FACULTAD DE INGENIERIA
Universidad de Buenos Aires

TP1 - Byte left

[75.73] Arquitectura de Software - FIUBA
Segundo cuatrimestre de 2024

Nombre	Padrón	Email
Maria Fernanda Pont Tovar	104229	mpont@fi.uba.ar
Martin Andres Maddalena	107610	mmaddalena@fi.uba.ar
Pratto, Florencia	110416	fnpratto@fi.uba.ar

Índice

1. Introducción

Este trabajo práctico tiene como objetivo principal comparar diversas tecnologías y evaluar cómo diferentes aspectos impactan en los atributos de calidad de un sistema. Al explorar este tema, buscamos no solo identificar las mejores prácticas y herramientas, sino también entender el impacto de las decisiones tecnológicas en el rendimiento y la confiabilidad de los servicios.

A lo largo de este TP, nos enfocaremos en una serie de tecnologías ampliamente utilizadas en la industria, incluyendo Node.js con Express para la construcción de APIs, Docker y Docker Compose para la containerización y gestión de servicios, Nginx como servidor proxy y balanceador de carga, y Redis para el almacenamiento en caché. También utilizaremos Artillery como generador de carga para simular distintos escenarios de tráfico, y herramientas como cAdvisor, StatsD, Graphite y Grafana para la recolección y visualización de métricas en tiempo real.

2. Funcionamiento

2.1. Levantar la aplicacion y servicios

En primer lugar, tenemos a Nginx corriendo en localhost:5555, que actúa como un servidor inverso. A través de este servidor, se habilitan diversos servicios que permiten interactuar con diferentes APIs. A continuación, se detallan los endpoints disponibles:

2.2. 1. Ping

- **Endpoint:** `/ping`
- **Descripción:** Este servicio devolverá un valor constante sin procesamiento. Se utilizará como *healthcheck* y como referencia (baseline) para comparar con los demás servicios.

2.3. 2. Dictionary

- **Endpoint:** `/dictionary?word=<word>`
- **Descripción:** Este endpoint consultará la *Free Dictionary API* para devolver la fonética (*phonetics*) y los significados (*meanings*) de las palabras en inglés. Es importante tener en cuenta que la API devuelve más campos de los que debemos incluir en nuestra respuesta.

2.4. 3. Spaceflight News

- **Endpoint:** `/spaceflight_news`
- **Descripción:** Este servicio devolverá únicamente los títulos de las cinco últimas noticias sobre actividad espacial, obtenidas de la *Spaceflight News API*. Se debe consultar la documentación de la API para obtener más detalles sobre la implementación.

2.5. 4. Random Quote

- **Endpoint:** `/quote`
- **Descripción:** Este endpoint devolverá un dato inútil de manera aleatoria por cada invocación, utilizando la API de *Useless Facts* (https://uselessfacts.jsph.pl/).

3. Definiendo pruebas de Rendimiento

3.1. Pruebas de Carga

Las pruebas de carga son un tipo de prueba no funcional que evalúa el comportamiento de un sistema cuando se incrementa progresivamente la carga, ya sea por número de usuarios concurrentes o transacciones, hasta alcanzar su capacidad máxima. El propósito de esta prueba es medir tanto el tiempo de respuesta como la estabilidad de la aplicación bajo una carga alta, garantizando que el sistema pueda manejar el volumen esperado de tráfico.

Estas pruebas se ejecutan en un entorno controlado para comparar el rendimiento entre diferentes sistemas. También se les conoce como "pruebas de volumen.º "pruebas de resistencia".

El éxito de las pruebas de carga se determina si los casos de prueba se completan sin errores dentro del tiempo asignado.

Para cada *endpoint*, las siguientes etapas son ejecutadas:

- **WarmUp**: Durante esta fase, el sistema se prepara con una carga inicial baja para estabilizarse. Duración: 30 segundos, tasa de llegada: 5 usuarios por segundo.
- **Ramp**: Se incrementa la carga progresivamente para observar cómo el sistema se adapta al aumento de usuarios. Duración: 30 segundos, tasa de llegada inicial: 5 usuarios por segundo, aumentando a 20 usuarios.
- **Plain**: Esta fase somete al sistema a la carga máxima planificada de usuarios o transacciones, manteniéndola constante durante un período prolongado. Duración: 60 segundos, tasa de llegada: 20 usuarios por segundo.
- **CoolDown**: Finalmente, se reduce la carga de manera gradual, permitiendo al sistema volver a su estado normal. Duración: 30 segundos, tasa de llegada: 5 usuarios por segundo.

