

# Trabajo Práctico N°2

Escalando una app:

Conceptos teóricos y herramientas utilizadas.

- **Materia:** Programación Distribuida 2.
- **Carrera:** Ingeniería en Informática.
- **Docente:** Ing. Lagostena, Juan Pablo.
- **Estudiante:** Calonge, Federico Matias.

## ÍNDICE:

1. Objetivo.....	Pág. 3
2. Escalamiento de una App.....	Pág. 4
2.1- Escalamiento horizontal y vertical.....	Pág. 4
2.2- Teorema CAP / Conjetura de Brewer.....	Pág. 4
3. Virtualización.....	Pág. 6
3.1- Containers con Docker.....	Pág. 6
3.2- Imágenes vs contenedores vs registros.....	Pág. 6
3.3- Ventajas Docker.....	Pág. 7
3.4- Instalación Docker.....	Pág. 7
3.5- Docker-compose.....	Pág. 7
4. Almacenamiento de los datos.....	Pág. 8
4.1- Redis.....	Pág. 8
4.2-Replicación en Redis.....	Pág. 9
4.3- Ventajas Redis. ....	Pág. 9
4.4- Redis Sentinel.....	Pág. 10
4.5-Implementación de Redis Sentinel.....	Pág. 10
5. Servidores web.....	Pág. 12
5.1- Apache vs Apache Tomcat vs NGINX.....	Pág. 12
5.2- Proxy y proxy reverso.....	Pág. 12
5.3- Ventajas NGINX.....	Pág. 13
6. Esquema final de nuestra app.....	Pág. 14

## 1. Objetivo.

Este Informe tiene como objetivo detallar los conceptos teóricos y las herramientas utilizadas en el Trabajo Práctico N° 2 alojado en Github (<https://github.com/FedericoCalonge/TP2-Docker-Redis-NGINIX>).

En este Trabajo Práctico lo que se hizo fue, en simples palabras, escalar una app: nos proveen una pequeña REST API realizada en Spring Boot, que lo único que hace es demorar en responder y ocupar recursos del contenedor en donde vive. El objetivo es hacerla escalable mediante herramientas que detallaremos en este documento.

## 2. Escalamiento de una App.

**La escalabilidad en una aplicación es la medida del número de usuarios que pueden efectivamente hacer requests en la app al mismo tiempo.** Cuando una app llega al límite de escalabilidad es porque no puede administrar más los request / usuarios. Escalamiento se trata de manejar los recursos que tenemos eficientemente para poder “contener” a todos los usuarios / requests.

### 2.1- Escalamiento horizontal y vertical.

El escalamiento HORIZONTAL (o escalamiento “hacia los costados”) consiste básicamente en agregar más nodos de los que ya tenemos disponibles; y el escalamiento VERTICAL (o escalamiento “hacia arriba”) consiste en “mejorar” los nodos que ya tenemos: asignándoles mayor memoria, procesamiento, incrementando los cores de CPU, etc.

### 2.2- Teorema CAP / Conjetura de Brewer:

El Teorema de CAP define que es imposible para un sistema de cómputo DISTRIBUIDO que garantice simultáneamente:

1-La consistencia (Consistency): nos garantiza que una lectura nos retornará la escritura más reciente de un registro dado. Lo que esto implica es que siempre que se haga alguna modificación a un dato dicho cambio debe reflejarse en todos los nodos de la base de datos, esto garantiza que siempre que se acceda a la información cualquiera de los nodos puede responder y la información siempre será la misma en cada uno de los nodos.

2-La disponibilidad (Availability): debe garantizar que cada petición a un nodo retorne una respuesta / confirmación de si ha sido o no resuelta satisfactoriamente en un periodo de tiempo razonable (sin error ni timeout). Esto implica que si un nodo falla, otro debe estar disponible para guardar los datos.

