Writing Audio Applications using GStreamer

Stefan KOST
GStreamer community, Nokia/Meego
Majurinkatu 12 B 43
Espoo, Finland, 02600
ensonic@sonicpulse.de

Abstract

GStreamer is mostly known for its use in media players. Although the API and the plugin collection has much to offer for audio composing and editing applications as well. This paper introduces the framework, focusing on features interesting for audio processing applications. The author reports about practical experience of using GStreamer inside the Buzztard project.

Keywords

GStreamer, Framework, Composer, Audio.

1 GStreamer framework

GStreamer [1] is an open source, cross platform, graph based, multimedia framework. Development takes place on freedesktop.org. GStreamer is written in C, but in an object oriented fashion using the GObject framework. Numerous language bindings exist, e.g. for Python, C#, Java and Perl. GStreamer has been ported to a wide variety of platforms such as Linux, Windows, MacOS, BSD and Solaris. The framework and most plugins are released under LGPL license. The project is over 10 years old now and has many contributors. The current 0.10 series is API and ABI stable and being actively developed since about 5 years.

A great variety of applications is using GStreamer already today. To give some examples, then GNOME desktop is using it for its Mediaplayers Totem and Rhythmbox and the Chat&Voip client Empathy, platforms like Nokias Maemo and Intels Moblin use GStreamer for all multimedia tasks. In the audio domain two applications to mention are Jokosher (a multitrack audio editor) and Buzztard [2] (a tracker like musical composer). The latter is the pet project of the article's author and will be used later on as an example case and thus deserves a little introduction. A song in Buzztard is a network of sound generators, effects and an audio sink. All parameters on each of the elements can be controlled interactively or by the sequencer. All audio is rendered on the fly by algorithms and by optionally accessing a wavetable of sampled sounds. [3] gives a nice overview of live audio effects in Buzztard.

1.1 Multimedia processing graph

A GStreamer application constructs its processing graph in a GstPipeline object. This is a container for actual elements (source, effect and sinks). A container element is an element itself, supporting the same API as the real elements it contains. A pipeline is built by adding more elements/containers and linking them through their “pads”. A pipeline can be anything from a simple linear sequence to a complex hierarchical directed graph (see Illustration 1).

As mentioned above, elements are linked by connecting two pads – a source pad of the source element and a sink pad from the target element. GStreamer knows about several types of pads – always-, sometimes- and request-pads:

- always-pads are static (always available)
- sometimes-pads appear and disappear

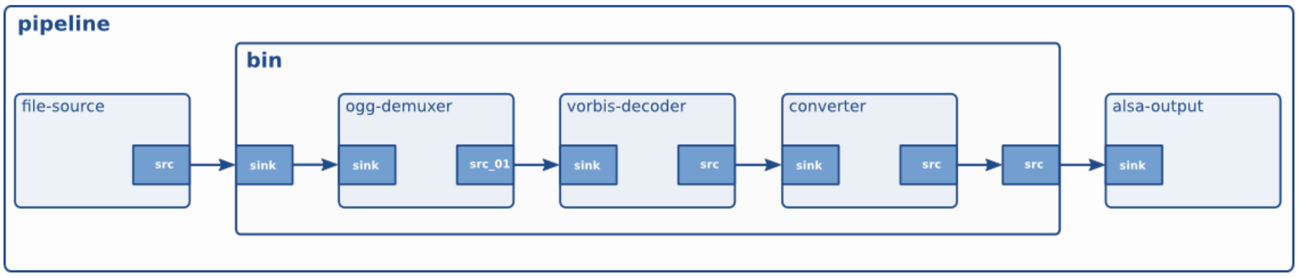

Illustration 1: A hierarchical GStreamer pipeline

according to data-flow (e.g. new stream on multiplexed content)

- request-pads are a template that can be instantiated multiple times (e.g. inputs for a mixer element)

Besides generic containers (containers to structure a pipeline like GstBin and GstPipeline) several specialized containers exist:

- GstPlaybin2: builds a media playback pipeline based on the media content, e.g. used in Totem
- GstCamerabin: abstract still image/video capture workflow, e.g. used in Nokia N900
- Gnonlin: multitrack audio/video editor, e.g. used in Jokosher and Pitivi

