GStreamer Application Development Manual

Wim Taymans

`GStreamer` Application Development Manual

by Wim Taymans

Table of Contents

List of Figures

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 streaming multimedia application. The framework includes several components to build a full featured media player capable of playing MPEG1, MPEG2, AVI, MP3, WAV, AU, ...

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

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

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

Chapter 2. Motivation

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

Current problems

We descibe the typical problems in todays 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

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

If I wanted to convert an MPEG2 video stream into an AVI file, my best option would be to take all of the MPEG2 decoding algorithms out of the player and duplicate them into my own AVI encoder. These algorithms cannot easily be shared accross applications.

Non unified plugin mechanisms

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

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

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

Provision for network transparency

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

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

Catch up with the Windows(tm) world

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

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

Chapter 3. Goals

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

The design goals

We descibe what we try to achieve with GStreamer.

Clean and powerfull

GStreamer wants to provide a clean interface to:

- The application programmer who wants to build a media pipeline. The programmer can use an extensive set of powerfull tools to create media pipelines without writing a single line of code. Performing complex media manipulations becomes very easy.
- The plugin programmer. Plugin programmers are provided a clean and simple API to create self contained plugins. An extensive debugging and tracing mechanism has been integrated. GStreamer also comes with an extensive set of real-life plugins that serve as an example too.

Object oriented

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

GStreamer uses the mechanism of signals and object arguments.

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

Extensible

All GStreamer Objects can be extended using the GTK+ 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 GtkObject arguments, there is no need to have any header files installed for the plugins.

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

High performance

High performance is obtained by:

- Using glib g_mem_chunk where possible to minimize dynamic memory allocation.
- Connections between plugins are extremely light-weight. Data can travel the pipeline with minimal overhead.
- Provide a mechanism to directly work on the target memory. A plugin can for example directly write to the X servers shared mem. Buffers can also point to arbitrary memory like kernel memory.
- Refcounting and copy on write to minimize the amount of memcpy. Subbufers to efficiently split the data in a buffer.
- Pipelines can be constructed using cothreads to minimize the threading overhead. Cothreads are a simple user-space method for switching between subtasks.
- HW acceleration is possible by writing a specialized plugin.
- Uses a plugin registry with the specifications of the plugins so that the plugin loading can be delayed until the plugin is actually used.

Chapter 4. GstElement

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

What is a GstElement

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

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

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

GStreamer source elements

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

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

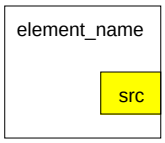

Figure 4-1. Visualisation of a source element

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

GStreamer filter elements

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

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

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

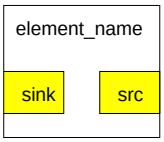

Figure 4-2. Visualisation of a filter element

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

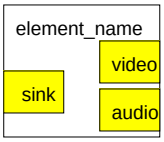

Figure 4-3. Visualisation of a filter element with more than one output pad

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

GStreamer sink elements

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

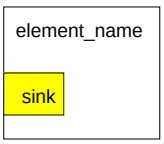

Figure 4-4. Visualisation of a sink element

Creating a GstElement

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

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

```
GstElementFactory *factory;

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

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

```
GstElement *element;

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

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

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

```
GstElement *element;

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

An element can be destroyed with:

```
GstElement *element;

 ...
gst_element_destroy (element);
```

Chapter 5. What are Plugins

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

- one or more elementfactories
- one or more typedefinitions
- one or more autopluggers

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

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

The basic plugin structure has the following fields:

```
struct _GstPlugin {
  gchar *name;                  /* name of the plugin */
  gchar *longname;              /* long name of plugin */
  gchar *filename;              /* filename it came from */

  GList *types;                 /* list of types provided */
  gint numtypes;
  GList *elements;              /* list of elements provided */
  gint numelements;
  GList *autopluggers;          /* list of autopluggers provided */
  gint numautopluggers;

  gboolean loaded;              /* if the plugin is in memory */
};
```

You can query a GList of available plugins with:

```
    GList *plugins;

    plugins = gst_plugin_get_list ();

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

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

      plugins = g_list_next (plugins);
    }
```

Chapter 6. GstPad

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

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

Getting pads from an element

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

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

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

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

```
GList *pads;
   ...
pads = gst_element_get_pad_list (element);
while (pads) {
  GstPad *pad = GST_PAD (pads->data);

  g_print ("pad name %s\n", gst_pad_get_name (pad));

  pads = g_list_next (pads);
}
   ...
```

Useful pad functions

You can get the name of a pad with gst_pad_get_name () and set its name with get_pad_set_name();

gst_pad_get_direction (GstPad *pad) can be used to query if the pad is a sink or a src pad. Remember a src pad is a pad that can output data and a sink pad is one that accepts data.

You can get the parent of the pad, this is the element that this pad belongs to, with get_pad_set_parent(GstPad *pad). This function will return a pointer to a GstObject.

Dynamic pads

Some elements might not have their pads when they are created. This can, for example, happen with an MPEG2 system demuxer. The demuxer will create its pads at runtime when it detects the different elementary streams in the MPEG2 system stream.

Running `gstreamer-inspect mpeg2parse` will show that the element has only one pad: a sink pad called 'sink'. The other pads are "dormant" as you can see in the padtemplates from the 'Exists: Sometimes' property. Depending on the type of MPEG2 file you play, the pads are created. We will see that this is very important when you are going to create dynamic pipelines later on in this manual.

You can attach a signal to an element to inform you when the element has created a new pad from one of its padtemplates. The following piece of code is an example of how to do this:

```
static void
pad_connect_func (GstElement *parser, GstPad *pad, GstElement *pipeline)
{
  g_print("***** a new pad %s was created\n", gst_pad_get_name(pad));

  gst_element_set_state (pipeline, GST_STATE_PAUSED);

  if (strncmp (gst_pad_get_name (pad), "private_stream_1.0", 18) == 0) {
    // set up an AC3 decoder pipeline
    ...
    // connect pad to the AC3 decoder pipeline
    ...
  }
  gst_element_set_state (GST_ELEMENT (audio_thread), GST_STATE_READY);
}

int
main(int argc, char *argv[])
{
  GstElement *pipeline;
  GstElement *mpeg2parser;

  // create pipeline and do something usefull
  ...

  mpeg2parser = gst_elementfactory_make ("mpeg2parse", "mpeg2parse");
  gtk_signal_connect (GTK_OBJECT (mpeg2parser), "new_pad", pad_connect_func, pipelin
  ...

  // start the pipeline
  gst_element_set_state (GST_ELEMENT (pipeline), GST_STATE_PLAYING);
  ...
}
```

Note: You need to set the pipeline to READY or NULL if you want to change it.

Request pads

An element can also have request pads. These pads are not created automatically but are only created on demand. This is very usefull for muxers, aggregators and tee elements.

The tee element, for example, has one input pad and a request padtemplate for the output pads. Whenever an element wants to get an output pad from the tee element, it has to request the pad.

The following piece of code can be used to get a pad from the tee element. After the pad has been requested, it can be used to connect another element to it.

```
  ...
GstPad *pad;
  ...
element = gst_elementfactory_make ("tee", "element");

pad = gst_element_request_pad_by_name (element, "src%d");
g_print ("new pad %s\n", gst_pad_get_name (pad));
  ...
```

The gst_element_request_pad_by_name method can be used to get a pad from the element based on the name_template of the padtemplate.

It is also possible to request a pad that is compatible with another padtemplate. This is very usefull if you want to connect an element to a muxer element and you need to request a pad that is compatible. The gst_element_request_compatible_pad is used to request a compatible pad, as is shown in the next example.

```
  ...
GstPadTemplate *templ;
GstPad *pad;
  ...
element = gst_elementfactory_make ("tee", "element");
mp3parse = gst_elementfactory_make ("mp3parse", "mp3parse");

templ = gst_element_get_padtemplate_by_name (mp3parse, "sink");

pad = gst_element_request_compatible_pad (element, templ);
g_print ("new pad %s\n", gst_pad_get_name (pad));
...
```

Capabilities of a GstPad

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 pad by using capabilities.

We will briefly describe what capabilities are, enough for you to get a basic understanding of the concepts. You will find more information on how to create capabilities in the filter-writer-guide.

What is a capability

A capability is attached to a pad in order to describe what type of media the pad can handle.

A capability is named and consists of a MIME type and a set of properties. Its data structure is:

```
struct _GstCaps {
  gchar *name;                  /* the name of this caps */

  guint16 id;                   /* type id (major type) */
```

```
  GstProps *properties;         /* properties for this capability */
};
```

Below is a dump of the capabilities of the element mpg123, as shown by **gstreamer-inspect**. You can see two pads: sink and src. Both pads have capability information attached to them.

The sink pad (input pad) is called 'sink' and takes data of MIME type 'audio/mp3'. It also has three properties: layer, bitrate and framed.

The src pad (output pad) is called 'src' and outputs data of MIME type 'audio/raw'. It also has four properties: format, depth, rate and channels.