3.2. Pruebas de Estrés

Las pruebas de estrés son otro tipo de prueba no funcional cuyo objetivo es evaluar la estabilidad del sistema cuando se enfrenta a condiciones extremas que exceden sus capacidades operativas normales, como la falta de recursos de hardware (CPU, memoria, disco). Se utiliza para verificar cómo el sistema responde cuando se le somete a una carga superior a la esperada y su capacidad de recuperación ante fallos.

Se ejecutan en un entorno controlado para evitar riesgos antes del lanzamiento, y son clave para identificar cuán robusto es un sistema bajo presión.

Para cada *endpoint*, las fases son:

- **WarmUp**: Esta fase inicial tiene como propósito estabilizar el sistema antes de someterlo a cargas más altas. Durante 30 segundos, se simula la llegada de 5 usuarios por segundo para que el sistema se prepare y alcance un estado operativo estable.
- **Ramp**: En esta fase, la carga se incrementa progresivamente para probar cómo el sistema se comporta bajo un aumento de usuarios. Comienza con una tasa de llegada de 5 usuarios por segundo y, a lo largo de 30 segundos, la carga se incrementa de manera gradual hasta llegar a 40 usuarios por segundo. Este incremento permite observar la respuesta del sistema frente a situaciones de carga creciente.
- **Plain**: En esta fase, el sistema opera bajo una carga sostenida de alto estrés. Durante 60 segundos, la tasa de llegada es de 40 usuarios por segundo, simulando condiciones de carga muy superiores a las esperadas en un entorno normal. Esta fase es crucial para identificar cómo el sistema maneja el estrés prolongado.

- **CoolDown**: Finalmente, en la fase de enfriamiento, la carga se reduce de nuevo a niveles bajos, con una tasa de llegada de 5 usuarios por segundo durante 30 segundos. El objetivo es permitir que el sistema vuelva a condiciones normales de operación y evaluar si hay problemas durante este proceso de recuperación.

4. Métricas visibles en el dashboard en Grafana

- Demora de cada endpoint en responder (esto es, API remota + procesamiento propio) - Demora de cada API remota en responder (solo API remota)

4.1. Escenarios lanzados (stacked)

Esta métrica muestra la cantidad de escenarios que se lanzan durante las pruebas de rendimiento. Se visualizan con una gráfica en formato de líneas apiladas, lo que permite ver cómo evolucionan las ejecuciones de los diferentes escenarios en el tiempo. Los principales datos mostrados en esta métrica incluyen:

- **Total de escenarios lanzados**: Muestra la cantidad total de escenarios que se han lanzado en cada momento de tiempo.
- **Escenarios individuales**: Cada serie representa un escenario específico que se ejecuta durante la prueba.

La visualización permite observar el comportamiento de las diferentes simulaciones y su evolución en función del tiempo.

4.2. Estado de las solicitudes (stacked)

Esta métrica está enfocada en el estado de las solicitudes procesadas durante las pruebas. Se muestran las solicitudes completadas, pendientes, limitadas (por rate limiting) y las que han fallado. Cada estado se representa mediante un color diferente:

- **Solicitudes completadas (Completed)**: Representa las solicitudes que han sido procesadas con éxito (códigos 200).
- **Solicitudes pendientes (Pending)**: Muestra las solicitudes que están en cola o aún no se han procesado.
- **Errores (Errored)**: Se muestra la cantidad de errores durante las solicitudes.
- **Solicitudes limitadas (Limited)**: Representa las solicitudes que han sido rechazadas por exceder los límites de tasa (códigos 429).

Esta métrica es crucial para identificar problemas relacionados con el procesamiento de solicitudes y los errores que ocurren durante las pruebas.

4.3. Latencia de API y de Endpoint

hemos implementado latencia de API y latencia de endpoint, ambas implementadas utilizando la librería hot-shots para enviar métricas a StatsD. Estas métricas nos permiten observar y medir en tiempo real el rendimiento de los diferentes endpoints de nuestra aplicación y entender cómo responden a las solicitudes de los usuarios. Estp tambien nos sirve para identificar si los tiempos de respuesta largos están relacionados con nuestro propio código o con solicitudes externas a APIs de terceros.

