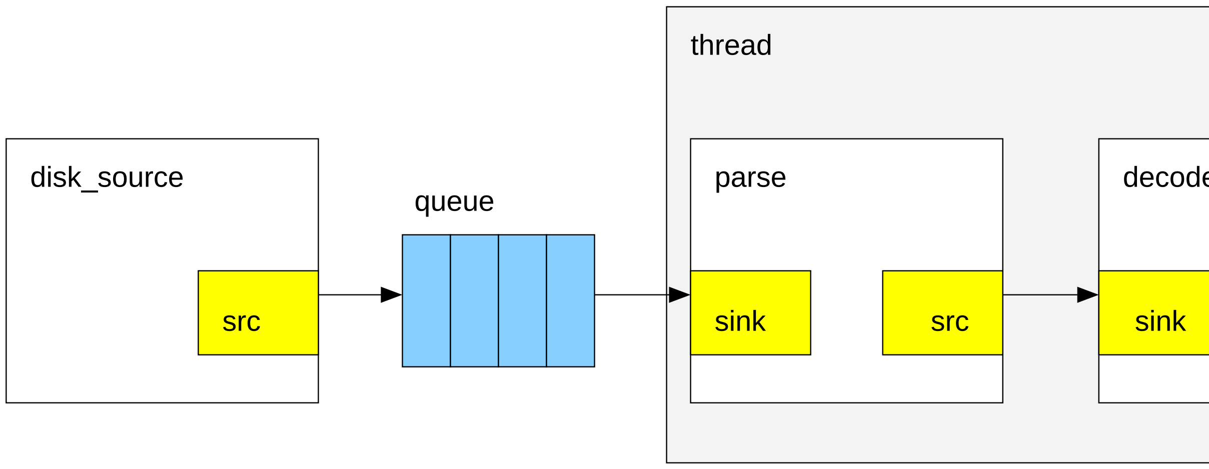

Figure 15-2. a two-threaded decoder with a queue

Above, we've mentioned the "queue" element several times now. A queue is a thread boundary element. It does so by using a classic provider/receiver model as learned in threading classes at universities all around the world. By doing this, it acts both as a means to make data throughput between threads threadsafe, and it can also act as a buffer. Queues have several `GObject` properties to be configured for specific uses. For example, you can set lower and upper tresholds for the element. If there's less data than the lower treshold (default: disabled), it will block output. If there's more data than the upper treshold, it will block input or (if configured to do so) drop data.

Constraints placed on the pipeline by the GstThread

Within the pipeline, everything is the same as in any other bin. The difference lies at the thread boundary, at the link between the thread and the outside world (containing bin). Since `GStreamer` is fundamentally buffer-oriented rather than byte-oriented, the natural solution to this problem is an element that can "buffer" the buffers between the threads, in a thread-safe fashion. This element is the "queue" element. A queue should be placed in between any two elements whose pads are linked together while the elements live in different threads. It doesn't matter if the queue is placed in the containing bin or in the thread itself, but it needs to be present on one side or the other to enable inter-thread communication.

If you are writing a GUI application, making the top-level bin a thread will make your GUI more responsive. If it were a pipeline instead, it would have to be iterated by your application's event loop, which increases the latency between events (say, keyboard presses) and responses from the GUI. In addition, any slight hang in the GUI would delay iteration of the pipeline, which (for example) could cause pops in the output of the sound card, if it is an audio pipeline.

A problem with using threads is, however, thread contexts. If you connect to a signal that is emitted inside a thread, then the signal handler for this thread *will be executed in that same thread*! This is very important to remember, because many graphical toolkits can not run multi-threaded. Gtk+, for example, only allows threaded access to UI objects if you explicitely use mutexes. Not doing so will result in random crashes and X errors. A solution many people use is to place an idle handler in the signal handler, and have the actual signal emission code be executed in the idle handler, which will be executed from the mainloop.

Generally, if you use threads, you will encounter some problems. Don't hesitate to ask us for help in case of problems.

A threaded example application

As an example we show the helloworld program that we coded in Chapter 9 using a thread. Note that the whole application lives in a thread (as opposed to half of the application living in a thread and the other half being another thread or a pipeline). Therefore, it does not need a queue element in this specific case.

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

GstElement *thread, *source, *decodebin, *audiosink;

static gboolean
idle_eos (gpointer data)
{
  g_print ("Have idle-func in thread %p\n", g_thread_self ());
  gst_main_quit ();

  /* do this function only once */
  return FALSE;
}

/*
 * EOS will be called when the src element has an end of stream.
 * Note that this function will be called in the thread context.
 * We will place an idle handler to the function that really
 * quits the application.
 */
static void
cb_eos (GstElement *thread,
 gpointer data)
{
  g_print ("Have eos in thread %p\n", g_thread_self ());
  g_idle_add ((GSourceFunc) idle_eos, NULL);
}

/*
 * On error, too, you'll want to forward signals to the main
 * thread, especially when using GUI applications.
 */

static void
cb_error (GstElement *thread,
   GstElement *source,
   GError *error,
   gchar *debug,
   gpointer data)
{
  g_print ("Error in thread %p: %s\n", g_thread_self (), error->message);
  g_idle_add ((GSourceFunc) idle_eos, NULL);
}

/*
 * Link new pad from decodebin to audiosink.
 * Contains no further error checking.
 */

static void
cb_newpad (GstElement *decodebin,
    GstPad *pad,
    gboolean last,
    gpointer data)
{
  gst_pad_link (pad, gst_element_get_pad (audiosink, "sink"));
  gst_bin_add (GST_BIN (thread), audiosink);
  gst_bin_sync_children_state (GST_BIN (thread));
}
```

```
gint
main (gint   argc,
      gchar *argv[])
{
  /* init GStreamer */
  gst_init (&argc, &argv);

  /* make sure we have a filename argument */
  if (argc != 2) {
    g_print ("usage: %s <Ogg/Vorbis filename>\n", argv[0]);
    return -1;
  }

  /* create a new thread to hold the elements */
  thread = gst_thread_new ("thread");
  g_signal_connect (thread, "eos", G_CALLBACK (cb_eos), NULL);
  g_signal_connect (thread, "error", G_CALLBACK (cb_error), NULL);

  /* create elements */
  source = gst_element_factory_make ("filesrc", "source");
  g_object_set (G_OBJECT (source), "location", argv[1], NULL);
  decodebin = gst_element_factory_make ("decodebin", "decoder");
  g_signal_connect (decodebin, "new-decoded-pad",
      G_CALLBACK (cb_newpad), NULL);
  audiosink = gst_element_factory_make ("alsasink", "audiosink");

  /* setup */
  gst_bin_add_many (GST_BIN (thread), source, decodebin, NULL);
  gst_element_link (source, decodebin);
  gst_element_set_state (audiosink, GST_STATE_PAUSED);
  gst_element_set_state (thread, GST_STATE_PLAYING);

  /* no need to iterate. We can now use a mainloop */
  gst_main ();

  /* unset */
  gst_element_set_state (thread, GST_STATE_NULL);
  gst_object_unref (GST_OBJECT (thread));

