José Treviño Olvera

Proyecto final: Red de distribución de contenido

En este proyecto se desarrollará un primer acercamiento a una red de distribución de contenido utilizando microservicios contenerizados en la plataforma Docker.

El servicio tendrá los siguientes elementos:

- Un cliente para generar contenido en forma de archivos de texto y que sea capaz de comunicarse con el microservicio distribuidor del contenido.
- Tres nodos que se encargarán de almacenar el contenido en un volumen particular además de tener la capacidad de reportar las estadísticas del uso de sus recursos.
- Un microservicio distribuidor que tendrá la capacidad de obtener el reporte de estadísticas de los nodos de almacenamiento y tomar una decisión basado en la utilización de los recursos para asignar el nodo en el que se va a almacenar el contenido.
- El nodo distribuidor deberá ser capaz de manejar los metadatos para mantener un registro del contenido y que sea posible ubicar un archivo luego de ser almacenado.
- Los nodos de almacenamiento harán peticiones para crear, obtener, actualizar y eliminar el registro de los metadatos de los contenidos.

La figura 1 muestra un diagrama de la arquitectura del servicio.

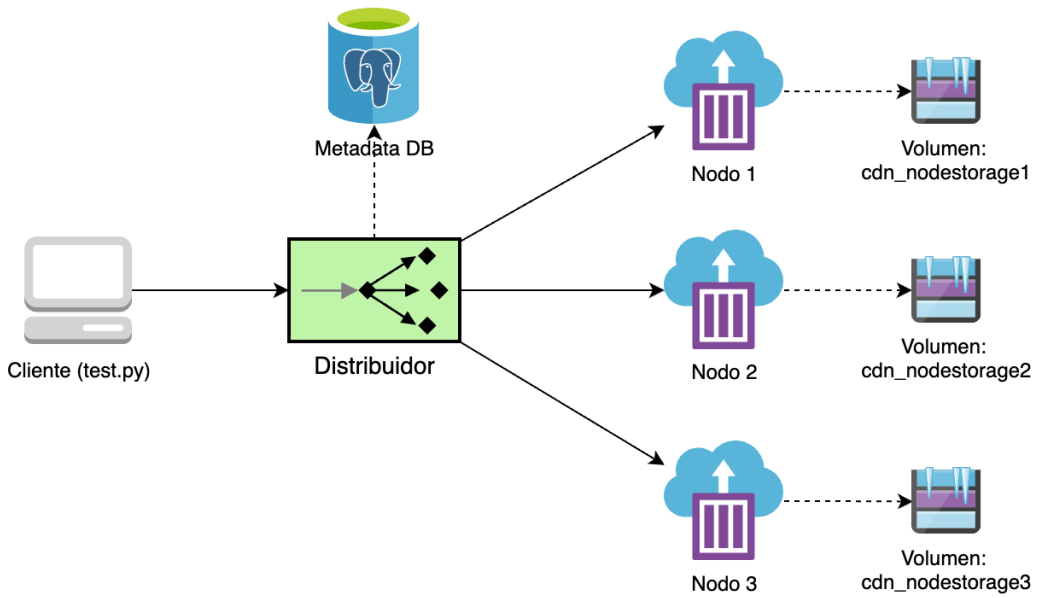


Figura 1: Arquitectura del servicio.

1. Propuesta de solución

Se desarrollaron los microservicios necesarios para el diseño de la arquitectura del servicio implementando API's que se comunican con el protocolo HTTP utilizando FastAPI el cual es un framework moderno de desarrollo de alto desempeño para construir aplicaciones web mediante API's en el lenguaje de programación Python.

Los microservicios se empaquetaron en contenedores y se utilizó la herramienta Docker Compose para establecer un archivo de configuración de los contenedores de la aplicación desarrollada además de permitir arrancar todos los contenedores asociados.

Tecnologías utilizadas:

- Plataforma de contenedores: Docker.
- Framework de desarrollo: FastAPI (https://fastapi.tiangolo.com/) versión 0.68.0.
- Base de datos: PostgreSQL versión 14.1-alpine.
- Python versión 3.9 con librería requests.