4.4. Tiempos de respuesta (client-side)

El tiempo de respuesta se mide desde el lado del cliente, mostrando tanto la duración máxima como la mediana. Los tiempos de respuesta se visualizan en milisegundos (ms) y ayudan a identificar cuán eficientes o lentas son las respuestas de los diferentes escenarios.

- **Duración máxima (Upper)**: El valor máximo registrado en los tiempos de respuesta de los escenarios.
- **Mediana (Median)**: La mediana del tiempo de respuesta, útil para obtener una referencia más estable del tiempo típico.

El análisis de esta métrica es fundamental para evaluar la experiencia del usuario en cuanto a tiempos de respuesta.

4.5. Resources

La seccion de Recursos suele referirse al monitoreo de los recursos del sistema. Estas métricas son importante para entender cómo se utilizan los recursos y detectar posibles problemas de rendimiento o capacidad. Mide el uso de la CPU y memoria. El uso de CPU muestra qué porcentaje del procesador está siendo utilizado por tu aplicación. El uso de memoria indica cuánta memoria RAM está consumiendo tu aplicación.

5. Corriendo escenarios

5.1. Caso Base

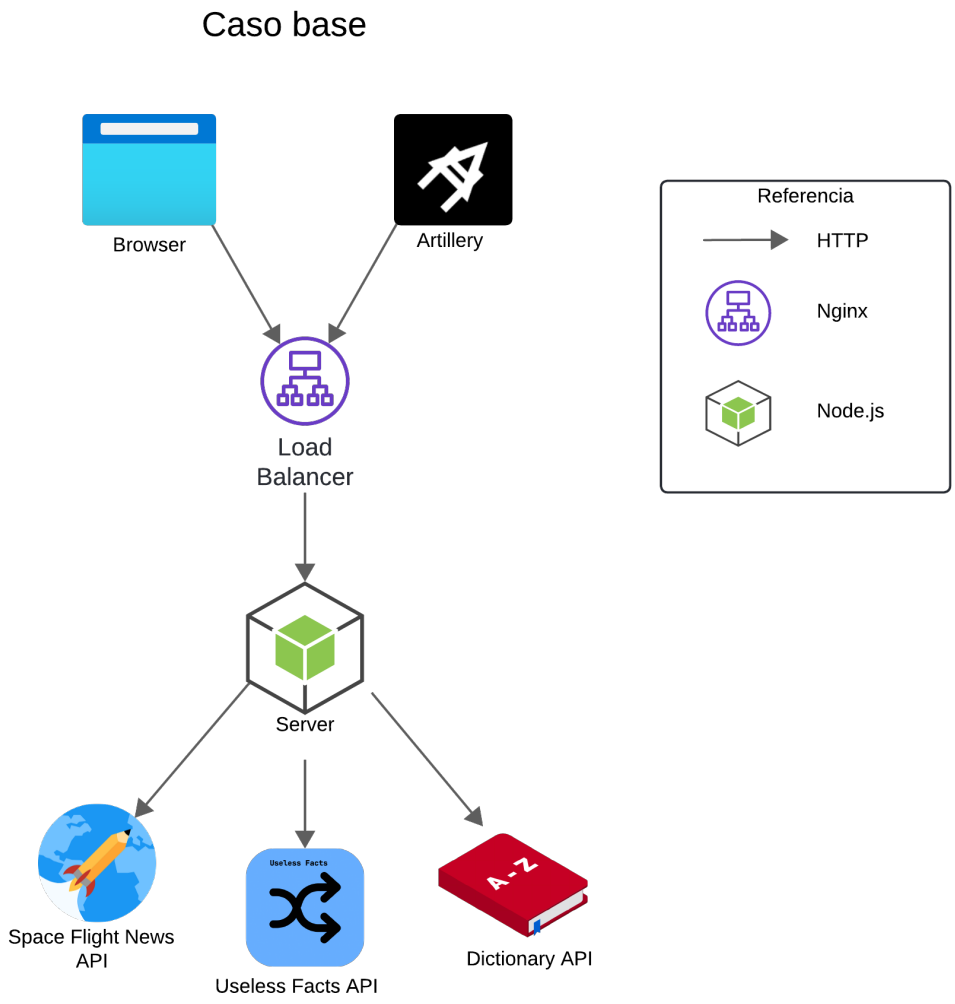

Figura 1: Diagrama de componentes de caso base

El caso base no utiliza tácticas de caching, replicación o rate limiting. Consiste de una sola instancia containerizada de la aplicación Node.js (denominada 'app') donde el endpoint 'ping' devuelve instantáneamente la línea 'pong', mientras que el resto de los endpoints hacen un único request HTTP a las APIs externas utilizadas (Spaceflight, Quote, Dictionary etc.) Existe un contenedor proxy 'nginx' que recibe las requests y se las envía a la app como indica la consigna.