  return 0;
}
```

Notes

1. http://gstreamer.freedesktop.org/data/doc/gstreamer/stable/gstreamer/html/GstThread.html
2. http://gstreamer.freedesktop.org/data/doc/gstreamer/stable/gstreamer/html/GstBin.html

Chapter 16. Scheduling

By now, you've seen several example applications. All of them would set up a pipeline and call `gst_bin_iterate ()` to start media processing. You might have started wondering what happens during pipeline iteration. This whole process of media processing is called scheduling. Scheduling is considered one of the most complex parts of `GStreamer`. Here, we will do no more than give a global overview of scheduling, most of which will be purely informative. It might help in understanding the underlying parts of `GStreamer`.

The scheduler is responsible for managing the plugins at runtime. Its main responsibilities are:

- Managing data throughput between pads and elements in a pipeline. This might sometimes imply temporary data storage between elements.
- Calling functions in elements that do the actual data processing.
- Monitoring state changes and enabling/disabling elements in the chain.
- Selecting and distributing the global clock.

The scheduler is a pluggable component; this means that alternative schedulers can be written and plugged into GStreamer. There is usually no need for interaction in the process of choosing the scheduler, though. The default scheduler in `GStreamer` is called "opt". Some of the concepts discussed here are specific to opt.

Managing elements and data throughput

To understand some specifics of scheduling, it is important to know how elements work internally. Largely, there are four types of elements: `_chain ()`-based elements, `_loop ()`-based elements, `_get ()`-based elements and decoupled elements. Each of those have a set of features and limitations that are important for how they are scheduled.

- `_chain ()`-based elements are elements that have a `_chain ()`-function defined for each of their sinkpads. Those functions will receive data whenever input data is available. In those functions, the element can *push* data over its source pad(s) to peer elements. `_chain ()`-based elements cannot *pull* additional data from their sinkpad(s). Most elements in `GStreamer` are `_chain ()`-based.
- `_loop ()`-based elements are elements that have a `_loop ()`-function defined for the whole element. Inside this function, the element can pull buffers from its sink pad(s) and push data over its source pad(s) as it sees fit. Such elements usually require specific control over their input. Muxers and demuxers are usually `_loop ()`-based.
- `_get ()`-based elements are elements with only source pads. For each source pad, a `_get ()`-function is defined, which is called whenever the peer element needs additional input data. Most source elements are, in fact, `_get ()`-based. Such an element cannot actively push data.
- Decoupled elements are elements whose source pads are `_get ()`-based and whose sink pads are `_chain ()`-based. The `_chain ()`-function cannot push data over its source pad(s), however. One such element is the "queue" element, which is a thread boundary element. Since only one side of such elements are interesting for one particular scheduler, we can safely handle those elements as if they were either `_get ()`- or `_chain ()`-based. Therefore, we will further omit this type of elements in the discussion.

Obviously, the type of elements that are linked together have implications for how the elements will be scheduled. If a get-based element is linked to a loop-based element and the loop-based element requests data from its sinkpad, we can just call the get-function and be done with it. However, if two loop-based elements are linked to each other, it's a lot more complicated. Similarly, a loop-based element linked to a chain-based element is a lot easier than two loop-based elements linked to each other.

The default `GStreamer` scheduler, "opt", uses a concept of chains and groups. A group is a series of elements that can that do not require any context switches or intermediate data stores to be executed. In practice, this implies zero or one loop-based elements, one get-based element (at the beginning) and an infinite amount of chain-based elements. If there is a loop-based element, then the scheduler will simply call this elements loop-function to iterate. If there is no loop-based element, then data will be pulled from the get-based element and will be pushed over the chain-based elements.

A chain is a series of groups that depend on each other for data. For example, two linked loop-based elements would end up in different groups, but in the same chain. Whenever the first loop-based element pushes data over its source pad, the data will be temporarily stored inside the scheduler until the loop-function returns. When it's done, the loop-function of the second element will be called to process this data. If it pulls data from its sinkpad while no data is available, the scheduler will "emulate" a get-function and, in this function, iterate the first group until data is available.

The above is roughly how scheduling works in `GStreamer` . This has some implications for ideal pipeline design. An pipeline would ideally contain at most one loop-based element, so that all data processing is immediate and no data is stored inside the scheduler during group switches. You would think that this decreases overhead significantly. In practice, this is not so bad, however. It's something to keep in the back of your mind, nothing more.

Chapter 17. Autoplugging

In Chapter 9, you've learned to build a simple media player for Ogg/Vorbis files. By using alternative elements, you are able to build media players for other media types, such as Ogg/Speex, MP3 or even video formats. However, you would rather want to build an application that can automatically detect the media type of a stream and automatically generate the best possible pipeline by looking at all available elements in a system. This process is called autoplugging, and `GStreamer` contains high-quality autopluggers. If you're looking for an autoplugger, don't read any further and go to Chapter 19. This chapter will explain the *concept* of autoplugging and typefinding. It will explain what systems `GStreamer` includes to dynamically detect the type of a media stream, and how to generate a pipeline of decoder elements to playback this media. The same principles can also be used for transcoding. Because of the full dynamicity of this concept, `GStreamer` can be automatically extended to support new media types without needing any adaptations to its autopluggers.

We will first introduce the concept of MIME types as a dynamic and extendible way of identifying media streams. After that, we will introduce the concept of typefinding to find the type of a media stream. Lastly, we will explain how autoplugging and the `GStreamer` registry can be used to setup a pipeline that will convert media from one mimetype to another, for example for media decoding.

MIME-types as a way to identity streams

We have previously introduced the concept of capabilities as a way for elements (or, rather, pads) to agree on a media type when streaming data from one element to the next (see the Section called *Capabilities of a pad* in Chapter 7). We have explained that a capability is a combination of a mimetype and a set of properties. For most container formats (those are the files that you will find on your hard disk; Ogg, for example, is a container format), no properties are needed to describe the stream. Only a MIME-type is needed. A full list of MIME-types and accompanying properties can be found in the Plugin Writer's Guide[1].

An element must associate a MIME-type to its source and sink pads when it is loaded into the system. `GStreamer` knows about the different elements and what type of data they expect and emit through the `GStreamer` registry. This allows for very dynamic and extensible element creation as we will see.

In Chapter 9, we've learned to build a music player for Ogg/Vorbis files. Let's look at the MIME-types associated with each pad in this pipeline. Figure 17-1 shows what MIME-type belongs to each pad in this pipeline.

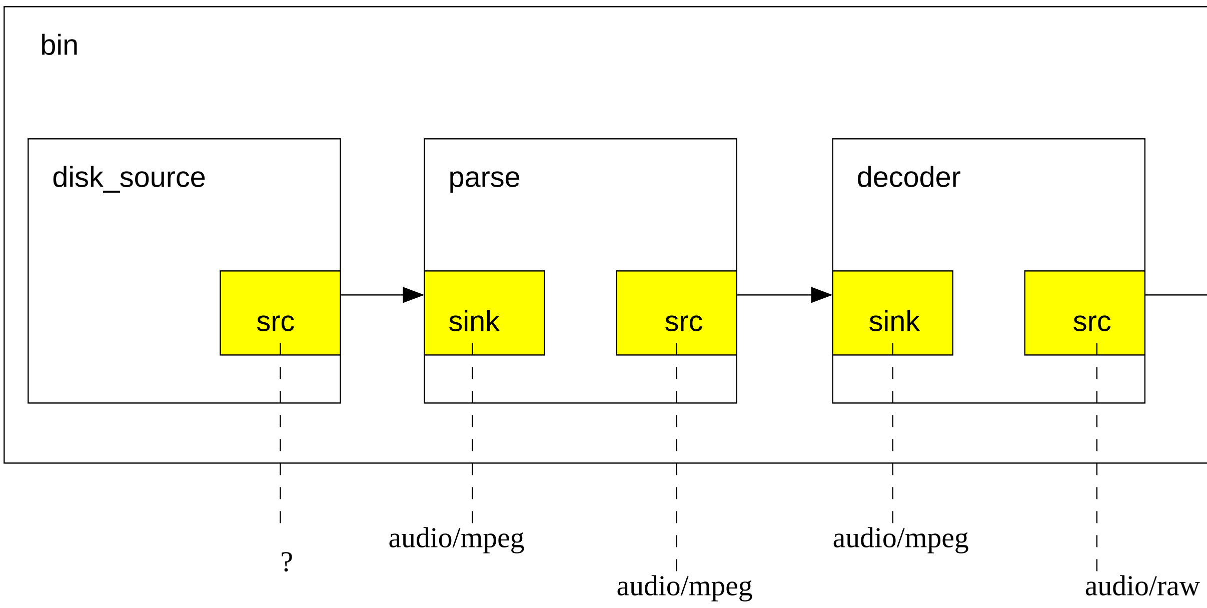

Figure 17-1. The Hello world pipeline with MIME types

Now that we have an idea how `GStreamer` identifies known media streams, we can look at methods `GStreamer` uses to setup pipelines for media handling and for media type detection.

Media stream type detection

Usually, when loading a media stream, the type of the stream is not known. This means that before we can choose a pipeline to decode the stream, we first need to detect the stream type. `GStreamer` uses the concept of typefinding for this. Typefinding is a normal part of a pipeline, it will read data for as long as the type of a stream is unknown. During this period, it will provide data to all plugins that implement a typefinder. when one of the typefinders recognizes the stream, the typefind element will emit a signal and act as a passthrough module from that point on. If no type was found, it will emit an error and further media processing will stop.

Once the typefind element has found a type, the application can use this to plug together a pipeline to decode the media stream. This will be discussed in the next section.

Plugins in `GStreamer` can, as mentioned before, implement typefinder functionality. A plugin implementing this functionality will submit a mimetype, optionally a set of file extensions commonly used for this media type, and a typefind function. Once this typefind function inside the plugin is called, the plugin will see if the data in this media stream matches a specific pattern that marks the media type identified by that mimetype. If it does, it will notify the typefind element of this fact, telling which mediatype was recognized and how certain we are that this stream is indeed that mediatype. Once this run has been completed for all plugins implementing a typefind functionality, the typefind element will tell the application what kind of media stream it thinks to have recognized.

The following code should explain how to use the typefind element. It will print the detected media type, or tell that the media type was not found. The next section will introduce more useful behaviours, such as plugging together a decoding pipeline.

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

static void
cb_typefound (GstElement *typefind,
        guint          probability,
        GstCaps       *caps,
        gpointer       data)
{
  gchar  *type;

  type = gst_caps_to_string (caps);
  g_print ("Media type %s found, probability %d%%\n", type, probability);
  g_free (type);

  /* done */
  (* (gboolean *) data) = TRUE;
}

static void
cb_error (GstElement *pipeline,
    GstElement    *source,
    GError        *error,
    gchar         *debug,
    gpointer       data)
{
  g_print ("Error: %s\n", error->message);

  /* done */
  (* (gboolean *) data) = TRUE;
}

gint
main (gint    argc,
      gchar  *argv[])
{
  GstElement *pipeline, *filesrc, *typefind;
  gboolean done = FALSE;

  /* init GStreamer */
  gst_init (&argc, &argv);

  /* check args */
  if (argc != 2) {
    g_print ("Usage: %s <filename>\n", argv[0]);
    return -1;
  }

  /* create a new pipeline to hold the elements */
  pipeline = gst_pipeline_new ("pipe");
  g_signal_connect (pipeline, "error", G_CALLBACK (cb_error), &done);

  /* create file source and typefind element */
  filesrc = gst_element_factory_make ("filesrc", "source");
  g_object_set (G_OBJECT (filesrc), "location", argv[1], NULL);
  typefind = gst_element_factory_make ("typefind", "typefinder");
  g_signal_connect (typefind, "have-type", G_CALLBACK (cb_typefound), &done);

  /* setup */
  gst_bin_add_many (GST_BIN (pipeline), filesrc, typefind, NULL);
  gst_element_link (filesrc, typefind);
  gst_element_set_state (GST_ELEMENT (pipeline), GST_STATE_PLAYING);

  /* now iterate until the type is found */
  do {
    if (!gst_bin_iterate (GST_BIN (pipeline)))
      break;
  } while (!done);

  /* unset */
  gst_element_set_state (GST_ELEMENT (pipeline), GST_STATE_NULL);
  gst_object_unref (GST_OBJECT (pipeline));

  return 0;
}
```

