Tecnológico de Costa Rica

Sede Cartago

Escuela Ingeniería en Computación

Sistemas Operativos

Profesor:
Esteban Arias Méndez

Estudiantes:
Jose Enrique Alvarado Chaves
Ariel Herrera Fernández

2018

Abstract:

On this document It will give description and explanation about the project JukebOS, desitions made it by the team, solutions, problems and the state of the project. Also it shows the architecture develop and the structure of the main server, developed mainly on c language.

JukebOS

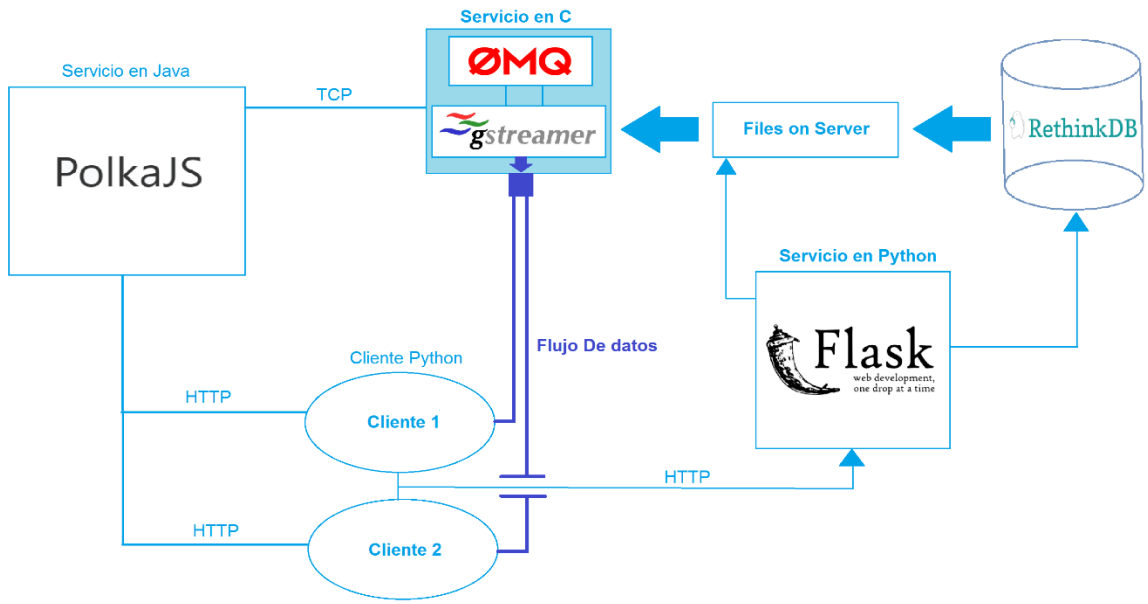
El siguiente proyecto es un sistema que permitirá a usuarios poder ingresar, agregar y reproducir música por medio de flujo de datos. La arquitectura se basa en un sistema de microservicios altmente escalable, desarrollado en diferentes lenguajes:

- El core principal en C
- Javascript
- Python.

Las principales herramientas utilizadas fueron:

- Gstreamer
- Zeromq
- PolkaJS
- FLask

Para empezar, es necesario comprender de qué manera está estructurado el proyecto, por lo que se mostrará una pequeña descripción de la arquitectura desarrollada.



Servicio en C:

- Para el servicio en C, se determinó principalmente el uso de la herramienta e zeromq para l manipulación y comunicación de las diferentes conexiones se van a encontrar activas en el servicio y la herramienta de GStreamer, que ayuda crear aplicaciones de transmisión de media, permitiendo formatos mp3, Ogg-Vorbis, MPEG-1/2, AVI, entre otros. Además, se pueden editar pipelines y guardarse como XML, por lo que librerías de pipeline puede ser creadas con un mínimo de esfuerzo.
    - El equipo quería probar nuevas tecnologías como lo es Zeromq (Reconocido por establecer cero de latencia en servicios), el cual es una herramienta utilizada en el colisionador de hadrones del CERN.

