Aplicaciones Escalables y Orquestación de Servicios

Nicolás Aguado*

Diseño y Proyectos de Redes — 10 de diciembre de 2024

1. Introducción

Se propone un sistema basado en aplicaciones (replicadas) virtualizadas mediante `Docker` [5]; orquestadas y enlazadas entre sí de manera programática mediante `Docker Compose` [4].

Se describe un sistema con una amplia gama de aplicaciones, tales como balanceadores de carga, servicios *API REST*, almacenaje de datos, etc.

2. Descripción General

En la Figura 1. se expone el esquema general del sistema orquestado. El sistema está basado en un *reverse proxy* escuchando a peticiones, con dos servicios expuestos en dos rutas posibles.

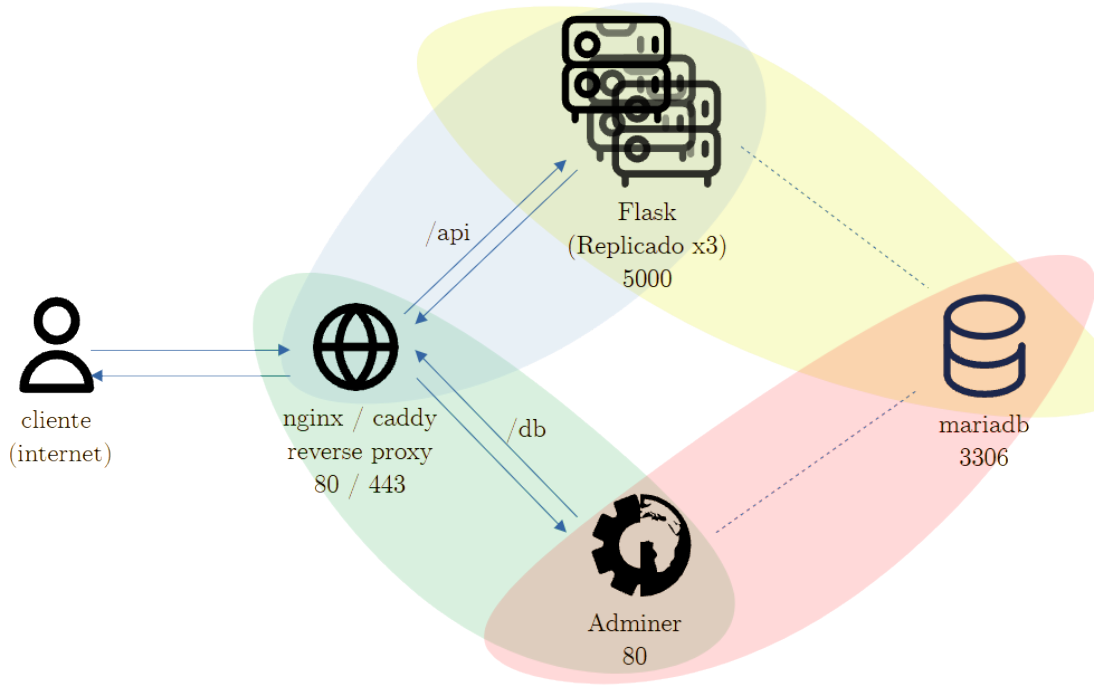

Figura 1: Diagrama del Sistema

*nico@nico.eus - Facultad de Informática - UPV/EHU

Por la rama `/api` se encuentra disponible un servicio *API REST*. Esta interfaz tiene diversos endpoints disponibles en los que se opera con un sistema de "mensajes". Se pueden publicar nuevos mensajes y se pueden consultar los ya recibidos. Este sistema está basado en el *framework* de Python `Flask` y almacena los datos en una base de datos `mariadb`, también incluida dentro del sistema orquestado.

Por la otra rama (`/db`), se encuentra una instancia de un servidor web (Apache y PHP) corriendo la aplicación `adminer`. Esta aplicación es, precisamente, un panel de control para operar más fácilmente con bases de datos y se usa para administrar la base de datos previamente mencionada.

