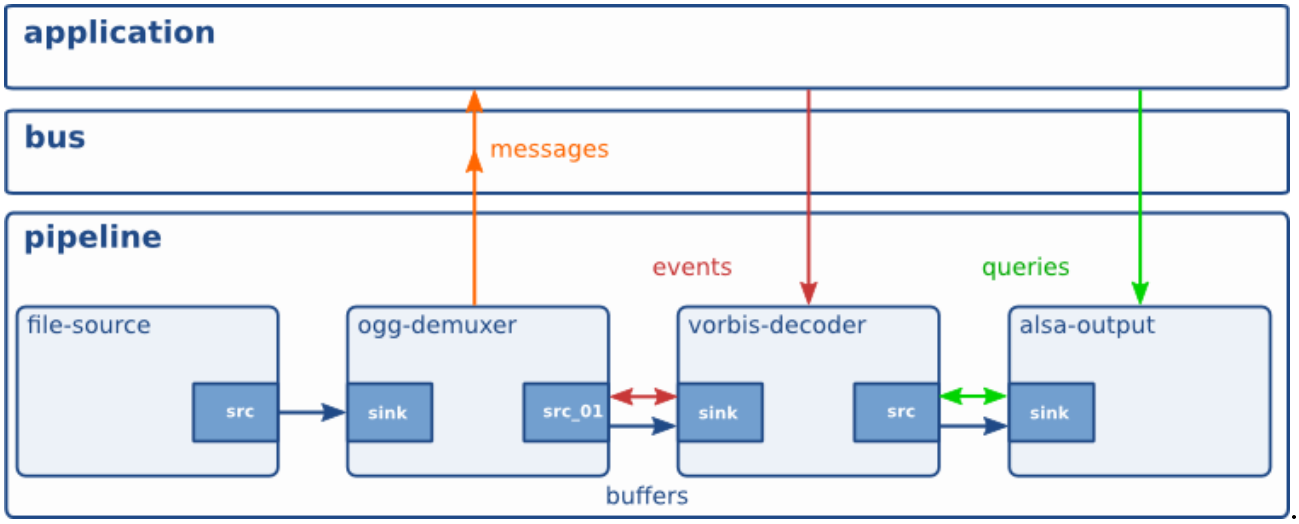
M1 OIM TP5

1. PROGRAMMATION GSTREAMER

Nous allons maintenant réaliser des traitements en écrivant un programme en C.

Communications.



Gstreamer met en place plusieurs méchanismes de communication et d'échange de données entre les application et le tuyau d'execution (« pipeline ») :

- **buffers** (mémoires tampon) : sont des objets pour passer des données en flux entre différents éléments du pipeline. Les buffers travaillent toujours des sources (src) vers les sorties (sink).
- **events** (événements) : ce sont eds objets qui sont envoyés entre les éléments ou depuis les applications vers les éléments. Les événements peuvent voyager du haut vers le bas (upstream) et du bas vers le haut (downstream). Les événements downstream peuvent être synchronisés au flot de données.
- **messages** : ce sont des objets postés par des éléments sur le bus de messages, où ils seront pris en charge par l'application. Les messages peuvent être interceptées simultanément par le contexte du processus gérant le flux de l'élément qui poste le message, mais sont en général utilisés de manière asynchrone pa rle processus principal de l'application. Les messages servent à transmettre de l'information comme les erreurs, les changements d'état, l'état du buffer, les redirections...
- **queries** (demandes) permettent aux applications de demander de l'information comme la durée ou la position courante du playback dans le pipeline. Les demandes sont toujours traitées de manière synchrone. Des élements peuvent également utiliser des demandes pour récupérer de l'information (comme la taille ou la durée). Elles peuvent être utilisées dans les deux sens à l'interieur du piipeline, mais en général on retrouve des demandes du haut vers le bas.