3-La tolerancia al particionado (Partition Tolerance): es decir, que el sistema siga funcionando a pesar que de algunos nodos fallen (ya que la información es consistente).

En conclusión, este Teorema afirma que **un sistema de computación distribuida sólo es posible asegurar 2 de estas 3 características fundamentales**. De esta manera, un entorno distribuido que asegure la consistencia y la disponibilidad, carecerá de tolerancia a particiones (sistemas tipo CA); un sistema que posea disponibilidad y tolerancia a particiones será poco consistente (sistemas tipo AP); y un sistema consistente y tolerante a particiones no podrá estar siempre disponible (sistemas tipo CP).

#### Ejemplos:

1-AP: Cassandra y CouchDB.

2-CP: MongoDB y Paxos.

3-CA: RDBMS. (sistemas de gestión de BD RELACIONALES)

Sin embargo, ningún sistema distribuido está a salvo de las fallas de red, por lo tanto, **la partición de red ( P ) generalmente tiene que ser tolerada**. Y en presencia de una partición / falla, nos quedamos con 2 opciones: consistencia o disponibilidad. Si elegimos C sobre A el sistema devolverá un error o un timeout de espera si no puede garantizar que la información esté actualizada debido a la partición de red. Al elegir A sobre C entonces el sistema siempre procesará la consulta e intentará devolver la versión disponible más reciente de la información en caso de la partición de la red.

De esta forma, en ausencia de la falla de red (cuando el sistema distribuido se está ejecutando normalmente), se puede satisfacer tanto la A como la C. **Con frecuencia CAP se malinterpreta como si se tuviera que elegir abandonar A, D o PT... sin embargo esto NO es así: la elección es realmente entre C y A solo cuando ocurre una partición de red o falla (PT), en cualquier otro momento no hay que hacer concesiones.**

En los sistemas con filosofía ACID (como RDBMS) eligen consistencia (C) sobre disponibilidad (A). Y, en los sistemas con filosofía BASE (como bases NO SQL y **Redis**) eligen la **A sobre la C**. De esta manera **estos sistemas están disponibles / funcionando la mayoría del tiempo**. Y como la C se deja de lado entonces los almacenamientos no tienen que escribir información consistente todo el tiempo.

Esto nos sucederá en nuestra aplicación, ya que utilizamos **Redis** para el almacenamiento de datos.

### 3. Virtualización.

Virtualizar es la creación a través de **Software** de una versión virtual de algún recurso tecnológico (por ej. una plataforma de Hardware, un SO, un dispositivo de almacenamiento, recursos de red, etc.). En resumen, es **crear versiones virtuales de RECURSOS**.

La ventaja de virtualizar es obtener una eficiencia de costos, energía o de almacenamiento, u obtener una flexibilización para, por ejemplo, probar nuevos sistemas y reproducir con gran facilidad distintos entornos.

**En nuestra aplicación utilizaremos Docker para realizar una virtualización y poder reproducir distintos entornos y servicios con gran facilidad: maven, java, redis, redis-sentinel nginx, gatling, etc.**

#### 3.1- Containers con Docker.

Mediante la tecnología de **Docker** creamos containers. Un **container** es un método de virtualización que nos permite con gran facilidad tener múltiples entornos y servicios.

Docker es un software que nos permite crear y ejecutar software dentro de containers. Nos permite correr estos containers de una manera segura y desacoplada SIN necesidad de utilizar un “hipervisor” que controle todo y haga de “nexo” con el Hardware. Por esta razón utilizar Docker tiene mucha performance ya que **usamos el Hardware del host directamente**.

De esta manera Docker nos permite **encapsular nuestras aplicaciones en containers** de Docker; y nos permite también distribuir y empaquetar estos contenedores entre nuestros equipos para desarrollo y/o testing.

**Docker Engine** es el “motor” de Docker, es la aplicación cliente/servidor que se compone de:

- Un servidor (programa que corre como demonio en un servidor).
- Una API REST (usada para comunicarse con el servidor).
- Una interfaz de comandos o CLI para hacer de wrapper/contenedor de esa API REST.

