Report

Spotify. A music distributed system.

Autores:

- Niley González Ferrales C-411 @NileyGF
- Josué Rodríguez Ramírez C-412 @josueRdgz

Proyecto en github: https://github.com/NileyGF/A-music-distributed-system

Introducción

Presentamos el diseño de un sistema distribuido compuesto por cuatro tipos distintos de nodos, cada uno con una responsabilidad (role) específica. Nuestra arquitectura consiste de un servidor DNS encargado de resolver nombres de hosts en direcciones IP, servidores de datos almacenando y gestionando la música, servidores de enrutamiento encargados de atender a los clientes y redirigirlos, balanceadamente, a los servidores que proveen los archivos multimedia y los nodos clientes accediendo a los recursos del sistema.

Nuestro sistema garantiza su funcionamiento si hay al menos un nodo de cada role. Sin embargo, si en esas condiciones se conectan muchos clientes es posible que el rendimiento no sea el mejor.

Ejecución

En el archivo Instructions.txt aparecen los comandos a utilizar para ejecutar el sistema.

Solo es necesario tener instalado docker, ya que la imagen de docker tiene todas las dependencias.

Servidores

En la carpeta `server` se ejecuta:

```
docker build -t server .
```

o

```
docker load -i <path/server.tar>
```

Eso crea la imagen de los servidores. Despúes es necesario crear una red de docker, donde interactuarán los contenedores de los servidores.

```
docker network create --driver bridge --subnet=172.20.0.0/16
dispotify_network
```

Luego, por cada servidor que se desee activar hay que seguir los siguientes pasos:

1. Crear un contenedor. Seguido el parámetro --ip hay que introducir una dirección IP válida que esté en el rango de la subred especificada al crear la red de docker, que tiene ser distinta para cada contenedor/servidor creado. Además, después de --name debe introducirse el nombre con que se identificará el contenedor

   ```
   docker run --net dispotify_network --ip <valid ip in 172.20.0.0/16, like
   172.20.0.2> -it -d --name <container_name> server bash
   ```

2. Luego, se ejecuta el contenedor en un terminal bash con el siguiente comando:

   ```
   docker exec -it <container_name> /bin/bash
   ```

En este nuevo terminal, se puede comprobar que los requerimientos han sido instalados ejecutando, por ejemplo `python3 --version`.

3. Para ejecutar un servidor específico es necesario aclarar algunos parámetros como el tipo de servidor y la dirección del seridor.

   Los tipos de servidores son:
   0 - Servidor sin Role predefinido
   1 - Servidor de datos
   2 - Servidor DNS
   3 - Servidor router

   El ip del servidor que hay que poner es el del contenedor donde se está ejecutando. Además si la dirección del DNS no es la default (172.20.0.2), también hay que proveerla.

   ```
   python3 server_class.py <server_type> <container_ip> <DNS_ip, optional>
   ```

En `<server_type>` corresponde un número entre 0 y 3, según lo explicado anteriormente. En `<container_ip>` va la dirección asignada al contenedor. En `<DNS_ip, optional>` va la dirección del contenedor donde se ejecute el DNS y solo es necesaria si es distinta a `'172.20.0.2'`.

Clientes

En la carpeta `client` se ejecuta:

```
docker build -t client .
```

o

```
docker load -i <path/client.tar>
```

Eso crea la imagen de los clientes.

1. Por cada cliente que se desee unir al sistema se crea un contenedor, con nombres distintos, utilizando el siguiente comando:

   ```
   docker run --net dispotify_network -it -d --name <client_x> client bash
   ```

2. Luego, se ejecuta el contenedor en un terminal bash con el siguiente comando:

   ```
   docker exec -it <client_x> /bin/bash
   ```

3. Por último se ejecuta el cliente con :

   ```
   python3 client_class.py <DNS_ip, optional>
   ```

Roles del sistema:

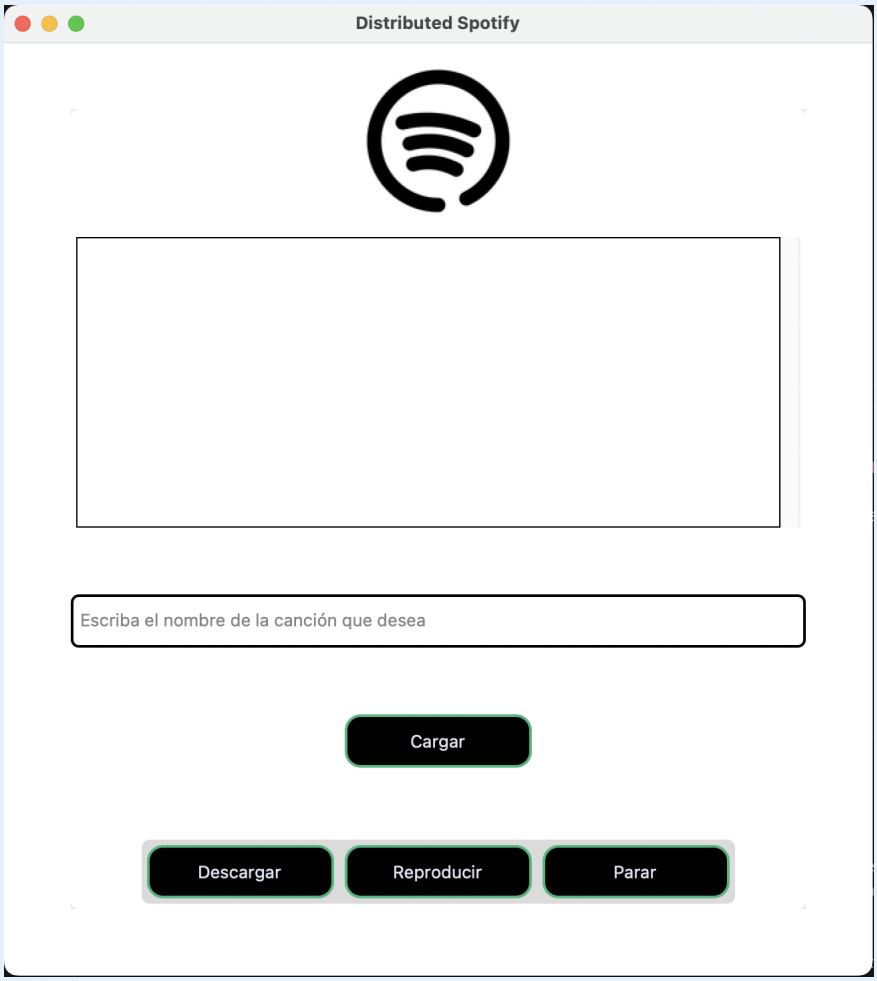
- **DNS**: implementamos este role para resolver dinámicamente la entrada al sistema tanto desde los clientes como desde otros servidores. Tiene tres funciones principales: agregar registros, resolución de nombre y mantenimiento de consistencia.

La adición de registros implica asociar un dominio con su dirección y un TTL (tiempo de vida), que indica el tiempo que debe durar ese registro en el DNS.

La resolución de nombre permite obtener todas las direcciones IP de los nodos activos en el sistema que corresponden al dominio solicitado.

Finalmente, la función de mantenimiento de consistencia conlleva a que cuando se inicia un nodo DNS se crea un subproceso que revisa periódicamente todos los registros almacenados para determinar si han expirado. Si es así, se contacta con el servidor correspondiente para comprobar su estado y, si responde correctamente, se renueva el TTL. Además, asegura que la dirección devuelta sea válida en el momento de la consulta.

- **Client**: este rol se encarga de manejar las solicitudes del usuario y comunicarse con el sistema de servidores distribuidos según sea necesario. Sus dos tareas principales incluyen actualizar la lista de metadatos de la música y solicitar canciones específicas al sistema para almacenarlas en cache y reproducirlas para el usuario.

> **App en desarrollo**
>
> 
>

- **Router**: su papel principal es manejar las solicitudes de los clientes, proporcionándoles información actualizada sobre listas de canciones y, dada una canción que el cliente desee, localizar los proveedores activos que ofrezcan la canción solicitada y enviar sus

direcciones al cliente. Todo ello mientras intenta mantener balance en el sistema con estrategias estocásticas.

- **Data**: el role de manejo de datos fue implementado como un nodo que utiliza SQLite para almacenar las canciones como fragmentos cortos (por ejemplo, 10 segundos) en una base de datos dividida en dos tablas: una para los tags musicales y otra para la multimedia en sí misma. Está diseñado para manejar cuatro tipos de solicitudes: mostrar la lista de canciones disponibles, confirmar si puede ofrecer una canción específica, entregar un fragmento específico a partir de un tiempo en milisegundos, y suministrar varios fragmentos a partir de un tiempo determinado.

De esta forma garantizamos que puede proveer todos los fragmento de una canción, los fragmentos a partir de un tiempo t, o solo un fragmento específico. Esta estrategia se desarrolló como medida de tolerancia a fallos y para mejorar la velocidad general del sistema, permitiendo al cliente empezar a reproducir la canción sin tener que esperar hasta que se haya descargado completamente.

