Memoria Proyecto Final CDPS

CDPSFY

Alejandro Docasar Moreno, Carlos Padilla Salazar
CDPS, G46 | 20-1-2016

ÍNDICE:

1. Objetivo:

Implementación de un **servicio de reproducción de canciones** que cumpla un conjunto de requisitos en cuanto a escalabilidad y fiabilidad se refiere.
Para ello nos han encargado el **diseño, implementación y despliegue** de dicho servicio en la dirección http://www.server.cdpsfy.es.
El servicio a desarrollar debe permitir al usuario subir sus propios ficheros de audio y reproducir los ya existentes.
El servicio se dividirá en dos partes:

- **server.cdpsfy.es** será el servidor web que contenga la lógica de la aplicación, así como el modelo de datos de canciones.
- **tracks.cdpsfy.es** será una red de servidores ubicada detrás de un balanceador de carga que contendrá una serie de servidores con discos y servidores web. Estos últimos serán los encargados de subir y borrar canciones, que es lo que más cargará la aplicación.

Además, se deberá crear otra máquina adicional donde se instalará **Nagios** para monitorizar toda la infraestructura.

2. Diseño de solución de alta disponibilidad:

La arquitectura de nuestra solución va a ser la siguiente:

- **Lado cliente:**
  - Disponemos de dos clientes **C1 y C2** que simularán dos usuarios que quieran hacer uso de la aplicación.
  - Aquí instalaremos también la máquina **M1** en la que se efectuará la monitorización del escenario completo, lo cual nos permitirá saber en todo momento si hay algún servicio que no funcione.
- **Lado servidor:**
  - Tenemos el servidor **S1**, en el cual vamos a alojar la **aplicación CDPSfy**. Este servidor enviará y solicitará canciones a la granja de servidores, donde están ubicados los servidores Nasx, que son los encargados de almacenar todas las canciones que suban los usuarios. Además, en S1 almacenaremos una **base de datos** que nos permitirá añadir persistencia al servicio, de modo que las canciones que suba un usuario permanezcan guardadas en los servidores Nasx hasta que alguien las borre o se destruya el escenario.
  - Disponemos también de los servidores **S2, S3 y S4** que serán los encargados de subir y bajar canciones a los Nasx. Esta tarea es la que más carga la aplicación, por lo tanto hemos decidido que este trabajo esté dividido de forma homogénea entre los tres servidores. Si en algún momento la demanda de servicio de nuestra aplicación desbordase la capacidad de estos servidores, sería tan fácil como añadir más **servidores en paralelo** para distribuirse el trabajo entre todos. Podemos garantizar por lo tanto que disponemos de una **solución escalable**. Además, disponer de más de un solo servidor para realizar esta tarea también aporta una **gran disponibilidad** de servicio, pues si alguno de los tres servidores se cae los otros dos podrían seguir ofreciéndolo, de modo que tendrían que caerse los tres servidores en un corto período de tiempo para que nuestro servicio no estuviese disponible, lo cual calculamos que es muy

improbable.
    - Por último, disponemos de los servidores Nas1, Nas2 y Nas3, que almacenan las canciones que suben los usuarios a través de la aplicación. Se ha creado un sistema de ficheros que replica la información que contienen los servidores S2, S3 y S4 en cada uno de los Nas, de modo que ofrecemos un servicio fiable al cliente. Al añadir redundancia al sistema de ficheros garantizamos que si alguno de los servidores Nas deja de funcionar no nos quedamos sin la opción de poder acceder a las canciones que los usuarios han subido, puesto que aún tenemos todas esas canciones en los otros dos servidores Nas.
- Balanceador de tráfico: Este elemento será el encargado de distribuir el tráfico de canciones generado por los usuarios de manera equitativa entre los tres servidores. Para ello, se ha configurado una división de tráfico según el algoritmo "round robin" visto en anteriores prácticas, que distribuye el tráfico entre los servidores que se encuentren disponibles en cada momento de forma totalmente equitativa. Al ser los tres servidores de idénticas características hemos considerado que esta opción era la más eficiente de las estudiadas, puesto que todos los servidores van a tener aproximadamente la misma carga.

Para poder tener una visión más clara de la solución propuesta, se adjunta en el anexo una imagen con la topología de la red montada.

3. Configuración del escenario:

3.1. Configurar Glusterfs:

Se va a configurar un sistema de ficheros glusterfs en los servidores de discos de la granja de servidores (a partir de ahora servidores Nasx) el cual podrá ser montado desde los servidores web de la granja de servidores (a partir de ahora servidores Sx).
Para ello, en la máquina Nas1 hay que ejecutar las siguientes órdenes:

```
gluster peer probe nas2
gluster peer probe nas3
gluster volume create nas replica 3 nas1:/nas nas2:/nas nas3:/nas force
```