```
Pads:
  SINK: 'sink'
    ....
      Capabilities:
      'mpg123_sink':
       MIME type: 'audio/mp3':
        layer: Integer range: 1 - 3
        bitrate: Integer range: 8 - 320
        framed: Boolean: TRUE

  SRC: 'src'
    ....
    Capabilities:
    'mpg123_src':
     MIME type: 'audio/raw':
      format: Integer: 16
      depth: Integer: 16
      rate: Integer range: 11025 - 48000
      channels: List:
        Integer: 1
        Integer: 2
```

What are properties

Properties are used to describe extra information for the capabilities. The properties basically exist of a key (a string) and a value. There are different possibile value types that can be used:

- An integer value: the property has this exact value.
- An integer range value. The property denotes a range of possible values. In the case of the mpg123 element: the src pad has a property rate that can go from 11025 to 48000.
- A boolean value.
- a fourcc value: this is a value that is commonly used to describe an encoding for video, as used be the AVI specification.
- A list value: the property can take any value from a list.
- A float value: the property has this exact floating point value.
- A float range value: denotes a range of possible floating point values.
- A string value.

What are the capabilities used for?

Capabilities describe in great detail the type of media that is handled by the pads. They are mostly used for:

- Autoplugging: automatically finding plugins for a set of capabilities
- Compatibility detection: when two pads are connected, `GStreamer` can verify if the two pads are talking about the same media types.

Getting the capabilities of a pad

A pad can have a GList of capabilities attached to it. You can get the capabilities list with:

```
GList *caps;
   ...
caps = gst_pad_get_caps_list (pad);

g_print ("pad name %s\n", gst_pad_get_name (pad));

while (caps) {
  GstCaps *cap = (GstCaps *) caps->data;

  g_print (" Capability name %s, MIME type\n", gst_caps_get_name (cap),
                                               gst_caps_get_mime (cap));

  caps = g_list_next (caps);
}
   ...
```

Chapter 7. Connecting elements

You can connect the different pads of elements together so that the elements form a chain.

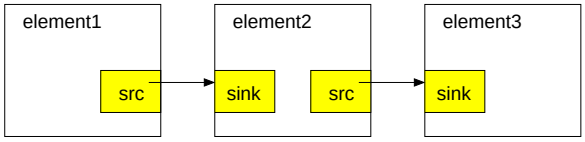

Figure 7-1. Visualisation of three connected elements

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

Imagine the above graph as a simple mpeg audio decoder. The source element is a disk source, the filter element is the mpeg decoder and the sink element is your audiocard. We will use this simple graph to construct an mpeg player later in this manual.

You can connect two pads with:

```
GstPad *srcpad, *sinkpad;

srcpad = gst_element_get_pad (element1, "src");
sinpad = gst_element_get_pad (element2, "sink");

// connect them
gst_pad_connect (srcpad, sinkpad);
  ....
// and disconnect them
gst_pad_disconnect (srcpad, sinkpad);
```

A convenient shortcut for the above code is done with the gst_element_connect () function:

```
// connect them
gst_element_connect (element1, "src", element2, "sink");
  ....
// and disconnect them
gst_element_disconnect (element1, "src", element2, "sink");
```

You can query if a pad is connected with GST_PAD_IS_CONNECTED (pad).

To query for the `GstPad` this srcpad is connected to, use gst_pad_get_peer (srcpad).

Chapter 8. Bins

A Bin is a container element. You can add elements to a bin. Since a bin is an `GstElement` itself, it can also be added to another bin.

Bins allow you to combine connected 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 powerfull 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.

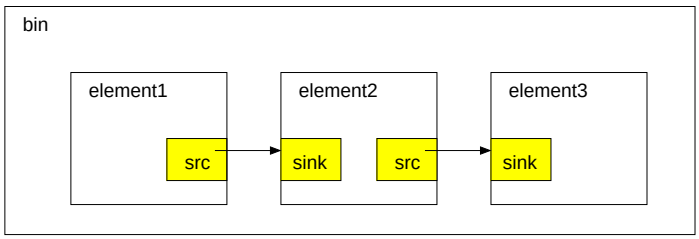

Figure 8-1. Visualisation of a `GstBin` element with some elements in it

There are two standard bins available to the GStreamer programmer:

- A pipeline (`GstPipeline`). Which is a generic container you will use most of the time.
- A thread (`GstThread`). All the elements in the thread bin will run in a separate thread. You will have to use this bin if you carfully have to synchronize audio and video for example. You will learn more about threads in..

Creating a bin

You create a bin with a specified name 'mybin' with:

```
GstElement *bin;

gst_bin_new ("mybin");
 ...
```

A thread can be created with:

```
GstElement *thread;

gst_thread_new ("mythread");
```

```
  ...
```

Pipelines are created with gst_pipeline_new ("name");

Adding elements to a bin

Elements are added to a bin with the following code sample:

```
GstElement *element;
GstElement *bin;

bin = gst_bin_new ("mybin");

element = gst_elementfactory_make ("mpg123", "decoder");
gst_bin_add (GST_BIN (bin), element);
 ...
```

Bins and threads can be added to other bins too. This allows you to create nested bins.

To get an element from the bin you can use:

```
GstElement *element;

element = gst_bin_get_by_name (GST_BIN (bin), "decoder");
 ...
```

You can see that the name of the element becomes very handy for retrieving the element from an bin by using the elements name. gst_bin_get_by_name () will recursively search nested bins.

To get a list of elements in a bin, use:

```
GList *elements;

elements = gst_bin_get_list (GST_BIN (bin));

while (elements) {
  GstElement *element = GST_ELEMENT (elements->data);

  g_print ("element in bin: %s\n", gst_element_get_name (element));

  elements = g_list_next (elements);
}
 ...
```

To remove an element from a bin use:

```
GstElement *element;

gst_bin_remove (GST_BIN (bin), element);
 ...
```

Custom bins

The application programmer can create custom bins packed with elements to perform a specific task. This allow you to write an MPEG audio decoder with just the follwing lines of code:

```
// create the mp3player element
GstElement *mp3player = gst_elementfactory_make("mp3player","mp3player");
// set the source mp3 audio file
gtk_object_set(GTK_OBJECT(mp3player), "location", "helloworld.mp3", NULL);
// start playback
gst_element_set_state(GST_ELEMENT(mp3player),GST_STATE_PLAYING);
 ...
// pause playback
gst_element_set_state(GST_ELEMENT(mp3player),GST_STATE_PAUSED);
 ...
// stop
gst_element_set_state(GST_ELEMENT(mp3player),GST_STATE_NULL);
```

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

Ghostpads

You can see from figure ... how a bin has no pads of its own. This is where Ghostpads come into play.

A ghostpad is a pad from some element in the bin that has been promoted to the bin. This way, the bin also has a pad. The bin becomes just another element with a pad and you can then use the bin just like any other element. This is a very important feature for creating custom bins.

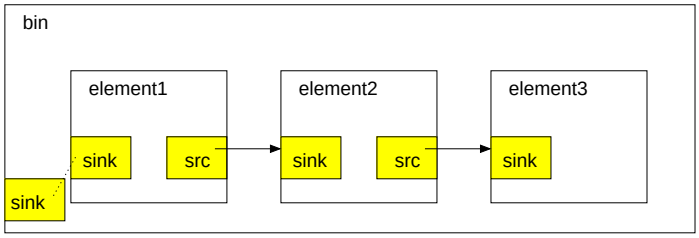

Figure 8-2. Visualisation of a `GstBin` element with a ghostpad

Above is a representation of a ghostpad. the sinkpad of element one is now also a pad of the bin.

Ghostpads can actually be added to all `GstElement`s and not just `GstBin`s. Use the following code example to add a ghostpad to a bin:

```
GstElement *bin;
GstElement *element;
```

```
element = gst_elementfactory_create ("mpg123", "decoder");
bin = gst_bin_new ("mybin");

gst_bin_add (GST_BIN (bin), element);

gst_element_add_ghost_pad (bin, gst_element_get_pad (element, "sink"));
```

In the above example, the bin now also has a pad: the pad called 'sink' of the given element. We can now, for example, connect the srcpad of a disksrc to the bin with:

```
GstElement *disksrc;

disksrc = gst_elementfactory_create ("disksrc", "disk_reader");

gst_element_connect (disksrc, "src", bin, "sink");
  ...
```

Chapter 9. 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 the 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.

The most important information in the buffer is:

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

GStreamer provides functions to create custom buffer create/destroy algorithms, called a `GstBufferPool`. This makes it possible to efficiently allocate and destroy buffer memory. It also makes it possible to exchange memory between elements by passing the `GstBufferPool`. A video element can, for example, create a custom buffer allocation algorithm that creates buffers with XSHM as the buffer memory. An element can use this algorithm to create and fill the buffer with data.

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

A more complex case is when the filter modifies the data in place. It does so and simply passes on the buffer to the next element. This is just as easy to deal with. An element that works in place has to be carefull when the buffer is used in more than one element; a copy on write has to made in this situation.

Chapter 10. Element states

One you have created a pipeline packed with elements, nothing will happen yet. This is where the different states come into play.

The different element states

All elements can be in one of the following four states:

- NULL: this is the default state all elements are in when they are created and are doing nothing.
- READY: An element is ready to start doing something.
- PLAYING: The element is doing something.
- PAUSED: The element is paused for a period of time.

All elements start with the NULL state. The elements will go throught the following state changes:

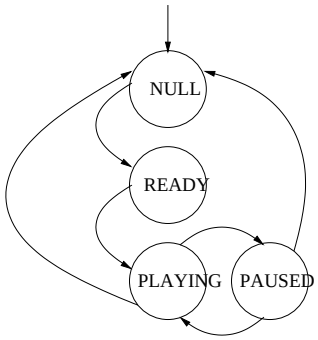

Figure 10-1. The different states of a `GstElement` and the state transitions

The state of an element can be changed with the following code:

```
GstElement *bin;

// create a bin, put elements in it and connect them
...
gst_element_set_state (bin, GST_STATE_PLAYING);
...
```

You can set the following states to an element:

`GST_STATE_NONE_PENDING` The element is in the desired state.

`GST_STATE_NULL` Reset the state of an element.

`GST_STATE_READY` will make the element ready to start processing data.

`GST_STATE_PLAYING` means there really is data flowing through the graph.

`GST_STATE_PAUSED` temporary stops the data flow.

The NULL state

When you created the pipeline all of the elements will be in the NULL state. There is nothing spectacular about the NULL state.

> **Note:** Don't forget to reset the pipeline to the NULL state when you are not going to use it anymore. This will allow the elements to free the resources they might use.

The READY state

You will start the pipeline by first setting it to the READY state. This will allow the pipeline and all the elements contained in it to prepare themselves for the actions they are about to perform.

The typical actions that an element will perform in the READY state might be to open a file or an audio device. Some more complex elements might have a non trivial action to perform in the READY state such as connecting to a media server using a CORBA connection.

> **Note:** You can also go from the NULL to PLAYING state directly without going through the READY state. this is a shortcut, the framework will internally go through the READY state for you.

The PLAYING state

A Pipeline that is in the READY state can be started by setting it to the PLAYING state. At that time data will start to flow all the way through the pipeline.

The PAUSED state

A pipeline that is playing can be set to the PAUSED state. This will temporarily stop all data flowing through the pipeline.

You can resume the data flow by setting the pipeline back to the PLAYING state.

> **Note:** The PAUSED state is available for temporarily freezing the pipeline. Elements will typically not free their resources in the PAUSED state. Use the NULL state if you want to stop the data flow permanantly.

The pipeline has to be in the PAUSED or NULL state if you want to insert or modify an element in the pipeline. We will cover dynamic pipeline behaviour in ...

Chapter 11. Your first application

This chapter describes the most rudimentary aspects of a GStreamer application, including initializing the libraries, creating elements, packing them into a pipeline and playing, pause and stop the pipeline.

Hello world

We will create a simple first application. In fact it will be a complete MP3 player, using standard GStreamer components. The player will read from a file that is given as the first argument of the program.

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

int
main (int argc, char *argv[])
{
  GstElement *bin, *disksrc, *parse, *decoder, *audiosink;

  gst_init(&argc, &argv);

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

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

  /* create a disk reader */
  disksrc = gst_elementfactory_make ("disksrc", "disk_source");
  gtk_object_set (GTK_OBJECT (disksrc),"location", argv[1], NULL);

  /* now it's time to get the parser */
  parse = gst_elementfactory_make ("mp3parse", "parse");
  decoder = gst_elementfactory_make ("mpg123", "decoder");

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

  /* add objects to the main pipeline */
  gst_bin_add (GST_BIN (bin), disksrc);
  gst_bin_add (GST_BIN (bin), parse);
  gst_bin_add (GST_BIN (bin), decoder);
  gst_bin_add (GST_BIN (bin), audiosink);

  /* connect src to sink */
  gst_pad_connect (gst_element_get_pad (disksrc, "src"),
                   gst_element_get_pad (parse, "sink"));
  gst_pad_connect (gst_element_get_pad (parse, "src"),
                   gst_element_get_pad (decoder, "sink"));
  gst_pad_connect (gst_element_get_pad (decoder, "src"),
                   gst_element_get_pad (audiosink, "sink"));

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

  while (gst_bin_iterate (GST_BIN (bin)));
```

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

  gst_object_destroy (GST_OBJECT (audiosink));
  gst_object_destroy (GST_OBJECT (parse));
  gst_object_destroy (GST_OBJECT (decoder));
  gst_object_destroy (GST_OBJECT (disksrc));
  gst_object_destroy (GST_OBJECT (bin));

  exit (0);
}
```

Let's go through this example step by step.

The first thing you have to do is to include the standard `GStreamer` headers and initialize the framework.

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

  ...

int
main (int argc, char *argv[])
{
  ...
  gst_init(&argc, &argv);
  ...
```

We are going to create 4 elements and one bin. Since all objects are in fact elements, we can define them as:

```
  ...
  GstElement *bin, *disksrc, *parse, *decoder, *audiosink;
  ...
```

Next, we are going to create an empty bin. As you have seen in the basic introduction, this bin will hold and manage all the elements we are going to stuff into it.

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

We use the standard constructor for a bin: gst_bin_new ("name").

We then create a disk source element. The disk source element is able to read from a file. We use the standard GTK+ argument mechanism to set a property of the element: the file to read from.

```
  /* create a disk reader */
  disksrc = gst_elementfactory_make ("disksrc", "disk_source");
  gtk_object_set (GTK_OBJECT (disksrc),"location", argv[1], NULL);
```

Note: You can check if the disksrc != NULL to verify the creation of the disk source element.

We now create the MP3 decoder element. GStreamer requires you to put a parser in front of the decoder. This parser will cut the raw data from the disk source into MP3 frames suitable for the decoder. In the advanced concepts chapter we will see how this can be avoided.

```
/* now it's time to get the parser */
parse = gst_elementfactory_make ("mp3parse", "parse");
decoder = gst_elementfactory_make ("mpg123", "decoder");
```

gst_elementfactory_make() takes two arguments: a string that will identify the element you need and a second argument: how you want to name the element. The name of the element is something you can choose yourself and might be used to retrieve the element from a bin.

Finally we create our audio sink element. This element will be able to playback the audio using OSS.

```
/* and an audio sink */
audiosink = gst_elementfactory_make ("audiosink", "play_audio");
```

We then add the elements to the bin.

```
/* add objects to the main pipeline */
gst_bin_add (GST_BIN (bin), disksrc);
gst_bin_add (GST_BIN (bin), parse);
gst_bin_add (GST_BIN (bin), decoder);
gst_bin_add (GST_BIN (bin), audiosink);
```

We connect the different pads of the elements together like this:

```
/* connect src to sink */
gst_pad_connect (gst_element_get_pad (disksrc, "src"),
                 gst_element_get_pad (parse, "sink"));
gst_pad_connect (gst_element_get_pad (parse, "src"),
                 gst_element_get_pad (decoder, "sink"));
gst_pad_connect (gst_element_get_pad (decoder, "src"),
                 gst_element_get_pad (audiosink, "sink"));
```

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

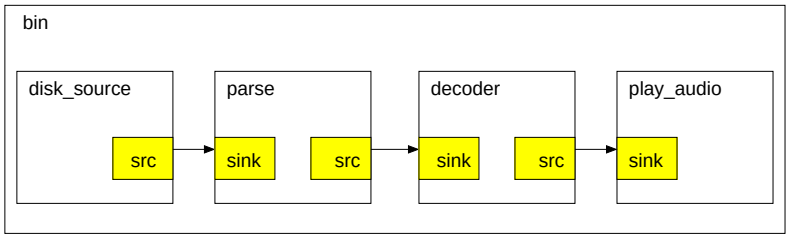

Figure 11-1. The Hello world pipeline

Everything is now set up to start the streaming. We use the following statements to change the state of the bin:

```
/* start playing */
gst_element_set_state (bin, GST_STATE_PLAYING);
```

Note: `GStreamer` will take care of the READY state for you when going from NULL to PLAYING.

Since we do not use threads, nothing will happen yet. We manually have to call gst_bin_iterate() to execute one iteration of the bin.

```
while (gst_bin_iterate (GST_BIN (bin)));
```

The gst_bin_iterate() function will return TRUE as long as something interesting happended inside the bin. When the end-of-file has been reached the _iterate function will return FALSE and we can end the loop.