Finalmente, se exponen una serie de redes internas que interconectan a los servicios entre sí, señaladas en la Figura 1 con distintos colores.

3. Servicios

A continuación se propone una descripción más detallada de la composición y orquestación de las aplicaciones contenidas en el sistema. También se argumentan las decisiones de diseño y se describen los aspectos que se consideran importantes.

3.1. Base de Datos

Inicialmente, de cara a poder llevar una buena gestión de los datos, se propone orquestar una base de datos `mysql`. Ésta, en principio, aunque pueda parecer una buena idea, presenta ciertas limitaciones. La orquestación en docker de `mysql` puede dar lugar ciertos problemas de rendimiento, además de que la propia orquestación en sí es ciertamente compleja y contradictoria en algunos aspectos.

Esto tiene una explicación muy sencilla. `Mysql` es una aplicación de pago [9], propiedad de Oracle Corporation. Existe una versión denominada `Mysql Community Edition` [10], que es gratuita, pero muy limitada. Esto es así porque esta imagen realmente no está mantenida por la comunidad *open-source*. La comunidad *open-source* hizo un *fork* de `mysql` antes de que fuera comprada por Oracle Corporation (año 2009), y ésta se denominó `mariadb` [7]. El *fork* es mucho más completo en cuanto a funcionalidad, rendimiento y en general tolerancia a fallos y facilidad de configuración. Además, es enteramente compatible con los principales drivers de `mysql` para los distintos lenguajes de programación, ya que su API es -por diseño- la misma.

En resumen, y usando términos más coloquiales, Oracle Corporation ya tiene un sistema de bases de datos (`Oracle Database` [11]), y por tanto no le dedica el mismo ″cariño″ a `Mysql`, ni mucho menos a `Mysql Community Edition`.

Entonces, a la hora de orquestar una base de datos para el sistema, se ha optado por la opción de usar `mariadb`. El apartado relativo a la configuración se detalla a continuación.

docker-compose.yml

```
dprnico-mysql:
image: mariadb:latest
volumes:
- ./dbdata:/var/lib/mysql
- ./db-init/startcmd.sql:/docker-entrypoint-initdb.d/cmd.sql
environment:
- MARIADB_ROOT_PASSWORD=reallyXlong1mysql2root3
- MARIADB_USER=dbuserwriter
- MARIADB_PASSWORD=P4ssw0rd
- MARIADB_DATABASE=dprdb
restart: unless-stopped
command: --collation-server=utf8mb4_unicode_ci
networks:
- adminer-db
- flask-db
```

Como se desea persistir datos entre ejecuciones del contenedor (la propia base de datos en sí), se desea recalcar que se monta un volumen en la carpeta local `dbdata`. Además, de cara a que en la primera ejecución se creen las bases de datos con las tablas y su estructura inicial se provee un fichero `.sql` con todos los comandos SQL necesarios para la inicialización.

Además, para personalizar la instalación se definen 4 variables de entorno mediante las que definir la contraseña del usuario root y un usuario (*dbuserwriter*) y contraseña (*P4ssw0rd*) con acceso completo a una base de datos especificada. De cara a mejorar la tolerancia a fallos, se incorpora también la opción de reinicio automático del contenedor.

A modo de detalle extra, en las últimas versiones de `mariadb` pueden llegar a existir problemas de compatibilidades con los drivers y sus conjuntos de caracteres predeterminados [6]. Para todo el sistema se ha definido el conjunto *utf8mb4* y el *flag* que se añade al comando de ejecución lo refleja en consecuencia.

3.2. API Rest

Se expone un servicio web de tipo *API REST* para poder interactuar con la base de datos al que se accede en la ruta `/api/`. Se ha diseñado un servicio sencillo de mensajería (enviar mensajes al servidor, leer mensajes en el servidor). Para esta implementación, se ha usado el framework de python `flask` [13].

