4. Desarrollo de aplicaciones

Una vez explicadas todas las tecnologías y expuestos los objetivos de este trabajo, se dará utilidad a todos estas herramientas. Para ello se han creado dos aplicaciones. La primera **SurveillanceApp,** se trata de una aplicación web capaz de manejar eventos en YouTube. La segunda …… continuar.

4.1 SurveillanceApp

Como se ha comentado anteriormente el objetivo de esta aplicación es poder manejar eventos programados en YouTube y retransmitir el contenido capturado por un elemento hardware que en nuestro caso será una cama web. La aplicación incorpora las siguientes funcionalidades.

1. Permite crear evento programados asociados a un canal de YouTube
2. Permite iniciar y detener la retransmisión del evento a través de ffmpeg
3. Permite añadir subtítulos al evento retransmitido.
4. Muestra en su pantalla principal los eventos en directo asociados a un canal de YouTube.

Antes de pasar a analizar el código se explicara la arquitectura con el objetivo de entender mejor su funcionamiento.

La aplicación usa un servidor programado en nodeJS, dicho servidor es el encargado de recoger los datos e instrucciones dados por el usuario a través de la interfaz web y comunicárselo a los scripts desarrollados con las librerías de YouTube API. Dichos scripts se encuentran desarrollados en Python.

Para la ejecución de los scripts de Python se ha utilizado *child_process*, un módulo de nodeJS capaz de ejecutar comandos en la terminal de esta forma podemos comunicarnos con Youtube.

El flujo audiovisual es recogido a través de ffmpeg mediante la herramienta *video4linux2* , que se encarga de recoger el contenido capturado vía hardware por una cámara web con micrófono incorporado. Para la ejecución de ffmpeg se sigue la misma técnica que con los scripts de YouTube. También es ffmpeg quien se encarga de retransmitir el flujo hacia los servidores de YouTube.

Cabe destacar que la aplicación implementa una seguridad adicional a la de YouTube. Esto se debe a que en la aplicación se manejan datos privados relacionados con la cuenta de google del usuario, por ello se ha implementado un mecanismo de seguridad basado en un *middleware* y el establecimiento de sesión. El *middleware* es un mecanismo que protege ciertas rutas de acceso, por ejemplo si queremos acceder a la parte de creación de eventos el *middleware* servirá un formulario de identificación. Una vez identificado se crea una sesión de usuario para no identificarnos continuamente.

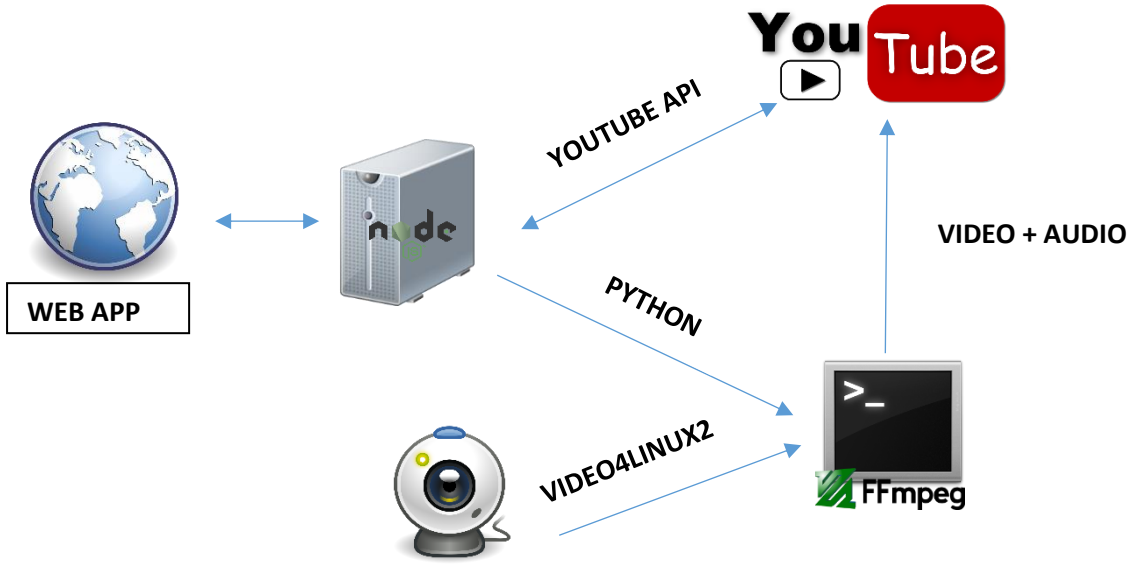

Figura 4.1 Arquitectura SurveillanceApp

4.1.1 Comunicación con Python

Una de las dificultades encontradas a la hora de desarrollar la aplicación es que las librerías proporcionadas por YouTube no presentan una versión estable para JavaScript. Como se ha comentado anteriormente las funcionalidades relacionadas con YouTube han sido desarrolladas con Python, como se podrá ver más adelante.