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

gst_object_destroy (GST_OBJECT (audiosink));
gst_object_destroy (GST_OBJECT (decoder));
gst_object_destroy (GST_OBJECT (disksrc));
gst_object_destroy (GST_OBJECT (bin));

exit (0);
```

Note: don't forget to set the state of the bin to NULL. This will free all of the resources held by the elements.

compiling helloworld.c

To compile the helloworld example, use:

```
gcc -Wall ‘gstreamer-config --cflags --libs‘ helloworld.c \
      -o helloworld
```

This uses the program gstreamer-config, which comes with `GStreamer`. This program "knows" what compiler switches are needed to compile programs that use `GStreamer`. gstreamer-config --cflags will output a list of include directories for the compiler to look in, and gstreamer-config --libs will output the list of libraries for the compiler to link with and the directories to find them in.

You can run the example with (substitute helloworld.mp3 with you favorite MP3 file):

```
./helloworld helloworld.mp3
```

conclusion

This concludes our first example. As you see, setting up a pipeline is very lowlevel but powerfull. You will later in this manual how you can create a custom MP3 element with a more high level API.

It should be clear from the example that we can very easily replace the disksrc element with an httpsrc, giving you instant network streaming. An element could be build to handle icecast connections, for example.

We can also choose to use another type of sink instead of the audiosink. We could use a disksink to write the raw samples to a file, for example. It should also be clear that inserting filters, like a stereo effect, into the pipeline is not that hard to do. The most important thing is that you can reuse allready existing elements.

Chapter 12. More on factories

The small application we created in the previous chapter used the concept of a factory to create the elements. In this chapter we will show you how to use the factory concepts to create elements based on what they do instead of how they are called.

We will first explain the concepts involved before we move on to the reworked helloworld example using autoplugging.

The problems with the helloworld example

If we take a look at how the elements were created in the previous example we used a rather crude mechanism:

```
...
/* now it's time to get the parser */
parse = gst_elementfactory_make("mp3parse","parse");
decoder = gst_elementfactory_make("mpg123","decoder");
...
```

While this mechanism is quite effective it also has some big problems: The elements are created based on their name. Indeed, we create an element mpg123 by explicitly stating the mpg123 elements name. Our little program therefore always uses the mpg123 decoder element to decode the MP3 audio stream, even if there are 3 other MP3 decoders in the system. We will see how we can use a more general way to create an MP3 decoder element.

We have to introduce the concept of MIME types and capabilities added to the source and sink pads.

more on MIME Types

GStreamer uses MIME types to indentify the different types of data that can be handled by the elements. They are the high level mechanisms to make sure that everyone is talking about the right kind of data.

A MIME (Multipurpose Internet Mail Extension) types are a set of string that denote a certain type of data. examples include:

- audio/raw : raw audio samples
- audio/mpeg : mpeg audio
- video/mpeg : mpeg video

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. This allows for very dynamic and extensible element creation as we will see.

As we have seen in the previous chapter, the MIME types are added to the Capability structure of a pad.

In our helloworld example the elements we constructed would have the following MIME types associated with their source and sink pads:

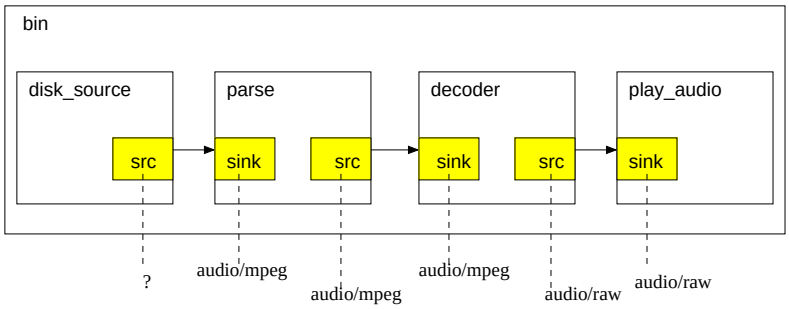

Figure 12-1. The Hello world pipeline with MIME types

We will see how you can create an element based on the MIME types of its source and sink pads. This way the end-user will have the ability to choose his/her favorite audio/mpeg decoder without you even having to care about it.

The typing of the source and sink pads also makes it possible to 'autoplug' a pipeline. We will have the ability to say: "construct me a pipeline that does an audio/mpeg to audio/raw conversion".

> **Note:** The basic GStreamer library does not try to solve all of your autoplug problems. It leaves the hard decisions to the application programmer, where they belong.

GStreamer types

GStreamer assigns a unique number to all registered MIME types. GStreamer also keeps a reference to a function that can be used to determine if a given buffer is of the given MIME type.

There is also an association between a MIME type and a file extension.

The type information is maintained in a list of `GstType`. The definition of a `GstType` is like:

```
typedef GstCaps (*GstTypeFindFunc) (GstBuffer *buf,gpointer *priv);

typedef struct _GstType GstType;

struct _GstType {
  guint16 id;                   /* type id (assigned) */

  gchar *mime;                  /* MIME type */
  gchar *exts;                  /* space-delimited list of extensions */

  GstTypeFindFunc typefindfunc; /* typefind function */
};
```

All operations on `GstType` occur via their `guint16 id` numbers, with `GstType` structure private to the GStreamer library.

MIME type to id conversion

We can obtain the id for a given MIME type with the following piece of code:

```
guint16 id;

id = gst_type_find_by_mime("audio/mpeg");
```

This function will return 0 if the type was not known.

id to `GstType` conversion

We can obtain the `GstType` for a given id with the following piece of code:

```
GstType *type;

type = gst_type_find_by_id(id);
```

This function will return NULL if the id was associated with any known `GstType`

extension to id conversion

We can obtain the id for a given file extension with the following piece of code:

```
guint16 id;

id = gst_type_find_by_ext(".mp3");
```

This function will return 0 if the extension was not known.

id to `GstElementFactory` conversion

When we have obtained a given type id using one of the above methods, we can obtain a list of all the elements that operate on this MIME type or extension.

Obtain a list of all the elements that use this id as source with:

```
GList *list;

list = gst_type_gst_srcs(id);
```

Obtain a list of all the elements that use this id as sink with:

```
GList *list;

list = gst_type_gst_sinks(id);
```

When you have a list of elements, you can simply take the first element of the list to obtain an appropriate element.

Note: As you can see, there might be a multitude of elements that are able to operate on video/raw types. some might include:

- an MP3 audio encoder.
- an audio sink.
- an audio resampler.
- a spectrum filter.

Depending on the application, you might want to use a different element. This is why GStreamer leaves that decision up to the application programmer.

id to id path detection

You can obtain a `GList` of elements that will transform the source id into the destination id.

```
GList *list;

list = gst_type_gst_sink_to_src(sourceid, sinkid);
```

This piece of code will give you the elements needed to construct a path from sourceid to sinkid. This function is mainly used in autoplugging the pipeline.

creating elements with the factory

In the previous section we described how you could obtain an element factory using MIME types. One the factory has been obtained, you can create an element using:

```
GstElementFactory *factory;
GstElement *element;

// obtain the factory
factory = ...

element = gst_elementfactory_create(factory, "name");
```

This way, you do not have to create elements by name which allows the end-user to select the elements he/she prefers for the given MIME types.

GStreamer basic types

GStreamer only has two builtin types:

- audio/raw : raw audio samples
- video/raw and image/raw : raw video data

All other MIME types are maintained by the plugin elements.

Chapter 13. Autoplugging

GStreamer provides an API to automatically construct complex pipelinebased on source and destination capabilities. This feature is very usefull if you want to convert type X to type Y but don't care about the plugins needed to accomplish this task. The autoplugger will consult the plugin repository, select and connect the elements needed for the conversion.

The autoplugger API is implemented in an abstract class. Autoplugger implementations reside in plugins and are therefore optional and can be optimized for a specific task. Two types of autopluggers exist: renderer ones and non renderer ones. the renderer autopluggers will not have any src pads while the non renderer ones do. The renderer autopluggers are mainly used for media playback while the non renderer ones are used for arbitrary format conversion.

Using autoplugging

You first need to create a suitable autoplugger with gst_autoplugfactory_make(). The name of the autoplugger must be one of the registered autopluggers..

A list of all available autopluggers can be obtained with gst_autoplugfactory_get_list().

If the autoplugger supports the RENDERER API, use gst_autoplug_to_renderers() call to create a bin that connects the src caps to the specified render elements. You can then add the bin to a pipeline and run it.

```
        GstAutoplug *autoplug;
GstElement  *element;
GstElement  *sink;

/* create a static autoplugger */
autoplug = gst_autoplugfactory_make ("staticrender");

/* create an osssink */
sink = gst_elementfactory_make ("osssink", "our_sink");

/* create an element that can play audio/mp3 through osssink */
element = gst_autoplug_to_renderers (autoplug,
                                     gst_caps_new (
                                       "sink_audio_caps",
                                       "audio/mp3",
                                       NULL
                                     ),
                                     sink,
                                     NULL);

/* add the element to a bin and connect the sink pad */
...
```

If the autoplugger supports the CAPS API, use the gst_autoplug_to_caps() function to connect the src caps to the destination caps. The created bin will have src and sink pads compatible with the provided caps.