Asignación de roles dinámicamente

Los servidores de datos y routers conforman un anillo para garantizar la disponibilidad del sistema. En esta estructura, cuando se une un nuevo servidor comprueba que role es más necesario en ese momento y se asigna dinámicamente. Para ello contacta con el DNS para consultar cuantos servidores hay activos por cada role e intenta mantener una relación de 2:3 de servidores de data y servidores routers.

En este anillo, cada integrante tiene una referencia a los 2 nodos que hay hacia adelante, y la referencia a su antecesor. Cuando un nuevo servidor llega, contacta con algún participante en el anillo y le pide unirse. Ese a su vez le proporciona sus referencias hacia adelante y actualiza al recién llegado como su nuevo sucesor.

Además todos los nodos, al unirse al anillo crean un subproceso de supervisión para monitorear la consistencia del anillo y enviar informes periódicos a su sucesor. Si un servidor se desconecta y es detectado por este mecanismo, se envía un mensaje de alerta por todo el anillo y cada nodo incluye su información en el mensaje al reenviarlo. Cuando el mensaje vuelve al que detectó la falla (que se convierte en un Manager temporal), se contabilizan la cantidad de nodos en el anillo de cada role. Si se alcanza el mínimo en alguno, se envía un mensaje de Desbalance por el anillo, explicando que role es necesario. Cuando el primer servidor con el role opuesto al necesario recibe el mensaje, cambia de role para mantener la disponibilidad y funcionalidad del sistema distribuidado.

Características generales de los servidores:

- **Concurrencia**: Los servidores tienen la capacidad de manejar varias solicitudes simultáneamente gracias a su diseño multiproceso. Al ser creados y establecer su role, designan un proceso que espera solicitudes y ante la llegada de una petición se crea otro proceso que se encarga de manejarla.
- **Localización**: El punto de entrada para las conexiones es desconocido y se resuelve mediante un servicio de nombres (lo que implica que todos conozcan la dirección del DNS).
- **Control**: Hay un canal de comunicación claramente definido entre los servidores y los usuarios, y los usuarios esperan una respuesta antes de volver a intentar comunicarse. Si reciben una respuesta, deben enviar un "acknowledgement" (ACK) para confirmar que han recibido la información correctamente.
- **Estado interno**: Los servidores carecen de información interna relacionada con el estado de los usuarios o conversaciones previas. Por tanto, no almacenan información personalizada sobre los usuarios.

Servidor de nombre (DNS)

El nodo de DNS gestiona registros de tipo A. Un registro A posee los siguientes campos:

```
label (nombre del recurso, en nuestro caso el dominio `dispotify.data`,
`dispotify.router`, etc );

tipo (tipo de información, en nuestro caso A, que significa IPv4);

clase (usualmente omitida ya que se utiliza solo `IN`);

TTL (time to live, cantidad de tiempo que debe ser guardado el registro) y

datos (datos reales del registro, osea, dirección: (IP, puerto) ).
```

Al inicializarce una instancia se crea un diccionario de 'headers' que por cada solicitud que puede recibir tiene como valor la función que la maneja. También empieza un subproceso para actualizar los registros basándose en sus valores de TTL.

Coordinación

Decidimos implementar una arquitectura descentralizada de anillo sin líder (explicada anteriormente). Dónde cualquiera puede insertar nuevos nodos al anillo y puede ejercer de Manager temporal si detecta una desconexión. Esta arquitectura se enfoca en evitar la introducción de más fuentes de falla; minimizando los posibles problemas de estabilidad que surgen de modelos centralizados, como sería la inclusión de un líder.

Transacciones en la base de datos

Los nodos de acceso a datos utilizan operaciones de transacción al acceder y modificar la base de datos. Por ejemplo al insertar valores solo se hace commit luego de que todas las operaciones indicadas se hallan ejecutado exitosamente, si falla alguna inserción no se hace commit y se indica el error que ocurrió.

```
try:
    connection = sqlite3.connect(data_base_file_path)
    cursor = connection.cursor()
    print("Connected to SQLite")

    sqlite_query = "INSERT INTO " + table_name + " ( " + columns_names + " )
VALUES "

    value = '(' + ('?,'*(len(row_tuples_tuple[0])-1)) + '?);'

    insert = sqlite_query + value
    cursor.executemany(insert, row_tuples_tuple)
    connection.commit()
    print("Values inserted successfully into the table")
    cursor.close()

except sqlite3.Error as error:
    print("Failed to insert data into sqlite table:",error)
finally:
    if connection:
        connection.close()
        print("The sqlite connection is closed")
```

Replicación y consistencia

Al unirse al anillo, un servidor de datos, se comunica con alguno de los servidores de datos existentes para replicar los datos. Para mantener la consistencia y el sistema funcional ntentamos garantizar que existan al menos 2 servidores de datos en todo momento, con los datos replicados. De esta forma si alguno se desconecta, aún persisten los datos importantes y a través del anillo se cambian los roles de ser necesario para mantener 2 nodos de datos.