La figura 2 muestra la estructura del proyecto desarrollado.

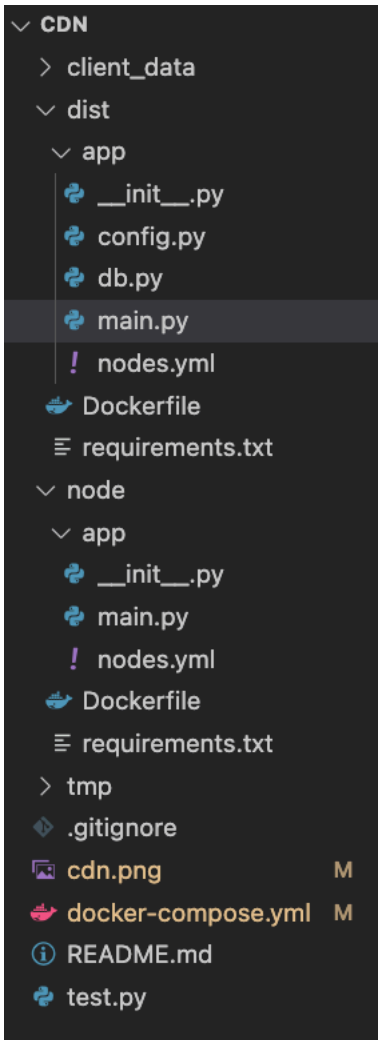


Figura 2: Estructura del proyecto.

Directorio o archivo	Descripción
client_data/	Directorio que almacena los archivos generados por el cliente (test.py).
dist/	Directorio del código de fuente del microservicio distribuidor de carga que gestiona los metadatos de los contenidos.
dist/app/nodes.yml	Archivo de configuración para el host y puerto de los nodos del servicio.
node/	Directorio del código de fuente del microservicio del nodo de almacenamiento de contenidos.
node/app/nodes.yml	Archivo de configuración para el host y puerto del microservicio distribuidor.
test.py	Cliente que realiza la generación de archivos, subida, actualización, descarga y eliminación de contenido.
tmp/	Directorio en donde se descarga el contenido que solicita el cliente (test.py)
docker-compose.yml	Archivo de configuración Docker Compose en donde se configuran los contenedores de los microservicios.

En el archivo de configuración (docker-compose.yml) se declararon un total de 4 volúmenes de Docker con la siguiente configuración por defecto:

```
volumes:
  db:
    driver: local

  nodestorage1:

  nodestorage2:

  nodestorage3:
```

El volumen “db” corresponde a la base de datos en donde se almacenan los metadatos.

Los volúmenes “*nodestorage1*”, “*nodestorage2*”, “*nodestorage3*” corresponden a los volúmenes en donde se almacenarán los archivos para el nodo 1, nodo 2 y nodo 3 respectivamente.

Se utilizó PostgreSQL como sistema gestor de base de datos para almacenar los metadatos del contenido con la siguiente estructura:

Tabla: content	
Campo	**Atributos**
id	INTEGER PK AUTO
filename	STR NULLABLE
size	INTEGER DEFAULT = 0
created_at	DATETIME TIMEZONE=TRUE DEFAULT=CURRENT

La base de datos se definió como el servicio “db” en la configuración del Docker Compose:

```
db:
    image: postgres:14.1-alpine
    restart: always
    environment:
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
      - POSTGRES_DB=cdn_metadata
    ports:
      - '5432:5432'
    volumes:
      - db:/var/lib/postgresql/data
```

1.1 Microservicio de distribución

Este microservicio se encarga tanto de distribuir el almacenamiento de los contenidos del cliente como de la gestión de los metadatos del contenido.