#### 3.2-Imágenes vs contenedores vs registros.

Podemos imaginarnos las **imágenes** de Docker como lo que en P.O.O. es una **CLASE**. Las imágenes son un template de solo lectura que tienen **INSTRUCCIONES** para crear un Container. Estas instrucciones van en un archivo de texto “**DOCKERFILE**”. Se puede empezar una imagen desde 0 o podemos extender una imagen ya existente descargandola de la web (a través de “**registros**”). Las imágenes son la parte de Docker donde se **construye**.

En cambio, un **contenedor**, siguiendo nuestra analogía con P.O.O., son los **OBJETOS** (o las INSTANCIAS de una Clase: son las instancias ejecutables de una imagen). Los containers son la parte de Docker donde se **ejecuta**.

Por último, los **registros** de Docker son un **CONJUNTO DE IMÁGENES**: similar a un servidor donde uno puede publicar una imagen de docker para que otra persona la descargue. DockerHUB es un ejemplo de repositorio público. Los registros son la parte de docker encargada de la **distribución**.

### 3.3- Ventajas Docker.

Docker nos permite:

- Estandarizar nuestros ambientes: de esta manera establecemos un ambiente estándar para correr nuestra app y ganar mucho tiempo de setup.
- Escalar fácilmente: docker nos permite escalar HORIZONTALMENTE una aplicación muy fácilmente.
- Resolver problemas de dependencias: esto nos permite desacoplarnos del SO y tener nuestras dependencias en un solo lugar.

### 3.4- Instalación Docker.

Para instalar Docker primero tenemos que asegurarnos de tener actualizada la lista de paquetes del repositorio utilizado en nuestro sistema:

**>sudo apt-get update**

Luego para instalar docker:

**>sudo apt install docker.io**

Además, con los siguientes comandos obtendremos la versión instalada de Docker (tanto del servidor como la del cliente del motor Docker) y un detalle con la configuración actual del motor Docker instalado en el sistema:

**>sudo docker version**

**>sudo docker info**

### 3.5- Docker-compose.

Docker-compose es una herramienta para correr aplicaciones multi-containers en Docker. Con docker compose usamos un archivo YAML para configurar nuestros servicios de la aplicación. Y luego mediante un único comando en CLI podemos crear y arrancar todos estos servicios y así correr nuestra app entera:

**>docker-compose up**

Esto nos sirve para “ahorrar” tirar comandos y tener múltiples archivos dockerfile, ya que teniendo todo en un archivo yaml luego corremos “>sudo docker-compose up” y ya se ejecuta todo lo que tiene el docker-compose.yml.

**Docker-compose lo utilizamos en nuestra app para instanciar dentro del docker-compose.yml nuestros containers con nuestras apis, containers de redis-sentinel (master, slaves y sentinelas) y el container de nginx.**

## 4. Almacenamiento de los datos.

Para almacenar los datos en nuestra app utilizamos Redis, que luego “actualizamos” a Redis-Sentinel.

### 4.1- Redis.

Redis es una base de datos de tipo “clave-valor”. Este tipo de DBs funciona mediante el uso de diccionarios (los cuales contienen una gran cantidad de registros que se encuentran identificados por un valor único). Dichos registros pueden poseer diversas variedades de campos, por ejemplo, en una BD tipo clave-valor puede existir un registro con solo campos numéricos y simultáneamente tener otro registro con campos numéricos y alfanuméricos, dicho ejemplo se ilustra en la **Imagen 1**.