Once a media type has been detected, you can plug an element (e.g. a demuxer or decoder) to the source pad of the typefind element, and decoding of the media stream will start right after.

Plugging together dynamic pipelines

In this chapter we will see how you can create a dynamic pipeline. A dynamic pipeline is a pipeline that is updated or created while data is flowing through it. We will create a partial pipeline first and add more elements while the pipeline is playing. The basis of this player will be the application that we wrote in the previous section (the Section called *Media stream type detection*) to identify unknown media streams.

Once the type of the media has been found, we will find elements in the registry that can decode this streamtype. For this, we will get all element factories (which we've seen before in the Section called *Creating a `GstElement`* in Chapter 5) and find the ones with the given MIME-type and capabilities on their sinkpad. Note that we will only use parsers, demuxers and decoders. We will not use factories for any other element types, or we might get into a loop of encoders and decoders. For this, we will want to build a list of "allowed" factories right after initializing `GStreamer`.

```
static GList *factories;

/*
 * This function is called by the registry loader. Its return value
 * (TRUE or FALSE) decides whether the given feature will be included
 * in the list that we're generating further down.
 */

static gboolean
cb_feature_filter (GstPluginFeature *feature,
     gpointer          data)
{
  const gchar *klass;
  guint rank;

  /* we only care about element factories */
  if (!GST_IS_ELEMENT_FACTORY (feature))
    return FALSE;

  /* only parsers, demuxers and decoders */
  klass = gst_element_factory_get_klass (GST_ELEMENT_FACTORY (feature));
  if (g_strrstr (klass, "Demux") == NULL &&
      g_strrstr (klass, "Decoder") == NULL &&
      g_strrstr (klass, "Parse") == NULL)
    return FALSE;

  /* only select elements with autoplugging rank */
  rank = gst_plugin_feature_get_rank (feature);
  if (rank < GST_RANK_MARGINAL)
    return FALSE;

  return TRUE;
}

/*
 * This function is called to sort features by rank.
 */

static gint
cb_compare_ranks (GstPluginFeature *f1,
    GstPluginFeature *f2)
{
  return gst_plugin_feature_get_rank (f2) - gst_plugin_feature_get_rank (f1);
}

static void
init_factories (void)
{
  /* first filter out the interesting element factories */
  factories = gst_registry_pool_feature_filter (
      (GstPluginFeatureFilter) cb_feature_filter, FALSE, NULL);

  /* sort them according to their ranks */
  factories = g_list_sort (factories, (GCompareFunc) cb_compare_ranks);
}
```

From this list of element factories, we will select the one that most likely will help us decoding a media stream to a given output type. For each newly created element, we will again try to autoplug new elements to its source pad(s). Also, if the element has dynamic pads (which we've seen before in the Section called *Dynamic (or sometimes) pads* in Chapter 7), we will listen for newly created source pads and handle those, too. The following code replaces the `cb_type_found` from the previous section with a function to initiate autoplugging, which will continue with the above approach.

```
static void try_to_plug (GstPad *pad, const GstCaps *caps);

static GstElement *audiosink;

static void
cb_newpad (GstElement *element,
    GstPad      *pad,
    gpointer     data)
{
  GstCaps *caps;

  caps = gst_pad_get_caps (pad);
  try_to_plug (pad, caps);
  gst_caps_free (caps);
}

static void
close_link (GstPad       *srcpad,
     GstElement   *sinkelement,
     const gchar  *padname,
     const GList  *templlist)
{
  gboolean has_dynamic_pads = FALSE;

  g_print ("Plugging pad %s:%s to newly created %s:%s\n",
    gst_object_get_name (GST_OBJECT (gst_pad_get_parent (srcpad))),
    gst_pad_get_name (srcpad),
    gst_object_get_name (GST_OBJECT (sinkelement)), padname);

  /* add the element to the pipeline and set correct state */
  gst_element_set_state (sinkelement, GST_STATE_PAUSED);
  gst_bin_add (GST_BIN (pipeline), sinkelement);
  gst_pad_link (srcpad, gst_element_get_pad (sinkelement, padname));
  gst_bin_sync_children_state (GST_BIN (pipeline));
```

```
    /* if we have static source pads, link those. If we have dynamic
     * source pads, listen for new-pad signals on the element */
    for ( ; templlist != NULL; templlist = templlist->next) {
      GstPadTemplate *templ = GST_PAD_TEMPLATE (templlist->data);

      /* only sourcepads, no request pads */
      if (templ->direction != GST_PAD_SRC ||
          templ->presence == GST_PAD_REQUEST) {
        continue;
      }

      switch (templ->presence) {
        case GST_PAD_ALWAYS: {
          GstPad *pad = gst_element_get_pad (sinkelement, templ->name_template);
          GstCaps *caps = gst_pad_get_caps (pad);

          /* link */
          try_to_plug (pad, caps);
          gst_caps_free (caps);
          break;
        }
        case GST_PAD_SOMETIMES:
          has_dynamic_pads = TRUE;
          break;
        default:
          break;
      }
    }

    /* listen for newly created pads if this element supports that */
    if (has_dynamic_pads) {
      g_signal_connect (sinkelement, "new-pad", G_CALLBACK (cb_newpad), NULL);
    }
  }

  static void
  try_to_plug (GstPad        *pad,
        const GstCaps *caps)
  {
    GstObject *parent = GST_OBJECT (gst_pad_get_parent (pad));
    const gchar *mime;
    const GList *item;
    GstCaps *res, *audiocaps;

    /* don't plug if we're already plugged */
    if (GST_PAD_IS_LINKED (gst_element_get_pad (audiosink, "sink"))) {
      g_print ("Omitting link for pad %s:%s because we're already linked\n",
        gst_object_get_name (parent), gst_pad_get_name (pad));
      return;
    }

    /* as said above, we only try to plug audio... Omit video */
    mime = gst_structure_get_name (gst_caps_get_structure (caps, 0));
    if (g_strrstr (mime, "video")) {
      g_print ("Omitting link for pad %s:%s because mimetype %s is non-audio\n",
        gst_object_get_name (parent), gst_pad_get_name (pad), mime);
      return;
    }

    /* can it link to the audiopad? */
    audiocaps = gst_pad_get_caps (gst_element_get_pad (audiosink, "sink"));
    res = gst_caps_intersect (caps, audiocaps);
    if (res && !gst_caps_is_empty (res)) {
      g_print ("Found pad to link to audiosink - plugging is now done\n");
      close_link (pad, audiosink, "sink", NULL);
      gst_caps_free (audiocaps);
    gst_caps_free (res);
    return;
  }
  gst_caps_free (audiocaps);
  gst_caps_free (res);

  /* try to plug from our list */
  for (item = factories; item != NULL; item = item->next) {
    GstElementFactory *factory = GST_ELEMENT_FACTORY (item->data);
    const GList *pads;

    for (pads = gst_element_factory_get_pad_templates (factory);
         pads != NULL; pads = pads->next) {
      GstPadTemplate *templ = GST_PAD_TEMPLATE (pads->data);

      /* find the sink template - need an always pad*/
      if (templ->direction != GST_PAD_SINK ||
          templ->presence != GST_PAD_ALWAYS) {
        continue;
      }

      /* can it link? */
      res = gst_caps_intersect (caps, templ->caps);
      if (res && !gst_caps_is_empty (res)) {
        GstElement *element;

        /* close link and return */
        gst_caps_free (res);
        element = gst_element_factory_create (factory, NULL);
        close_link (pad, element, templ->name_template,
      gst_element_factory_get_pad_templates (factory));
        return;
      }
      gst_caps_free (res);

      /* we only check one sink template per factory, so move on to the
       * next factory now */
      break;
    }
  }

  /* if we get here, no item was found */
  g_print ("No compatible pad found to decode %s on %s:%s\n",
    mime, gst_object_get_name (parent), gst_pad_get_name (pad));
}

static void
cb_typefound (GstElement *typefind,
       guint       probability,
       GstCaps    *caps,
       gpointer    data)
{
  gchar *s;

  s = gst_caps_to_string (caps);
  g_print ("Detected media type %s\n", s);
  g_free (s);

  /* actually plug now */
  try_to_plug (gst_element_get_pad (typefind, "src"), caps);
}
```