Se asignó el nombre de “dist” al contenedor de este microservicio y se utilizó la siguiente configuración en el Docker Compose:

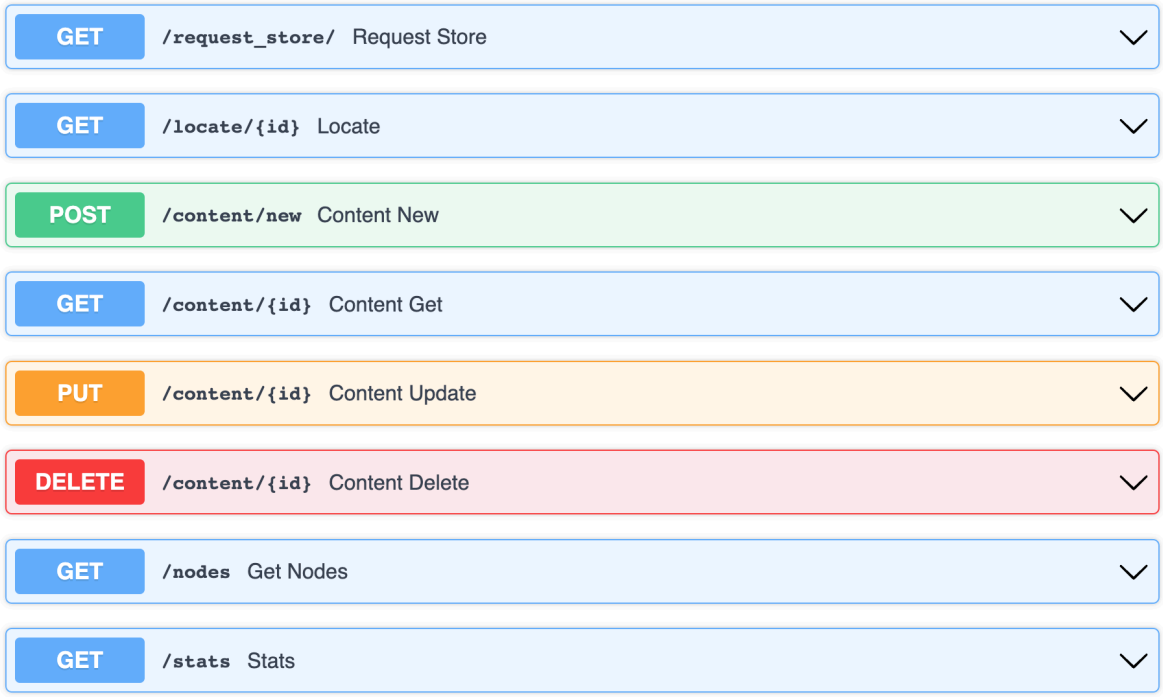
```
dist:
    image: "cdn-dist-img"
    build: dist/
    ports:
      - "8000:80"
    environment:
      - DATABASE_URL=postgresql://postgres:postgres@db:5432/cdn_metadata

    depends_on:
      - db
```

Este microservicio requiere que la base de datos esté funcionando por ello se declara la cláusula “depends_on” además de pasar la configuración de acceso a la base de datos mediante una variable de entorno.

José Treviño Olvera

Se programaron las siguientes rutas en el API:



Ruta	Descripción
request_store/ (GET)	Se utiliza para solicitar una URL para subir un nuevo contenido. El distribuidor recopila las estadísticas de utilización de los recursos de cada uno de los nodos de almacenamiento y toma la decisión de almacenarlo en el nodo que tenga menor porcentaje de utilización. Antes de considerar a un nodo como candidato, el distribuidor comprueba que el nodo tenga la capacidad de almacenar el archivo. Ejemplo: `{ "url": "http://192.168.0.4:8002", "status": "OK", "message": "Store URL assigned." }`
locate/{id} (GET)	Retorna el URL del nodo de almacenamiento en donde se encuentra el contenido solicitado asociado al identificador como argumento. `{ "uri": "http://192.168.0.4:8003" }`
content/new (POST)	Crea una nueva fila en el registro de los metadatos y retorna un objeto JSON con el identificador del contenido.