Grupos

La principal aproximación para tolerar la falla de procesos es organizar varios procesos idénticos en grupos. Nuestra implementación es de grupos planos, no jerárquicos. Ante una falla, aunque el grupo tenga un integrante menos continua su funcionamiento, a menos que quede vacío, por supuesto. Tanto los nodos de datos como los routers tienen un grupo. De forma que cuando se quiere solicitar algo a uno de estos roles se pide al DNS la dirección del grupo, y este retorna una lista con todos los integrantes activos en el momento, en el grupo. De esta forma se pueden hacer peticiones a uno o más miembros del grupo, de forma iterativa o simultánea.

Balance de cargas

Implementamos técnicas de balance de cargas en varios momentos. En la instancia más abstracta, tenemos los grupos de roles para que las conexiones no recaigan sobre solo un nodo, ya que cualquier integrante del grupo puede responder igual a las peticiones. Además, al elegir a cuál servidor solicitar la petición siempre se selecciona estocásticamente con distribución uniforme, intentando que cada integrante del grupo tenga la misma carga.

Tolerancia a fallas

La tolerancia a fallas está fuertemente relacionada con los Sistemas Dependientes. La dependencia cubre un conjunto de requerimientos donde se incluyen los siguientes:

- Disponibilidad : un sistema altamente disponible es aquel se encuentra trabajando en cualquier instante de tiempo. Nosotros garantizamos disponibilidad siempre que exista al menos un nodo en cada rol.
- Confiabilidad : la confiabilidad está definida en términos de intervalos de tiempo en vez de instantes de tiempo. Un sistema altamente confiable es aquel que puede funcionar sin interrupción por largos períodos de tiempo. Dado que depende de la estabilidad de la conexión solo podemos garantizar que tenemos mecanismos de reintentar las peticiones varias veces antes de darle a conocer al usuario que hubo fallo. Introduciento cierta medida de transparencia y dando oportunidad a que el sistema falle y pueda reconectarse y completar la tarea.
- Seguridad : se refiere a la posibilidad de un sistema de que a pesar de fallar temporalmente en su correcto funcionamiento no genera situaciones catastróficas. Dado que pensamos en muchísimas de las formas en que puede fallar el sistema y en todas o casi todas tenemos mecanismos de detección de fallas; los errores no suelen ser inesperados y existen medidas para frenar las consecuencias y que no sean catástroficas como borrar arbitrariamente datos o que se cierren solos los servidores.
- Mantenibilidad : se refiere a la facilidad con que un sistema con fallas puede ser reparado. En este apartado tenemos varias medidas. Por ejemplo al enviar datos, solo se considera correcto el envio si se envió correctamente cada fragmento de información y además se

recibió el ACK. Si algo falla se espera un tiempo y se vuelve a intentar el envio, creando la oportunidad para que el sistema se repare y la operación ocurra correctamente y de forma transparente. Además, al tener una noción de grupos, si la operación falla con uno de los integrantes, se puede intentar con otro que esté funcional. Esta medida de respaldo funciona gracias a que el DNS no brinda una sola dirección, sino varias, permitiendo intentar por todas ellas si alguna no funciona. Entre las técnicas empleadas está la redundancia de tiempo y redundancia física.

En otro tema de tolerancia a fallas está la comunicación punto a punto:

La comunicación en el sistema es a través de socket TCP, los cuáles poseen sus propios mecanismos de tolerancia a fallas. Sin embargo en casos donde la conexión TCP se cierra abruptamente producto de una falla externa a la conexión, nuestro sistema detecta la falla y dependiendo del contexto; crea una nueva conexión, espera un tiempo antes, decide manejarlo como un error y envía la información de error, etc.

Seguridad

Aunque no indagamos mucho en el apartado de seguridad, ya que no implementamos un sistema de autetificación explícito, no lo dejamos totalmente de lado. Toda la información que se envía está codificada, de esta forma si una entidad ajena recibe un paquete, no debe ser capaz de entenderlo. Por otro lado, solo se puede unir al sistema y ser reconocido por él, entidades que hablen el "idioma" que hablan nuestros nodos. Ya que solo así se puede saber la naturaleza de la petición, e incluso saber identificar el final de un mensaje y el inicio de otro.