- Servicio en javascript: se estableció el servicio en c, como un intermediario entre el sistema que provee el flujo de datos y la aplicación cliente. Esto con el objetivo de administrar las múltiples peticiones que va a tener el sistema por parte del usuario.

- Servicio de Python: Permite una administración correcta y ordenada de manejar todas las canciones que se desean subir al sistema, aprovechando de paso para utilizar una base de datos documental, que permite almacenar, el usuario las canciones y la dirección respectiva de cada una de ellas.
    - Se escogió la base de datos, NoSQL, RethinkDB, por su alta escalabilidad y por su uso conocido en algunos juegos, para la administración de media.

Explicación del servicio principal:

En esta sección se va a explicar cada una de las principales partes del servicio en c.

Primero se establece una declaración del servicio que se quedara escuchando peticiones por parte del servicio en JavaScript.

```
void *context = zmq_ctx_new ();
    void *responder = zmq_socket (context, ZMQ_REP);
    int rc = zmq_bind (responder, "tcp://*:5500");
    assert (rc == 0);
```

Se establece un tamaño para los responses por parte del servicio Polka, y un tamaño especifico al buffer que se comunica con los clientes.

```
int portId=6500;
    char * response = malloc(40*sizeof(char));
    memset(response, 32, 40);

    printf ("Service Dispatcher listening....\n");
    char * buffer = malloc(25*sizeof(char));
    memset(buffer, 0, 25);
```

Luego, una de las partes más importantes es la de generar procesos hijos por cada petición de dispatch que se genere.

- Una vez llega una petición zeromq, se crea un fork con los datos recibidos por la petición, envía al hijo a su respectivo flujo de ejecución, mientras que el padre se encarga de responder como es debido a la petición del servicio de javascrip

```
while (1){
        zmq_recv (responder, buffer, 25, 0);
        sprintf(response, "{\"zmq_port\":%d,\"gst_port\":%d}",
portId,portId+1);
        pid = fork();
        if(pid==0){break;}//Pasa a ejecutar el hijo
        zmq_send (responder, response, 40, 0);
        portId=portId+2;
}
```

Una vez el hijo se crea, se empieza a establecer el nuevo proceso que se encargara de satisfacer la petición de música el cliente, estableciendo un flujo de datos que se puede consumir.

```
      void *contextChild = zmq_ctx_new ();
```

```
      void *responderChild = zmq_socket (contextChild, ZMQ_REP);
      char host [15];
      sprintf(host, "tcp://*:%d", portId);
      printf("%s\n", host);
      int rcChild = zmq_bind (responderChild, host);
      assert (rcChild == 0);
```

En la siguiente imagen se muestra la configuración del pipeline

```
pipeline = gst_parse_launch ("filesrc name=my_filesrc ! mpegaudioparse !
mpegtsmux ! rtpmp2tpay ! queue max-size-bytes=0 max-size-buffers=0 max-size-
time=0 leaky=downstream ! tcpserversink name=my_portsrc host=127.0.0.1",
&error);
      if (!pipeline) {
        g_print ("Parse error: %s\n", error->message);
        exit (1);
      }
```

Posteriormente se establece la ruta del path del archivo, al cual se le provee al flujo de datos para que sea consumido.

```
component = gst_bin_get_by_name (GST_BIN (pipeline), "my_portsrc");
      g_object_set (component, "port", portId+1, NULL);
      g_object_unref (component);
```

Además, se le establece la ruta en la que se encuentra:

```
component = gst_bin_get_by_name (GST_BIN (pipeline), "my_filesrc");
      g_object_set (component, "location", location, NULL);
      g_object_unref (component);
```

Una vez establecidos estos elementos, es posible establecer el estado de la reproducción de la siguiente manera:

```
gst_element_set_state (pipeline, GST_STATE_PAUSED);
```

Para el siguiente código se establece un siclo en el que se mantiene esperando peticiones del Usuario para determinar si se establece alguno de los tres posibles estados:

- Play: Establece el estado en GST_STATE_PLAYING
- Pause: Establece el estado en GST_STATE_PAUSE
- Stop: Detiene el proceso.