1.2 Data formats

The GStreamer core is totally data format agnostic. Formats like audio and video (and their properties) are only introduced with the gst-plugins-base module. New formats can be added without modifying the core (consider adding spectral audio processing plugins like VAMP [4]). Data formats are described by GstCaps which is an object containing a media-type and a list of {key, value} pairs. Values can be fixed or variable (a range or list of possible values). The GstCaps class implements operations on caps (union, intersection, iterator,...). This is quite different to e.g. ladspa where the standard specifies a fix format like float-audio with 1 channel.

Elements register template caps for their pads. This describes the data formats that they can accept. When starting the pipeline, elements agree on the formats that they will use and store this in their pads. Different pads can use different formats and formats can also change while the pipeline is running.

GStreamer plugin packages provide various data format conversion plugins such as e.g. audioconvert and audioresample. Such plugins switch to pass-through mode if no conversion is needed. For performance and quality reason it is of course best to avoid conversion.

1.3 Data Flow

The framework provides various communication and I/O mechanisms between elements and the application. The main data-flow is done by transferring buffers from pad to pad. A pipeline can operate “push based”, “pull based” and in “mixed mode”. In push mode, source elements are active and deliver data downstream. In pull mode sink elements request data form upstream elements when needed. Pure pull based mode is interesting for audio applications, but not yet fully supported by all elements. Regardless of the scheduling, buffers always travel downstream (from sources to sinks).

Events are used for control flow between elements and from the application to the pipeline. Events inside the pipeline are bidirectional – some go upstream, some go downstream. E.g. seeking is done by using events.

Messages are used to inform the application from the inside of the pipeline. Messages are sent to a bus. There they are marshalled to the application thread. The application can subscribe to interesting messages. E.g. multimedia metadata (such as id3 tags) are send as a tag-message.

Queries are mostly used by the application to check the status of the pipeline. E.g. one can use a position-query to check the current playback position. Queries can also be used by elements, both up- and downstream.

Type	Direction
Buffer	Between elements
Event	From application to elements and between elements
Message	From elements to application via the message bus
Query	From applications to elements and between elements.

Table 1: Communication primitives

All the communication object are lightweight object (GstMiniObject). They support subclassing, but no properties and signals.

1.4 Multi threading

GStreamer pipelines always use at least one thread distinct from the main ui thread[1]. Depending on the scheduling model and pipeline structure more threads are used. The message bus described in the previous chapter is used to transfer messages to the applications main thread.

[1]This also applies to commandline applications

Together these mechanisms decouple the UI from the media handling.

Sink elements usually run a rendering thread. In pull based scheduling also all the upstream data processing would run in this thread.

In push based scheduling (which is the default for most elements), sources start a thread. Whenever data-flow branches out, one would add queue elements which again start new threads for their source pad (see Illustration 2 for an example).

Threads are taken from a thread-pool. GStreamer provides a default thread-pool, but the application can also provide its own. This allows (some) processing to run under different scheduling modes. The threads from the default pool are run with default scheduling (inherited from parent and thus usually SCHED_OTHER).

1.5 Plugin/Element API

GStreamer elements are GObjects subclassing GstElement or any other base-class build on top of that. Also structural components like GstBin and GstPipeline are subclassed from GstElement. GStreamer has many specialized base-classes, covering audio-sinks and -sources, filters and many other use cases.

Elements are normally provided by installed plugins. Plugins can be installed system wide or locally. Besides applications can also register elements directly from their code. This is useful for application specific elements. Buzztard uses this for loading sounds through a memory-buffer sink.

The use of the GObject paradigm provides full introspection of an element's capabilities. This allows applications to use elements in a generic way. New elements can be used without that the application needs special knowledge about them.

1.6 Plugin wrappers

The GStreamer plugin packages provide bridge plugins that integrate other plugin standards. In the audio area, GStreamer applications can use ladspa, lv2 and buzzmachine plugins. The bridge plugins register elements for each of the wrapped plugins. This eases application development, as one does not have to deal with the different APIs on that level any more.