3.2.1. Diseño de la Aplicación

Usando el framework de python `flask`, es una tarea relativamente sencilla el diseñar un servicio *API REST* con 3 endpoints para la mensajería. La implementación relativa a este apartado se encuentra en el fichero `flask-files/app.py`

El primer endpoint (`GET /api/test`) sirve como prueba, simplemente devuelve el texto ″ALIVE″ y el hostname de la máquina.

El segundo endpoint (`POST /api/message`) es el método de entrada de mensajes en la aplicación. Acepta dos parámetros dentro del *body*: ″From″ y ″Content″. Si uno de los parámetros

no existe o esté mal formado en la petición, devuelve el error oportuno. Si en cambio va todo bien, inserta el mensaje proporcionado dentro de la base de datos.

El tercer endpoint (`GET /api/message`) es la operación contraria al endpoint previo. Esto es, sirve para obtener los mensajes guardados en (la base de datos d)el servidor. Si en la petición se incluye alguno (o todos) de los parámetros ″From″, ″Content″ ó ″ID″ entonces se hace un filtrado.

3.2.2. Orquestación de la Aplicación

Para el despliegue de la aplicación python de tipo `flask` dentro de un contenedor se ha optado por realizar una construcción a medida. Básicamente, se parte desde una base mínima de python para luego instalar las dependencias necesarias, exponer el puerto correspondiente (5000) y copiar la aplicación dentro del contenedor. Así se obtiene una imagen completa y lista para el despliegue en producción. El *Dockerfile* correspondiente se expone a continuación.

Dockerfile-flask

```
FROM python:3-slim
WORKDIR /app
COPY ./flask-files/app.py /app/app.py
COPY ./flask-files/requirements.txt /app/req.txt
RUN pip install --trusted-host pypi.python.org -r req.txt
EXPOSE 5000
CMD ["gunicorn","-b" ,"0.0.0.0:5000", "app:app"]
```

A modo de preferencia subjetiva, se ha decidido usar como servidor que sirve la aplicación `flask` al servidor `gunicorn` [2]; ya que el servidor integrado que trae éste primero, tal y como se indica al ejecutarlo y en su documentación [12], no es adecuado para su uso en producción y por tanto no se debe de usar en imágenes finales.

Además, se propone una implementación en la que el servicio web está replicado (en concreto, en hasta 3 réplicas). El sistema propuesto no requiere de un almacenamiento de sesiones, así que se usa al propio docker como balanceador de carga. Al acceder al servicio por su nombre DNS, (en este caso, `dprnico-flask`) es el propio docker el que reparte las peticiones entre las réplicas disponibles. El apartado relativo a la configuración se detalla a continuación.

docker-compose.yml

```
dprnico-flask:
build:
dockerfile: flask-files/Dockerfile-flask
restart: unless-stopped
deploy:
mode: replicated
replicas: 3
depends_on:
- dprnico-mysql
networks:
- nginx-flask
- flask-db
```

En este caso, se especifica la construcción de la imagen en el *Dockerfile* mencionado anteriormente y se establece la política de reinicio.

3.3. Aplicación Gestión de BBDD

En la otra ruta (`/db/`) [1] se establece la aplicación de gestión `adminer` para la administración de la base de datos.

3.3.1. Construcción de la Imagen

El proyecto `adminer` se encuentra abandonado y desactualizado, así que se usa el *fork* mantenido por la comunidad `adminerevo` [3] en su lugar. `Adminerevo` (y `adminer`) son sistemas de administración de bases de datos basados en un único fichero PHP.

Para ejecutar un fichero PHP es necesario tener instalado el motor base de *php*, un soporte para renderizado de php en la web (normalmente *fastcgi*) y un servidor web y/o reverse proxy que se encargue de las peticiones HTTP. En el sistema completo conjunto ya se dispone de un reverse proxy (Sección 3.5), pero con motivo de mantener la ″independencia″ entre servicios éste no se usará para esta aplicación y en cambio se ha diseñado a medida una imagen con los tres elementos mencionados.

