GStreamer Training

Topic 1: Installing Gstreamer

Installing Gstreamer on Ubuntu is done by running the following command on the terminal,

```
$apt-get install libgstreamer1.0-dev libgstreamer-plugins-base1.0-dev
libgstreamer-plugins-bad1.0-dev gstreamer1.0-plugins-base gstreamer1.0-
plugins-good gstreamer1.0-plugins-bad gstreamer1.0-plugins-ugly
gstreamer1.0-libav gstreamer1.0-tools gstreamer1.0-x gstreamer1.0-alsa
gstreamer1.0-gl gstreamer1.0-gtk3 gstreamer1.0-qt5 gstreamer1.0-pulseaudio
```

After running the above command, now we have to install the following packages *(here I have installed 1.8.3 version)*:

- gstreamer-1.8.3
- gst-plugins-base-1.8.3
- gst-plugins-good-1.8.3
- gst-plugins-ugly-1.8.3
- gst-plugins-bad-1.8.3
- gst-libav-1.8.3

In order to compile the Gstreamer code which uses Gstreamer core library we have to attach a string i.e ***`pkg-config --cflags --libs gstreamer-1.0`*** during compilation process.

If we have a file named as **gstreamer.c**, to compile the file we use following command:

$gcc gstreamer.c `pkg-config --cflags --libs gstreamer-1.0`

Topic 2: Generic Concepts and Foundation

Gstreamer is a frame work which came into picture for creating streaming media applications. Gstremer is mostly used to build a media player. This frame work is based on plugins which provide various codec and other functionality. It also provides API's to write applications using the various plugins.

Gstreamer is packaged into:

- gstreamer : the core package.
- gst-plugins-base : an essential exemplary set of elements.
- gst-plugins-good : a set of good quality plugins under LGPL.
- gst-plugins-ugly : a set of good quality plugins that might pose distribution problems.
- gst-plugins-bad : a set of plugins that need more quality.
- gst-libav : a set of plugins that wrap libav for decoding and encoding.

and few other packages.

Gstreamer have some basic design principles nothing but clean and powerful, object oriented, extensible, allow binary-only plugins, high performance, clean core/plugins seperation and it also provide a frame work for codec experimentation.

FOUNDATIONS:

The main four foundations of Gstreamer are elements, pads, bins & pipelines and communication.

1. **Elements**: These are the most important class of objects in Gstreamer. We can chain many elements together to make a pipeline which helps in performing some specific tasks.
2. **Pads**: Pads are nothing but elements input and output where we can connect other elements. These pads help in the data flow between elements.
3. **Bins and Pipelines**: Bin is a container for a collection of elements. Whereas pipeline is a toplevel bin. This pipeline also provides a bus for the application and manages the synchronization.
4. **Communication**: We have buffers, events, messages and queries for the communication.

Topic 3: Gstreamer application development (Basics)

Initializing Gstreamer:

To run a Gstreamer code we have to add **#include<gst/gst.h>** header file in our code, as this header file gives us access to run a Gstreamer code. Before the Gstreamer libraries can be used, **gst_init()** function has to be called from the main application.

Whenever we use the gst_init() function, this function initialises all internal structures, checks what plugins are available and also executes any command-line option intended for Gstreamer.

gst_init() function have two arguments. We can declare it as **gst_init(&argc, &argv)**. We can also keep NULL in the place of &argc and &argv.

Elements:

Elements are the main blocks in the Gstreamer. The data type for these elements are **GstElement**. If we consider these elements as boxes, input is send from one side, particular type of element performs particular type of operation and provides an output from other side.

We have many types of elements which are used for specfic tasks. They are: source elements, sink elements, filters, converters, demuxers, muxers and codecs etc.

To create an GstElement we use the function known as **gst_element_factory_make()** and to dereference the element we have created we use the function known as **gst_object_unrefer()**. Instead of gst_element_factory_make() function we can use other two functions to create an element. Those functions are **gst_element_factory_find()** to find the factory name and **gst_element_factory_create()** to create the element.