Para manejar estos scripts se ha recurrido al módulo *child_process*, que forma parte de NodeJS. A través de este módulo Node nos proporciona la opción de crear un proceso hijo de forma asíncrona, es decir sin bloquear el flujo principal del servidor. *Child_proccess* propociona varios métodos para realizar esta tarea, en esta aplicación se usara el método *spawn*, que es capaz de generar un nuevo proceso a partir de un comando dado, en nuestro caso el comando será el de ejecución de ficheros Python.

El método toma tres argumentos ***child_process.spawn(command[, args][, options])*** , el primero es el comando a ejecutar, el segundo un array con los argumentos del comando si este los tuviera y por ultimo opciones del comando. De esta forma se crea un proceso paralelo al servidor capaz que ejecuta el comando en un terminal del sistema sin bloquear al servidor.

A continuación se incluye un fragmento de código donde se puede ver el uso del módulo dentro de la aplicación. En el podemos observar una primera sentencia donde se crea el proceso haciendo uso de *spawn* , ejecutando el script de Python. Una vez generado el proceso este tiene proporciona dos salidas stdout y stderr.

- **stdout,** recojo los datos devueltos por Python. De esta forma si desde Python queremos enviar datos al flujo principal debemos hacerlo mediante el método *print* y serán recogidos por stdout.
- **stderr,** aquí se recoge la salida de error para errores de ejecución del comando como errores que puedan surgir dentro del script.

```
exports.retrieveStreamList =  function(req,res,status){

        var process = spawn('python',['./public/python/streamList.py', '--status' , status]);
        processOutput(res,process);
}

function processOutput(res,process){

        var py_data;
        var py_err;

        process.stdout.on("data",function(data){
                        py_data += data;//obtenemos el string
        })

        process.stdout.on("end",function(){

        if(typeof py_data !== "undefined"){
    py_data = py_data.substring(9);
    console.log(py_data)
    if(py_data.localeCompare("ERROR") == 1){
      res.status(500).end();
    }else{
                    res.end(py_data)
    }
                }
        })

        process.stderr.on("data", function(data){
                py_err += data;
        })
        process.stderr.on("end",function(){

                if(typeof py_err !== 'undefined'){
                        res.status(500).end()
                        console.log("Python Error: " + py_err)
                }
        })
}
```

4.1.2 Comunicación con YouTube

Como ya sabemos para esta comunicación se han usado las librerías proporcionadas por YouTube API. YouTube API proporciona muchas alternativas, para esta aplicación se han elegido dos recursos *livebroadcast* y *livestream* ambos explicados en el capítulo tres. Con estos recursos hemos dado tres funcionalidades relacionadas con YouTube a la aplicación, separadas en tres scripts de Python. Todas ellas tienen en común la parte de autenticación que implementa el protocolo OAuth 2.0.

```
# The CLIENT_SECRETS_FILE variable specifies the name of a file that contains
# the OAuth 2.0 information for this application, including its client_id and
# client_secret.


CLIENT_SECRETS_FILE = "./private/client_secret.json"

YOUTUBE_READ_WRITE_SCOPE = "https://www.googleapis.com/auth/youtube"
YOUTUBE_API_SERVICE_NAME = "youtube"
YOUTUBE_API_VERSION = "v3"

MISSING_CLIENT_SECRETS_MESSAGE = """
WARNING: Please configure OAuth 2.0

"""
def get_authenticated_service(args):


  flow = flow_from_clientsecrets(CLIENT_SECRETS_FILE,
    scope=YOUTUBE_READ_WRITE_SCOPE,
    message=MISSING_CLIENT_SECRETS_MESSAGE)

  storage = Storage("%s-oauth2.json" % sys.argv[0])
  credentials = storage.get()

  if credentials is None or credentials.invalid:
    credentials = run_flow(flow, storage, args)
```

En este fragmento de código aparecen varias cosas importantes para poder entender el funcionamiento de las librerías. Como se ha comentado en el capítulo tres YouTube ha adoptado Oauth 2.0 como sistema de autenticación. Para poder usar las librerías de YouTube primeramente debes registrarte en la consola de desarrolladores de google[1] y dar de alta tu proyecto y una vez dado de alta debes habilitar las API que tu proyecto usara. Terminado este proceso obtendrás el secreto de cliente (clien_secret.json), es un fichero el cual contiene las credenciales que te identifican como propietario de la cuenta de google asociada al proyecto.

Otra parte importante es la de los ámbitos o scope. Esta parte especifica que permisos tiene para interactuar con YouTube la aplicación. Por ejemplo en el código escrito arriba el scope, permite tanto leer como modificar los recursos asociados a la cuenta de YouTube. Los otros parámetros indican el API que va a ser usada y su versión.