La intención del equipo es establecer una conexión por cada cliente que desee reproducir una canción, de esta manera solamente cuando se este escuchando una canción por parte del cliente, se van a mantener los procesos

```
while(1){
        printf("Waiting a instruction:\n");
        char i_buffered [1];
        zmq_recv (responderChild, i_buffered, 1, 0);
        int i = atoi(i_buffered);
        printf( "You entered: %d\n", i);
        if (i == 0){
          gst_element_set_state (pipeline, GST_STATE_PLAYING);
          zmq_send (responderChild, "playing", 7, 0);
        }
        else if (i == 1){
          gst_element_set_state (pipeline, GST_STATE_PAUSED);
          zmq_send (responderChild, "Paused", 6, 0);
        }
        else if (i == 2){
          zmq_send (responderChild, "Exiting", 7, 0);
          gst_element_set_state (pipeline, GST_STATE_NULL);
          gst_object_unref (pipeline);
          gst_object_unref (bus);
          kill(getpid(), SIGKILL);
        }
        else{
          printf("\nActions: \n  0. Play \n  1. Pause\n  2. Exit\n");
          zmq_send (responderChild, "Showing menu", 12, 0);
        }
      }
```

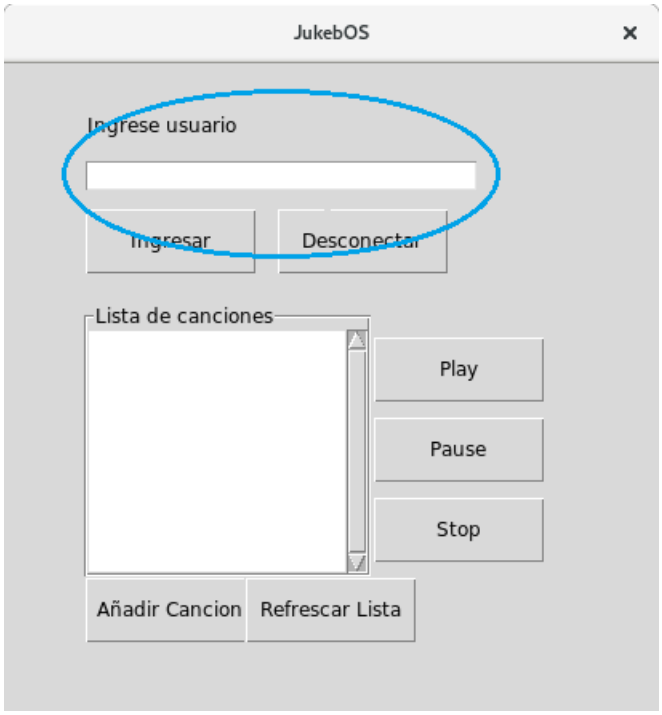
Siendo de esta manera en la que se establece cada proceso hijo.

Por otra parte, el padre se va a quedar ejecutando hasta que el desarrollador detenga el servicio en c. Así finaliza el manejo / trabajo del dispatcher y los multiprocesos.