Linking elements:

By linking the source element, filter like elements and sink element at the last we set up a basic media pipeline.

Element States:

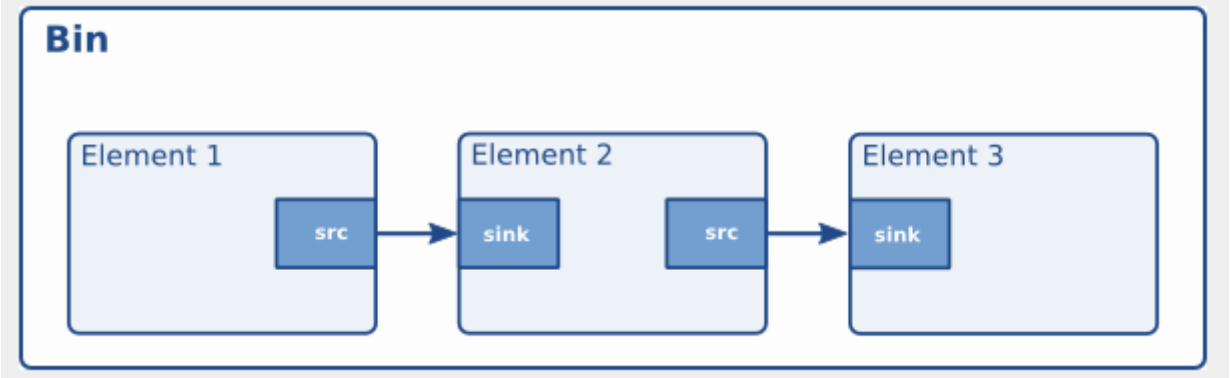
There are four element states. To make element perform some specific actions we use these element states. We can set these element states by using the **gst_element_set_state()** function.

Below are the element states:

1. **GST_STATE_NULL**
2. **GST_STATE_READY**
3. **GST_STATE_PAUSED**
4. **GST_STATE_PLAYING**

Bins:

Bin is a container element. We can say that bin allow you to combine a group of linked elements into one logical element. This bin performs state changes on the elements and also collect and forward bus messages.



To create a bin we can use **gst_element_factory_make()**. There are also some other functions through which we can create bins. They are: **gst_bin_new()** and **gst_pipeline_new()**. To add elements to bin and also to remove elements from bin we use **gst_bin_add()** and **gst_bin_remove()** functions.

Bus:

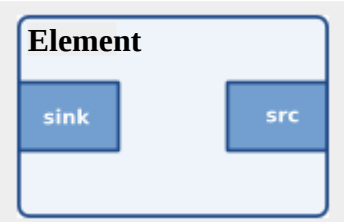
Bus is used for forwarding messages from the streaming threads to an application. Every pipeline have a bus internally. To know the bus which is attached to pipeline we use **gst_pipeline_get_bus()**. The data type to initialize the bus is **GstBus**.

As we know that every pipeline contains bus, there is no need to create the bus. The only thing we must do is to initialize the message_handler function. When the

mainloop is running, this bus check for the new messages and the callback will be called when any message is available. To call the message_handler we use bus watch id i.e, **gst_bus_add_watch()** where bus and message_handler functions are input to it.

Pads and Capabilities:

Pads are nothing but element interface. Gstreamer defines two pad directions: source pads and sink pads. Elements receive data on sinks pad and generate data on their source pads.



To link the pad of one element with the pad of other element we use **gst_element_link_pads()** function. This function is used to link the source pad of one Gstreamer element with the sink pad of another Gstreamer element.

Dynamic pads:

Dynamic pads are also known as sometimes pads. Some elements may not have all of their pads when created. To overcome this dynamic pads came into picture. After the usage of the pad which is dynamically created, it gets deleted automatically.