By doing all this, we will be able to make a simple autoplugger that can automatically setup a pipeline for any media type. In the example below, we will do this for audio only. However, we can also do this for video to create a player that plays both audio and video.

The example above is a good first try for an autoplugger. Next steps would be to listen for “pad-removed” signals, so we can dynamically change the plugged pipeline if the stream changes (this happens for DVB or Ogg radio). Also, you might want special-case code for input with known content (such as a DVD or an audio-CD), and much, much more. Moreover, you’ll want many checks to prevent infinite loops during autoplugging, maybe you’ll want to implement shortest-path-finding to make sure the most optimal pipeline is chosen, and so on. Basically, the features that you implement in an autoplugger depend on what you want to use it for. For full-blown implementations, see the “playbin”, “decodebin” and “spider” elements.

Notes

1. http://gstreamer.freedesktop.org/data/doc/gstreamer/head/pwg/html/section-types-definitions.html

Chapter 18. Pipeline manipulation

This chapter will discuss how you can manipulate your pipeline in several ways from your application on. Parts of this chapter are downright hackish, so be assured that you'll need some programming knowledge before you start reading this.

Topics that will be discussed here include how you can insert data into a pipeline from your application, how to read data from a pipeline, how to manipulate the pipeline's speed, length, starting point and how to listen to a pipeline's data processing.

Data probes

Probes are best envisioned as pad listeners. They are attached to a pad in a pipeline, and you can add callback functions to this probe. Those callback functions will be called whenever data is being sent over this pad. The callback can then decide whether the data should be discarded or it can replace the piece of data with another piece of data. In this callback, it can also trigger actions in the application itself. For pipeline manipulation, probes are rather limited, but for pipeline tracking, they can be very useful.

Manually adding or removing data from/to a pipeline

Many people have expressed the wish to use their own sources to inject data into a pipeline. Some people have also expressed the wish to grab the output in a pipeline and take care of the actual output inside their application. While either of these methods are stongly discouraged, `GStreamer` offers hacks to do this. *However, there is no support for those methods.* If it doesn't work, you're on your own. Also, synchronization, thread-safety and other things that you've been able to take for granted so far are no longer guanranteed if you use any of those methods. It's always better to simply write a plugin and have the pipeline schedule and manage it. See the Plugin Writer's Guide for more information on this topic. Also see the next section, which will explain how to embed plugins statically in your application.

After all those disclaimers, let's start. There's three possible elements that you can use for the above-mentioned purposes. Those are called "fakesrc" (an imaginary source), "fakesink" (an imaginary sink) and "identity" (an imaginary filter). The same method applies to each of those elements. Here, we will discuss how to use those elements to insert (using fakesrc) or grab (using fakesink or identity) data from a pipeline, and how to set negotiation.

Inserting or grabbing data

The three before-mentioned elements (fakesrc, fakesink and identity) each have a "handoff" signal that will be called in the `_get ()`- (fakesrc) or `_chain ()`-function (identity, fakesink). In the signal handler, you can set (fakesrc) or get (identity, fakesink) data to/from the provided buffer. Note that in the case of fakesrc, you have to set the size of the provided buffer using the "sizemax" property. For both fakesrc and fakesink, you also have to set the "signal-handoffs" property for this method to work.

Note that your handoff function should *not* block, since this will block pipeline iteration. Also, do not try to use all sort of weird hacks in such functions to accomplish something that looks like synchronization or so; it's not the right way and will lead to issues elsewhere. If you're doing any of this, you're basically misunderstanding the `GStreamer` design.

Forcing a format

Sometimes, when using fakesrc as a source in your pipeline, you'll want to set a specific format, for example a video size and format or an audio bitsize and number of channels. You can do this by forcing a specific `GstCaps` on the pipeline, which is possible by using *filtered caps*. You can set a filtered caps on a link by using `gst_pad_link_filtered ()`, where the third argument is the format to force on the link.

Example application

This example application will generate black/white (it switches every second) video to an X-window output by using fakesrc as a source and using filtered caps to force a format. Since the depth of the image depends on your X-server settings, we use a colorspace conversion element to make sure that the output to your X server will have the correct bitdepth. You can also set timestamps on the provided buffers to override the fixed framerate.

```
#include <string.h> /* for memset () */
#include <gst/gst.h>

static void
cb_handoff (GstElement *fakesrc,
     GstBuffer *buffer,
     GstPad *pad,
     gpointer user_data)
{
  static gboolean white = FALSE;

  /* this makes the image black/white */
  memset (GST_BUFFER_DATA (buffer), white ? 0xff : 0x0,
   GST_BUFFER_SIZE (buffer));
  white = !white;
}

gint
main (gint argc,
      gchar *argv[])
{
  GstElement *pipeline, *fakesrc, *conv, *videosink;
  GstCaps *filter;

  /* init GStreamer */
  gst_init (&argc, &argv);

  /* setup pipeline */
  pipeline = gst_pipeline_new ("pipeline");
  fakesrc = gst_element_factory_make ("fakesrc", "source");
  conv = gst_element_factory_make ("ffmpegcolorspace", "conv");
  videosink = gst_element_factory_make ("ximagesink", "videosink");

  /* setup */
  filter = gst_caps_new_simple ("video/x-raw-rgb",
    "width", G_TYPE_INT, 384,
    "height", G_TYPE_INT, 288,
    "framerate", G_TYPE_DOUBLE, (gdouble) 1.0,
    "bpp", G_TYPE_INT, 16,
    "depth", G_TYPE_INT, 16,
    "endianness", G_TYPE_INT, G_BYTE_ORDER,
    NULL);
  gst_element_link_filtered (fakesrc, conv, filter);
  gst_element_link (conv, videosink);
  gst_bin_add_many (GST_BIN (pipeline), fakesrc, conv, videosink, NULL);
  /* setup fake source */
  g_object_set (G_OBJECT (fakesrc),
  "signal-handoffs", TRUE,
  "sizemax", 384 * 288 * 2,
  "sizetype", 2, NULL);
  g_signal_connect (fakesrc, "handoff", G_CALLBACK (cb_handoff), NULL);

  /* play */
  gst_element_set_state (pipeline, GST_STATE_PLAYING);
  while (gst_bin_iterate (GST_BIN (pipeline))) ;

  /* clean up */
  gst_element_set_state (pipeline, GST_STATE_NULL);
  gst_object_unref (GST_OBJECT (pipeline));

  return 0;
}
```

Embedding static elements in your application

The Plugin Writer's Guide[1] describes in great detail how to write elements for the `GStreamer` framework. In this section, we will solely discuss how to embed such elements statically in your application. This can be useful for application-specific elements that have no use elsewhere in `GStreamer`.