Usamos la herramienta Artillery para generar dos tipos de escenarios, de carga (load testing) y de sobrecarga (stress testing) que fueron idénticos en cada endpoint para facilitar la experi-

mentación. Para el endpoint diccionario, se utilizó un archivo 'words.csv' para poder alimentar al endpoint con palabras distintas en cada request artificial generada.

5.2. Rate limiting

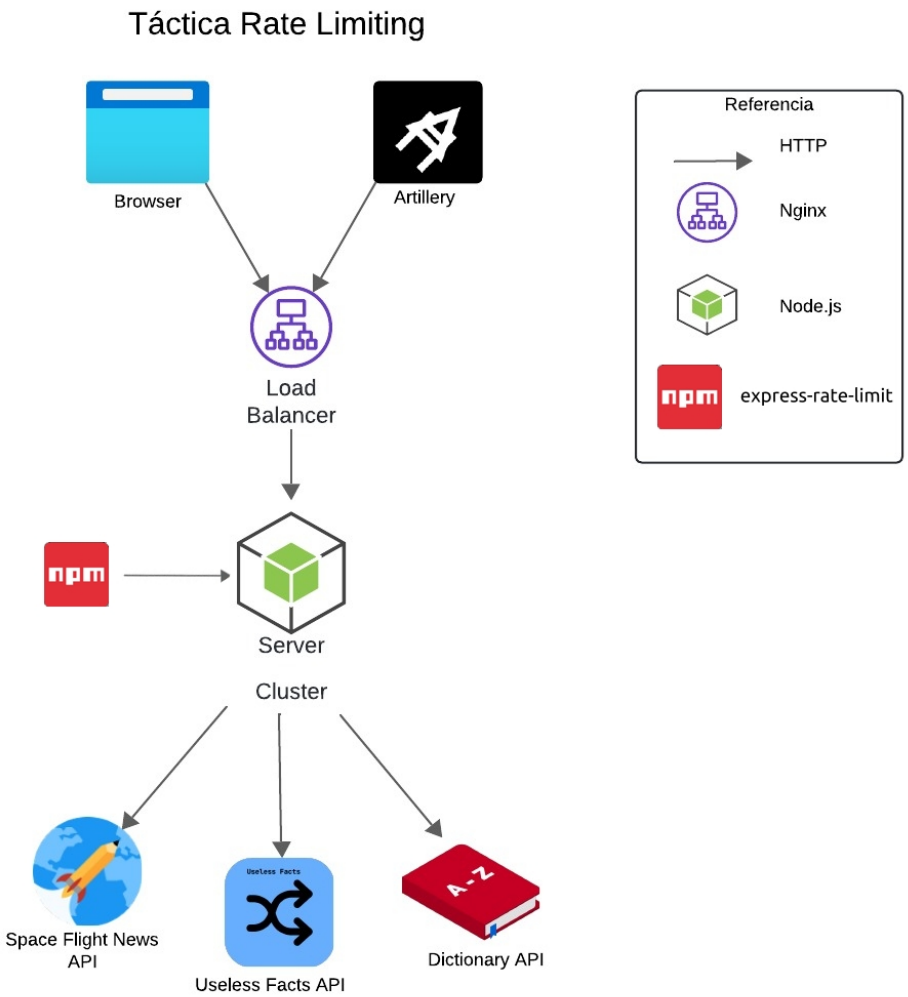

Figura 2: Diagrama de componentes de Rate limiting

Para experimentar con el rate limiting escogimos la API spaceflight-news. Al correr nuestra prueba load testing en el caso base observamos que tras un determinado tiempo la API comienza a ignorar todos nuestros requests, resultando en falla del 100 % de ahí en adelante.

Figura 3: Spaceflight: caso base

Dado esto nos pareció un buen candidato para la estrategia de Rate Limiting. El Rate Limiting nos permite rechazar requests de un IP determinado al llegar más de cierta cantidad dentro de una ventana dada, en nuestro caso de 10 segundos, del lado de nuestra aplicación. De esta forma evitamos hacer el request a Spaceflight News y prevenimos que se nos bloquee totalmente el acceso a la API.