Etat des éléments. Après avoir été crée, un élément ne fera aucune action. Vous devez changer l'état de l'élément pour qu'il fasse quelquechose. GStreamer connait quatre états différents, chacun ayant un sens très précis. Les quatre états sont :

- GST_STATE_NULL : état par défaut. Aucune ressource n'est allouée dans cet état, donc si vous arrivez dans cet état celà va libérer toutes les ressources. L'élément doit être dans cet état quand son compteur de référence (refcount) atteint 0 et qu'il doit être libéré.

- GST_STATE_READY : dans l'état disponible, un élément a alloué toutes les ressources nécessaires pour gérer les flux. cela concerne les ouvertures de périphériques, l'allocation des mémoires tampons, etc. Cependant, le flux n'est pas ouvert à ce stade, donc la position du stream est automatiquement zéro. Si un flux avait précédament été ouvert et positionné, les propriétés sont remises à zéro.
- GST_STATE_PAUSED : dans cet état, un élément a ouvert un flux, mais il n'est pas encore en train de le traiter activement. L'élément est autorisé à modifier la position du flux, lire et traiter des données et donc se préparer à à effectuer la lecture dès que son état sera changé en PLAYING, mais il n'est pas autorisé à lire le média, ce qui aurait pour conséquence de faire tourner l'horloge. Pour résumer, PAUSED est la même chose que PLAYING mais sans que l'horloge tourne. Les éléments qui vont dans l'état PAUSE doivent se préparer à repasser dans l'état PLAYING aussi rapidement que possible. Les sorties audio ou vidéos, par exemple, attendent que les données arrivent et les mettent dans la queue afin de pouvoir les lire juste après le changement d'état. Les sorties vidéos peuvent déja lire la première trame (puisque ca n'affecte pas l'horloge). Les connecteurs automatiques (autoplugger) peuvent également utiliser cet état transitoire pour connecter le pipeline. La plupart des autres éléments, comme les codec ou les filtres, ne doivent pas explicitement réaliser quelque chose dans cet état.
- GST_STATE_PLAYING : dans l'état PLAYING,l'élement fait exactement la même chose que dans l'état PAUSED, excepté que l'horloge tourne maintenant.

Vous pouvez changer l'état d'un élément à l'aide de la fonction `gst_element_set_state ()`. Si vous changez l'état d'un élément, GStreamer va traverser en interne tous les états intermédiaires. Donc si vous passer l'état de NULL à PLAYING, GStreamer va en interne passer par les états READY et PAUSED.

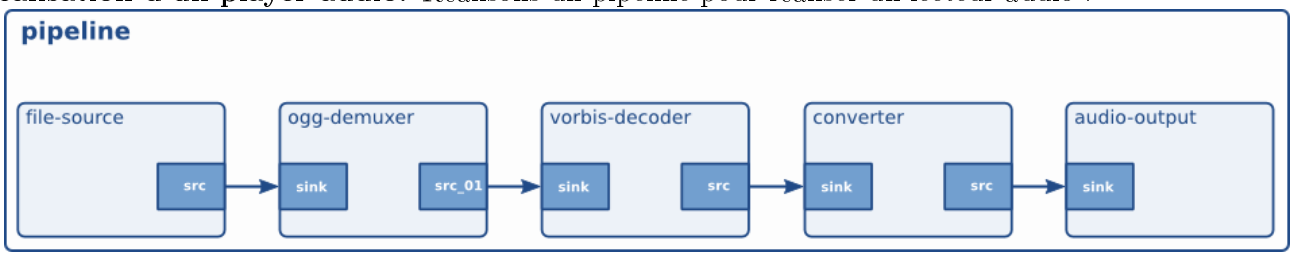
Quand l'état arrive à GST_STATE_PLAYING, les pipelines vont traiter des données automatiquement. Ils n'ont pas besoin dêtre itérés. En interne, GStreamer va démarrer les flux nécessaires au processus. GStreamer va aussi s'occuper de faire passer les messages du processus du pipeline vers le processus de l'application, en utilisant `GstBus`.