Dynamically loaded plugins contain a structure that's defined using `GST_PLUGIN_DEFINE ()`. This structure is loaded when the plugin is loaded by the `GStreamer` core. The structure contains an initialization function (usually called `plugin_init`) that will be called right after that. It's purpose is to register the elements provided by the plugin with the `GStreamer` framework. If you want to embed elements directly in your application, the only thing you need to do is to manually run this structure using `_gst_plugin_register_static ()`. The initialization will then be called, and the elements will from then on be available like any other element, without them having to be dynamically loadable libraries. In the example below, you would be able to call `gst_element_factory_make ("my-element-name", "some-name")` to create an instance of the element.

```
/*
 * Here, you would write the actual plugin code.
 */

[..]

static gboolean
register_elements (GstPlugin *plugin)
{
  return gst_element_register (plugin, "my-element-name",
         GST_RANK_NONE, MY_PLUGIN_TYPE);
}

static GstPluginDesc plugin_desc = {
  GST_VERSION_MAJOR,
  GST_VERSION_MINOR,
  "my-private-plugins",
  "Private elements of my application",
  register_elements,
  NULL,
  "0.0.1",
  "LGPL",
  "my-application",
  "http://www.my-application.net/",
  GST_PADDING_INIT
};

/*
 * Call this function right after calling gst_init ().
 */

void
my_elements_init (void)
{
  _gst_plugin_register_static (&plugin_desc);
}
```

Notes

1. http://gstreamer.freedesktop.org/data/doc/gstreamer/head/pwg/html/index.html

Chapter 19. Components

GStreamer includes several higher-level components to simplify your applications life. All of the components discussed here (for now) are targetted at media playback. The idea of each of these components is to integrate as closely as possible with a GStreamer pipeline, but to hide the complexity of media type detection and several other rather complex topics that have been discussed in Part III in *GStreamer Application Development Manual (0.8.9.2)*.

We currently recommend people to use either playbin (see the Section called *Playbin*) or decodebin (see the Section called *Decodebin*), depending on their needs. The other components discussed here are either outdated or deprecated. The documentation is provided for legacy purposes. Use of those other components is not recommended.

Playbin

Playbin is an element that can be created using the standard GStreamer API (e.g. `gst_element_factory_make ()`). The factory is conveniently called "playbin". By being a `GstElement`, playbin automatically supports all of the features of this class, including error handling, tag support, state handling, getting stream positions, seeking, and so on.

Setting up a playbin pipeline is as simple as creating an instance of the playbin element, setting a file location (this has to be a valid URI, so "<protocol>://<location>", e.g. file:///tmp/my.ogg or http://www.example.org/stream.ogg) using the "uri" property on playbin, and then setting the element to the `GST_STATE_PLAYING` state. Internally, playbin uses threads, so there's no need to iterate the element or anything. However, one thing to keep in mind is that signals fired by playbin might come from another than the main thread, so be sure to keep this in mind in your signal handles. Most application programmers will want to use a function such as `g_idle_add ()` to make sure that the signal is handled in the main thread.

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

static void
cb_eos (GstElement *play,
 gpointer data)
{
  gst_main_quit ();
}

static void
cb_error (GstElement *play,
   GstElement *src,
   GError *err,
   gchar *debug,
   gpointer data)
{
  g_print ("Error: %s\n", err->message);
}

gint
main (gint argc,
      gchar *argv[])
{
  GstElement *play;

  /* init GStreamer */
  gst_init (&argc, &argv);

  /* make sure we have a URI */
  if (argc != 2) {
    g_print ("Usage: %s <URI>\n", argv[0]);
    return -1;
  }

  /* set up */
  play = gst_element_factory_make ("playbin", "play");
  g_object_set (G_OBJECT (play), "uri", argv[1], NULL);
  g_signal_connect (play, "eos", G_CALLBACK (cb_eos), NULL);
  g_signal_connect (play, "error", G_CALLBACK (cb_error), NULL);
  if (gst_element_set_state (play, GST_STATE_PLAYING) != GST_STATE_SUCCESS) {
    g_print ("Failed to play\n");
    return -1;
  }

  /* now run */
  gst_main ();

  /* also clean up */
  gst_element_set_state (play, GST_STATE_NULL);
  gst_object_unref (GST_OBJECT (play));

  return 0;
}
```

Playbin has several features that have been discussed previously:

- Settable video and audio output (using the "video-sink" and "audio-sink" properties).
- Mostly controllable and trackable as a `GstElement`, including error handling, eos handling, tag handling, state handling, media position handling and seeking.
- Buffers network-sources.
- Supports visualizations for audio-only media.
- Supports subtitles, both in the media as well as from separate files.
- Supports stream selection and disabling. If your media has multiple audio or subtitle tracks, you can dynamically choose which one to play back, or decide to turn it off alltogther (which is especially useful to turn off subtitles).

Decodebin

Decodebin is the actual autoplugger backend of playbin, which was discussed in the previous section. Decodebin will, in short, accept input from a source that is linked to its sinkpad and will try to detect the media type contained in the stream, and set up decoder routines for each of those. It will automatically select decoders. For each decoded stream, it will emit the "new-decoded-pad" signal, to let the client know about the newly found decoded stream. For unknown streams (which might be the whole stream), it will emit the "unknown-type" signal. The application is then responsible for reporting the error to the user.

The example code below will play back an audio stream of an input file. For readability, it does not include any error handling of any sort.

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

GstElement *pipeline, *audio;
GstPad *audiopad;

static void
cb_newpad (GstElement *decodebin,
    GstPad      *pad,
    gboolean     last,
    gpointer     data)
{
  GstCaps *caps;
  GstStructure *str;

  /* only link audio; only link once */
  if (GST_PAD_IS_LINKED (audiopad))
    return;
  caps = gst_pad_get_caps (pad);
  str = gst_caps_get_structure (caps, 0);
  if (!g_strrstr (gst_structure_get_name (str), "audio"))
    return;

  /* link'n'play */
  gst_pad_link (pad, audiopad);
  gst_bin_add (GST_BIN (pipeline), audio);
  gst_bin_sync_children_state (GST_BIN (pipeline));
}

gint
main (gint   argc,
      gchar *argv[])
{
  GstElement *src, *dec, *conv, *scale, *sink;

  /* init GStreamer */
  gst_init (&argc, &argv);

  /* make sure we have input */
  if (argc != 2) {
    g_print ("Usage: %s <filename>\n", argv[0]);
    return -1;
  }

  /* setup */
  pipeline = gst_pipeline_new ("pipeline");
  src = gst_element_factory_make ("filesrc", "source");
  g_object_set (G_OBJECT (src), "location", argv[1], NULL);
  dec = gst_element_factory_make ("decodebin", "decoder");
  g_signal_connect (dec, "new-decoded-pad", G_CALLBACK (cb_newpad), NULL);
  audio = gst_bin_new ("audiobin");
  conv = gst_element_factory_make ("audioconvert", "aconv");
  audiopad = gst_element_get_pad (conv, "sink");
  scale = gst_element_factory_make ("audioscale", "scale");
  sink = gst_element_factory_make ("alsasink", "sink");
  gst_bin_add_many (GST_BIN (audio), conv, scale, sink, NULL);
  gst_element_link_many (conv, scale, sink, NULL);
  gst_bin_add_many (GST_BIN (pipeline), src, dec, NULL);
  gst_element_link (src, dec);

  /* run */
  gst_element_set_state (audio, GST_STATE_PAUSED);
  gst_element_set_state (pipeline, GST_STATE_PLAYING);
  while (gst_bin_iterate (GST_BIN (pipeline))) ;

  /* cleanup */
  gst_element_set_state (pipeline, GST_STATE_NULL);
  gst_object_unref (GST_OBJECT (pipeline));

  return 0;
}
```

Decodebin, similar to playbin, supports the following features:

- Can decode an unlimited number of contained streams to decoded output pads.
- Is handled as a `GstElement` in all ways, including tag or error forwarding and state handling.

Although decodebin is a good autoplugger, there's a whole lot of things that it does not do and is not intended to do:

- Taking care of input streams with a known media type (e.g. a DVD, an audio-CD or such).
- Selection of streams (e.g. which audio track to play in case of multi-language media streams).
- Overlaying subtitles over a decoded video stream.