Con esto en mente observamos que el caso base empieza a estar sobrecargado alrededor de los 70 requests cada 10 segundos (7 rd/s).

Con esto en mente decidimos imponer un rate limit de 70 requests cada 10 segundos prediciendo que esto impediría que Spaceflight News rechace todas nuestras requests. Al hacer esto logramos que se ejecutara la prueba con este nivel de requests sin que estas comenzaran a fallar.

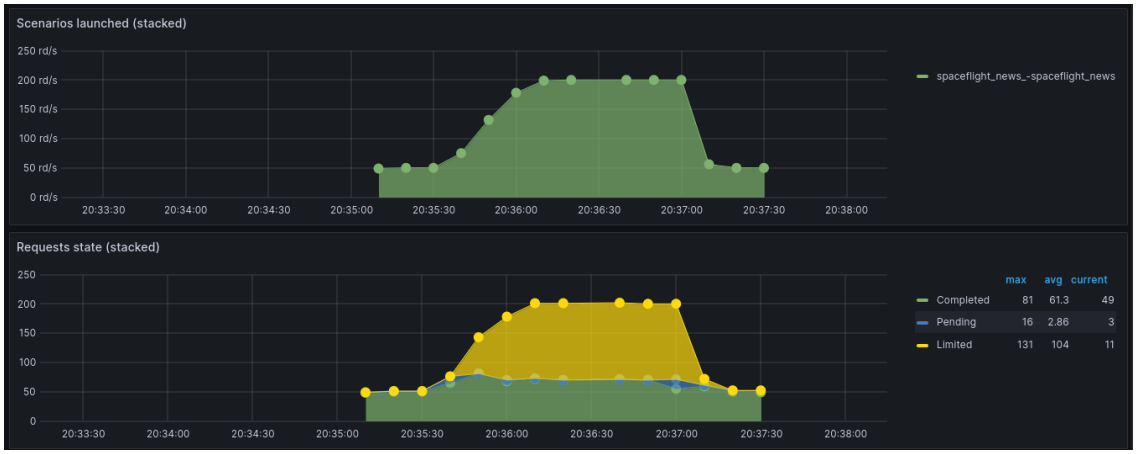

Figura 4: Spaceflight: rate limiting

Esto demuestra la capacidad del rate-limiting de prevenir fallas catastróficas del sistema, manteniendo servicio parcial en momentos de alta carga.