```
        GstAutoplug *autoplug;
GstElement  *element;
```

```
/* create a static autoplugger */
autoplug = gst_autoplugfactory_make ("static");

/* create an element that converts audio/mp3 to audio/raw */
element = gst_autoplug_to_caps (autoplug,
                                gst_caps_new (
                                  "sink_audio_caps",
                                  "audio/mp3",
                                  NULL
                                ),
                                gst_caps_new (
                                  "src_audio_caps",
                                  "audio/raw",
                                  NULL
                                ),
                                NULL);

/* add the element to a bin and connect the src/sink pads */
...
```

A complete autoplugging example

We will create and explain how a complete media player can be built with the autoplugger.

Chapter 14. Your second application

In the previous chapter we created a first version of the helloworld application. We then explained a better way of creating the elements using factories identified by MIME types.

In this chapter we will introduce you to autoplugging. Using the MIME types of the elements `GStreamer` can automatically create a pipeline for you.

Autoplugging helloworld

We will create a second version of the helloworld application using autoplugging. Its source code is considerably easier to write and it can also handle many more data types.

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

static gboolean playing;

/* eos will be called when the src element has an end of stream */
void
eos (GstSrc *src)
{
  g_print ("have eos, quitting\n");

  playing = FALSE;
}

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

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

  gst_init (&argc, &argv);

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

  /* create a disk reader */
  disksrc = gst_elementfactory_make ("disksrc", "disk_source");
  gtk_object_set (GTK_OBJECT (disksrc), "location", argv[1], NULL);
  gtk_signal_connect (GTK_OBJECT (disksrc), "eos",
                      GTK_SIGNAL_FUNC (eos), NULL);

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

  /* add objects to the main pipeline */
  gst_pipeline_add_src (GST_PIPELINE (pipeline), disksrc);
  gst_pipeline_add_sink (GST_PIPELINE (pipeline), audiosink);

  if (!gst_pipeline_autoplug (GST_PIPELINE (pipeline))) {
    g_print ("unable to handle stream\n");
```

```
    exit (-1);
  }

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

  playing = TRUE;

  while (playing) {
    gst_bin_iterate (GST_BIN (pipeline));
  }

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

  gst_pipeline_destroy (pipeline);

  exit (0);
}
```

First of all, we do not use any mpg123 or mp3parse element in this example. In fact, we only specify a source element and a sink element and add them to a pipeline.

The most interesting change however is the following:

```
  ...
  if (!gst_pipeline_autoplug (pipeline)) {
    g_print ("unable to handle stream\n");
    exit (-1);
  }
  ...
```

This piece of code does all the magic.

- The pipeline will try to connect the src and the sink element.
- Since the source has no type, a typedetection will be started on the source element.
- The best set of elements that connect the MIME type of the source element to the MIME type of the sink are found.
- The elements are added to the pipeline and their pads are connected.

After this autoplugging, the pipeline is ready to play. Remember that this pipeline will be able to playback all of the media types for which an appropriate plugin exists since the autoplugging is all done using MIME types.

If you really want, you can use the GSteamer components to do the autoplugging yourself. We will cover this topic in the dynamic pipeline chapter.

To compile the helloworld2 example, use:

```
gcc -Wall ‘gstreamer-config --cflags --libs‘ helloworld2.c \
      -o helloworld2
```

You can run the example with (substitute helloworld.mp3 with you favorite MP3 file):

```
./helloworld2 helloworld.mp3
```

You can also try to use an AVI or MPEG file as its input. Using autoplugging, `GStreamer` will automatically figure out how to handle the stream. Remember that only the audio part will be played because we have only added an audiosink to the pipeline.

```
./helloworld2 mymovie.mpeg
```

Chapter 15. Threads

GStreamer has support for multithreading throught the use of the `GstThread` object. This object is in fact a special `GstBin` that will become a thread when started.

To construct a new thread you will perform something like:

```
GstElement *my_thread;

// create the thread object
my_thread = gst_thread_new ("my_thread");
g_return_if_fail (audio_thread != NULL);

// add some plugins
gst_bin_add (GST_BIN (my_thread), GST_ELEMENT (funky_src));
gst_bin_add (GST_BIN (my_thread), GST_ELEMENT (cool_effect));

// connect the elements here...
...

// start playing
gst_element_set_state (GST_ELEMENT (my_thread), GST_STATE_PLAYING);
```

The above program will create a thread with two elements in it. As soon as it is set to the PLAYING state, the thread will start to iterate.

> **Note:** A thread should normally contain a source element. Most often, the thread is fed with data from a queue.

A thread will be visualised as below

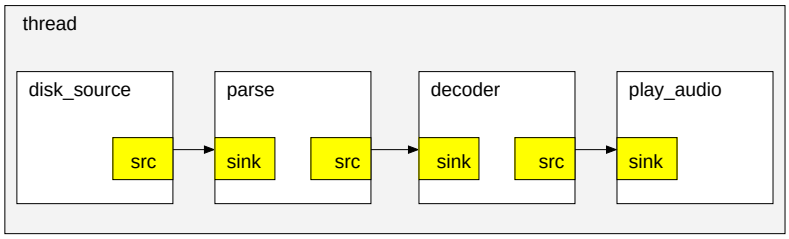

Figure 15-1. a thread

As an example we show the helloworld program using a thread.

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

/* eos will be called when the src element has an end of stream */
void
eos (GstSrc *src, gpointer data)
{
  GstThread *thread = GST_THREAD (data);
```

```
  g_print ("have eos, quitting\n");

  /* stop the bin */
  gst_element_set_state (GST_ELEMENT (thread), GST_STATE_NULL);

  gst_main_quit ();
}

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

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

  gst_init (&argc, &argv);

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

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

  /* create a disk reader */
  disksrc = gst_elementfactory_make ("disksrc", "disk_source");
  g_assert (disksrc != NULL);
  gtk_object_set (GTK_OBJECT (disksrc), "location", argv[1], NULL);
  gtk_signal_connect (GTK_OBJECT (disksrc), "eos",
                      GTK_SIGNAL_FUNC (eos), thread);

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

  /* add objects to the main pipeline */
  gst_bin_add (GST_BIN (pipeline), disksrc);
  gst_bin_add (GST_BIN (pipeline), audiosink);

  /* automatically setup the pipeline */
  if (!gst_pipeline_autoplug (GST_PIPELINE (pipeline))) {
    g_print ("unable to handle stream\n");
    exit (-1);
  }

  /* remove the source element from the pipeline */
  gst_bin_remove (GST_BIN (pipeline), disksrc);

  /* insert the source element in the thread, remember a thread needs at
     least one source or connection element */
  gst_bin_add (GST_BIN (thread), disksrc);

  /* add the pipeline to the thread too */
  gst_bin_add (GST_BIN (thread), GST_ELEMENT (pipeline));

  /* start playing */
  gst_element_set_state (GST_ELEMENT (thread), GST_STATE_PLAYING);
```

```
  /* do whatever you want here, the thread will be playing */
  ...

  gst_main ();

  gst_pipeline_destroy (thread);

  exit (0);
}
```

Chapter 16. Queues

A `GstQueue` is a filter element. Queues can be used to connect two elements in such way that the data can be buffered.

A buffer that is sinked to a Queue will not automatically be pushed to the next connected element but will be buffered. It will be pushed to the next element as soon as a gst_pad_pull () is called on the queues srcpad.

Queues are mostly used in conjunction with a `GstThread` to provide an external connection for the thread elements. You could have one thread feeding buffers into a `GstQueue` and another thread repeadedly calling gst_pad_pull () on the queue to feed its internal elements.

Below is a figure of a two-threaded decoder. We have one thread (the main execution thread) reading the data from a file, and another thread decoding the data.

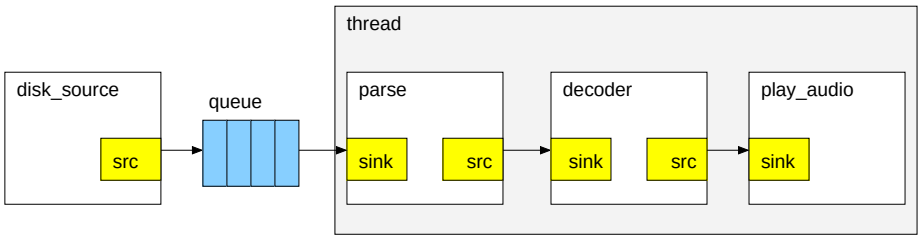

Figure 16-1. a two-threaded decoder with a queue

The standard `GStreamer` queue implementation has some properties that can be changed using the gtk_objet_set () method. To set the maximum number of buffers that can be queued to 30, do:

```
  gtk_object_set (GTK_OBJECT (queue), "max_level", 30, NULL);
```

The following mp3 player shows you how to create the above pipeline using a thread and a queue.

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

gboolean playing;

/* eos will be called when the src element has an end of stream */
void
eos (GstElement *element, gpointer data)
{
  g_print ("have eos, quitting\n");

  playing = FALSE;
}

int
main (int argc, char *argv[])
{
```

```
GstElement *disksrc, *audiosink, *queue, *parse, *decode;
GstElement *bin;
GstElement *thread;

gst_init (&argc,&argv);

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

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

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

/* create a disk reader */
disksrc = gst_elementfactory_make ("disksrc", "disk_source");
g_assert (disksrc != NULL);
gtk_object_set (GTK_OBJECT (disksrc), "location", argv[1], NULL);
gtk_signal_connect (GTK_OBJECT (disksrc), "eos",
                    GTK_SIGNAL_FUNC (eos), thread);

queue = gst_elementfactory_make ("queue", "queue");

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

parse = gst_elementfactory_make ("mp3parse", "parse");
decode = gst_elementfactory_make ("mpg123", "decode");

/* add objects to the main bin */
gst_bin_add (GST_BIN (bin), disksrc);
gst_bin_add (GST_BIN (bin), queue);

gst_bin_add (GST_BIN (thread), parse);
gst_bin_add (GST_BIN (thread), decode);
gst_bin_add (GST_BIN (thread), audiosink);

gst_pad_connect (gst_element_get_pad (disksrc,"src"),
                 gst_element_get_pad (queue,"sink"));

gst_pad_connect (gst_element_get_pad (queue, "src"),
                 gst_element_get_pad (parse, "sink"));
gst_pad_connect (gst_element_get_pad (parse, "src"),
                 gst_element_get_pad (decode, "sink"));
gst_pad_connect (gst_element_get_pad (decode, "src"),
                 gst_element_get_pad (audiosink, "sink"));

gst_bin_add (GST_BIN (bin), thread);

/* make it ready */
gst_element_set_state (GST_ELEMENT (bin), GST_STATE_READY);
/* start playing */
gst_element_set_state (GST_ELEMENT (bin), GST_STATE_PLAYING);

playing = TRUE;

while (playing) {
```

```
    gst_bin_iterate (GST_BIN (bin));
  }

  gst_element_set_state (GST_ELEMENT (bin), GST_STATE_NULL);

  exit (0);
}
```

Chapter 17. Cothreads

Cothreads are user-space threads that greatly reduce context switching overhead introduced by regular kernel threads. Cothreads are also used to handle the more complex elements.

A cothread is created by a `GstBin` whenever an element is found inside the bin that has one or more of the following properties:

- The element is loop-based instead of chain-based
- The element has multiple input pads
- The element has the MULTI_IN flag set

The `GstBin` will create a cothread context for all the elements in the bin so that the elements will interact in cooperative multithreading.

Before proceding to the concept of loop-based elements we will first explain the chain-based elements

Chain-based elements

Chain based elements receive a buffer of data and are supposed to handle the data and perform a gst_pad_push.

The basic main function of a chain-based element is like:

```
static void
chain_function (GstPad *pad, GstBuffer *buffer)
{
  GstBuffer *outbuffer;

  ....
  // process the buffer, create a new outbuffer
  ...

  gst_pad_push (srcpad, outbuffer);
}
```

Chain based function are mainly used for elements that have a one to one relation between their input and output behaviour. An example of such an element can be a simple video blur filter. The filter takes a buffer in, performs the blur operation on it and sends out the resulting buffer.

Another element, for example, is a volume filter. The filter takes audio samples as input, performs the volume effect and sends out the resulting buffer.

Loop-based elements

As opposed to chain-based elements, Loop-based elements enter an infinite loop that looks like this:

```
  GstBuffer *buffer, *outbuffer;

  while (1) {
```

```
  buffer = gst_pad_pull (sinkpad);
  ...
  // process buffer, create outbuffer
  while (!done) {
    ....
    // optionally request another buffer
    buffer = gst_pad_pull (sinkpad);
    ....
  }
  ...
  gst_pad_push (srcpad, outbuffer);
}
```

The loop-based elements request a buffer whenever they need one.

When the request for a buffer cannot immedialty satisfied, the control will be given to the source element of the loop-based element until it performs a push on its source pad. At that time the control is handed back to the loop-based element, etc... The the execution trace can get fairly complex using cothreads when there are multiple input/output pads for the loop-based element.

Loop based elements are mainly used for the more complex elements that need a specific amount of data before they can start to produce output. An example of such an element is the mpeg video decoder. the element will pull a buffer, performs some decoding on it and optionally requests more buffers to decode, when a complete video frame has been decoded, a buffer is send out.

There is no problem in putting cothreaded elements into a `GstThread` to create even more complex pipelines with both user and kernel space threads.

Chapter 18. Dynamic pipelines

this chapter we will see how you can create a dynamic pipleine. A dynamic pipeline is a pipeline that is updated or created while media is flowing through it. We will create a partial pipeline first and add more elements while the pipeline is playing. Dynamic pipelines cause all sorts of scheduling issues and will remain a topic of research for a long time in GStreamer.

We will show how to create an mpeg1 video player using dynamic pipelines. As you have seen in the pad section, we can attach a signal to an element when a pad is created. We will use this to create our MPEG1 player.

We'll start with a simple main function:

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

void eof(GstElement *src) {
  g_print("have eos, quitting\n");
  exit(0);
}

gboolean
idle_func (gpointer data)
{
  gst_bin_iterate (GST_BIN (data));
  return TRUE;
}

int
main(int argc, char *argv[])
{
  GstElement *pipeline, *src, *parse;

  gst_init (&argc, &argv);
  gnome_init ("MPEG1 Video player","0.0.1", argc, argv);

  pipeline = gst_pipeline_new ("pipeline");
  g_return_val_if_fail (pipeline != NULL, -1);

  src = gst_elementfactory_make ("disksrc", "src");
  g_return_val_if_fail (src != NULL, -1);
  gtk_object_set (GTK_OBJECT (src), "location", argv[1], NULL);

  parse = gst_elementfactory_make ("mpeg1parse", "parse");
  g_return_val_if_fail (parse != NULL, -1);

  gst_bin_add (GST_BIN (pipeline), GST_ELEMENT (src));
  gst_bin_add (GST_BIN (pipeline), GST_ELEMENT (parse));

  gtk_signal_connect (GTK_OBJECT (parse), "new_pad",
                      GTK_SIGNAL_FUNC (new_pad_created), pipeline);

  gtk_signal_connect (GTK_OBJECT (src), "eos",
                      GTK_SIGNAL_FUNC (eof), NULL);

  gst_pad_connect (gst_element_get_pad (src, "src"),
                   gst_element_get_pad (parse, "sink"));

  gst_element_set_state (GST_ELEMENT (pipeline), GST_STATE_PLAYING);
```

```
  g_idle_add (idle_func, pipeline);

  gdk_threads_enter ();
  gst_main ();
  gdk_threads_leave ();

  return 0;
}
```

We create two elements: a disksrc (the element that will read the file from disk) and an mpeg1parser. We also add an EOS (End Of Stream) signal to the disksrc so that we will be notified when the file has ended. There's nothing special about this piece of code except for the signal 'new_pad' that we connected to the mpeg1parser using:

```
  gtk_signal_connect (GTK_OBJECT (parse), "new_pad",
                      GTK_SIGNAL_FUNC (new_pad_created), pipeline);
```

When an elementary stream has been detected in the system stream, mpeg1parse will create a new pad that will provide the data of the elementary stream. A function 'new_pad_created' will be called when the pad is created:

```
void
new_pad_created (GstElement *parse, GstPad *pad, GstElement *pipeline)
{
  GstElement *parse_audio, *parse_video, *decode, *decode_video, *play, *videoscale,
  GstElement *audio_queue, *video_queue;
  GstElement *audio_thread, *video_thread;

  GtkWidget *appwindow;

  g_print ("***** a new pad %s was created\n", gst_pad_get_name (pad));

  gst_element_set_state (GST_ELEMENT (pipeline), GST_STATE_PAUSED);

  // connect to audio pad
  if (strncmp (gst_pad_get_name (pad), "audio_", 6) == 0) {

    // construct internal pipeline elements
    parse_audio = gst_elementfactory_make ("mp3parse", "parse_audio");
    g_return_if_fail (parse_audio != NULL);
    decode = gst_elementfactory_make ("mpg123", "decode_audio");
    g_return_if_fail (decode != NULL);
    play = gst_elementfactory_make ("audiosink", "play_audio");
    g_return_if_fail (play != NULL);

    // create the thread and pack stuff into it
    audio_thread = gst_thread_new ("audio_thread");
    g_return_if_fail (audio_thread != NULL);
    gst_bin_add (GST_BIN (audio_thread), GST_ELEMENT (parse_audio));
    gst_bin_add (GST_BIN (audio_thread), GST_ELEMENT (decode));
    gst_bin_add (GST_BIN (audio_thread), GST_ELEMENT (play));