Request pads:

Some pads are not created automatically, they are created on request, such pads are known as request pads. We can request a pad by **gst_element_request_pad_simple()** and also **gst_element_get_compatible_pad()**.

Capabilities:

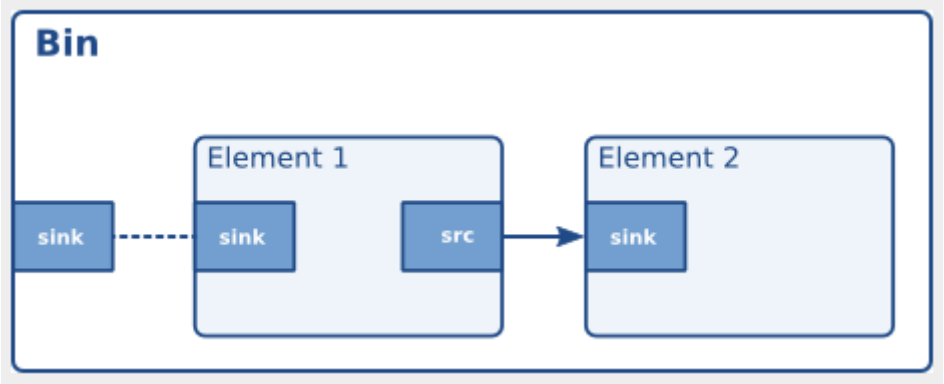
A mechanism is implemented to describe the data that can flow or currently flows through the pad by using capabilities. These capabilities are attached to pads *(it can either be a list of possible caps, or it is thetypr of media that currently streams over the pad)* or pad templates *(it will describe the types of media that may stream over a pad created from this template)*.

Capabilities also know as “caps” in short describes the type of data that is streamed between two pads. Capabilities are used in autoplugging, compatibility detection, metadata and filtering. By using the function **gst_caps_get_structure()** we can get a structure from caps. And also to get the number of structures in a **GstCaps**

using **gst_caps_get_size()**. To create your own GstCaps we have to use the function **gst_caps_new_simple()**.

Ghost pad:

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



Buffers and Events:

The data which we are sending through pipelines consists of buffers and events. Buffer is nothing but the media data whereas event contains control information.

Topic 4: Gstreamer application development (Advanced)

Seeking:

Seeking is nothing but configuring the pipeline for playback of the media between a certain start and stop time, called as playback segment. Playback describes media files containg audio or video stored on the computer that can be played at any time.

A seek is performed by sending a **SEEK** event to the sink element. There are some flags provided by the Gstreamer. They are:

- **GST_SEEK_FLAG_FLUSH** flag -> All pending data in the pipeline is discovered and playback starts from the new position immediately.
- **GST_SEEK_FLAG_KEY_UNIT** flag -> Can be performed to a nearby key unit or to the exact unit in the media (estimated).
- **GST_SEEK_FLAG_ACCURATE** flag -> Performed by using an estimated target position or in an accurate way.
- **GST_SEEK_FLAG_INSTANT_RATE_CHANGE** flag -> It is possible to control the playback rate instantly by sending a seek with the mentioned flag.

Seek event is created with **gst_event_new_seek()**.

Metadata:

In simple **metadata** is nothing but **“data about data”**. There are two types of metadata in Gstreamer, they are nothing but Stream tags and Stream-info.

- **Stream tags**: These stream tags describe the content of a stream in a non-technical way.
- **Stream-Info**: This is somewhat the technical description of the properties of the stream.