<table>
<tr><td>content/{id}
(GET)</td><td>Obtiene la fila en el registro de los metadatos y lo retorna como un objeto JSON.

Ejemplo:
<pre>{
  "id": 33,
  "filename": "4fe91914-2760-11ed-bf73-86a8a97b280d",
  "size": 3072,
  "uri": "http://192.168.0.4:8003",
  "created_date": "2022-08-29T06:03:24.181883+00:00",
  "status": "OK"
}</pre></td></tr>
<tr><td>content/{id}
(PUT)</td><td>Actualiza el valor de los metadatos asociados a un contenido identificado por el “id” como argumento. Recibe como parámetros el valor de los atributos:
filename, size, uri.</td></tr>
<tr><td>content/{id}
(DELETE)</td><td>Elimina la fila en el registro de los metadatos.</td></tr>
<tr><td>nodes/
(GET)</td><td>Retorna un objeto JSON con la información de los nodos de almacenamiento asociados.

Ejemplo:
<pre>{
  "nodes": {
    "dist": {
      "host": "192.168.0.4",
      "port": 8000
    },
    "node1": {
      "host": "192.168.0.4",
      "port": 8001
    },
    "node2": {
      "host": "192.168.0.4",
      "port": 8002
    },
    "node3": {
      "host": "192.168.0.4",
      "port": 8003
    }
  }
}</pre></td></tr>
<tr><td>stats/
(GET)</td><td>Retorna un objeto JSON con las estadísticas de utilización de memoria y almacenamiento de cada uno de los nodos asociados.

Ejemplo:</td></tr>
</table>

```
{
  "stats": [
    {
      "memory": 32296960,
      "storage": 15703152,
      "storagep": 15.703152,
      "max_storage": 100000000,
      "node": "http://192.168.0.4:8001"
    },
    {
      "memory": 25157632,
      "storage": 7281695,
      "storagep": 7.281695,
      "max_storage": 100000000,
      "node": "http://192.168.0.4:8002"
    },
    {
      "memory": 32616448,
      "storage": 16441739,
      "storagep": 16.441739,
      "max_storage": 100000000,
      "node": "http://192.168.0.4:8003"
    }
  ]
```


1.2 Microservicio de almacenamiento

Este microservicio se encarga de almacenar archivos.

Se asignó el nombre de “nodeX” al contenedor de este microservicio y se utilizó la siguiente configuración en el Docker Compose:

```
node1:
    image: "cdn-node-img"
    build: node/
    ports:
      - "8001:80"
    volumes:
      - nodestorage1:/storage

    environment:
      # SET max storage
      MAX_STORAGE: 100M
```

Se especifica el tamaño máximo del almacenamiento del nodo mediante la variable de entorno “MAX_STORAGE”. El valor de esta variable se puede introducir en formato de bytes legible por el humano, por ejemplo: “100GB”. Se implementó una librería para realizar la conversión de formato humano a bytes.

José Treviño Olvera

Se programaron las siguientes rutas en el API:

Ruta	Descripción
store/ (POST)	Ruta para subir un nuevo archivo para ser almacenado. Retorna un objeto JSON con el estado de la operación y el identificador del nuevo contenido en caso de ser exitoso, de lo contrario retorna un mensaje de error.
retrieve/{id} (GET)	Responde con el archivo asociado al contenido con el identificador de contenido "id" como argumento.
update/ (PUT)	Reemplaza un nuevo archivo asociado a un contenido previamente existente y se recibe el identificador de contenido como parámetro.
delete/{id} (DELETE)	Elimina un archivo asociado a un contenido existente identificado con el argumento "id".
stats/ (GET)	Obtiene las estadísticas de uso de la memoria y almacenamiento y las retorna en un objeto JSON. Ejemplo: `{ "memory": 32292864, "storage": 15703152, "storagep": 15.703152, "max_storage": 100000000 }` Atributos: • "memory": Uso de memoria RAM en bytes. • "storage": Uso del almacenamiento en bytes. • "storagep": Uso del almacenamiento en porcentaje %. • "max_storage": Capacidad máxima de almacenamiento en bytes.