    // set up pad connections
    gst_element_add_ghost_pad (GST_ELEMENT (audio_thread),
                               gst_element_get_pad (parse_audio, "sink"));
    gst_pad_connect (gst_element_get_pad (parse_audio,"src"),
                     gst_element_get_pad (decode,"sink"));
    gst_pad_connect (gst_element_get_pad (decode,"src"),
                     gst_element_get_pad (play,"sink"));
```

```
    // construct queue and connect everything in the main pipelie
    audio_queue = gst_elementfactory_make ("queue", "audio_queue");

    gst_bin_add (GST_BIN (pipeline), GST_ELEMENT (audio_queue));
    gst_bin_add (GST_BIN (pipeline), GST_ELEMENT (audio_thread));

    gst_pad_connect (pad,
                     gst_element_get_pad (audio_queue, "sink"));
    gst_pad_connect (gst_element_get_pad (audio_queue, "src"),
                     gst_element_get_pad (audio_thread, "sink"));

    // set up thread state and kick things off
    g_print ("setting to READY state\n");
    gst_element_set_state (GST_ELEMENT (audio_thread), GST_STATE_READY);

  }
  else if (strncmp (gst_pad_get_name (pad), "video_", 6) == 0) {

    // construct internal pipeline elements
    parse_video = gst_elementfactory_make ("mp1videoparse", "parse_video");
    g_return_if_fail (parse_video != NULL);
    decode_video = gst_elementfactory_make ("mpeg_play", "decode_video");
    g_return_if_fail (decode_video != NULL);

    show = gst_elementfactory_make ("videosink", "show");
    g_return_if_fail (show != NULL);

    appwindow = gnome_app_new ("MPEG1 player", "MPEG1 player");
    gnome_app_set_contents (GNOME_APP (appwindow),
                gst_util_get_widget_arg (GTK_OBJECT (show), "widget"));
gtk_widget_show_all (appwindow);

    // create the thread and pack stuff into it
    video_thread = gst_thread_new ("video_thread");
    g_return_if_fail (video_thread != NULL);
    gst_bin_add (GST_BIN (video_thread), GST_ELEMENT (parse_video));
    gst_bin_add (GST_BIN (video_thread), GST_ELEMENT (decode_video));
    gst_bin_add (GST_BIN (video_thread), GST_ELEMENT (show));

    // set up pad connections
    gst_element_add_ghost_pad (GST_ELEMENT (video_thread),
                               gst_element_get_pad (parse_video, "sink"));
    gst_pad_connect (gst_element_get_pad (parse_video, "src"),
                     gst_element_get_pad (decode_video, "sink"));
    gst_pad_connect (gst_element_get_pad (decode_video, "src"),
                     gst_element_get_pad (show, "sink"));

    // construct queue and connect everything in the main pipeline
    video_queue = gst_elementfactory_make ("queue", "video_queue");

    gst_bin_add (GST_BIN (pipeline), GST_ELEMENT (video_queue));
    gst_bin_add (GST_BIN (pipeline), GST_ELEMENT (video_thread));

    gst_pad_connect (pad,
                     gst_element_get_pad (video_queue, "sink"));
    gst_pad_connect (gst_element_get_pad (video_queue, "src"),
                     gst_element_get_pad (video_thread, "sink"));

    // set up thread state and kick things off
    gtk_object_set (GTK_OBJECT (video_thread), "create_thread", TRUE, NULL);
    g_print ("setting to READY state\n");
    gst_element_set_state (GST_ELEMENT (video_thread), GST_STATE_READY);
```

```
  }
  g_print("\n");
  gst_element_set_state (GST_ELEMENT (pipeline), GST_STATE_PLAYING);
}
```

In the above example, we created new elements based on the name of the newly created pad. We added them to a new thread There are other possibilities to check the type of the pad, for example, by using the MIME type and the properties of the pad.

Chapter 19. Typedetection

Sometimes the capabilities of a pad are not specificied. The disksrc, for example, does not know what type of file it is reading. Before you can attach an element to the pad of the disksrc, you need to determine the media type in order to be able to choose a compatible element.

To solve this problem, a plugin can provide the `GStreamer` core library with a typedefinition library with a typedefinition. The typedefinition will contain the following information:

- The MIME type we are going to define.
- An optional string with a list of possible file extensions this type usually is associated with. the list entries are separated with a space. eg, ".mp3 .mpa .mpg".
- An optional typefind function.

The typefind functions give a meaning to the MIME types that are used in GStreamer. The typefind function is a function with the following definition:

```
typedef GstCaps *(*GstTypeFindFunc) (GstBuffer *buf, gpointer priv);
```

This typefind function will inspect a GstBuffer with data and will output a GstCaps structure describing the type. If the typefind function does not understand the buffer contents, it will return NULL.

`GStreamer` has a typefind element in its core elements that can be used to determine the type of a given pad.

The next example will show how a typefind element can be inserted into a pipeline to detect the media type of a file. It will output the capabilities of the pad into an XML representation.

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

void    type_found      (GstElement *typefind, GstCaps* caps);

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

  gst_init(&argc,&argv);

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

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

  /* create a disk reader */
  disksrc = gst_elementfactory_make("disksrc", "disk_source");
  g_assert(disksrc != NULL);
  gtk_object_set(GTK_OBJECT(disksrc),"location", argv[1],NULL);
```

```
  /* create the typefind element */
  typefind = gst_elementfactory_make("typefind", "typefind");
  g_assert(typefind != NULL);

  /* add objects to the main pipeline */
  gst_bin_add(GST_BIN(bin), disksrc);
  gst_bin_add(GST_BIN(bin), typefind);

  gtk_signal_connect (GTK_OBJECT (typefind), "have_type",
  type_found, NULL);

  gst_pad_connect(gst_element_get_pad(disksrc,"src"),
                  gst_element_get_pad(typefind,"sink"));

  /* start playing */
  gst_element_set_state(GST_ELEMENT(bin), GST_STATE_PLAYING);

  gst_bin_iterate(GST_BIN(bin));

  gst_element_set_state(GST_ELEMENT(bin), GST_STATE_NULL);

  exit(0);
}
```

We create a very simple pipeline with only a disksrc and the typefind element in it. The sinkpad of the typefind element has been connected to the src pad of the disksrc.

We attached a signal 'have_type' to the typefind element which will be called when the type of the media stream as been detected.

the typefind function will loop over all the registered types and will execute each of the typefind functions. As soon as a function returns a GstCaps pointer, the type_found function will be called:

```
void
type_found (GstElement *typefind, GstCaps* caps)
{
  xmlDocPtr doc;
  xmlNodePtr parent;

  doc = xmlNewDoc ("1.0");
  doc->root = xmlNewDocNode (doc, NULL, "Capabilities", NULL);

  parent = xmlNewChild (doc->root, NULL, "Caps1", NULL);
  gst_caps_save_thyself (caps, parent);

  xmlDocDump (stdout, doc);
}
```

In the type_found function we can print or inspect the type that has been detected using the GstCaps APIs. In this example, we just print out the XML representation of the caps structure to stdout.

A more usefull option would be to use the registry to look up an element that can handle this particular caps structure, or we can also use the autoplugger to connect this caps structure to, for example, a videosink.

Chapter 20. Utility functions

while you can use the regular gtk_object_getv () function to query the value of an object property, `GStreamer` provides some easy wrappers for this common operation.

Instead of writing the following Gtk+ code to query the GTK_STRING value of an object:

```
GtkArg arg;
guchar *value;

arg.name = argname;
gtk_object_getv (GTK_OBJECT (object), 1, &arg);
value = GTK_VALUE_STRING (arg);
```

You can also use:

```
value = gst_util_get_string_arg (object, argname);
```

These convenience functions exist for the following types:

- gint: with gst_util_get_int_arg ();
- gboolean: with gst_util_get_bool_arg ();
- glong: with gst_util_get_long_arg ();
- gfloat: with gst_util_get_float_arg ();
- gdouble: with gst_util_get_double_arg ();
- guchar*: with gst_util_get_string_arg ();
- gpointer: with gst_util_get_pointer_arg ();
- GtkWidget*: with gst_util_get_widget_arg ();

There is also another utility function that can be used to dump a block of memory on the console. This function is very usefull for plugin developers. The function will dump size bytes of the memory pointed to by mem.

```
void gst_util_dump_mem(guchar *mem, guint size);
```

Chapter 21. 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 save it to disk with gst_xml_write (). The following code constructs an mp3 player pipeline with two threads and finaly writes it to disk. 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 *disksrc, *audiosink, *queue, *queue2, *parse, *decode;
  GstElement *bin;
  GstElement *thread, *thread2;

  gst_init (&argc,&argv);

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

  /* create a new thread to hold the elements */
  thread = gst_elementfactory_make ("thread", "thread");
  g_assert (thread != NULL);
  thread2 = gst_elementfactory_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 */
  disksrc = gst_elementfactory_make ("disksrc", "disk_source");
  g_assert (disksrc != NULL);
  gtk_object_set (GTK_OBJECT (disksrc), "location", argv[1], NULL);