Dockerfile-adminer

```
FROM php:8.2-apache
COPY ./adminer-files/adminer-4.8.4.php
      /var/www/html/db/adminer.php
COPY ./adminer-files/custom-adminer.php
      /var/www/html/db/index.php

# Instalar extensión
RUN docker-php-ext-install mysqli pdo_mysql
# Activar extensión
RUN mv /usr/local/etc/php/php.ini-production
      /usr/local/etc/php/php.ini
RUN sed -i 's/;extension=mysqli/extension=mysqli/'
      /usr/local/etc/php/php.ini

EXPOSE 80
# No es necesaria una instrucción CMD; se hereda
```

Como punto de partida se obtiene la imagen con el servidor web *Apache* junto con *PHP* y *PHP-FastCGI*. De ahí, se copian los archivos necesarios para la ejecución de `adminerevo` en la ruta adecuada del servidor web.

Después, como la aplicación necesita conectarse a una base de datos *mysql*, se instala el plugin de *mysql* para *php* (usando la utilidad que ya viene en la imagen base, *docker-php-ext-install*). Una vez instalada la extensión, ésta se activa en el fichero de configuración *php.ini*. Finalmente, una vez expuesto el puerto necesario, ya se obtiene una imagen lista para producción.

[1] Nótese la barra final / después de *db*.

3.4. Orquestación de la Aplicación

Para orquestar la aplicación se usa una configuración parecida a las ya expuestas anteriormente (Secciones 3.2.2 y 3.1), construyendo la imagen manualmente y con una política de reinicio.

docker-compose.yml

```
dprnico-adminer:
build:
dockerfile: adminer-files/Dockerfile-adminer
restart: unless-stopped
depends_on:
- dprnico-mysql
networks:
- nginx-adminer
- adminer-db
```

3.5. *Reverse Proxy*

Como punto de acceso desde el exterior se propone un servicio de *reverse proxy* que se encargue de recibir los paquetes y de reenviarlos a su destino final.

Por la ruta `/db` se reenvian los paquetes al servicio de gestión de la BBDD (Sección 3.3) y por la ruta `/api` se reenvian los paquetes a (las replicas del) servicio *API REST* (Sección 3.2).

Como la elección del servicio de reverse proxy es algo meramente subjetivo y que muchas veces depende de la infraestructura ya existente, se proporcionan configuraciones para dos de las aplicaciones más populares:

- **nginx** : Aplicación con una larga trayectoria, ya establecida y de las más probadas para servicios en producción. Fichero de configuración en `nginx-files/nginx.conf`.
- **caddy** : Aplicación relativamente nueva, rompedora en el mercado por su facilidad de uso y su integración automática de TLS en nombres de dominio, ya ha obtenido una cuota relativamente alta del mercado. Fichero de configuración en `caddy-init/Caddyfile`

A grandes rasgos, ambas configuraciones son muy parecidas.

A modo de detalle, se desea mencionar que en ambas configuraciones se ha hecho explícita la eliminación del header *Server* y del header *X-Powered-By*. El hecho de revelar **exactamente** cuál es la versión del servicio que se está ejecutando en cada caso puede suponer un grave fallo de seguridad. En caso de no realizar un mantenimiento adecuado y de no tener siempre la última versión de todos los servicios (muchas veces esto no es posible), el revelar el número de versión es dar un pase gratuito a los atacantes.

Entonces, sin ningún otro motivo particular, aquí se expone el fichero de configuración de la aplicación `nginx`.

nginx.conf

```
server {
listen 80;
listen [::]:80;
server_name _;

server_tokens off;
proxy_hide_header X-Powered-By;

location /db {
proxy_pass http://dprnico-adminer:80;
proxy_set_header Host $host;
proxy_set_header X-Real-IP $remote_addr;
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
}

location /api {
proxy_pass http://dprnico-flask:5000;
proxy_set_header Host $host;
proxy_set_header X-Real-IP $remote_addr;
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
}

# Default location to return if no matching route is found
location / {
return 404;
}
}
```

De cara a la orquestación de este servicio, los ficheros de configuración correspondientes tienen los puertos 80 y 443 expuestos al exterior, junto con sus volúmenes correspondientes para la incorporación de los ficheros de configuración. Se ha optado por montar los ficheros de configuración en un volumen en lugar de construir una imagen personalizada por la alta volatilidad y las muchas modificaciones que éstos pueden surgir en un entorno activo.