Decodebin can be easily tested on the commandline, e.g. by using the command **gst-launch-0.8 filesrc location=file.ogg ! decodebin ! audioconvert ! audioscale ! alsasink**.

Spider

Spider is an autoplugger that looks and feels very much like decodebin. On the commandline, you can literally switch between spider and decodebin and it'll mostly just work. Try, for example, **gst-launch-0.8 filesrc location=file.ogg ! spider ! audioconvert ! audioscale ! alsasink**. Although the two may seem very much alike from the outside, they are very different from the inside. Those internal differences are the main reason why spider is currently considered deprecated (along with the fact that it was hard to maintain).

As opposed to decodebin, spider does not decode pads and emit signals for each detected stream. Instead, you have to add output sinks to spider by create source request pads and connecting those to sink elements. This means that streams decoded by spider cannot be dynamic. Also, spider uses many loop-based elements internally, which is rather heavy scheduler-wise.

Code for using spider would look almost identical to the code of decodebin, and is therefore omitted. Also, featureset and limitations are very much alike, except for the above-mentioned extra limitations for spider with respect to decodebin.

GstPlay

GstPlay is a GtkWidget with a simple API to play, pause and stop a media file.

GstEditor

GstEditor is a set of widgets to display a graphical representation of a pipeline.

Chapter 20. XML in GStreamer

GStreamer uses XML to store and load its pipeline definitions. XML is also used internally to manage the plugin registry. The plugin registry is a file that contains the definition of all the plugins GStreamer knows about to have quick access to the specifics of the plugins.

We will show you how you can save a pipeline to XML and how you can reload that XML file again for later use.

Turning GstElements into XML

We create a simple pipeline and write it to stdout with gst_xml_write_file (). The following code constructs an MP3 player pipeline with two threads and then writes out the XML both to stdout and to a file. Use this program with one argument: the MP3 file on disk.

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

gboolean playing;

int
main (int argc, char *argv[])
{
  GstElement *filesrc, *osssink, *queue, *queue2, *decode;
  GstElement *bin;
  GstElement *thread, *thread2;

  gst_init (&argc,&argv);

  if (argc != 2) {
    g_print ("usage: %s <mp3 filename>\n", argv[0]);
    exit (-1);
  }

  /* create a new thread to hold the elements */
  thread = gst_element_factory_make ("thread", "thread");
  g_assert (thread != NULL);
  thread2 = gst_element_factory_make ("thread", "thread2");
  g_assert (thread2 != NULL);

  /* create a new bin to hold the elements */
  bin = gst_bin_new ("bin");
  g_assert (bin != NULL);

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

  queue = gst_element_factory_make ("queue", "queue");
  queue2 = gst_element_factory_make ("queue", "queue2");

  /* and an audio sink */
  osssink = gst_element_factory_make ("osssink", "play_audio");
  g_assert (osssink != NULL);

  decode = gst_element_factory_make ("mad", "decode");
  g_assert (decode != NULL);

  /* add objects to the main bin */
  gst_bin_add_many (GST_BIN (bin), filesrc, queue, NULL);
  gst_bin_add_many (GST_BIN (thread), decode, queue2, NULL);

  gst_bin_add (GST_BIN (thread2), osssink);

  gst_element_link_many (filesrc, queue, decode, queue2, osssink, NULL);

  gst_bin_add_many (GST_BIN (bin), thread, thread2, NULL);

  /* write the bin to stdout */
  gst_xml_write_file (GST_ELEMENT (bin), stdout);

  /* write the bin to a file */
  gst_xml_write_file (GST_ELEMENT (bin), fopen ("xmlTest.gst", "w"));

  exit (0);
}
```

The most important line is:

```
  gst_xml_write_file (GST_ELEMENT (bin), stdout);
```

gst_xml_write_file () will turn the given element into an xmlDocPtr that is then formatted and saved to a file. To save to disk, pass the result of a fopen(2) as the second argument.

The complete element hierarchy will be saved along with the inter element pad links and the element parameters. Future GStreamer versions will also allow you to store the signals in the XML file.

Loading a GstElement from an XML file

Before an XML file can be loaded, you must create a GstXML object. A saved XML file can then be loaded with the gst_xml_parse_file (xml, filename, rootelement) method. The root element can optionally left NULL. The following code example loads the previously created XML file and runs it.

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

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

  gst_init (&argc, &argv);

  xml = gst_xml_new ();

  ret = gst_xml_parse_file(xml, "xmlTest.gst", NULL);
  g_assert (ret == TRUE);

  bin = gst_xml_get_element (xml, "bin");
  g_assert (bin != NULL);

  gst_element_set_state (bin, GST_STATE_PLAYING);

  while (gst_bin_iterate(GST_BIN(bin)));
  gst_element_set_state (bin, GST_STATE_NULL);

  exit (0);
}
```

gst_xml_get_element (xml, "name") can be used to get a specific element from the XML file.

gst_xml_get_topelements (xml) can be used to get a list of all toplevel elements in the XML file.

In addition to loading a file, you can also load a from a xmlDocPtr and an in memory buffer using gst_xml_parse_doc and gst_xml_parse_memory respectively. Both of these methods return a gboolean indicating success or failure of the requested action.

Adding custom XML tags into the core XML data

It is possible to add custom XML tags to the core XML created with gst_xml_write. This feature can be used by an application to add more information to the save plugins. The editor will for example insert the position of the elements on the screen using the custom XML tags.

It is strongly suggested to save and load the custom XML tags using a namespace. This will solve the problem of having your XML tags interfere with the core XML tags.

To insert a hook into the element saving procedure you can link a signal to the GstElement using the following piece of code:

```
xmlNsPtr ns;

  ...
  ns = xmlNewNs (NULL, "http://gstreamer.net/gst-test/1.0/", "test");
    ...
  thread = gst_element_factory_make ("thread", "thread");
  g_signal_connect (G_OBJECT (thread), "object_saved",
         G_CALLBACK (object_saved), g_strdup ("decoder thread"));
    ...
```

When the thread is saved, the object_save method will be called. Our example will insert a comment tag:

```
static void
object_saved (GstObject *object, xmlNodePtr parent, gpointer data)
{
  xmlNodePtr child;

  child = xmlNewChild (parent, ns, "comment", NULL);
  xmlNewChild (child, ns, "text", (gchar *)data);
}
```

Adding the custom tag code to the above example you will get an XML file with the custom tags in it. Here's an excerpt:

```
          ...
        <gst:element>
          <gst:name>thread</gst:name>
          <gst:type>thread</gst:type>
          <gst:version>0.1.0</gst:version>
    ...
```

```
          </gst:children>
          <test:comment>
            <test:text>decoder thread</test:text>
          </test:comment>
        </gst:element>
            ...
```

To retrieve the custom XML again, you need to attach a signal to the GstXML object used to load the XML data. You can then parse your custom XML from the XML tree whenever an object is loaded.

We can extend our previous example with the following piece of code.

```
  xml = gst_xml_new ();

  g_signal_connect (G_OBJECT (xml), "object_loaded",
           G_CALLBACK (xml_loaded), xml);

  ret = gst_xml_parse_file (xml, "xmlTest.gst", NULL);
  g_assert (ret == TRUE);
```

Whenever a new object has been loaded, the xml_loaded function will be called. This function looks like:

```
static void
xml_loaded (GstXML *xml, GstObject *object, xmlNodePtr self, gpointer data)
{
  xmlNodePtr children = self->xmlChildrenNode;

  while (children) {
    if (!strcmp (children->name, "comment")) {
      xmlNodePtr nodes = children->xmlChildrenNode;

      while (nodes) {
        if (!strcmp (nodes->name, "text")) {
          gchar *name = g_strdup (xmlNodeGetContent (nodes));
          g_print ("object %s loaded with comment '%s'\n",
                   gst_object_get_name (object), name);
        }
        nodes = nodes->next;
      }
    }
    children = children->next;
  }
}
```

As you can see, you'll get a handle to the GstXML object, the newly loaded GstObject and the xmlNodePtr that was used to create this object. In the above example we look for our special tag inside the XML tree that was used to load the object and we print our comment to the console.

Chapter 21. Things to check when writing an application

This chapter contains a fairly random selection of things that can be useful to keep in mind when writing `GStreamer`-based applications. It's up to you how much you're going to use the information provided here. We will shortly discuss how to debug pipeline problems using `GStreamer` applications. Also, we will touch upon how to acquire knowledge about plugins and elements and how to test simple pipelines before building applications around them.