1.7 Input and Outputs

GStreamer supports sources and sinks for various platforms and APIs. For audio on Linux these include Alsa, OSS, OSS4, Jack, PulseAudio, ESound, SDL as well as some esoteric options (e.g. an Apple Airport Express Sink). Likewise there are video sinks for XVideo, SDL, OpenGL and others.

1.8 Audio plugins

Besides the plugin wrappers described in section 1.6, the input/output elements mentioned in 1.7 and generic data flow and tool elements (tee, input/output-selector, queue, adder, audioconvert), GStreamer has a couple of native audio elements already. There is an equalizer, a spectrum analyser, a level meter, some filters and some effects in the gst-plugins-good module. The Buzztard project has a simple monophonic synthesizer (simsyn) and a fluidsynth wrapper in the gst-buzztard module. Especially simsyn is a good starting point for an instrument plugin (audio generator).

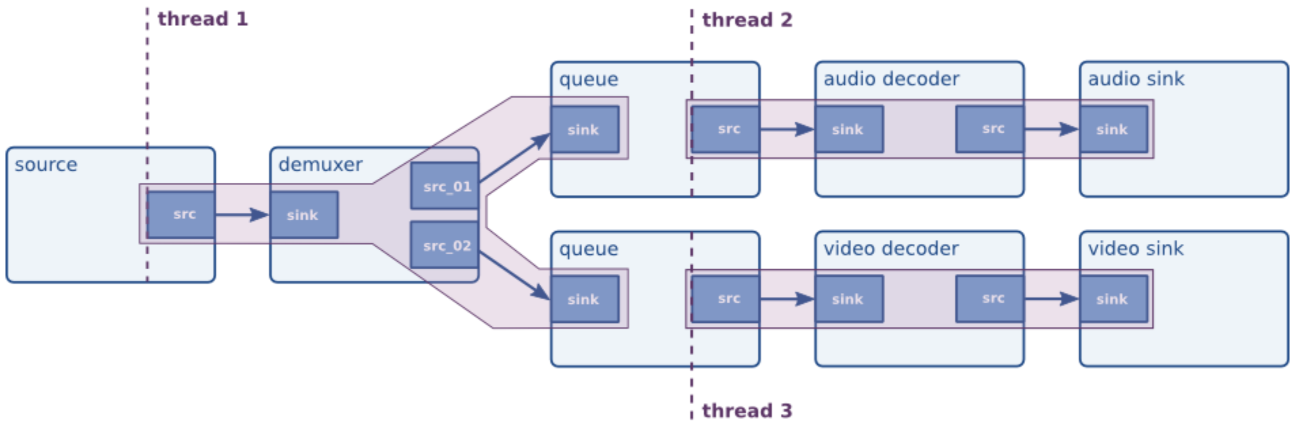

Illustration 2: Thread boundaries in a pipeline

1.9 A/V Sync and Clocks

Time is important in multimedia application. A common timebase is needed for mixing and synchronisation. A GStreamer pipeline contains a clock object. This clock can be provided by an element inside the pipeline (e.g. the audio clock from the audio sink) or a system clock is used as a fall back. The system clock is based on POSIX monotonic timers if available. Elements tag buffers with timestamps. This is the stream-time for that particular media object (see Illustration 3 for the relation of different timebases). Sink elements can now sync received buffers to the clock. If multiple sinks are in use, they all sync against the same clock source.

GStreamer also provides a network clock. This allows to construct pipelines that span multiple computers.

One can use seek-events to configure from which time to play (and until what time or the end). Seek-events are also used to set the playback speed to achieve fast forward or playing media backwards.

1.10 Sequencer

A sequencer records events over time. Usually the sequence is split into multiple tracks too.

GStreamer elements are easy targets for event automation. Each element comes with a number of GObject properties as part of its interface. The properties can be enumerated by the application. One can query data types, ranges and display texts. Changing the value of an element's property has the desired effect almost immediately. Audio elements update the processing parameters at least once per buffer. In video elements a buffers is one frame and thus parameter changes are always immediate.