Interfaces:

Gstreamer uses interfaces based on the **GObject GtypeInterface** type. Following are the interfaces which are provided under Gstreamer:

- **GstURIHandler -** To know URI specifics
- **GstColorBalance -** To control video related properties on an element, such as brighteness, contrast and so on

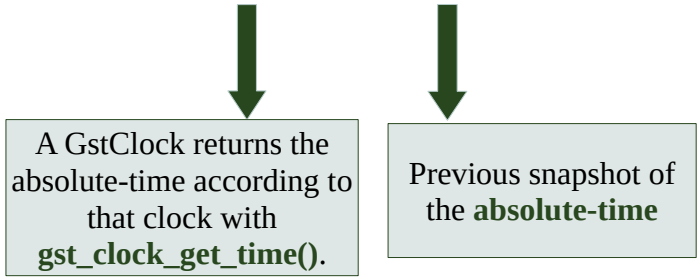
- **GstVideoOverlay -** To solve the problem of embedding video streams in an application window
- **GstChildProxy -** For access to internal element's properties on multi-child elements
- **GstNavigation -** For the sending and parsing of navigation events
- **GstPreset -** For handling element presets
- **GstRTSPExtension -** An RTSP Extension interface
- **GstStream -** Volume Interface to provide access and control stream volume levels
- **GstTagSetter -** For handling media metadata
- **GstTagXmpWriter -** For elements that provide XMP serialization
- **GstTocSetter -** For setting and retrieval of TOC-like data
- **GstTuner -** For elements providing RF tunning operations
- **GstVideoDirection -** For video rotation and flipping controls
- **GstVideoOrientation -** For video orientation controls
- **GstWaylandVideo -** Wayland video interface

Clocks and Synchronization:

Gstreamer provides **synchronization** mechanism for a specific purpose i.e., playing a video or audio in a specific order at a specific time. To synchronize streams in a pipeline Gstreamer uses a **GstClock** object, buffer timestamps and a **SEGMENT** event.

Clock running-time: **running-time = absolute-time – base-time**

A Gstreamer **GstPipeline** object maintains a **GstClock** object and a **base-time** when it goes to the **PLAYING** state.



Buffer running-time:

We need a buffer timestamp and the **SEGMENT** to calculate the buffer running-time. First we have to convert the **SEGMENT** event into a **GstSegment** object and then we can use **gst_segment_to_running_time()** function to calculate the buffer running-time.

Similarly we also have **buffer stream-time,** inshort it is explained as the position in the stream.

Buffering:

To accumulate enough data in the pipeline so that playback can occur smoothly without interruptions, **buffering** came into picture.

Gstreamer provides the buffering mainly in three cases. Those are: Stream buffering, Download buffering and Timeshift buffering.

- **Stream buffering**: In this the buffering takes place up to a specific amount of data, in memory, before starting playback. This helps in minimizing the network fluctuations.
- **Download buffering**: In this type, the network file is downloaded to a local disk with fast seeking in the downloaded data.
- **Timeshift buffering**: In this mode, a fixed size of ringbuffer is kept to download the server content.

Dynamic Controllable Parameters:

Controller subsystem offers a lightweight way to adjust GObject properties over stream-time. This controller works by attaching **GstControlSource**s to properties using **control-bindings**. The base classes for **GstControlSource**s and control-bindings are includes in **gstcontroller** library. Thus you must include the following header files:

```
#include <gst/gst.h>
#include <gst/controller/gstinterpolationcontrolsource.h>
#include <gst/controller/gstdirectcontrolbinding.h>
...
```

To compile the application which is linked to the gstreamer-controller shared library we must link **`pkg-config --libs --cflags gstreamer-controller-1.0`**.

Setting up parameter control:

If we had a pipeline setup and want to control some parameters, we first need to create **GstControlSource** using **gst_interpolation_control_source_new()**. After creating we must attach **GstControlSource** to the **gobject** property.

```
csource = gst_interpolation_control_source_new ();
g_object_set (csource, "mode", GST_INTERPOLATION_MODE_LINEAR, NULL);
```

Then we have to add new property values.

```
gst_object_add_control_binding (object, gst_direct_control_binding_new (object, "prop1", csource));
```

We can also set some control points.

```
GstTimedValueControlSource *tv_csource = (GstTimedValueControlSource *)csource;
gst_timed_value_control_source_set (tv_csource, 0 * GST_SECOND, 0.0);
gst_timed_value_control_source_set (tv_csource, 1 * GST_SECOND, 1.0);
```

Now everything is ready to play. It works based on the property we use.