Base de datos relacional			Base de datos Clave-Valor	
ID (Int)	Name (Varchar)	Age(int)	Key	Value
1	Sergio	22	1	Sergio, Andres, 22,19/09/1994
2	Ana	48	2	Ana, 12/08/1969
3	Pablo	49	3	Pablo
4	Juan	12	4	Juan, 12

**Imagen 1 - Cómo almacena Redis sus datos.**

Además, **Redis puede guardar una tabla de hashes relacionados a una clave (esto es lo que utilizamos para guardar nuestros datos en nuestra app: utilizando ‘HASHOPERATIONS’).**

Redis se compone de un **Servidor** que actúa como almacén de datos en memoria y un **Cliente** que se conecta contra el, el cual es accedido por aplicaciones a través de una **API**. De esta manera almacena en la memoria del servidor los datos y los recupera a medida que se necesitan. **Estos datos deberían estar permanentemente accesibles (ya que Redis tiene alta disponibilidad)...** por esto surge la **necesidad de tener un sistema de FAILOVER que permite a Redis tener una mínima infraestructura tolerante a fallo: acá entra en juego Redis Sentinel (ver 4.3- Redis Sentinel).**



## 4.2-Replicación en Redis.

La manera más sencilla de replicación de datos en Redis es configurando una replicación “leader follower” (o master-slave): esto permite a las instancias slaves de Redis ser copias EXACTAS de las instancias master. El slave automáticamente se reconectará con el master cada vez que se caiga el link entre ellos.

**Por default Redis usa replicación ASINCRÓNICA** (esto significa que el master le pasa los datos para replicar a los slaves cuando se necesite y no cada cierto tiempo). Esto da **baja latencia y alta performance**. De esta manera periódica y asincrónicamente los Redis Slaves reciben la data del Redis Master. Un master puede tener múltiples slaves. Los slaves se pueden conectar entre sí aparte de estar conectados al master. La replicación sincrónica de ciertos datos pueden ser pedidos por los clientes usando el comando “WAIT”. Esta replicación puede ser usada para tener mayor escalabilidad, teniendo así múltiples slaves para operaciones de lectura; y también sirve para tener los datos seguros y con alta disponibilidad.

Para que funcione la replicación lo que hace el Redis master es tener un “replication ID” (un string random y muy largo que marca toda la historia del dataset) y un “offset” (es un número que incrementa por cada byte de datos replicado y que debe ser enviado a los slaves). De esta manera a los slaves les llega un “replication ID” y un “offset” cuando hacen la replicación y, en caso de tener que reconectarse con el master por alguna falla que hubo en el medio le enviará este replication ID y offset que le indicará al master que data guardada tiene el slave.

Para configurar esta replicación entre Master y Slaves simplemente se debe ejecutar un comando, el cual agregamos en nuestro docker-compose.yml cuando instanciamos cada uno de los Slaves (ver *Imagen 2*).

```
replica_1:
  image: redis:6
  container_name: "redis-replica_1_V2"
  command: redis-server --replicaof redis-master 6379
  links:
    - redis-master
```

*Imagen 2 - Command para replicación en Redis*

## 4.3- Ventajas Redis.

- Es opensource.
- Su almacenamiento es muy rápido debido a que mantiene los datos en memoria pero también se persisten en disco.
- Puede atender cientos de miles de operaciones por segundo y es escalable.
- Como vimos anteriormente, soporta replicación maestro-esclavo.

## 4.4- Redis Sentinel.

Como mencionamos previamente, Sentinel nos brinda un **sistema de FAILOVER** que **permite a Redis tener una mínima infraestructura tolerante a fallo**. De esta manera los datos podrán estar permanentemente accesibles / disponibles mediante dicho sistema de failover.

### Componentes Redis Sentinel:

1-Servidor Redis: proceso encargado de almacenar y mantener los datos en memoria. A la **instancia maestra** de Redis la llamaremos **M** y a las instancias **‘slave’** las llamaremos **R (rèplicas)**.

**2-Procesos Sentinel:** procesos encargados de monitorear las instancias de Redis (R y M) y de iniciar el proceso de Failover en caso de que sea necesario promover una instancia de R a M. A estos los llamaremos **S**.