Good programming habits

- Always connect to the "error" signal of your topmost pipeline to be notified of errors in your pipeline.
- Always check return values of `GStreamer` functions. Especially, check return values of `gst_element_link ()` and `gst_element_set_state ()`.
- Always use your pipeline object to keep track of the current state of your pipeline. Don't keep private variables in your application. Also, don't update your user interface if a user presses the "play" button. Instead, connect to the "state-changed" signal of your topmost pipeline and update the user interface whenever this signal is triggered.

Debugging

Applications can make use of the extensive `GStreamer` debugging system to debug pipeline problems. Elements will write output to this system to log what they're doing. It's not used for error reporting, but it is very useful for tracking what an element is doing exactly, which can come in handy when debugging application issues (such as failing seeks, out-of-sync media, etc.).

Most `GStreamer`-based applications accept the commandline option `--gst-debug=LIST` and related family members. The list consists of a comma-separated list of category/level pairs, which can set the debugging level for a specific debugging category. For example, `--gst-debug=oggdemux:5` would turn on debugging for the Ogg demuxer element. You can use wildcards as well. A debugging level of 0 will turn off all debugging, and a level of 5 will turn on all debugging. Intermediate values only turn on some debugging (based on message severity; 2, for example, will only display errors and warnings). Here's a list of all available options:

- `--gst-debug-help` will print available debug categories and exit.
- `--gst-debug-level=`*`LEVEL`* will set the default debug level (which can range from 0 (no output) to 5 (everything)).
- `--gst-debug=`*`LIST`* takes a comma-separated list of category_name:level pairs to set specific levels for the individual categories. Example: `GST_AUTOPLUG:5,avidemux:3`.
- `--gst-debug-no-color` will disable color debugging.
- `--gst-debug-disable` disables debugging alltogether.
- `--gst-plugin-spew` enables printout of errors while loading `GStreamer` plugins.

Conversion plugins

`GStreamer` contains a bunch of conversion plugins that most applications will find useful. Specifically, those are videoscalers (videoscale), colorspace convertors (ffmpegcolorspace), audio format convertors and channel resamplers (audioconvert) and audio samplerate convertors (audioscale). Those convertors don't do anything when not required, they will act in passthrough mode. They will activate when the hardware doesn't support a specific request, though. All applications are recommended to use those elements.

Utility applications provided with `GStreamer`

`GStreamer` comes with a default set of command-line utilities that can help in application development. We will discuss only **gst-launch** and **gst-inspect** here.

gst-launch

gst-launch is a simple script-like commandline application that can be used to test pipelines. For example, the command **gst-launch sinesrc ! alsasink** will run a pipeline which generates a sine-wave audio stream and plays it to your ALSA audio card. **gst-launch** also allows the use of threads (using curly brackets, so "{" and "}") and bins (using brackets, so "(" and ")"). You can use dots to imply padnames on elements, or even omit the padname to automatically select a pad. Using all this, the pipeline **gst-launch filesrc location=file.ogg ! oggdemux name=d { d. ! theoradec ! ffmpegcolorspace ! xvimagesink } { d. ! vorbisdec ! alsasink }** will play an Ogg file containing a Theora video-stream and a Vorbis audio-stream. You can also use autopluggers such as decodebin on the commandline. See the manual page of **gst-launch** for more information.

gst-inspect

gst-inspect can be used to inspect all properties, signals, dynamic parameters and the object hierarchy of an element. This acn be very useful to see which `GObject` properties or which signals (and using what arguments) an element supports. Run **gst-inspect fakesrc** to get an idea of what it does. See the manual page of **gst-inspect** for more information.

Chapter 22. Integration

GStreamer tries to integrate closely with operating systems (such as Linux and UNIX-like operating systems, OS X or Windows) and desktop environments (such as GNOME or KDE). In this chapter, we'll mention some specific techniques to integrate your application with your operating system or desktop environment of choice.

Linux and UNIX-like operating systems

GStreamer provides a basic set of elements that are useful when integrating with Linux or a UNIX-like operating system.

- For audio input and output, GStreamer provides input and output elements for several audio subsystems. Amongst others, GStreamer includes elements for ALSA (alsasrc, alsamixer, alsasink), OSS (osssrc, ossmixer, osssink) and Sun audio (sunaudiosrc, sunaudiomixer, sunaudiosink).
- For video input, GStreamer contains source elements for Video4linux (v4lsrc, v4lmjpegsrc, v4lelement and v4lmjpegisnk) and Video4linux2 (v4l2src, v4l2element).
- For video output, GStreamer provides elements for output to X-windows (ximagesink), Xv-windows (xvimagesink; for hardware-accelerated video), direct-framebuffer (dfbimagesink) and openGL image contexts (glsink).

GNOME desktop

GStreamer has been the media backend of the GNOME[1] desktop since GNOME-2.2 onwards. Nowadays, a whole bunch of GNOME applications make use of GStreamer for media-processing, including (but not limited to) Rhythmbox[2], Totem[3] and Sound Juicer[4].

Most of these GNOME applications make use of some specific techniques to integrate as closely as possible with the GNOME desktop:

- GNOME applications call `gnome_program_init ()` to parse command-line options and initialize the necessary gnome modules. GStreamer applications would normally call `gst_init ()` to do the same for GStreamer. This would mean that only one of the two can parse command-line options. To work around this issue, GStreamer can provide a `poptOption` array which can be passed to `gnome_program_init ()`.

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

gint
main (gint   argc,
      gchar *argv[])
{
  struct poptOption options[] = {
    {NULL, '\0', POPT_ARG_INCLUDE_TABLE, NULL, 0, "GStreamer", NULL},
    POPT_TABLEEND
  };