Threads:

Gstreamer is a multi-threaded. It notifies when threads are created so that we can configure things such as the thread priority or threadpool to use.

Streaming threads are also known as **GstTask** objects. These threads are created from a **GstTaskPool**. To know the status of the streaming threads **STREAM_STATUS** message is posted on the bus. Some situations where threads can be useful are during buffering, sunchronizing, etc.

Queue:

Queue is a thread boundary element. Through queues we can force the use of threads. These queues have several **GObject** properties to be configured for special uses.

Autoplugging:

The process of building an application that can automatically detect the media type of a stream and automatically generate the best possible pipeline by looking at all available elements in a system is known as Autoplugging.

For this autoplugging process we have some autopluggers. They are: Playbin, Decodebin, URIDecodebin and Playsink *(these four are media playback autopluggers)* etc.

Pipeline Manipulation:

Using probes:

Probe notifies about the activity going on, in a pipeline. It is nothing but a callback function that can be attached to a pad by using **gst_pad_add_probe()** function, and can also be removed by using **gst_pad_remove_probe()** function. We can also define what kind of notifications we are interested in when we add probe. Some of the probe types are,

- **GST_PAD_PROBE_TYPE_BUFFER**
- **GST_PAD_PROBE_TYPE_PUSH**
- **GST_PAD_PROBE_TYPE_PULL**
- **GST_PAD_PROBE_TYPE_BUFFER_LIST**
- **GST_PAD_PROBE_TYPE_EVENT_DOWNSTREAM**
- **GST_PAD_PROBE_TYPE_EVENT_UPSTREAM**
- **GST_PAD_PROBE_TYPE_EVENT_BOTH**
- **GST_PAD_PROBE_TYPE_BLOCK** etc.

Data probes:

Data probes notify you when there is a flow of data in the pipeline. To create this kind of data probe, we have to pass GST_PAD_PROBE_TYPE_BUFFER or GST_PAD_PROBE_TYPE_BUFFER_LIST to the function **gst_pad_add_probe()**.

appsrc and appsink:

appsrc is an imaginary source and **appsink** is an imaginary sink. We can insert and grab the data by using this **appsrc** and **appsink**.

appsrc can be operated in both **push** or **pull** modes. The main way of handling data to **appsrc** is by using the **gst_app_src_push_buffer()** function. This will put the buffer onto a queue from which **appsrc** will read in its streaming thread.

Similarly **appsink** also supports both pull and push based modes for getting data from the pipeline. To retrieve data from **appsink** we use **gst_app_sink_pull_sample()** and **gst_app_sink_pull_preroll()** functions.

In this **pipeline manipulation** we can also *change format, change pipeline dynamically and also change elements in a pipeline.*

Playback Components:

We have four playback components which are targetted at media playback. They are: Playbin, Decodebin, URIDecodebin and Playsink.

1. **Playbin**: Playbin is an element which can be created by using the function **gst_element_factory_make()** by using **“playbin”** factory. This playbin supports all of the features of the class, including error handling, state handling, seeking and so on.
2. **Decodebin**: This decodebin is the actual autoplugger backend of playbin. This decodebin automatically detect decoders. In short we can say that it accepts input from the source that is linked to its sinkpad and will try to detect the media type contained in the stream, and set up decoder routines for each these. This decodebin emit **“pad-added”** signal for each decoded stream and **“unknown-type”** signal for unknown streams.
3. **URIDecodebin**: This URIDecodebin is similar to the decodebin. The difference is that it automatically plugs a source plugin based on the protocol of the URI given.
4. **Playsink**: Playsink has request pads for raw decoded audio, video and text and it will configure itself to play the media streames. It is a powerful sink element.