**3-Procesos Cliente:** procesos encargados de comunicar la aplicación con el servidor de Redis. Los llamaremos **C**. En nuestra app, el cliente de Redis que usamos es **Lettuce**.

### Notas:

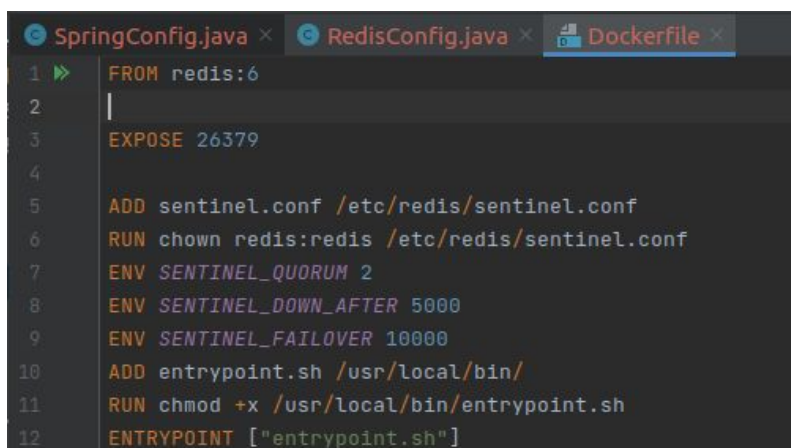
- De la instancia M solo puede haber 1 sola, pero de R, S y C pueden haber más de 1 instancia. El número de cada uno lo elegimos en función de nuestra arquitectura.
- El **QUORUM** es el número de instancias que tienen que estar “de acuerdo” a la hora de marcar un nodo M como caído y promocionar un R a M. **Por esto siempre tenemos que trabajar con al menos 3 instancias de Sentinel, para tener al menos un quorum de 2.**

## 4.5- Implementación de Redis Sentinel.

Para implementar Redis Sentinel en nuestra app se construyeron en el docker-compose.yml: 4 instancias de slaves/replicas, 3 sentinelas y 1 master.

Además, se tuvieron que crear 3 archivos para la configuración de los sentinelas, los cuales son:

**1-“Dockerfile”:** define nuestra imagen custom de sentinel.



```

1 FROM redis:6
2
3 EXPOSE 26379
4
5 ADD sentinel.conf /etc/redis/sentinel.conf
6 RUN chown redis:redis /etc/redis/sentinel.conf
7 ENV SENTINEL_QUORUM 2
8 ENV SENTINEL_DOWN_AFTER 5000
9 ENV SENTINEL_FAILOVER 10000
10 ADD entrypoint.sh /usr/local/bin/
11 RUN chmod +x /usr/local/bin/entrypoint.sh
12 ENTRYPOINT ["entrypoint.sh"]

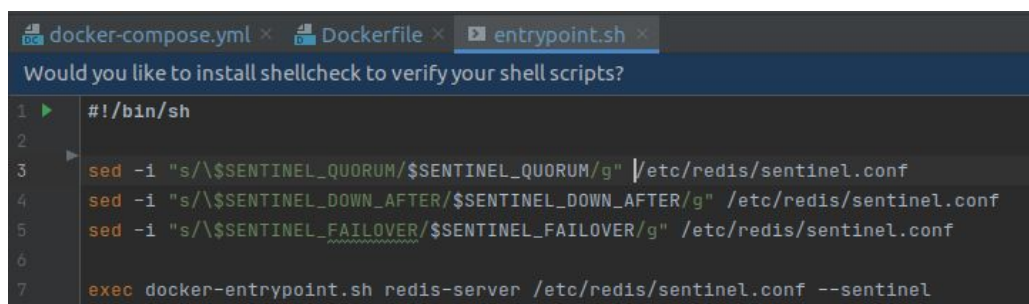
```

**Imagen 3 - Dockerfile Sentinel.**

Lo que hay que destacar de este archivo es:

- Línea 1: hacemos un pull de la imagen oficial de Redis como imagen base.
- Línea 3: exponemos el puerto 26379 que es el que tiene por defecto configurado redis sentinel.
- Línea 6: con "RUN" corremos comandos, y con "chown user[:group] filename(s)" cambiamos de usuario en el contenedor (usuario y grupo "redis").
- Líneas 7, 8 y 9: seteamos valores para la configuración de Sentinel como VARIABLES DE ENTORNO (ENV).
- Línea 5 y 10: Con "ADD" copiamos los archivos desde nuestro directorio local (el sentinel.conf por ej. para la línea 5) al CONTENEDOR (en /etc/redis/sentinel.conf por ej. para la línea 5).
- Línea 11: con chmod +x le decimos que el archivo sea ejecutable.
- Línea 12: definimos al "entrypoint.sh" como nuestro entrypoint... este archivo es el que especifica el ejecutable que usará el contenedor.

**2-"Sentinel-entrypoint.sh":** script para setear los valores en nuestro config file.



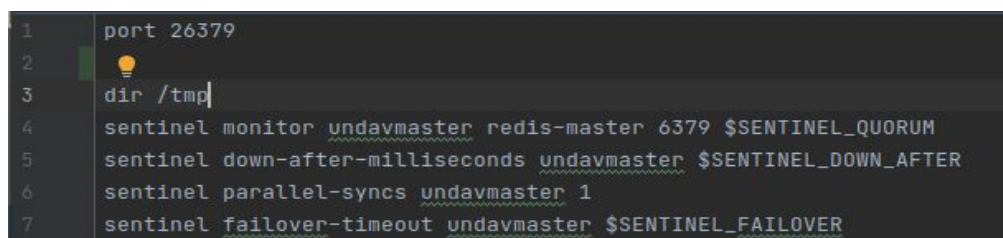
```

1  #!/bin/sh
2
3  sed -i "s/\${SENTINEL_QUORUM}/${SENTINEL_QUORUM}/g" /etc/redis/sentinel.conf
4  sed -i "s/\${SENTINEL_DOWN_AFTER}/${SENTINEL_DOWN_AFTER}/g" /etc/redis/sentinel.conf
5  sed -i "s/\${SENTINEL_FAILOVER}/${SENTINEL_FAILOVER}/g" /etc/redis/sentinel.conf
6
7  exec docker-entrypoint.sh redis-server /etc/redis/sentinel.conf --sentinel
  
```

**Imagen 4 - Entrypoint Sentinel.**

En este archivo solo reemplazamos unos strings con los valores predeterminados que definimos como variables de entorno (los ENV) en el Dockerfile, luego inicia el servidor Redis usando el indicador "--sentinel" mientras lo apunta a nuestro config file.

**3-"sentinel.conf":** nuestro config file, nos provee un template para la configuración de Sentinel.



```

1  port 26379
2
3  dir /tmp
4  sentinel monitor undavmaster redis-master 6379 ${SENTINEL_QUORUM}
5  sentinel down-after-milliseconds undavmaster ${SENTINEL_DOWN_AFTER}
6  sentinel parallel-syncs undavmaster 1
7  sentinel failover-timeout undavmaster ${SENTINEL_FAILOVER}
  
```

**Imagen 5 - Config Sentinel.**

Lo que hay que destacar de este archivo es:

- Línea 4: Aca le decimos a Redis Sentinel qué instancia es nuestro master y cual monitoreará (será una instancia que llamaremos "undavmaster"... donde "redis-master" es el nombre de nuestra imagen master definida en nuestro Docker Compose file). → EL \$SENTINEL\_QUORUM se lo pasamos desde dockerfile con la variable de entorno ENV: entonces al final se ejecutará un **">sentinel monitor undavmaster redis-master 6379 2"**.
- Línea 5: down-after-milliseconds es el tiempo en milisegundos que una instancia master se la considera "down" (o no responde a nuestros PING o responde con un error).
- Línea 6: parallel-syncs permite setear el número de replicas que pueden ser configuradas para usar el nuevo master luego de un failover al mismo tiempo. Mientras más chico sea el número, más tiempo tomará al proceso de failover completarse.
- Línea 7: failover-timeout es el tiempo que se debe esperar para que se intente un failover con el master nuevamente.