5.3. Lazy cache

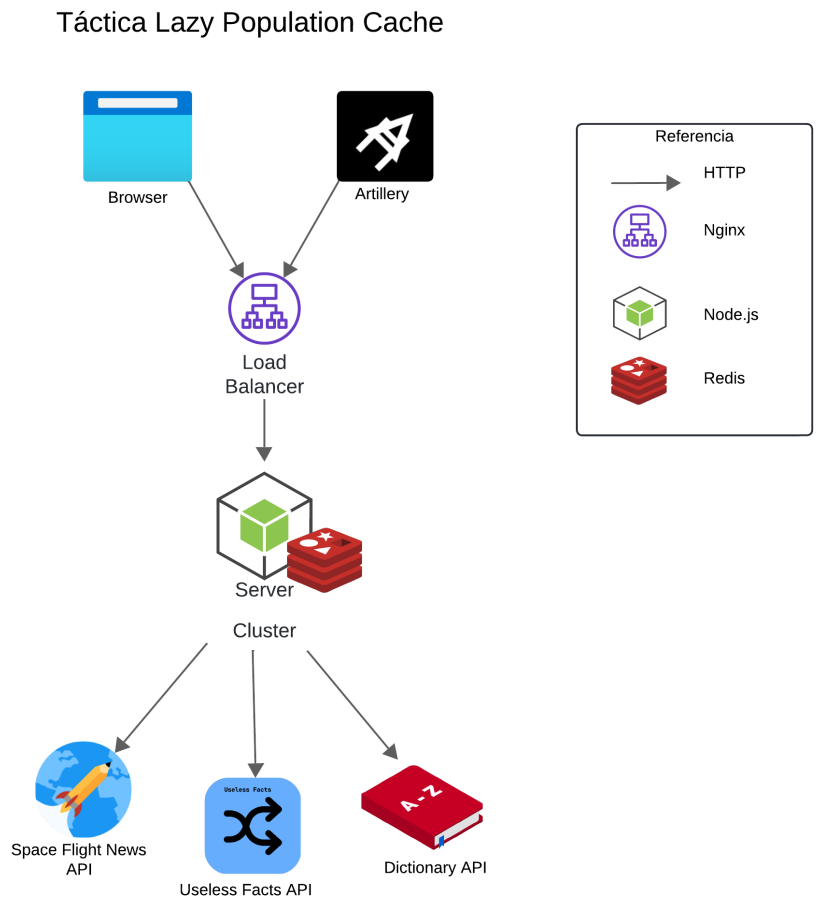

Figura 5: Diagrama de componentes de lazy cache

Para la táctica de Cache, escogimos el tipo de cache 'lazy' que hace caching de un endpoint después de llegar el pedido del primer usuario.

Nuestro primer experimento fue con el endpoint de Quotes. Nuestra predicción fue que el cache tendría un impacto más visible en el response time, mejorando el tiempo de respuesta en general para las requests (medido usando la mediana cada 10 segundos). También predecimos que habría un impacto sutil en el uso de recursos, dado que no haríamos peticiones a la API externa.

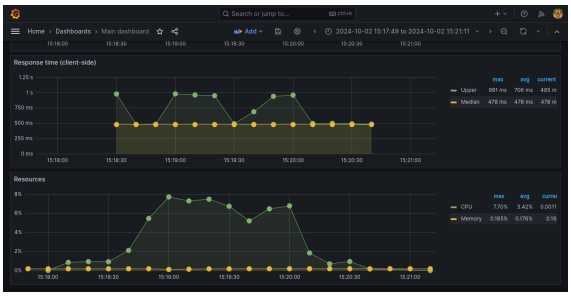

Figura 6: Quote: stress testing, caso base

Al ejecutar el caso baseline observamos un tiempo de respuesta a cada API de 500 ms (mediana), con picos de 1 o 2 segundos.

Después de activar el cache, observamos una reducción masiva en el tiempo de respuesta, tendiendo a alrededor de 10 ms. Esto demuestra el impacto de hacer caching de las respuestas del API.

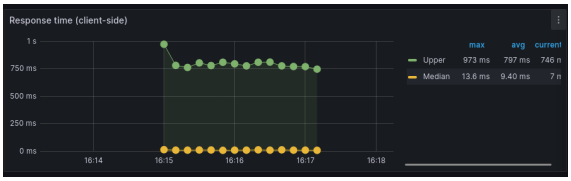

Figura 7: Enter Caption

Para nuestro segundo experimento, utilizamos el endpoint Dictionary, que tiene la particularidad de que nuestro simulador de carga, Artillery, selecciona palabras de forma secuencial del archivo "words.csv", siendo siempre distintas entre sí.

Nuestra hipótesis en este caso era que la estrategia de lazy cache no sería efectiva, dado que los datos del endpoint varían significativamente entre solicitudes. Esto haría que la táctica de caché fuera ineficaz o, en el mejor de los casos, poco útil.

Sin embargo, al ejecutar el experimento como lo hacemos habitualmente, observamos resultados contradictorios con respecto a nuestra hipótesis y conocimientos. Por ello, decidimos adaptar la prueba de load testing para analizar los detalles con mayor precisión.

Probamos un nuevo escenario en el que se ejecutó una carga de 30 segundos dos veces, seguido de una carga de 40 segundos y otra de 60 segundos, con una tasa de 1 solicitud por segundo. En este escenario (Ver figura 8 y 9), observamos una leve mejora en los tiempos de respuesta al utilizar lazy cache.