Quand vous passez un bin ou un pipeline à un certain état, le changement d'état sera propagé automatiquement à tous les éléments à l'intérieur du bin ou du pipeline, donc il est en général seulement nécessaire de définir l'état du pipeline le plus haut poru démarrer le pipeline ou l'éteindre. Cependant, quand vous rajoutez des éléments dynamiquement à un pipeline qui est déjà en train de tourner, par exemple avec un signal "pad-added" ou "new-decoded-pad", vous devez le configurer au bon état en utilisant `gst_element_set_state ()` ou `gst_element_sync_state_with_parent ()`.

Initialisation. Pour avoir accès aux fonctions GStreamer, il suffit d'inclure `gst/gst.h`. Mais vous devez aussi initialiser la bibliothèque avec un appel à `gst_init`.

```
#include <stdio.h>
#include <gst/gst.h>
int main (int   argc,
          char *argv[])
{
  const gchar *nano_str;
  guint major, minor, micro, nano;
  gst_init (&argc, &argv);
  gst_version (&major, &minor, &micro, &nano);
  if (nano == 1)
    nano_str = "(CVS)";
  else if (nano == 2)
    nano_str = "(Prerelease)";
  else
    nano_str = "";
  printf ("Ce programme est lié à GStreamer %d.%d.%d %s\n",
          major, minor, micro, nano_str);
  return 0;
}
```

Réalisation d'un player audio. Réalisons un pipeline pour réaliser un lecteur audio :



Code correspondant (disponible dans le fichier suivant : `lecteuraudio.c`) :

```
#include <gst/gst.h>
#include <glib.h>
static gboolean bus_call (GstBus     *bus,
                          GstMessage *msg,
                          gpointer    data)
{
  GMainLoop *loop = (GMainLoop *) data;

  switch (GST_MESSAGE_TYPE (msg)) {

   case GST_MESSAGE_EOS:
      g_print ("End of stream\n");
      g_main_loop_quit (loop);
      break;
    case GST_MESSAGE_ERROR: {
      gchar  *debug;
      GError *error;

      gst_message_parse_error (msg, &error, &debug);
      g_free (debug);

      g_printerr ("Error: %s\n", error->message);
      g_error_free (error);

      g_main_loop_quit (loop);
      break;
    }
    default:
      break;
  }
  return TRUE;
}

static void
on_pad_added (GstElement *element,
              GstPad     *pad,
              gpointer    data)
{
  GstPad *sinkpad;
  GstElement *decoder = (GstElement *) data;

  /* We can now link this pad with the vorbis-decoder sink pad */
  g_print ("Dynamic pad created, linking demuxer/decoder\n");
```

```
  sinkpad = gst_element_get_static_pad (decoder, "sink");

  gst_pad_link (pad, sinkpad);
  gst_object_unref (sinkpad);
}

int main (int   argc,
          char *argv[])
{
  GMainLoop *loop;

  GstElement *pipeline, *source, *demuxer, *decoder, *conv, *sink;
  GstBus *bus;

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

 loop = g_main_loop_new (NULL, FALSE);

  /* Verification des arguments d'entrée */
  if (argc != 2) {
    g_printerr ("Usage: %s <Ogg/Vorbis filename>\n", argv[0]);
    return -1;
  }

  /* Create gstreamer elements */
  pipeline = gst_pipeline_new ("audio-player");
  source   = gst_element_factory_make ("filesrc",       "file-source");
  demuxer  = gst_element_factory_make ("oggdemux",      "ogg-demuxer");
  decoder  = gst_element_factory_make ("vorbisdec",     "vorbis-decoder");
  conv     = gst_element_factory_make ("audioconvert",  "converter");
  sink     = gst_element_factory_make ("autoaudiosink", "audio-output");

  if (!pipeline || !source || !demuxer || !decoder || !conv || !sink) {
    g_printerr ("One element could not be created. Exiting.\n");
    return -1;
  }

  /* Mise en place du pipeline */
  /* on configurer le nom du fichier à l'élément source */
  g_object_set (G_OBJECT (source), "location", argv[1], NULL);

  /* on rajoute une gestion de messages */
  bus = gst_pipeline_get_bus (GST_PIPELINE (pipeline));
  gst_bus_add_watch (bus, bus_call, loop);
  gst_object_unref (bus);

  /* on rajoute tous les éléments dans le pipeline */
  /* file-source | ogg-demuxer | vorbis-decoder | converter | alsa-output */
  gst_bin_add_many (GST_BIN (pipeline),
                    source, demuxer, decoder, conv, sink, NULL);
```

```
  /* On relie les éléments entre eux */
  /* file-source -> ogg-demuxer ~> vorbis-decoder -> converter -> alsa-output */
  gst_element_link (source, demuxer);
  gst_element_link_many (decoder, conv, sink, NULL);
  g_signal_connect (demuxer, "pad-added", G_CALLBACK (on_pad_added), decoder);

  /* Notez que le demuxer va être lié au décodeur dynamiquement.
     la raison est que Ogg peut contenir plusieurs flux (par exemple
     audio et vidéo). Les connecteurs sources seront crées quand la
     lecture débutera, par le demuxer quand il détectera le nombre et
     la nature des flux. Donc nous connectons une fonction de rappel
     qui sera exécuté quand le "pad-added" sera émis. */

  /* passage à l'état "playing" du pipeline */
  g_print ("Lecture de : %s\n", argv[1]);
  gst_element_set_state (pipeline, GST_STATE_PLAYING);

  /* Iteration */
  g_print ("En cours...\n");
  g_main_loop_run (loop);

  /* En dehors de la boucle principale, on nettoie proprement */
  g_print ("Arret de la lecture\n");
  gst_element_set_state (pipeline, GST_STATE_NULL);
  g_print ("Suppression du pipeline\n");
  gst_object_unref (GST_OBJECT (pipeline));
  return 0;
}
```