## 5. Servidores web.

Los servidores web (web server) son un componente de los servidores que tienen como principal función almacenar, en web hosting, todos los archivos propios de una página web (imágenes, textos, videos, etc.) y transmitirlos a los usuarios a través de los navegadores mediante el protocolo **HTTP** (Hypertext Transfer Protocol).

Un servidor Web en Internet sirve principalmente para **almacenar y transmitir el contenido solicitado de un sitio web al navegador del usuario.**

Algunos de los servidores web más comunes:

- Apache.
- Apache Tomcat.
- NGINX.

### 5.1- Apache vs Apache Tomcat vs NGINX.

- **Apache:** Es el más común y utilizado en el mundo. Ventajas: Es de código abierto, con software gratuito y multiplataforma, y entre sus desventajas su bajo rendimiento cuando recibe miles de requests (peticiones) simultáneas en procesamiento de contenido dinámico o archivos estáticos.
- **Apache Tomcat:** El Servidor Apache nombrado anteriormente es un servidor http de propósito general. En cambio, Apache Tomcat, se desarrolló específicamente para aplicaciones Java: para interactuar con Java Servlets, JavaServer Pages (JSP), Java EL y WebSocket. --> Por defecto **las apps en Spring boot (como es nuestro caso) tienen Apache Tomcat como servidor web configurado.**
- **NGINX:** Una de las mejores alternativas de Apache. Servidor web de código abierto y gratuito (aunque también existe una versión comercial) que se destaca por su alto rendimiento. La principal diferencia entre Apache y Nginx (y la más grande) es su arquitectura, mientras que Apache abre un montón de procesos para servir peticiones, **Nginx abre solo los hilos de ejecución justos y necesarios permitiendo servir millones de peticiones en un corto espacio de tiempo**, ya que no requiere tiempo adicional para abrir nuevos procesos y además al no abrir nuevos procesos tampoco consume mas memoria RAM. La diferencia de rendimiento entre Nginx y Apache se nota, ya que el tiempo de respuesta conseguido por Nginx es casi un 150% más rápido que en el caso de Apache. Cuando el número de usuarios aumenta notamos que a Apache le empieza a costar trabajar con tantos usuarios al mismo tiempo, mientras que **Nginx se comporta mucho más rápido cuando tenemos mucho tráfico.**

### 5.2- Proxy y proxy reverso.

- Un servidor proxy "normal" es para que los clientes puedan acceder a Internet (USER → PROXY → Internet).
- Un servidor proxy REVERSO es un intermediario entre Internet y el Web Server: toma los requests y se los envía al web server y así permite distribuir la carga a varios servidores actuando como Load Balancer... (INTERNET → PROXY REVERSO → Web Server).