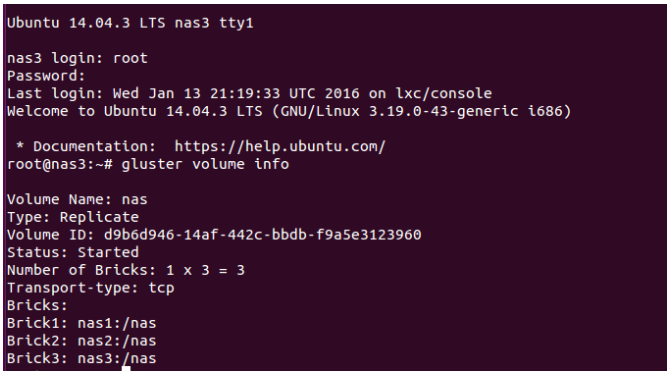
Con estas órdenes ya tenemos el volumen creado, y podemos proceder a arrancar el volumen con la orden:

```
gluster volume start nas
```

Para ver la información acerca del volumen creado, podemos ejecutar la orden:

```
gluster volume info
```

En la que se podrá comprobar que hemos creado un volumen de nombre "nas", donde la información estará replicada en 3 bricks (nas1, nas2 y nas3).
La siguiente captura ilustra la información del volumen creado:



Lo que se va a implementar a continuación servirá para que todos aquellos archivos que se guarden en los servidores Sx se repliquen automáticamente en los servidores Nasx. Es decir, las canciones que se escriban en tracks.cdpsfy.es estarán replicadas en los servidores de disco.
En cada uno de los servidores (menos s1 que aloja la aplicación web) web ejecutar:

```
mkdir /mnt/nas
mount -t glusterfs 10.1.3.21:/nas /mnt/nas
```

3.2. Alojar aplicación en S1.

Como se ha comentado al principio, server.cdpsfy.es será el servidor web que contenga la lógica de la aplicación. Vamos a alojar esta aplicación en el servidor S1.
Para ello, desde el host ejecutar la orden:

```
sudo cp -r /home/alex/CDPSfy /var/lib/lxc/s1/rootfs/tmp
```

Para poder ejecutar el proyecto, debemos instalar node, nodejs y node-legacy en s1:

```
sudo apt-get install node //Igual con nodejs y node-legacy
```

Una vez hecho esto, para arrancar el proyecto tan solo debemos ejecutar (en la carpeta donde esté ubicada la aplicación CDPSfy):

```
npm install
npm start
```

Para poder acceder remotamente con comodidad, en el archivo /etc/hosts de la máquina local escribir:

```
10.1.2.11 server.cdpsfy.es
10.1.1.1  tracks.cdpsfy.es
```

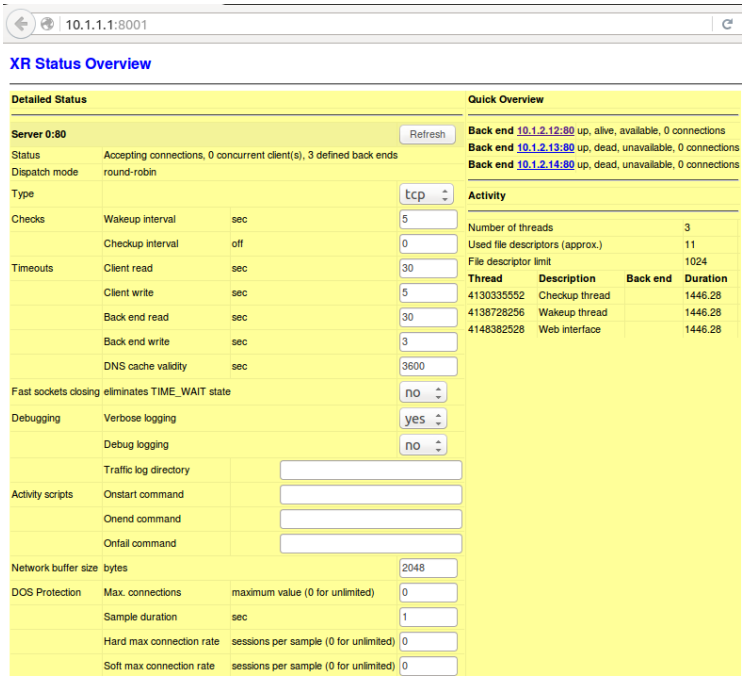
Si además cambiamos en el archivo www del proyecto CDPSfy el puerto de 8080 a 80, podremos acceder a la aplicación poniendo en el navegador: server.cdpsfy.es.