The GObject properties are a convenient interface for live control. Besides multimedia applications usually also need sequencer capabilities. A Sequencer would control element properties based on time. GStreamer provides such a mechanism with the GstController subsystem. The application can create control-source objects and attach them to element properties. Then the application would program the controller and at runtime the element fetches parameter changes by timestamps.

The control-source functionality comes as a base-class with a few implementations in GStreamer core: an interpolation control source and an lfo control source. The former takes a series of {timestamp, value} pairs and can provide intermediate values by applying various smoothing functions (trigger, step, linear, quadratic and cubic). The latter supports a few waveforms (sine, square, saw, reverse-saw and triangle) plus amplitude, frequency and phase control.

The GObject properties are annotated to help the application to assign control-sources only to meaningful parameters.

[5] has a short example demonstrating the interpolation control source.

The sequencer in Buzztard completely relies on this mechanism. All events recorded in the timeline are available as interpolation control sources on the parameters of the audio generators and effects.

1.11 Preset handling

The GStreamer framework defines several interfaces that can be implemented by elements. One that is useful for music applications is the preset interface. It defines the API for applications to browse and activate element presets. A preset itself is a named parameter set. The interface also defines where those are stored in the file system and how to merge system wide and user local files.

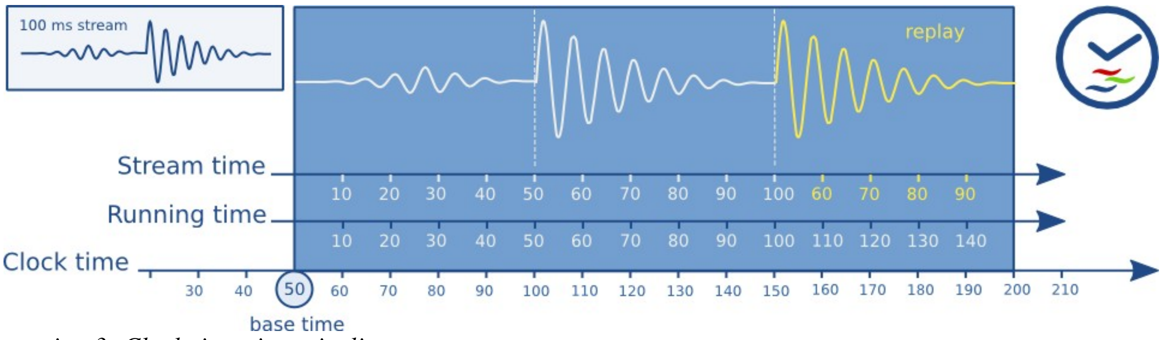

Illustration 3: Clock times in a pipeline

In Buzztard the preset mechanism is used to defines sound preset on sound generators and effects. Another use of presets are encoding profiles for rendering of content so that it is suitable for specific devices.

2 Development support

Previous chapters introduced the major framework features. One nice side effect of a widely used framework is that people write tools to support the development.

The GStreamer core comes with a powerful logging framework. Logs can be filtered to many criteria on the fly or analysed off-line using the gst-debug-viewer tool. Another useful feature is the ability to dump pipeline layouts with all kinds of details as graphviz dot graphs. This is how Illustration 1 was generated.

Small pipelines can by tested with gst-launch on the command-line. The needed elements can be found with gst-inspect or by browsing the installed plugins with its graphical counterpart gst-inspector.

Gst-tracelib can provide statistics and performance monitoring of any GStreamer based application.

3 Performance

One often asked questions is regarding to the framework overhead. This is discussed in the sections below, looking at it from different angles.

3.1 Startup time

When initializing the gstreamer library, it load and verifies the registry cache. The cache is a dictionary of all known plugins and the features they provide. The cache allows applications to lookup elements by features, even before their module is loaded.

The cache is built automatically if it is not present. This spawns a separate process that will load and introspect each plugin. Crashing plugins are blacklisted. Obviously this takes some time, especially if a large amount of plugins is installed.

This of course depends at lot on the amount of installed plugins (in my case: 235 plugins, 1633 features) and the system[2]. As can been seen from the above example, the 2nd round is quick.