## 5.3- Ventajas NGINX

- Es de configuración simple, ligera, rápida y excelente en cuanto a seguridad y rendimiento, además permite ser configurado para integrarse nativamente con casi cualquier tecnología y lenguaje de programación moderno.
- La principal ventaja de Nginx como servidor web es que consume muchos menos recursos al servir contenido estático, y esto convierte a **Nginx en una excelente opción para funcionar como proxy inverso o como balanceador de carga para otros servidores como Apache**, optimizando la entrega de contenidos.
- Lo que hace que Nginx sea diferente a otros servidores web es su arquitectura, que permite responder a millones de peticiones por segundo aprovechando al máximo los núcleos o hilos de ejecución del servidor con una configuración muy simple.
- Como mencionamos previamente, Nginx puede ser configurado como Reverse proxy y load balancer:
  - **Reverse proxy:** podemos usar Nginx como proxy inverso para otro servidor web como Apache. El funcionamiento de Nginx como proxy inverso para Apache es simple: Nginx sirve los contenidos estáticos (imágenes, css, javascript, etc...) mientras que Apache se encarga de servir el contenido dinámico siendo además el encargado de procesar los scripts PHP.
  - **Load balancer:** podemos utilizar Nginx como balanceador de carga para balancear el tráfico entrante entre varios servidores web. Para configurar Nginx como load balancer debemos utilizar el módulo "Upstream Module" y configurarlo para enviar tráfico a varias direcciones internas que corresponden con diferentes servidores web. En la versión opensource tendremos limitaciones pero podremos realizar algunas configuraciones interesantes, como por ejemplo: definir el peso de cada servidor web o configurar algunas directivas adicionales que nos ayudarán a realizar una correcta administración del balanceo de carga.

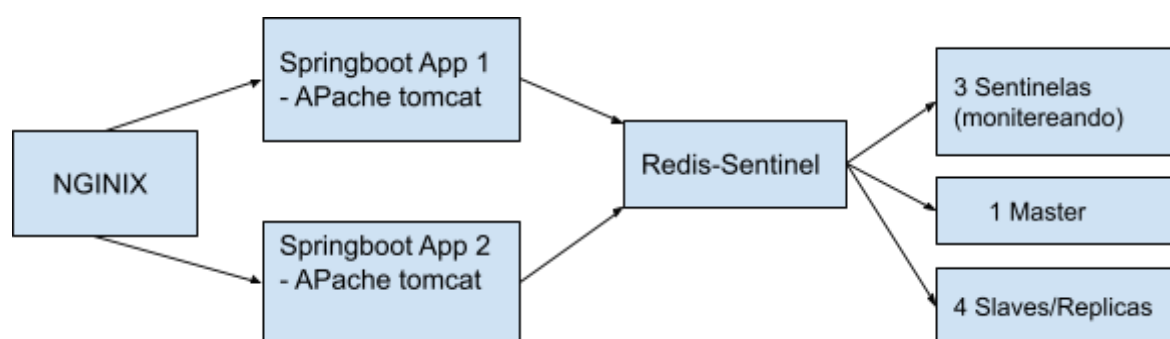
**Podemos configurar Nginx para que envíe el tráfico a uno u otro servidor web dependiendo del estado de cada uno de los servidores web y la carga de tráfico que estén soportando en un momento concreto.**

En nuestra app conseguimos implementar NGINX: de esta manera, haciendo un GET a localhost:80, NGINX redirigirá al web server Tomcat que esté disponible (localhost:8082 o al localhost:8086). Y, cuando sature un servidor empezará a enviar los GET al otro servidor. En conclusión, lo que logramos con NGINX es tener las 2 apps levantadas, y cuando una se sature, la otra pueda responder.

## 6. Esquema final de nuestra app.

Luego de haber implementado Redis, Redis Sentinel y Nginx, podemos decir que logramos tener una app escalable (una vez analizados los tests realizados con Gatling).

De esta manera, el esquema de la versión final de nuestra app es la observada en la **Imagen 6**.



**Imagen 6 - Esquema versión final app**

De esta manera tenemos a NGINX instalado en un contenedor apuntando a dos apps que están en otros contenedores. La app 1 y app 2 están ambas conectadas a Apache Tomcat, el cual está escuchando en el puerto 8080. Además, ambas están conectadas a Redis mediante los 3 sentinelas (que están en los puertos 26379, 26380 y 26381) y que monitorean al master en 6379 y a los slaves/Replicas; y en caso de falla o caída del master se producirá un failover y existirá un quorum entre los sentinelas para elegir al nodo slave/replica, ubicado en otro puerto, que tomará el nuevo rol de master.