3.3. Configuración del balanceador de tráfico:

Para configurar el balanceador de tráfico (a partir de ahora lb) se seguirán los mismos pasos que en la **práctica 3**. Las configuraciones previas que tuvimos que realizar en esta práctica ya están hechas aquí (cambiar hostname, hosts, configurar las direcciones IP, habilitar el ip_forwarding en el lb...etc).
Por lo tanto, la tarea que nos queda es la de especificar al lb como queremos efectuar el **balanceo de tráfico** entre los servidores Sx.
En el lb ejecutar:

```
service apache2 stop
xr --verbose --server tcp:0:80 --url-match tracks --backend 10.1.2.12:80 --backend 10.1.2.13:80 --backend 10.1.2.14:80 --web-interface 0:8001 -d r
```

Mediante esta última orden indicamos al balanceador que queremos que reparta la carga demandada por la aplicación CDPSfy entre **3 backends** (los servidores S2, S3 y S4) y que lo haga según el algoritmo de **"round robin",** es decir, que vaya alternando periódicamente la carga entre los 3 servidores disponibles.
A continuación podemos acceder al servidor web para la gestión del balanceador a través de la dirección **http://10.1.1.1:8001**. Desde aquí podemos ver cómo afecta a la distribución de tráfico la disponibilidad de cada uno de los servidores S2, S3, S4.



Para una visualización más clara sobre cómo está distribuyendo el tráfico el lb entre los distintos backend, podemos probar a escribir:

```
echo sx > /var/www/html/index.html
```

Desde cualquiera de los clientes (c1, c2 o host) hacemos:

```
while true; do curl 10.1.1.1; sleep 0.1; done
```

De esta forma podremos ver como se está efectuando el reparto de tráfico por parte del lb. La opción por defecto (dispatch mode: **least-connections**) efectúa la división de la forma S2>S3>S4, es decir, si S2 está activo todo el tráfico irá a través de él, pero si este cae o se desactiva, irá por S3, y en último caso por S4. No obstante, como ya se ha indicado nosotros efectuamos el balanceo según el algoritmo "round robin", que hemos pensado más apropiado para la ocasión.

3.4. Nueva máquina para monitorización:

Se va a crear una nueva máquina en el **lado cliente** para monitorizar el sistema montado. Para ello, en el archivo p7.xml copiar la descripción de la máquina C2, cambiarle el nombre por M1 y su dirección IP por **10.1.1.13/24**.
Para que esta máquina se cree se debe hacer un destroy del escenario.

A continuación vamos a proceder a instalar en la máquina creada el entorno **Nagios**, para lo cual deberemos ejecutar las órdenes:

```
apt-get update
apt-get install nagios3
```

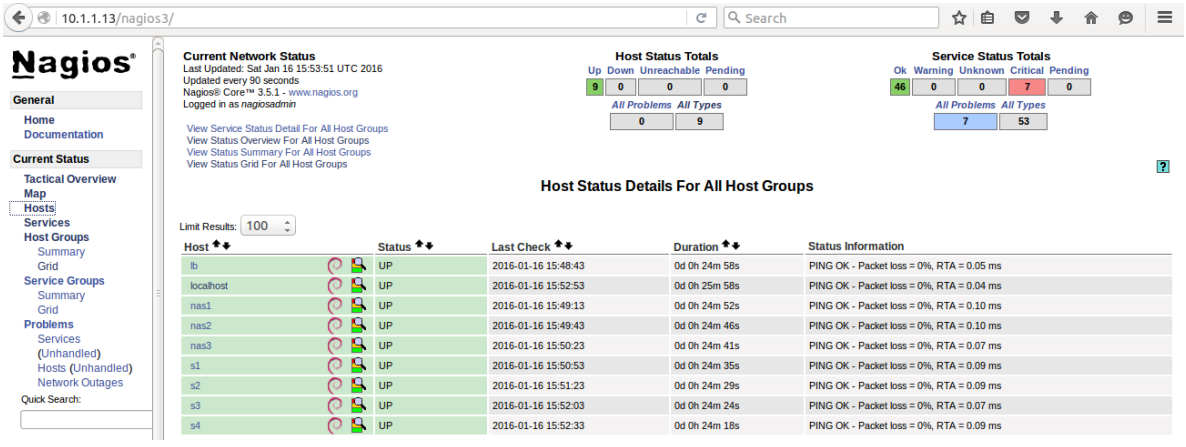
Cuando pregunte sobre la configuración del mail optamos por la opción "No configuration".
La contraseña del usuario 'nagiosadmin' será 'nagios'.

Una vez acabado el proceso de instalación, la interfaz web de Nagios estará accesible desde la dirección http://10.1.1.13/nagios3 .

Para monitorizar cada una de las máquinas se seguirá el siguiente procedimiento:

- Acceder al sistema de ficheros de M1 mediante cd /etc/nagios3/conf.d
- Copiar el archivo localhost_nagios2.cfg mediante cp localhost_nagios2.cfg s1_nagios2.cfg (se va a configurar S1 a modo de ejemplo).
- Modificar adecuadamente el archivo s1_nagios2.cfg, para ello, cambiar el parámetro **address** (poner ip de S1) y **hostname** (poner S1) de todas las entradas.
- Además, en el archivo hostgroups_nagios2.cfg habrá que añadir la máquina S1 en cada una de las entradas (localhost, s1).

Para el resto de máquinas se realizará el mismo proceso, hasta tener todas monitorizadas.