A continuación nos encontramos con el método que lleva a cabo el proceso de autenticación. Dicho método es proporcionado por google en la documentación de la API. Este método busca un fichero con las credenciales que autorizan a la aplicación a acceder a los datos almacenados en la cuenta de YouTube asociada a la misma. Si es la primera vez que se accede estos datos no estarán generados por lo cual se redirigirá automáticamente al usuario a los servidores de autenticación de google. Si la autenticación ha sido correcta se creara automáticamente un fichero que permite el acceso a la cuenta por parte de la aplicación.

[1] https://console.developers.google.com

- **Creación de eventos**

Una de las funcionalidades que nos permite la aplicación es la creación de eventos programados. Para ellos a través de un formulario contenido en la interfaz web son recogidos una serie de datos como el título del evento, el horario, la privacidad o la calidad del mismo, todas ellas propiedades de ambos recursos. Estos datos son enviados al servidor en formato JSON y proporcionados como entrada a *createEvent.py*. CreateEvent hace uso del método *insert* tanto de *livebroadcast* como de *livestream* para crear ambos recursos. Una vez creados se recupera su ID para proporcionárselo al método *bind* perteneciente a *livebroadcast* de forma que el evento es creado en el canal asociado.

```
# Create a liveBroadcast resource and set configuration
def insert_broadcast(youtube, options):
  insert_broadcast_response = youtube.liveBroadcasts().insert(
   part="snippet,status,contentDetails",
    body=dict(snippet=dict(
        title= args["broadcast_title"],
        scheduledStartTime= args["start_time"]),

      status=dict(
        privacyStatus= args["privacy"]),
      contentDetails=dict(
        monitorStream=dict(
          enableMonitorStream = 'true')
        )
      )
  ).execute()

  snippet = insert_broadcast_response["snippet"]
  return insert_broadcast_response["id"]

# Create a liveStream resource and set configuration
def insert_stream(youtube, options):
  insert_stream_response = youtube.liveStreams().insert(
    part="snippet,cdn",
    body=dict(
      snippet=dict(
        title= args["broadcast_title"]
      ),
      cdn=dict(
        format= args["format"],
        ingestionType="rtmp"
      )
    )
  ).execute()

  snippet = insert_stream_response["snippet"]
  return insert_stream_response["id"]

# Bind the broadcast to the video stream.
def bind_broadcast(youtube, broadcast_id, stream_id):
  bind_broadcast_response = youtube.liveBroadcasts().bind(
    part="id,contentDetails",
    id=broadcast_id,
    streamId=stream_id
  ).execute()
```

- **Recuperación de información**

Dentro de la aplicación en su pantalla principal podemos ver el video de aquellos eventos que se encuentran en directo sin necesidad de acudir a YouTube. Por otro lado dentro de la parte privada de la aplicación, únicamente accesible para administradores, se pueden ver los eventos programados e iniciar su retransmisión o los eventos que se están retransmitiendo para detenerlos.

Todas estas funcionalidades requieren de datos proporcionados por los servidores de YouTube, para recuperar estos datos se recurre al método *list*, perteneciente tanto al recurso *livebroadcast* como al *livestream*. Dicho método admite filtros para buscar los eventos que nos interesan, en nuestro caso filtramos por el estado del evento, activo o programado, y además especificamos las partes del recurso que queremos recuperar[2] por último se limitan los resultados a 50.

```
list_broadcasts_request = youtube.liveBroadcasts().list(
    broadcastStatus= status,
    part="snippet, contentDetails",
    maxResults=50
  )
```

En el caso de *livestream* recuperamos únicamente el stream asociado a la retransmisión mediante su ID, de este recurso nos interésa su propiedad *cdn* ya que de ahí obetenemos el *stream_name*, necesario para la posterior retransmisión ya que este ID identifica al recurso en los servidores de ingestión de contenido de YouTube, además recuperamos los datos de ingestión como la calidad del evento de forma que luego se pueden calcular datos como bitrate o resolución. Otras propiedades importantes que recuperamos son *monitor* que contiene un iframe que posteriormente se inserta en la aplicación web para poder visualizar el evento una vez activo.

Una vez recuperados son almacenados en *list_broadcast_request*. Este objeto contiene la información de todos los eventos y de él recuperamos la información que nos interesa de cada uno y formamos un JSON que será la respuesta a la petición web.