Además, en el caso del *reverse proxy* `caddy` se provee un volumen para el almacenamiento persistente de los certificados TLS. El apartado de configuración para la orquestación automática de ambos servicios se encuentra en el fichero `docker-compose.yml`

4. Despliegue en Local

Para desplegar todos los servicios descritos en una máquina local es suficiente con extraer los contenidos del fichero *zip* adjunto y ejecutar el siguiente comando en una terminal *UNIX-like*

bash

```
$ sudo docker compose up -d
```

Por defecto, el *reverse-proxy* está escuchando en todas las interfaces de red en el puerto 80, de modo que es suficiente con acceder a `http://localhost/api/` para interactuar con la interfaz API (Sección 3.2) y con acceder a `http://localhost/db/` para interactuar con el servicio de `adminerevo` (Sección 3.3)

5. Extra

A parte de todas las consideraciones y todos los detalles que se han mencionado anteriormente, se desea exponer una serie de consideraciones presentes en la orquestación propuesta.

5.1. Sistema de Redes Aisladas

De cara a mejorar la seguridad del sistema en su conjunto, se encuentra creado un sistema de redes aisladas entre sí (Véase formas coloreadas en la Figura 1).

El sistema se encuentra configurado de manera que cada aplicación solo tiene acceso mediante la red a las aplicaciones mínimas requeridas con las que se tiene que comunicar (política de aislamiento máximo). Además, el único servicio que se encontrará expuesto al exterior será el *reverse-proxy* (Sección 3.5). Esta estructura de red se consigue usando un conjunto de redes privadas internas interconectando servicios.

De este modo, todas las conexiones tienen que pasar por el reverse proxy, y a su vez, por la ruta indicada. No se puede acceder directamente al servicio *REST API*, ni mucho menos a la BBDD.

Por ejemplo, se provee un extracto del apartado de configuración para el servicio *API REST*. Se puede observar que no tiene acceso al exterior (no está en la red *public-gateway*) y que sólo se podrá comunicar con el *reverse proxy* (mediante la red *nginx-flask*) y con la base de datos (mediante la red *flask-db*)

`docker-compose.yml`

```
...
networks:
- nginx-flask
- flask-db
...
```

5.2. *Autofill* en Adminer

De cara a una mejor gestión, y existiendo sólo una base de datos que se puede gestionar mediante el sistema `adminer` (descrito en la Sección 3.3) por su característica de aislamiento de redes, se propone un sistema en el que el *login* está parcialmente autorellenado (ignorando las consideraciones que seguridad que esto pueda suponer, en favor de una mayor facilidad de uso).

Siguiendo algunos ejemplos de los plugins para esta herramienta [1], se ha sobreescrito la función que genera el código HTML para el formulario de login en el fichero `adminer-custom.php` para que contenga los campos adecuados ya pre-rellenos.

Se puede comprobar esta funcionalidad accediendo por primera vez al servicio (`/db`)

5.3. Desplegado en la Nube

De cara al despliegue en producción del sistema, se ha optado por integrarlo dentro de una infraestructura ya existente.

Una de las máquinas disponibles para el despliegue es una con 4 CPUs compartidas y 4GB de RAM en la localidad de Núremberg (Alemania) con el proveedor de infraestructuras *netcup Gmbh* [8]. Esta máquina ya contiene varios contenedores docker pertenecientes a otros proyectos, y en concreto la ″fachada″ (punto de entrada) a todos ellos está gestionada por una instancia de `caddy`. Además, se dispone del dominio `nico.eus` en propiedad.