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

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

  parse = gst_elementfactory_make ("mp3parse", "parse");
  decode = gst_elementfactory_make ("mpg123", "decode");
```

```
  /* add objects to the main bin */
  gst_bin_add (GST_BIN (bin), disksrc);
  gst_bin_add (GST_BIN (bin), queue);

  gst_bin_add (GST_BIN (thread), parse);
  gst_bin_add (GST_BIN (thread), decode);
  gst_bin_add (GST_BIN (thread), queue2);

  gst_bin_add (GST_BIN (thread2), audiosink);

  gst_pad_connect (gst_element_get_pad (disksrc,"src"),
                   gst_element_get_pad (queue,"sink"));

  gst_pad_connect (gst_element_get_pad (queue,"src"),
                   gst_element_get_pad (parse,"sink"));
  gst_pad_connect (gst_element_get_pad (parse,"src"),
                   gst_element_get_pad (decode,"sink"));
  gst_pad_connect (gst_element_get_pad (decode,"src"),
                   gst_element_get_pad (queue2,"sink"));

  gst_pad_connect (gst_element_get_pad (queue2,"src"),
                   gst_element_get_pad (audiosink,"sink"));

  gst_bin_add (GST_BIN (bin), thread);
  gst_bin_add (GST_BIN (bin), thread2);

  // write the bin to disk
  xmlSaveFile ("xmlTest.gst", gst_xml_write (GST_ELEMENT (bin)));

  exit (0);
}
```

The most important line is:

```
  xmlSaveFile ("xmlTest.gst", gst_xml_write (GST_ELEMENT (bin)));
```

gst_xml_write () will turn the given element into and xmlDocPtr that can be saved with the xmlSaveFile () function found in the gnome-xml package. The result is an XML file named xmlTest.gst.

The complete element hierarchy will be saved along with the inter element pad connections 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 respectivily. 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 connect 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_elementfactory_make("thread", "thread");
  gtk_signal_connect (GTK_OBJECT (thread), "object_saved", object_saved, g_strdup ("
coder thread"));
    ...
```

When the thread is saved, the object_save method will be caled. 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 ();

  gtk_signal_connect (GTK_OBJECT (xml), "object_loaded", 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 22. Debugging

GStreamer has an extensive set of debugging tools for plugin developers.

Command line options

Applications using the GStreamer libraries accept the following set of command line argruments to enable the debugging system.

- `--gst-debug-mask=`*`mask`* Sets the mask for the debugging output.
- `--gst-info-mask=`*`mask`* Sets the mask for the info output.
- `--gst-plugin-spew` Enable printout of errors while loading GST plugins.
- `--gst-plugin-path=`*`PATH`* Add a directory to the plugin search path.
- `--help` Print the a short desciption of the options and an overview of the current debugging/info masks set.

The follwing table gives an overview of the mask values and their meaning. (enabled) means that the corresponding flag has been set.

```
Mask (to be OR'ed)   info/debug         FLAGS
------------------------------------------------------
0x00000001     (enabled)/              GST_INIT
0x00000002              /              COTHREADS
0x00000004              /              COTHREAD_SWITCH
0x00000008              /              AUTOPLUG
0x00000010              /              AUTOPLUG_ATTEMPT
0x00000020              /              PARENTAGE
0x00000040              /              STATES
0x00000080              /              PLANING
0x00000100              /              SCHEDULING
0x00000200              /              OPERATION
0x00000400              /              BUFFER
0x00000800              /              CAPS
0x00001000              /              CLOCK
0x00002000              /              ELEMENT_PADS
0x00004000              /              ELEMENTFACTORY
0x00008000              /              PADS
0x00010000              /              PIPELINE
0x00020000              /              PLUGIN_LOADING
0x00040000              /              PLUGIN_ERRORS
0x00080000              /              PROPERTIES
0x00100000              /              THREAD
0x00200000              /              TYPES
0x00400000              /              XML
```

Adding a custom debug handler

Chapter 23. Programs

gstreamer-config

gstreamer-config is a script to get information about the installed version of `GStreamer`. This program "knows" what compiler switches are needed to compile programs that use `GStreamer`.

gstreamer-config accepts the following options:

- `--version` Print the currently installed version of `GStreamer` on the standard output.
- `--libs` Print the linker flags that are necessary to link a `GStreamer` program.
- `--cflags` Print the compiler flags that are necessary to compile a `GStreamer` program.
- `--prefix=`*`PREFIX`* If specified, use *`PREFIX`* instead of the installation prefix that `GStreamer` was built with when computing the output for the `--cflags` and `--libs` options. This option is also used for the exec prefix if `--exec-prefix` was not specified. This option must be specified before any `--libs` or `--cflags` options.
- `--exec-prefix=`*`PREFIX`* If specified, use *`PREFIX`* instead of the installation exec prefix that `GStreamer` was built with when computing the output for the `--cflags` and `--libs` options. This option must be specified before any `--libs` or `--cflags` options.

A simple `Makefile` will contain something like:

```
CC = gcc

helloworld2: helloworld2.c
   $(CC) -Wall ‘gstreamer-config --cflags --libs‘ helloworld2.c -o helloworld2

clean:
   rm -f *.o helloworld2
```

gstreamer-register

gstreamer-register is used to rebuild the database of plugins. It is used after a new plugin has been added to the system. The plugin database can be found in `/etc/gstreamer/reg.xml`.

gstreamer-launch

This is a tool that will construct pipelines based on a command-line syntax.

A simple commandline looks like:

```
gstreamer-launch disksrc location=hello.mp3 ! mp3parse ! mpg123 ! au-
diosink
```

A more complex pipeline looks like:

```
gstreamer-launch disksrc redpill.vob audio_00! (ac3parse ! ac3dec ! au-
diosink) \
video_00! (mpeg2dec ! videosink)
```

Note that the parser isn't capable of more complex pipelines yet, including the VOB player above. The minor tweaks will be made post 0.1.0.

You can also use the the parser in you own code. `GStreamer` provides a function gst_parse_launch () that you can use to construt a pipeline. The code of gstreamer-launch actually looks like:

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

int
main(int argc, char *argv[])
{
  GstElement *pipeline;
  char **argvn;
  gchar *cmdline;
  int i;

  gst_init(&argc,&argv);

  pipeline = gst_pipeline_new("launch");

  // make a null-terminated version of argv
  argvn = g_new0(char *,argc);
  memcpy(argvn,argv+1,sizeof(char*)*(argc-1));
  // join the argvs together
  cmdline = g_strjoinv(" ",argvn);
  // free the null-terminated argv
  g_free(argvn);

  gst_parse_launch(cmdline,pipeline);

  fprintf(stderr,"RUNNING pipeline\n");
  gst_element_set_state(pipeline,GST_STATE_PLAYING);

  while (1)
    gst_bin_iterate (GST_BIN (pipeline));

  return 0;
}
```

gstreamer-inspect

This is a tool to query a plugin or an element about its properties.

To query the information about the element mpg123, you would specify:

```
gstreamer-inspect mpg123
```

Below is the output of a query for the audiosink element:

```
Factory Details:
  Long name: Audio Sink (OSS)
  Class: Sink/Audio
  Description: Output to a sound card via OSS
  Version: 0.1.0
  Author(s): Erik Walthinsen <omega@cse.ogi.edu>
  Copyright: (C) 1999

Pad Templates:
  SINK template: 'sink'
    Exists: Always
    Capabilities:
      'audiosink_sink':
        MIME type: 'audio/raw':
        format: Integer: 16
        depth: List:
          Integer: 8
          Integer: 16
        rate: Integer range: 8000 - 48000
        channels: Integer range: 1 - 2

Element Flags:
  GST_ELEMENT_THREADSUGGESTED
  no flags set

Element Implementation:
  No loopfunc(), must be chain-based or not configured yet
  Has change_state() function

Pads:
  SINK: 'sink'
    Implementation:
      Has chainfunc(): 0x4001cde8
      Has default eosfunc() gst_pad_eos_func()
    Pad Template: 'sink'
    Capabilities:
      'audiosink_sink':
        MIME type: 'audio/raw':
        format: Integer: 16
        depth: List:
          Integer: 8
          Integer: 16
        rate: Integer range: 8000 - 48000
        channels: Integer range: 1 - 2

Element Arguments:
  GstAudioSink::mute: Boolean
  GstAudioSink::format: Enum (default 16)
    (8):  8 Bits
    (16):  16 Bits
  GstAudioSink::channels: Enum (default 2)
    (1):  Mono
```

```
    (2):  Stereo
  GstAudioSink::frequency: Integer
```

To query the information about a plugin, you would do:

```
gstreamer-inspect gstelements
```

gstmediaplay

A sample media player.

Chapter 24. Components

`GStreamer` includes components that people can include in their programs.

GstPlay

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

GstMediaPlay

GstMediaply is a complete player widget.

GstEditor

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

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.openprojects.net: the following are a selection of amusing[1] quotes from our conversations.

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 oru 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.