4. Fase de desarrollo del proyecto:

4.1. Desarrollo de la aplicación CDPSfy:

A la aplicación por defecto que se nos entregó había que realizar las modificaciones necesarias para que la gestión de ficheros de audio se realice a través de la solución diseñada para tracks.cdpsfy.es.
Básicamente se trata de **modificar** los métodos de **creación** y **borrado** de tracks:

- **create**: En la versión proporcionada este método simplemente añade al modelo de datos local los metadatos de la nueva canción. Debe modificarse el método para escribir el fichero en tracks.cdpsfy.es e introducir en el modelo de datos la verdadera URL generada para la canción en tracks.cdpsfy.es.
- **destroy**: en la versión proporcionada este método simplemente elimina la entrada correspondiente a la canción del modelo de datos local. Debe modificarse para eliminar el fichero de tracks.cdpsfy.es.

Para implementar esto, en la función create debemos hacer un **post** utilizando la herramienta **needle** con destino tracks.cdpsfy.es (la granja de servidores donde se almacenará la nueva canción), en el que enviaremos los datos de una nueva canción (para ello creamos una variable data en la que almacenamos en un **buffer** los metadatos de la nueva canción). Para escribir los metadatos de la nueva canción en el registro, modificamos la URL de destino para que quede de la forma:

```
var url = 'http://tracks.cdpsfy.es/'+name;
```

Por otro lado, en la función **destroy** eliminaremos las canciones utilizando de nuevo la herramienta **needle**, indicando la URL de la canción que se desea borrar y eliminándola también de registro de datos posteriormente a través del ID de la canción a borrar.

4.2. Desarrollo de tracks.cdpsfy.es:

Para desarrollar las **API REST** en cada uno de los servidores S2, S3 y S4 utilizaremos las herramientas **multer** y **fs**.
Con multer gestionamos que las canciones se envíen correctamente a su destino (los servidores Nasx) en la función **post** (router.post).
Disponemos también de una función **get** (router.get) para poder coger las canciones ya almacenadas en los Nasx y devolverlas a la aplicación CDPSfy.
Además, mediante la función **delete** (router.delete) eliminamos las canciones de los Nasx utilizando la herramienta **fs.unlink,** que las borra de las carpetas nas.

De este modo ya tenemos una **aplicación funcional** que nos permite subir y eliminar canciones y presentarlas en la aplicación para que los usuarios interactúen con ellas.

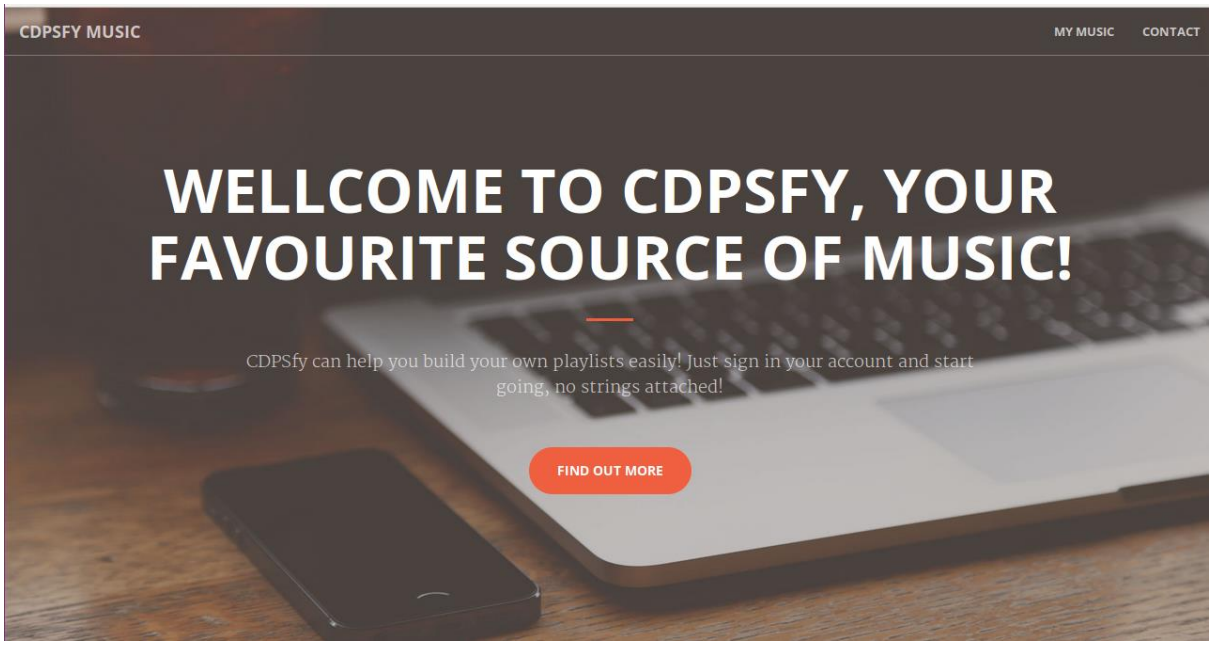
5. Mejoras:

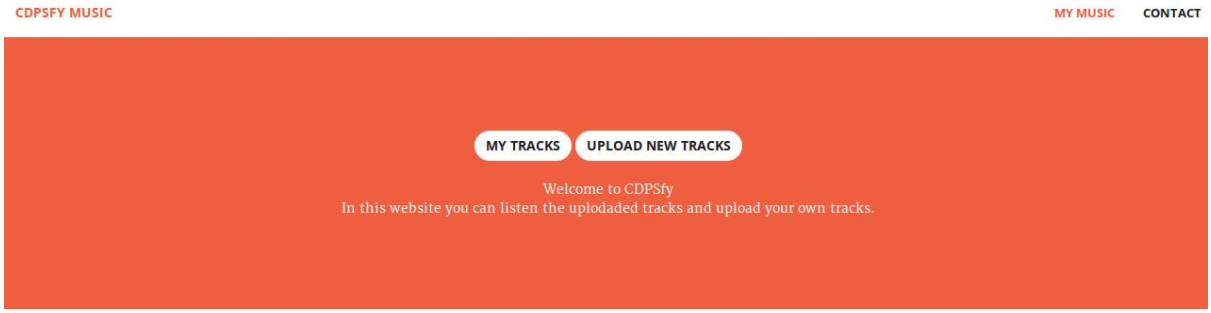
5.1. Script de automatización:

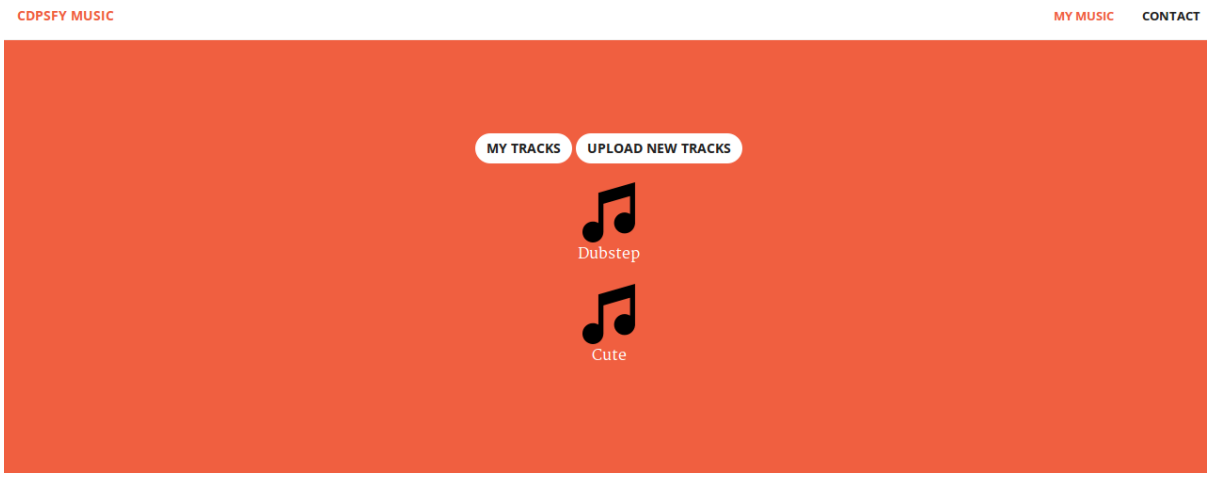
Con objeto de no tener que escribir muchas órdenes que resultan repetitivas y conllevan a una gran pérdida de tiempo, se ha creado un script que **automatiza** estas tareas de **configuración inicial** del escenario. Para ello se ha utilizado **Python**, tal y como se hizo en la práctica final p1.
Además, se han instalado en los servidores de forma permanente algunos paquetes que resultan imprescindibles para la práctica, como **node**, **nodejs** o **nagios3**. De esta forma nos ahorramos tener que descargarlos cada vez que hacemos un destroy del escenario.
En conclusión, se ha conseguido que la creación del escenario se lleve a cabo en mucho menos tiempo del inicial, lo cual consideramos que es un gran aporte para la práctica.

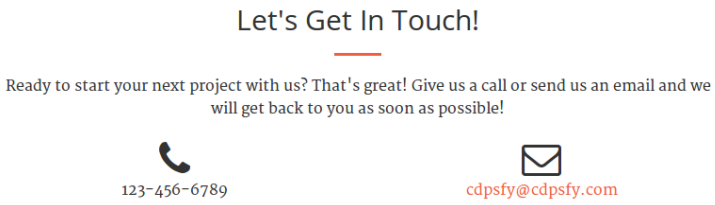
5.2. Diseño de la aplicación CDPSfy:

Hemos considerado oportuno adoptar un **nuevo diseño** más atractivo para el usuario de la aplicación CDPSfy.
Para realizar esta tarea exitosamente se han utilizado conocimientos adquiridos en las asignaturas **CORE** y **TEWC**.
Lo que se ha hecho es basar el diseño de nuestra aplicación en el framework **"Bootstrap"** creado por Twitter, que permite de una cómoda y rápida la creación de **diseños adaptativos** y muy vistosos.
El resultado final se puede apreciar en las siguientes capturas de pantalla:









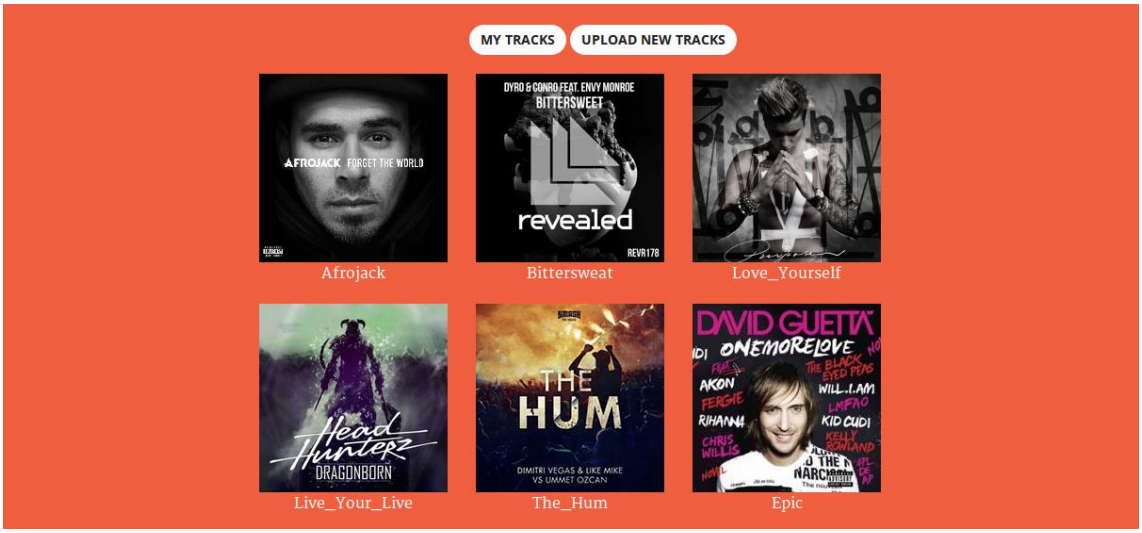
5.3. Base de datos:

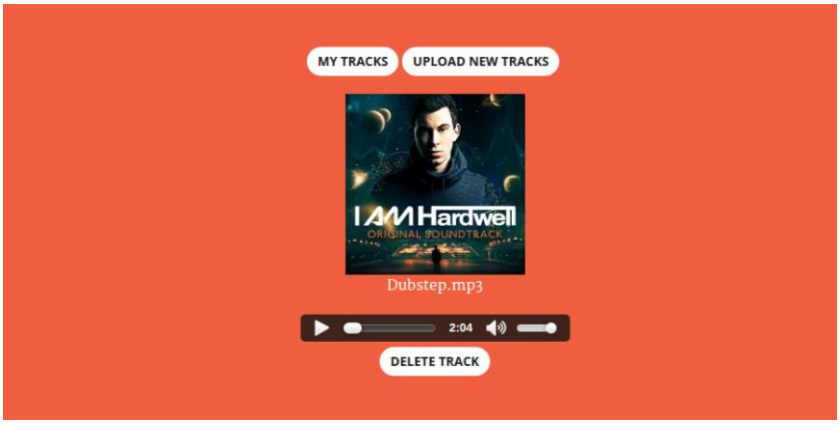
Para añadir **persistencia** a la aplicación y permitir a los usuarios que sus canciones queden almacenadas en los discos aunque se desactive el servicio, se ha decidido implementar una base de datos **MongoDB**.
Para ello hemos creado un modelo (**schema**) en el que añadimos los parámetros que van a ser guardados en la base de datos (id de la canción, nombre, url de la canción y url de la imagen).
Como el modelo de la práctica básica estaba alojado en **S1**, hemos decidido alojar la base de datos aquí.
El controlador "**track_controllers.js**" será el encargado de gestionar la base de datos.
A continuación describiremos cómo se maneja la base de datos en función de la acción que se ejecute en el controlador:

- list: El controlador se encarga de acceder a la base de datos e iterar por ella para enviar a index.ejs arrays que indican la información a presentar en la aplicación (nombre y url de la imagen de cada canción).
- show: Se encarga de encontrar la canción seleccionada por el usuario y le pasa a show.ejs la información necesaria para que presente la canción con su correspondiente imagen.
- create: Después de pasar la canción a los servidores REST, el controlador se encarga de buscar si hay alguna canción con el mismo nombre en la base de datos, y en caso negativo guarda la canción con sus respectivos parámetros.
- destroy: Busca la canción que se quiere borrar y después de mandar la petición DELETE al balanceador, borra la canción con ese nombre de la base de datos.