- Se ha creado un registro en el servidor DNS de tipo A apuntando hacia la IPv4 de la citada máquina.
- Se han incorporado las redes internas descritas en el fichero `docker-compose.yml` y las configuraciones de *reverse-proxy* descritas en `caddy-init/Caddyfile` para la instalación ya existente de `caddy`.
- Se ha modificado el fichero `docker-compose.yml`, eliminándo la parte relativa al *reverse proxy*.
- Se han copiado los ficheros del sistema a la máquina destino y se ha lanzado el despliegue con `docker-compose up -d`.

Así, el sistema ha sido desplegado en la dirección https://dpr.nico.eus

6. Posibles Problemas y Soluciones

A continuación se detallan algunos problemas que pueden llegar a surgir al configurar el sistema y las soluciones propuestas en cada caso.

- Tanto la imagen de Docker de `mysql` como la de `mariadb` NO son compatibles con el sistema de ficheros exFAT. Por tanto, al intentar ejecutar el sistema en memorias de almacenamiento volátiles, como lo pueden ser USBs o discos de estado sólido externos (donde este formato predomina), ocurrirán problemas. Se propone usar otro medio de almacenamiento formateado con otro sistema de ficheros.
- El *daemon* de Docker que ejecute el sistema deberá tener permisos de *superusuario* en sistemas UNIX, ya que se exponen los puertos reservados 80 y 443 y no se desea utilizar otro proceso como intermediario. Al mismo tiempo, si se obtiene algún error a la hora del despliegue, es conveniente verificar si estos puertos están siendo utilizados por algún otro proceso. El homólogo a esta configuración en sistemas Windows es aceptar los permisos de conexión a través del firewall para el proceso de Docker.
- Al acceder a las distintas rutas implementadas en la aplicación, se ha de considerar incluir la url completa con la barra al final. Por ejemplo, el acceder a la ruta `/db` puede dar problemas; y se deberá de acceder usando `/db/` en su lugar.
- A la hora de implementar una aplicación (o de realizar peticiones mediante *curl*) para usar la API , si se está usando un nombre de dominio, hay que tener cuidado de usar el nombre completo. Esto es, si se están haciendo peticiones a `example.com/api/`, insertar en su lugar `http://example.com/api/` para evitar problemas.

Referencias

[1] Adminer: Plugin Repository. https://www.adminer.org/en/plugins/, [Online; consultado 22-Nov-2024]

[2] Chesneau, B.: Gunicorn Python WSGI HTTP Server for UNIX. https://gunicorn.org/, [Online; consultado 22-Nov-2024]

[3] Contributors: AdminerEvo. https://github.com/adminerevo/adminerevo, [Online; consultado 29-Nov-2024]

[4] Docker: Docker Compose. https://docs.docker.com/compose/, [Online; consultado 21-Nov-2024]

[5] Docker: Docker Engine. https://docs.docker.com/engine/, [Online; consultado 21-Nov-2024]

[6] Frosty: mysql to mariadb unknown collation utf8mb4 0900 ai ci. https://dba.stackexchange.com/q/248904, [Online; consultado 21-Nov-2024]

[7] MariaDbFoundation: MariaDB in brief. https://mariadb.org/en/, [Online; consultado 10-Dic-2024]

[8] netcup: netcup Gmbh. Virtual Server (VPS) Offers. https://www.netcup.com/en/server/vps, [Online; consultado 22-Nov-2024]

[9] Oracle: Compare and Choose MySQL Editions. https://www.mysql.com/products/enterprise/compare/, [Online; consultado 10-Dic-2024]

[10] Oracle: MySQL Community Edition. https://www.mysql.com/products/community/, [Online; consultado 10-Dic-2024]

[11] Oracle: Oracle Database. https://www.oracle.com/database/, [Online; consultado 10-Dic-2024]

[12] Pallets: Deploying your Flask Application to Production. https://flask.palletsprojects.com/en/stable/deploying/, [Online; consultado 22-Nov-2024]

[13] Pallets: Flask WSGI Web Application Framework. https://flask.palletsprojects.com, [Online; consultado 22-Nov-2024]