[2] Capítulo 3. Propiedades livestream y livebroadcast

```
while list_broadcasts_request:
    broadcast_response = list_broadcasts_request.execute()
    for broadcast in broadcast_response.get("items", []):

      title = broadcast["snippet"]["title"]
      monitorStream = broadcast["contentDetails"]["monitorStream"]["embedHtml"]
      stream_id = broadcast["contentDetails"]["boundStreamId"]
      cdn = getStreamKey(youtube,stream_id)
      data_output["data"].append({"title" : title , "streamkey" :
cdn["ingestionInfo"]["streamName"],
                                  "monitor": monitorStream, "quality" :
cdn["format"],"broadcastID": broadcast["id"] })

    list_broadcasts_request =
youtube.liveBroadcasts().list_next(list_broadcasts_request, broadcast_response)

# Retrieve a livestream resource match with stream_id
def getStreamKey(youtube,stream_id):

  list_streams_request = youtube.liveStreams().list(
    part="cdn",
    id=stream_id,
    maxResults=1
  )
  list_streams_response = list_streams_request.execute()
  return list_streams_response["items"][0]["cdn"]
```

- **Terminar la retransmisión del evento**

Esta es la última funcionalidad que le se le ha dado a la aplicación apoyándonos en la las librerías de YouTube. Para conseguir este objetivo se ha usado el método *transition* perteneciente al recurso *livebroadcast*. Con dicho método recuperamos una emisión en directo a partir de su ID y modificamos su estado de activo a finalizado, de esta forma youtube dará por terminado el evento. Adicionalmente se añade el comando –pkill de Linux el cual acabara con la ejecución de ffmpeg una vez terminado el evento y se podrá disponer de la cámara para otros eventos.

```
def stop_broadcast(youtube,brID):

      broadcast_request = youtube.liveBroadcasts().transition(
    broadcastStatus= "complete",
    id = brID,
    part = "status")
      broadcast_request.execute()
```

- **Comunicación con ffmpeg**

Ffmpeg es el encargado de enviar el flujo audiovisual a los servidores de ingestión de contenido de YouTube. Ffmpeg presenta varias herramientas pero para este objetivo únicamente hemos usado la funcionalidad que permite manejarlo desde la terminal. Para ello se ha desarrollado un script de Python que escribe en un fichero .sh el comando de ffmpeg a ejecutar, tras esto mediante la instrucción *os.system* se ejecuta el fichero. Para la construcción del comando ffmpeg se toma como entrada primeramente la calidad del evento, en función de esta calidad se añade el bitrate y la resolución del video según las especificaciones de YouTube. Otro parámetro fundamental que recibe el script es el *stream_key*, representa un identificador único del *stream* dentro de los servidores de YouTube para que este pueda asociar el flujo audiovisual recibido al evento.

```
def list_streams(stream_key,resolution,bitrate):

   print stream_key
   os.system("chmod +rw ./public/static/ffmpeg.sh")
   outfile = open('./public/static/ffmpeg.sh', 'w')
         outfile.write('ffmpeg -f alsa -ac 2 -i default -f video4linux2'+
     '-framerate 15 -video_size ' + resolution +
     '-i dev/video0 -vcodec libx264 -preset veryfast -minrate ' + bitrate  +
     '-maxrate 1000k -bufsize 1000k' +' -vf "format=yuv420p"  -g 30 ' +
     '-vf drawtext= "fontfile=/usr/share/fonts/truetype/freefont/FreeSerif.ttf:'+
     'fontsize=24:''fontcolor=yellow:textfile=./public/static/subtitles.txt:reloa
     d=1 :x=100:y=50"' +
     ' -c:a libmp3lame -b:a 128k -ar 44100' +
     ' -force_key_frames 0:00:04 -f flv rtmp://a.rtmp.youtube.com/live2/' +
     Stream_key

   outfile.close()
   os.system("chmod 0755 ./public/static/ffmpeg.sh")
   os.system("./public/static/ffmpeg.sh")
```

Ffmpeg presenta una gran variedad de posibilidades a continuación se explican las principales partes del comando usado en el script:

- **Video4linux2 y alsa,** son componentes pertenecientes al SO Linux que se encargan de recoger los flujos de video y audio respectivamente. Como fuente de video usamos la cámara principal del sistema, /dev/video0, para el audio el micro definido por defecto.
- **Libx264**, códec de video H264, tras el aparecen las opciones de codificación del video, el perfil no está especificado pero por defecto ffmepg usa el perfil básico del códec.
- **Drawtext**, opción usada para superponer texto sobre el video, es la equivalencia a los subtítulos. El texto es leído de un fichero .txt, que tiene asociada la opción *reload* encargada de detectar el cambio en el fichero de texto. Las otras opciones lo acompañan se refieren al formato del texto mostrado en pantalla.
- **Force_key_frames**, a la hora de codificar el video se usan frames de referencia a partir de los cuales se codifican los siguientes, este comando obliga a enviar estos frames cada cuatro segundos.
- **Rtmp,** protocolo de comunicación usado entre el servidor de YouTube y ffmpeg para el intercambio del contenido.