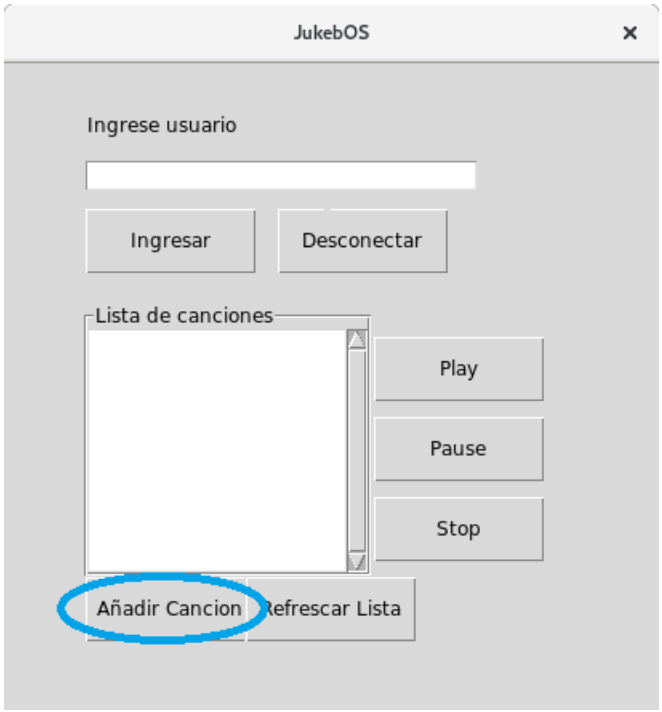
Guía para uso del cliente del sistema:

En las siguientes imágenes se muestra la ventana del cliente, con la cual podrá ejecutar las distintas funciones que el sistema JukebOS ofrece.

1) Ingresar al sistema: Es necesario establecer un nombre de usuario para que el Sistema establezca una solución

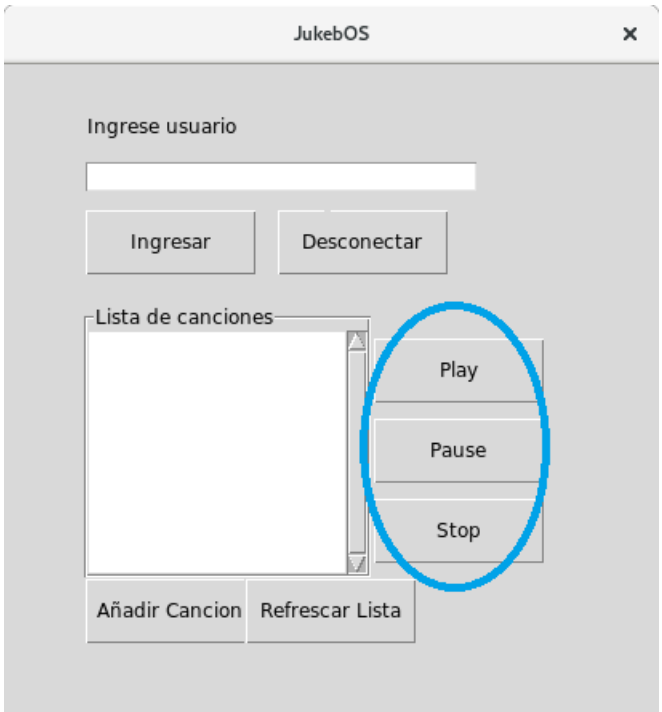


Añadir nuevo contenido: Para poder agregar canciones, el usuario debe de presionar el botón de ¨añadir canción¨, este le desplegara una ventana que permite la navegación por la máquina del cliente. Solo mostrará archivos con extensión mp3, y solo podrá subir contenido con ese formato.



Escuchar alguna canción

Para poder escuchar alguna canción se debe seleccionar en la lista de canciones que se muestran una vez el usuario ha iniciado sesión.