José Treviño Olvera

2. Escenario

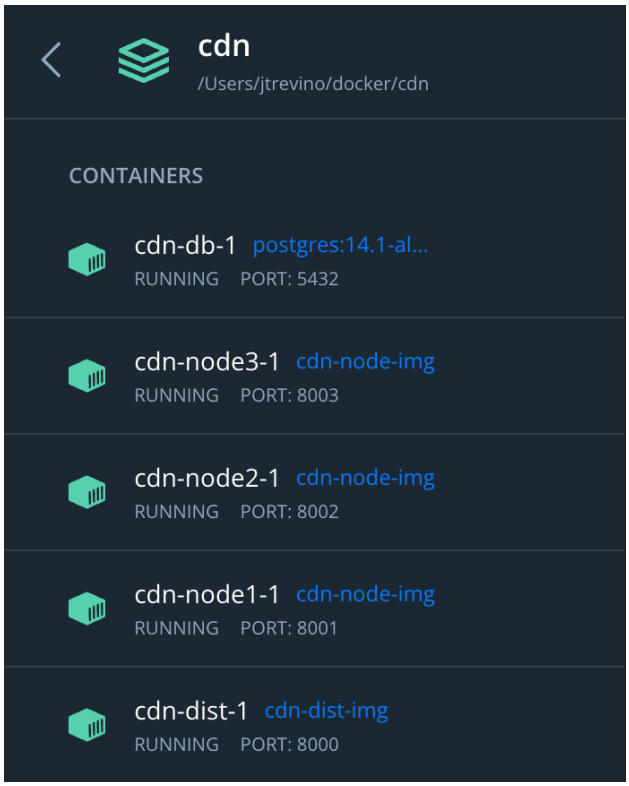
La prueba del servicio se realizó en una sola computadora con las siguientes características y con los requerimientos de software previamente instalados:

Host

Sistema operativo	MacOS 12.3.1
Memoria	16 GB
Almacenamiento	512 GB SSD
Procesador	Apple M1 8 cores (ARM)

3. Evidencias

Contenedores corriendo:



Ejecución del cliente (test.py):

```
→  cdn git:(main) ✗ python test.py
File generated:  client_data/4d18ebe2-2760-11ed-bf73-86a8a97b280d
Store URL:  http://192.168.0.4:8002/store
{'id': 33, 'message': 'Content successfully uploaded. ID: 33'}
{'date': 'Mon, 29 Aug 2022 06:03:31 GMT', 'server': 'uvicorn', 'content-type': 'application/octet-stream', 'content-disposition':
'attachment; filename="4d18ebe2-2760-11ed-bf73-86a8a97b280d"', 'content-length': '9567760', 'last-modified': 'Mon, 29 Aug 2022 06:
03:32 GMT', 'etag': '32d6d5c9da4ffd389be25044f597f121'}
File retrieved. Stored at:  tmp/4d18ebe2-2760-11ed-bf73-86a8a97b280d
File generated:  client_data/4fe91914-2760-11ed-bf73-86a8a97b280d
New Store URL:  http://192.168.0.4:8003/update
{'id': 33, 'message': 'Content successfully updated. ID: 33'}
{'uri': 'http://192.168.0.4:8003'}
Updated file located at:  http://192.168.0.4:8003
```

José Treviño Olvera

Uso del API para obtener las estadísticas de uso del primer nodo:

Responses

Curl

```
curl -X 'GET' \
  'http://127.0.0.1:8001/stats/' \
  -H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:8001/stats/
```

Server response

Code	Details
200	**Response body**

```
{
  "memory": 32292864,
  "storage": 15703152,
  "storagep": 15.703152,
  "max_storage": 100000000
}
```

Download

Response headers

```
content-length: 83
content-type: application/json
date: Mon,29 Aug 2022 08:02:42 GMT
server: uvicorn
```

Responses

Code	Description	Links
200	Successful Response	*No links*