5.4. Añadir caratulas personalizadas a las canciones:

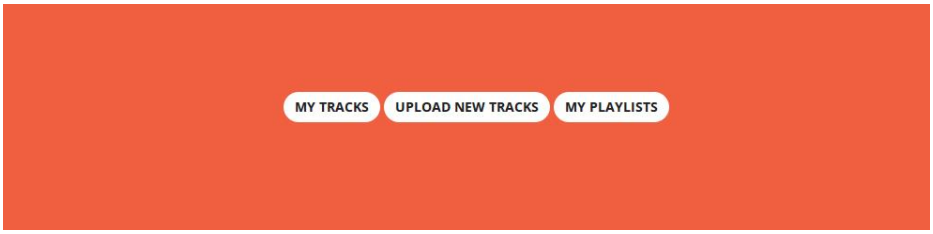
Para permitir una mayor personalización a los usuarios a la hora de subir sus canciones, se ha decidido implementar la posibilidad de asociar una imagen a cada canción, de modo que puedan ser identificadas más fácilmente.
Para ello, hemos añadido en la acción create la posibilidad de mandar los metadatos de una imagen en caso de que el usuario la agregase a la hora de subir una canción. En caso contrario hemos establecido una carátula por defecto.
La recepción de los datos mandados por la aplicación se realiza con multer, y en vez de usar upload.single usamos upload.fields, que ofrece la posibilidad de tanto si llega un archivo como si llegan dos, cogerlos y subirlos.
En el borrado volvemos a utilizar needle, pero ahora en vez de enviar null a los servidores REST enviamos el nombre de la imagen que se quiere borrar (no se podría usar el trackID porque el nombre de la imagen podría ser distinto).
Tanto en el show como en el list se deberán presentar las carátulas subidas previamente. Esto se hace a través de una petición GET a los servidores REST mediante una URL en el parámetro src de la imagen. En los servidores REST se atiende a la petición GET de la imagen de forma similar al de las canciones.
Los ejs han sido adaptados para que las distintas funcionalidades se implementen de forma correcta.
El resultado final se puede apreciar en las siguientes capturas de pantalla:

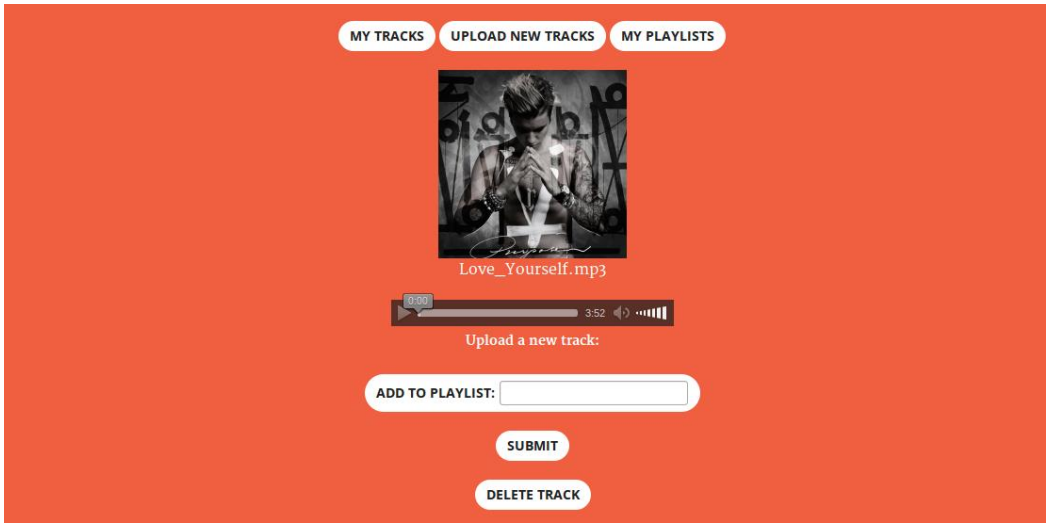


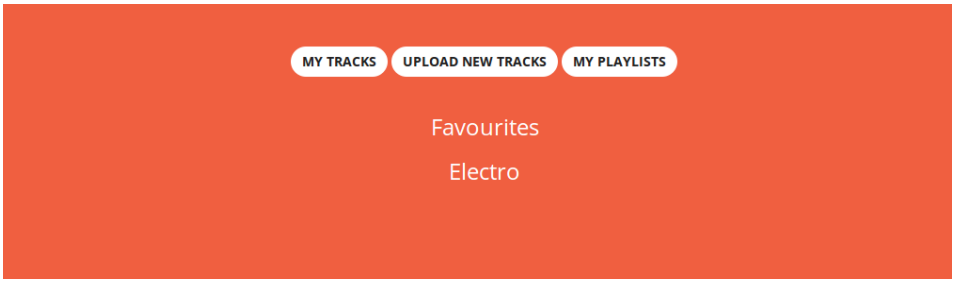


5.5. Añadir playlists:

Hemos añadido una nueva funcionalidad que permitirá a los usuarios agrupar sus **canciones favoritas** en grupos eligiendo el nombre de los mismos.
Para ello, en un primer lugar se ha añadido un parámetro más al **schema** de la base de datos llamado "**playlist"**, el cual nos dirá si la canción pertenece ya a alguna playlist.
En **track_controllers.js** hemos añadido más acciones como son: **list_playlist**, **add_playlist**, **new_playlist** y **show_playlist**, las cuales se encargan, respectivamente, de recoger las playlist y pasarlas al ejs para su muestra en pantalla, de añadir una canción a una playlist ya existente o nueva, de redirigir a la creación de una nueva playlist y, finalmente, de enseñar las canciones pertenecientes a una playlist.
Además, se han tenido que añadir ejs para la correcta visualización de los cambios, los cuales son: **new_playlist.ejs**, **playlist.ejs** y **show_playlist.ejs**, encargados de recibir los datos pasados por las acciones que se renderizarán para su visualización.
El resultado final podrá apreciarse en las siguientes capturas:





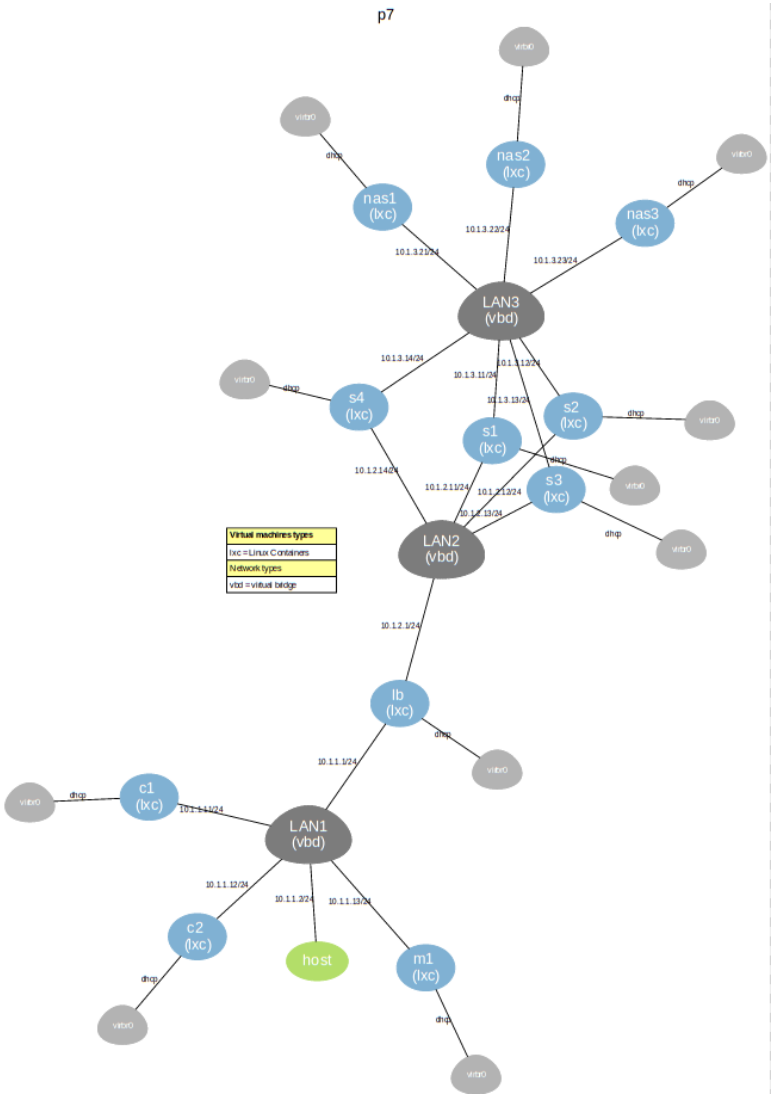


6. Cuestiones teóricas:

6.1 Despliegue con Amazon AWS/OpenStack:

Si realizásemos este proyecto utilizando **Amazon AWS,** no necesitaríamos crear nosotros máquinas virtuales, dado que éstas ya se las alquilaríamos a Amazon.
Así pues, la diferencia reside en que las máquinas que forman nuestro sistema se adquirirían a Amazon en lugar de tener que crearlas, y nosotros tan solo elegiríamos las características de éstas (memoria RAM, disco duro, soporte para base de datos…etc).
En caso de haberlo desplegado con **OpenStack**, la gestión de las máquinas se realizaría a través de Nova, y el almacenamiento de objetos a través de Swift. Además, en vez de utilizar Nagios podríamos usar la herramienta Ceilometer, que nos permite monitorizar el uso de tu nube OpenStack.
Para desplegar redes virtuales, en vez de utilizar VNX se emplearía Neutron o VPC, en función de si usamos OpenStack o Amazon AWS respectivamente.

7. Anexo


p7
nas2 (lxc)
nas1 (lxc)
nas3 (lxc)
LAN3 (vbd)
s4 (lxc)
s2 (lxc)
s1 (lxc)
s3 (lxc)
LAN2 (vbd)
lb (lxc)
c1 (lxc)
LAN1 (vbd)
c2 (lxc)
host
m1 (lxc)
Virtual machines types
lxc = Linux Containers
Network types
vbd = virtual bridge