Código correspondiente al servicio en c:

```c
#include <zmq.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <string.h>
#include <errno.h>
#include <signal.h>
#include <sys/types.h>
#include <gst/gst.h>
#include <glib.h>


int main (int argc, char *argv[])
{
    int pid;
    //  Socket to talk to clients
    void *context = zmq_ctx_new ();
    void *responder = zmq_socket (context, ZMQ_REP);
    int rc = zmq_bind (responder, "tcp://*:5500");
    assert (rc == 0);

    int portId=6500;
    char * response = malloc(40*sizeof(char));
    memset(response, 32, 40);

    printf ("Service Dispatcher listening....\n");
    char * buffer = malloc(25*sizeof(char));
    memset(buffer, 0, 25);

    while (1){
        zmq_recv (responder, buffer, 25, 0);
        sprintf(response, "{\"zmq_port\":%d,\"gst_port\":%d}", portId,
portId+1);
        pid = fork();
        if(pid==0){break;}
        zmq_send (responder, response, 40, 0);
        portId=portId+2;
    }

    const char * location = buffer;
    if(pid==0){
      printf("Hijo creado con: %s\n",response);
      GstElement *pipeline;
      GstElement *component;
```

```
      GstMessage *msg;
      GstBus *bus;
      GError *error = NULL;

      void *contextChild = zmq_ctx_new ();
      void *responderChild = zmq_socket (contextChild, ZMQ_REP);
      char host [15];
      sprintf(host, "tcp://*:%d", portId);
      printf("%s\n", host);
      int rcChild = zmq_bind (responderChild, host);
      assert (rcChild == 0);

      gst_init (&argc, &argv);

      pipeline = gst_parse_launch ("filesrc name=my_filesrc ! mpegaudioparse !
mpegtsmux ! rtpmp2tpay ! queue max-size-bytes=0 max-size-buffers=0 max-size-
time=0 leaky=downstream ! tcpserversink name=my_portsrc host=127.0.0.1",
&error);
      if (!pipeline) {
        g_print ("Parse error: %s\n", error->message);
        exit (1);
      }

      component = gst_bin_get_by_name (GST_BIN (pipeline), "my_filesrc");
      g_object_set (component, "location", location, NULL);
      g_object_unref (component);

      component = gst_bin_get_by_name (GST_BIN (pipeline), "my_portsrc");
      g_object_set (component, "port", portId+1, NULL);
      g_object_unref (component);

      gst_element_set_state (pipeline, GST_STATE_PAUSED);

      while(1){
        printf("Waiting a instruction:\n");
        char i_buffered [1];
        zmq_recv (responderChild, i_buffered, 1, 0);
        int i = atoi(i_buffered);
        printf( "You entered: %d\n", i);
        if (i == 0){
          gst_element_set_state (pipeline, GST_STATE_PLAYING);
          zmq_send (responderChild, "playing", 7, 0);
        }
        else if (i == 1){
          gst_element_set_state (pipeline, GST_STATE_PAUSED);
          zmq_send (responderChild, "Paused", 6, 0);
        }
        else if (i == 2){
```

```
          zmq_send (responderChild, "Exiting", 7, 0);
          return 0;
        }
        else{
          printf("\nActions: \n  0. Play \n  1. Pause\n  2. Exit\n");
          zmq_send (responderChild, "Showing menu", 12, 0);
        }
      }

      bus = gst_element_get_bus (pipeline);

      msg = gst_bus_poll (bus, GST_MESSAGE_EOS | GST_MESSAGE_ERROR, -1);

      switch (GST_MESSAGE_TYPE (msg)) {
        case GST_MESSAGE_EOS: {
          g_print ("EOS\n");
          break;
        }
        case GST_MESSAGE_ERROR: {
          GError *err = NULL;
;          gchar *dbg = NULL
          gst_message_parse_error (msg, &err, &dbg);
          if (err) {
            g_printerr ("ERROR: %s\n", err->message);
            g_error_free (err);
          }
          if (dbg) {
            g_printerr ("[Debug details: %s]\n", dbg);
            g_free (dbg);
          }
        }
        default:
          g_printerr ("Unexpected message of type %d", GST_MESSAGE_TYPE
(msg));
          break;
      }
      gst_message_unref (msg);

      gst_element_set_state (pipeline, GST_STATE_NULL);
      gst_object_unref (pipeline);
      gst_object_unref (bus);
      kill(getpid(), SIGKILL);
    }
    else{
      printf("El padre Sale");
    }
    return 0;
}
```