Figura 8: Dictionary: lazy-cache, response time

Figura 9: Dictionary: caso base, response time

5.4. Replicación

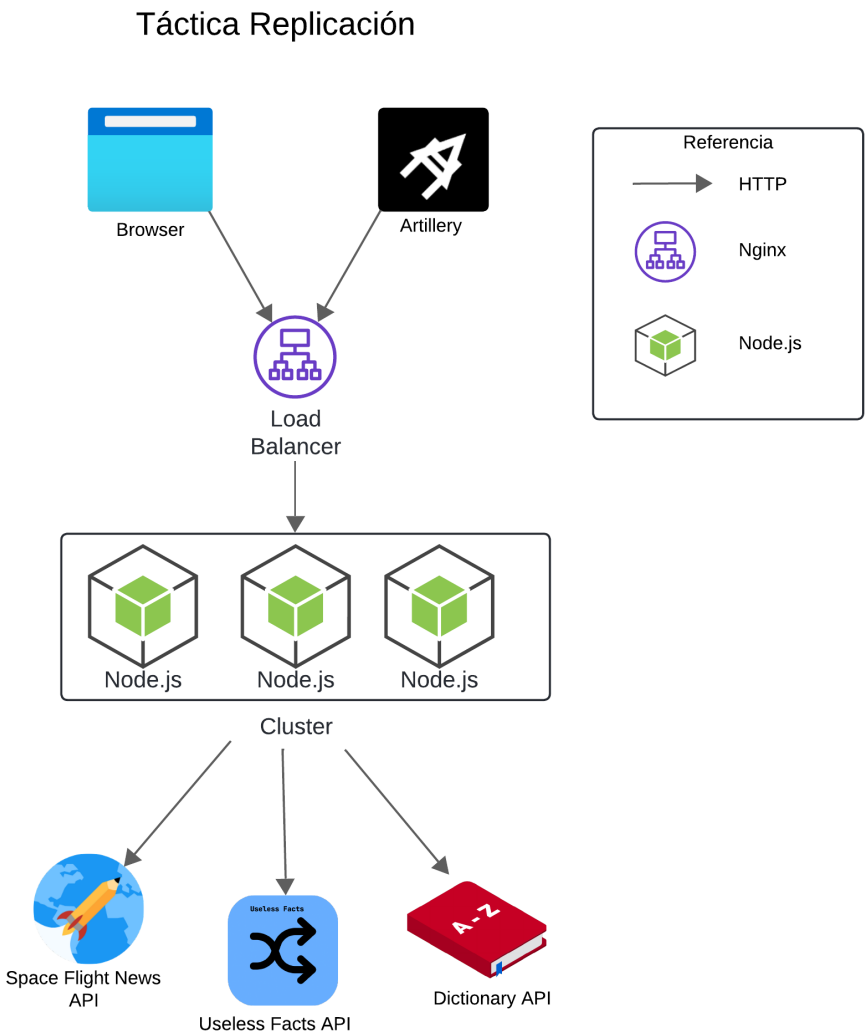

Figura 10: Diagrama de componentes de tactica de replicacion

Nuestra estrategia de replicación fue implementada a través del load balancer nginx. Levantamos tres contenedores distintos a idénticos de la app denominados app-1, app-2 y app-3, y configuramos nginx para que redirigiera requests equitativamente entre ellos. Para los escenarios de replicación, probamos los endpoints de spaceflight, quote y dictionary. Nuestra predicción fue que al estar realizando una replicación dentro del mismo hardware, y dadas las capacidades limitadas de este hardware de paralelizar la actividad de los distintos contenedores de nuestro sistema en una única computadora, los beneficios de la replicación no serían detectables.

Para este experimento se asume y observa que app-1, app-2 y app-3 tienen aproximadamente el mismo comportamiento y no se observarán diferencias importantes en las métricas. Se anexarán de todas formas los resultados para todas las instancias.