Compilation. Pour compiler l'exemple ci-dessus, il faut utiliser gcc :
`gcc -Wall lecteuraudio.c -o lecteuraudio $(pkg-config --cflags --libs gstreamer-0.10)`

GStreamer utilise pkg-config pour récupérer les paramètres du compilateur et du linker qui sont nécessaires pour compiler cette application.

$(...) : permet d'executer une commande dans un sous-shell. C'est une commande bash. Si vous êtes sous csh, utilisez plutôt les quotes inverses : `` `pkg-config --cflags --libs gstreamer-0.10` ``

Si vous utilisez une installation non standard (en ayant par exemple compilé GStreamer à partir des sources au lieu d'utiliser les paquets pré-compilés), vérifiez que la variable d'environnement `PKG_CONFIG_PATH` est définie et pointe vers la bonne direction (`$libdir/pkgconfig`).

Dans le cas non souhaitable où vous utilisez une configuration où GStreamer est non installé (ie. `gst-uninstalled`), vous allez avoir besoin d'utiliser libtool pour construire ce programme, par exemple comme ceci : `libtool --mode=link gcc -Wall lecteuraudio.c -o lecteuraudio $(pkg-config --cflags --libs gstreamer-0.10)`.

Vous pouvez compiler cet exemple d'application de la façon suivante : `./lecteuraudio fichier.ogg`. Substituez `fichier.ogg` par votre fichier Ogg/Vorbis favori.

Exercices.

(1) Détaillez le programme lectureaudio et comprenez toutes les étapes.

(2) Codez en C une chaîne de traitement audio-vidéo : par exemple, lecture de trailer_400p.ogg, passage en noir et blanc, baisse du volume de 50%.

(3) Réalisez un transcodeur audio-vidéo. utilisez des « bin » pour rendre le décodage universel.