[2] Lenovo T60 using a Intel® CPU T2400@ 1.83GHz (dual core)

```
> sync; echo 3 > /proc/sys/vm/drop_caches
> time gst-inspect >/dev/null 2>&1
real    0m2.710s
user    0m0.084s
sys     0m0.060s
> time gst-inspect >/dev/null 2>&1
real    0m0.074s
user    0m0.036s
sys     0m0.040s
```

Example 1: Registry initialisation times

3.2 Pipeline construction

Initial format negotiation can take some time in large pipelines. This gets especially worse if a lot of conversion elements are plugged (they will increase the number of possible formats). One way to address this, is to define a desired default format and plug conversion elements only as needed.

```
> export GST_DEBUG="bt-cmd:3"
>  ./buzztard-cmd 2>&1 -c p -i Aehnatron.bmw |
grep start
0:00:00.038075793 first lifesign from the app
0:00:02.583626430 song has been loaded
0:00:03.388285111 song is playing
```

Example 2: Playback start time for a Buzztard song

Example 2 is the authors poor mans approach to meassure the time to go to playing. The timings already use the optimization suggested in the previous paragraph. The song used in the example uses 44 threads and consists of 341 GStreamer elements structured into 62 GstBins.

3.3 Running time

The actual run time overhead is quite low. Pushing a buffer, will essentially just call the process function on the peer pad. Likewise it happens in pull mode. There is some overhead for sanity checking and renegotiation testing.

```
> time ./buzztard-cmd -c e -i Aehnatron.bmw -o
/tmp/Aehnatron.wav
11:48.480 / 11:48.480

real    0m29.687s
user    0m41.615s
sys     0m4.524s
```

Example 3: Render a Buzztard song

Example 3 show how long it takes to render a quite big song to a pcm wav file. Processing nicely saturates both cpu cores. This benchmark uses the same song as in Example 2.

4 Conclusion

GStreamer is more that just a plugin API. It implements a great detail of the plugin-host functionality. To compare GStreamer with some technologies more known in the Linux Audio ecosystem - one could understand GStreamer as a mix of the functionality that e.g. Jack + lv2 provide together. GStreamer manages the whole processing filter graph, but in one process, while jack would do something similar across multiple processes. GStreamer also describes how plugins are run together, while APIs like ladspa or lv2 leave that freedom to the host applications.

The GStreamer framework comes with a lot of features needed to write a multimedia application. It abstracts handling of various media formats, provides sequencer functionality, integrates with the different media APIs on platforms like Linux, Windows and MacOS. The multithreaded design helps to write applications that make use of modern multicore CPU architectures.

The media agnostic design is ideal for application that like to use other media besides audio as well.

As a downside all the functionality also brings quite some complexity. Pure audio projects still need to deal with some aspects irrelevant to their area. Having support for arbitrary formats makes plugins more complex. Pull based processing remains an area that needs more work, especially if it should scale as well as the push based processing on multicore CPUs.

If there is more interest in writing audio plugins as GStreamer elements, it would also be a good idea to introduce new base-classes. While there is already one for audio filters, there is none yet for audio generators (there is a GstBaseAudioSrc, but that is meant for a sound card source). Thus simsyn introduced in chapter 1.8 is build from 1428 lines of C, while the audiodelay effect in the same package only needs 620.

5 Acknowledgements

Thanks go the the GStreamer community for their great passionate work on the framework and my family for their patience regarding my hobby.

References

[1] GStreamer Project. 1999-2010. *GStreamer: Documentation.* ww.freedesktop.org.

[2] Stefan Kost. 2003-2010. *The Buzztard project*. www.buzztard.org

[3] Stefan Kost. 24 June 2007 . *Fun with GStreamer Audio Effects*. The Gnome Journal.

[4] VAMP Project. 1999-2010. *The Vamp audio analysis plugin system*. http://vamp-plugins.org

[5] GStreamer Project. 1999-2010. *GstController audio example*. http://cgit.freedesktop.org/gstreamer/gstreamer/tree/tests/examples/controller/audio-example.c