  /* init GStreamer and GNOME using the GStreamer popt tables */
  options[0].arg = (void *) gst_init_get_popt_table ();
  gnome_program_init ("my-application", "0.0.1", LIBGNOMEUI_MODULE, argc, argv,
          GNOME_PARAM_POPT_TABLE, options,
          NULL);
[..]
}
```

- GNOME stores the default video and audio sources and sinks in GConf. `GStreamer` provides a small utility library that can be used to get the elements from the registry using functions such as `gst_gconf_get_default_video_sink ()`. See the header file (`gst/gconf/gconf.h`) for details. All GNOME applications are recommended to use those variables.
- `GStreamer` provides data input/output elements for use with the GNOME-VFS system. These elements are called "gnomevfssrc" and "gnomevfssink".

KDE desktop

`GStreamer` has been proposed for inclusion in KDE-4.0. Currently, `GStreamer` is included as an optional component, and it's used by several KDE applications, including AmaroK[5] and JuK[6]. A backend for KMPlayer[7] is currently under development.

Although not yet as complete as the GNOME integration bits, there are already some KDE integration specifics available. This list will probably grow as `GStreamer` starts to be used in KDE-4.0:

- AmaroK contains a kiosrc element, which is a source element that integrates with the KDE VFS subsystem KIO.

OS X

`GStreamer` provides native video and audio output elements for OS X. It builds using the standard development tools for OS X.

Windows

`GStreamer` builds using Microsoft Visual C .NET 2003 and using Cygwin.

Notes

1. http://www.gnome.org/
2. http://www.rhythmbox.org/
3. http://www.hadess.net/totem.php3
4. http://www.burtonini.com/blog/computers/sound-juicer
5. http://amarok.kde.org/
6. http://developer.kde.org/~wheeler/juk.html
7. http://www.xs4all.nl/~jjvrieze/kmplayer.html

Chapter 23. Licensing advisory

How to license the applications you build with `GStreamer`

The licensing of GStreamer is no different from a lot of other libraries out there like GTK+ or glibc: we use the LGPL. What complicates things with regards to GStreamer is its plugin-based design and the heavily patented and proprietary nature of many multimedia codecs. While patents on software are currently only allowed in a small minority of world countries (the US and Australia being the most important of those), the problem is that due to the central place the US hold in the world economy and the computing industry, software patents are hard to ignore wherever you are. Due to this situation, many companies, including major GNU/Linux distributions, get trapped in a situation where they either get bad reviews due to lacking out-of-the-box media playback capabilities (and attempts to educate the reviewers have met with little success so far), or go against their own - and the free software movement's - wish to avoid proprietary software. Due to competitive pressure, most choose to add some support. Doing that through pure free software solutions would have them risk heavy litigation and punishment from patent owners. So when the decision is made to include support for patented codecs, it leaves them the choice of either using special proprietary applications, or try to integrate the support for these codecs through proprietary plugins into the multimedia infrastructure provided by GStreamer. Faced with one of these two evils the GStreamer community of course prefer the second option.

The problem which arises is that most free software and open source applications developed use the GPL as their license. While this is generally a good thing, it creates a dilemma for people who want to put together a distribution. The dilemma they face is that if they include proprietary plugins in GStreamer to support patented formats in a way that is legal for them, they do risk running afoul of the GPL license of the applications. We have gotten some conflicting reports from lawyers on whether this is actually a problem, but the official stance of the FSF is that it is a problem. We view the FSF as an authority on this matter, so we are inclined to follow their interpretation of the GPL license.

So what does this mean for you as an application developer? Well, it means you have to make an active decision on whether you want your application to be used together with proprietary plugins or not. What you decide here will also influence the chances of commercial distributions and Unix vendors shipping your application. The GStreamer community suggest you license your software using a license that will allow proprietary plugins to be bundled with GStreamer and your applications, in order to make sure that as many vendors as possible go with GStreamer instead of less free solutions. This in turn we hope and think will let GStreamer be a vehicle for wider use of free formats like the Xiph.org formats.

If you do decide that you want to allow for non-free plugins to be used with your application you have a variety of choices. One of the simplest is using licenses like LGPL, MPL or BSD for your application instead of the GPL. Or you can add a exceptions clause to your GPL license stating that you except GStreamer plugins from the obligations of the GPL.

A good example of such a GPL exception clause would be, using the Muine music player project as an example: The Muine project hereby grants permission for non-GPL-compatible GStreamer plugins to be used and distributed together with GStreamer and Muine. This permission goes above and beyond the permissions granted by the GPL license Muine is covered by.

Our suggestion among these choices is to use the LGPL license, as it is what resembles the GPL most and it makes it a good licensing fit with the major GNU/Linux desktop projects like GNOME and KDE. It also allows you to share code more openly with projects that have compatible licenses. Obviously, pure GPL code without the above-mentioned clause is not usable in your application as such. By choosing the LGPL, there is no need for an exception clause and thus code can be shared more freely.

I have above outlined the practical reasons for why the GStreamer community suggest you allow non-free plugins to be used with your applications. We feel that in the multimedia arena, the free software community is still not strong enough to set the agenda and that blocking non-free plugins to be used in our infrastructure hurts us more than it hurts the patent owners and their ilk.

This view is not shared by everyone. The Free Software Foundation urges you to use an unmodified GPL for your applications, so as to push back against the temptation to use non-free plug-ins. They say that since not everyone else has the strength to reject them because they are unethical, they ask your help to give them a legal reason to do so.

This advisory is part of a bigger advisory with a FAQ which you can find on the GStreamer website[1]

Notes

1. http://gstreamer.freedesktop.org/documentation/licensing.html

Chapter 24. Windows support

Building `GStreamer` under Win32

There are different makefiles that can be used to build GStreamer with the usual Microsoft compiling tools.

The Makefile is meant to be used with the GNU make program and the free version of the Microsoft compiler (http://msdn.microsoft.com/visualc/vctoolkit2003/). You also have to modify your system environment variables to use it from the command-line. You will also need a working Platform SDK for Windows that is available for free from Microsoft.

The projects/makefiles will generate automatically some source files needed to compile GStreamer. That requires that you have installed on your system some GNU tools and that they are available in your system PATH.

The GStreamer project depends on other libraries, namely :

- GLib
- popt
- libxml2
- libintl
- libiconv

There is now an existing package that has all these dependencies built with MSVC7.1. It exists either as precompiled librairies and headers in both Release and Debug mode, or as the source package to build it yourself. You can find it on http://mukoli.free.fr/gstreamer/deps/.

Notes: GNU tools needed that you can find on http://gnuwin32.sourceforge.net/

- GNU flex (tested with 2.5.4)
- GNU bison (tested with 1.35)

and http://www.mingw.org/

- GNU make (tested with 3.80)

the generated files from the -auto makefiles will be available soon separately on the net for convenience (people who don't want to install GNU tools).

Installation on the system

By default, GSTreamer needs a registry. You have to generate it using "gst-register.exe". It will create the file in c:\gstreamer\registry.xml that will hold all the plugins you can use.

You should install the GSTreamer core in c:\gstreamer\bin and the plugins in c:\gstreamer\plugins. Both directories should be added to your system PATH. The library dependencies should be installed in c:\usr

For example, my current setup is :

- `c:\gstreamer\registry.xml`
- `c:\gstreamer\bin\gst-inspect.exe`

- `c:\gstreamer\bin\gst-launch.exe`
- `c:\gstreamer\bin\gst-register.exe`
- `c:\gstreamer\bin\gstbytestream.dll`
- `c:\gstreamer\bin\gstelements.dll`
- `c:\gstreamer\bin\gstoptimalscheduler          .dll`
- `c:\gstreamer\bin\gstspider.dll`
- `c:\gstreamer\bin\libgtreamer-0.8.dll`
- `c:\gstreamer\plugins\gst-libs.dll`
- `c:\gstreamer\plugins\gstmatroska.dll`
- `c:\usr\bin\iconv.dll`
- `c:\usr\bin\intl.dll`
- `c:\usr\bin\libglib-2.0-0.dll`
- `c:\usr\bin\libgmodule-2.0-0.dll`
- `c:\usr\bin\libgobject-2.0-0.dll`
- `c:\usr\bin\libgthread-2.0-0.dll`
- `c:\usr\bin\libxml2.dll`
- `c:\usr\bin\popt.dll`

Notes

1. http://msdn.microsoft.com/visualc/vctoolkit2003/
2. http://mukoli.free.fr/gstreamer/deps/
3. http://gnuwin32.sourceforge.net/
4. http://www.mingw.org/

Chapter 25. Quotes from the Developers

As well as being a cool piece of software, `GStreamer` is a lively project, with developers from around the globe very actively contributing. We often hang out on the #gstreamer IRC channel on irc.freenode.net: the following are a selection of amusing[1] quotes from our conversations.

14 Oct 2004

* *zaheerm* wonders how he can break gstreamer today :)

ensonic: zaheerm, spider is always a good starting point

14 Jun 2004

teuf: ok, things work much better when I don't write incredibly stupid and buggy code

thaytan: I find that too

23 Nov 2003

Uraeus: ah yes, the sleeping part, my mind is not multitasking so I was still thinking about exercise

dolphy: Uraeus: your mind is multitasking

dolphy: Uraeus: you just miss low latency patches

14 Sep 2002

--- *wingo-party* is now known as *wingo*

* *wingo* holds head

16 Feb 2001

wtay: I shipped a few commerical products to >40000 people now but GStreamer is way more exciting...

16 Feb 2001

* *tool-man* is a gstreamer groupie

14 Jan 2001

Omega: did you run ldconfig? maybe it talks to init?

wtay: not sure, don't think so... I did run gstreamer-register though :-)

Omega: ah, that did it then ;-)

wtay: right

Omega: probably not, but in case GStreamer starts turning into an OS, someone please let me know?

9 Jan 2001

wtay: me tar, you rpm?

wtay: hehe, forgot "zan"

Omega: ?

wtay: me tar"zan", you ...

7 Jan 2001

Omega: that means probably building an agreggating, cache-massaging queue to shove N buffers across all at once, forcing cache transfer.

wtay: never done that before...

Omega: nope, but it's easy to do in gstreamer <g>

wtay: sure, I need to rewrite cp with gstreamer too, someday :-)

7 Jan 2001

wtay: GStreamer; always at least one developer is awake...

5/6 Jan 2001

wtay: we need to cut down the time to create an mp3 player down to seconds...

richardb: :)

Omega: I'm wanting to something more interesting soon, I did the "draw an mp3 player in 15sec" back in October '99.

wtay: by the time Omega gets his hands on the editor, you'll see a complete audio mixer in the editor :-)

richardb: Well, it clearly has the potential...

Omega: Working on it... ;-)

28 Dec 2000

MPAA: We will sue you now, you have violated our IP rights!

wtay: hehehe

MPAA: How dare you laugh at us? We have lawyers! We have Congressmen! We have *LARS*!

wtay: I'm so sorry your honor

MPAA: Hrumph.

* *wtay* bows before thy

4 Jun 2001

taaz: you witchdoctors and your voodoo mpeg2 black magic...

omega_: um. I count three, no four different cults there <g>

ajmitch: hehe

omega_: witchdoctors, voodoo, black magic,

omega_: and mpeg

Notes

1. No guarantee of sense of humour compatibility is given.