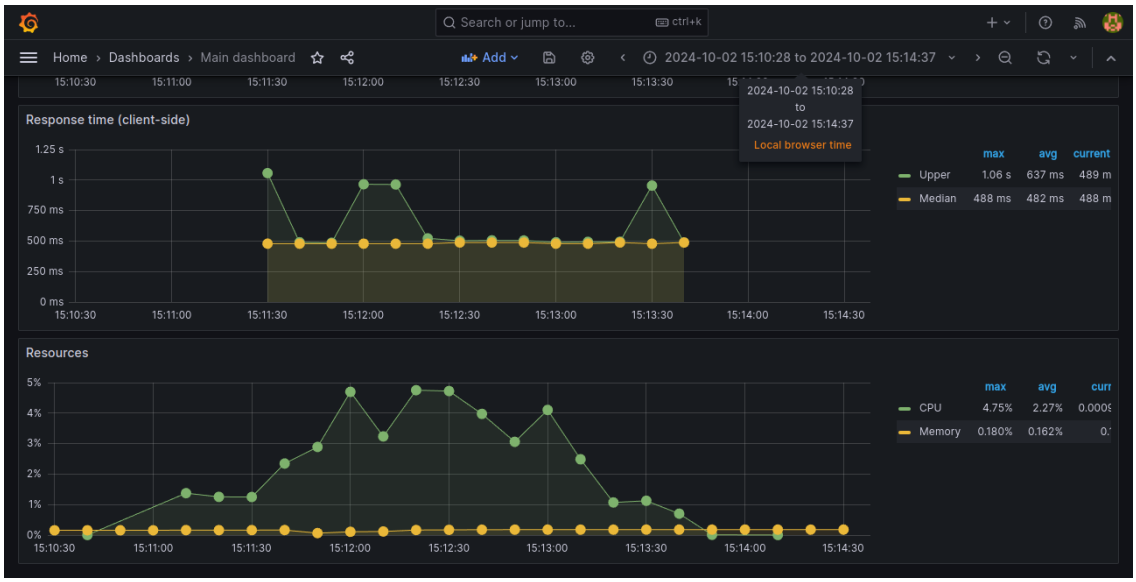

Figura 11: Caso base - recursos

Como fue esperado el response time y latencia no mejoraron ni empeoraron significativamente. La mayor diferencia es en el uso de recursos de CPU. El uso de CPU del caso base no supera el 5 % mientras que para una de las apps en el caso de replicación aumenta a entre el 7 y 10 %. Esto es de esperarse dado que la replicación ocurre en contenedores dentro de la misma computadora o sea que refleja el overhead de la replicación en estos recursos compartidos.

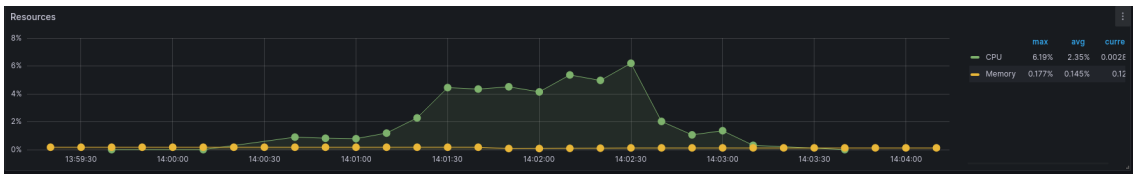

Figura 12: Replication: App 1 - recursos utilizados, quote

6. Comparaciones

Luego de estos experimentos comparamos las distintas técnicas y sus impactos en las métricas de nuestra aplicación.

El rate limiting, observamos que sólo tenía un impacto beneficioso en el caso de spaceflight news, donde llegábamos en nuestras pruebas al límite de la capacidad de la API externa lo cual causaba un colapso total del servicio. Aplicar el rate limiting nos permitió evitar sobrecargarlo con esta limitación y mantener funcionamiento parcial (atributo Availability). Sin embargo, en el endpoint de dictionary, observamos un rate limiting de alrededor de 1 request por segundo por parte del endpoint en sí, por lo que descartamos la táctica de rate limiting del lado de nuestra aplicación.

La replicación no obtuvo mejoras significativas en ninguna métrica, y resultó en peor uso de recursos de CPU y memoria, puesto que todas las réplicas eran ejecutadas en un mismo ordendor y no proveían mejoras de servicio. La replicación observa beneficios para la Availability y Reliability en casos donde se replica la aplicación entre distintos hosts, permitiendo que fallas de hardware independientes no causen la pérdida total del sistema, y además pueden mejorar la performance en comparación al caso base al escalar verticalmente el servicio.

Finalmente la táctica de cache tuvo enormes beneficios debido a eliminar la necesidad de esperar la respuesta de la misma API externa en todos los requests. Vale la pena recalcar que el caching solo tiene sentido para endpoints cuyo contenido es repetitivo, en nuestro caso el caching de requests como quote y spaceflight_news tenía sentido dentro de los parámetros de la consigna.

7. Conclusiones

Con los resultados de nuestros experimentos llegamos a la conclusión de que las tácticas utilizadas (replicación, cache y rate limiting) sirven propósitos diferentes, para tratar y mejorar distintos atributos de calidad (Reliability, availability, performance) y asimismo son apropiados sólo para ciertos tipos de servicios y endpoints.

8. Referencias

https://en.wikipedia.org/wiki/Nginx
https://redis.io/docs/latest/
https://en.wikipedia.org/wiki/Reliability_engineering#Software_reliability
https://en.wikipedia.org/wiki/Availability
https://jenkov.com/tutorials/software-architecture/caching-techniques.html