



UNIVERSIDAD DE BUENOS AIRES

FACULTAD DE INGENIERÍA

75.73 - ARQUITECTURA DE SOFTWARE

Primer cuatrimestre 2023

Trabajo Práctico

ANÁLISIS DE UNA ARQUITECTURA DE SOFTWARE

Nombre	Padrón	Mail
Cai, Ana María	102150	acai@fi.uba.ar
Guglielmone, Lionel	96963	lguglielmone@fi.uba.ar
Milhas, Facundo	102727	fmilhas@fi.uba.ar
Ramirez, Jeremias	102878	jnramirez@fi.uba.ar

Índice

1. Introducción y objetivo	3
2. Metodología	4
3. Tácticas	5
3.1. Caso base	5
3.2. <i>Cache</i>	6
3.3. <i>Replication</i>	7
3.4. <i>Rate Limiting</i>	8
4. Load tests	10
4.1. /ping	10
4.1.1. Caso base	10
4.1.2. <i>Replication</i>	12
4.1.3. <i>Rate limiting</i>	13
4.2. /metar	14
4.2.1. Caso base	14
4.2.2. <i>Cache</i>	16
4.2.3. <i>Replication</i>	18
4.2.4. <i>Rate limiting</i>	19
4.3. /space_news	20
4.3.1. Caso base	20
4.3.2. <i>Cache</i>	21
4.3.3. <i>Replication</i>	23
4.3.4. <i>Rate limiting</i>	24
4.4. /fact	25
4.4.1. Caso base	25
4.4.2. <i>Replication</i>	27
4.4.3. <i>Rate limiting</i>	28
5. Stress tests	29
5.1. /ping	29
5.1.1. Caso base	29
5.1.2. <i>Replication</i>	30
5.1.3. <i>Rate limiting</i>	31
5.2. /metar	32
5.2.1. Caso base	32
5.2.2. <i>Cache</i>	33
5.2.3. <i>Replication</i>	34
5.2.4. <i>Rate limiting</i>	36
5.3. /space_news	37
5.3.1. Caso base	37
5.3.2. <i>Cache</i>	38
5.3.3. <i>Replication</i>	39
5.3.4. <i>Rate limiting</i>	40

5.4. /fact	41
5.4.1. Caso base	41
5.4.2. <i>Replication</i>	43
5.4.3. <i>Rate limiting</i>	44
6. Conclusiones	45
7. Apéndice: métricas propias	46
7.0.1. Tiempo de respuesta local	46

1. Introducción y objetivo

El objetivo principal de este trabajo es analizar el impacto de algunas tecnologías en atributos de calidad seleccionados, ver que cambios se pueden realizar para mejorar dichos atributos y, como objetivo menor, reflejar el aprendizaje de ciertas tecnologías, tales como:

- Node.js (+ Express)
- Docker Compose
- Nginx
- Redis
- Artillery

Con el fin de alcanzar estos objetivos, se definieron 2 metodologías que nos permiten comparar los atributos de calidad seleccionados: **load test** (test de carga) y **stress test** (test de estrés).

Asimismo, la API desarrollada cuenta con cuatro *endpoints*, los cuales se comunican con servicios externos. Estos endpoints son:

- /ping: devuelve un valor constante, sin procesamiento.
- /metar: devuelve un reporte del estado meteorológico que se registra en un aeródromo pasado por parámetro.
- /space_news: devuelve las 5 últimas noticias sobre actividad espacial.
- /fact: devuelve 1 hecho sin utilidad por cada invocación a nuestro endpoint.

Cada uno de estos endpoints tiene una versión en la que se utiliza Redis para cachear los resultados y otra con un rate limiter aplicado (en los casos que estas dos tácticas apliquen).

2. Metodología

Para el análisis se utilizaron dos tipos de tests de performance con el objetivo de compararlos entre las distintas tácticas aplicadas:

Test de carga: El objetivo de este test es verificar la capacidad de tolerancia de una aplicación de manera constante y creciente. Esta prueba es llevada a cabo para evaluar los requerimientos no funcionales de una aplicación y asegurarse de que pueda soportar una carga determinada.

Las pruebas de performance realizaron con la ayuda de la herramienta llamada “artillery” que nos permite replicar estos escenarios.

Test de estrés: Se utiliza *spike testing*, que es un tipo de prueba de estrés que tiene por objetivo validar las características de rendimiento de un sistema bajo cargas de trabajo y volúmenes de carga que superan repetidamente las operaciones de producción anticipadas durante períodos cortos de tiempo.

En primer lugar, se define un **baseline** para cada uno de los *endpoints*. Este **baseline** consiste en el llamado a cada endpoint sin la aplicación de ninguna táctica en particular. La idea es utilizar los resultados obtenidos en esta instancia para entender el impacto de las diferentes tácticas aplicadas. A este escenario lo denominamos **caso base**.

3. Tácticas

En el siguiente trabajo se analizan distintas tácticas para mejorar distintos atributos de calidad en ciertos endpoints con cualidades distintas.

A fin de dar claridad a cada táctica aplicada, se muestran los diagramas de componentes de las tácticas empleadas.

3.1. Caso base

En primer lugar, se analizara el endpoint `/ping`. La idea es utilizar sus resultados como baseline y healthcheck para entender el impacto de las diferentes tácticas aplicadas. A esto lo llamamos caso base.

A continuación se presenta el diagrama de componentes del caso base.

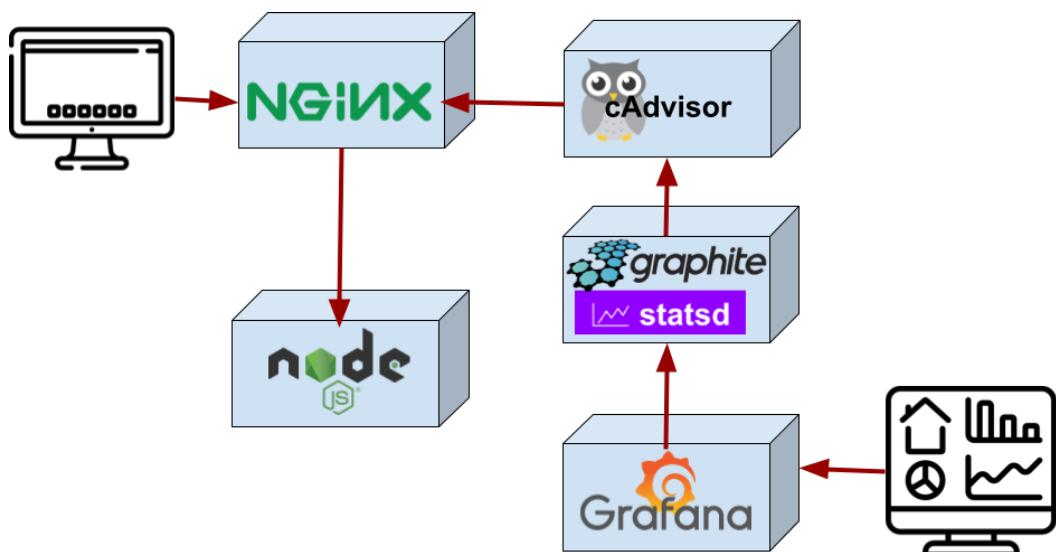


Figura 1: Diagrama de componentes del **caso base**

Se observa que cada cubo representa un contenedor de **Docker** de nuestro sistema con diferentes tecnologías utilizadas.

Tal como muestra el diagrama, la versión básica de la aplicación cuenta con un componente de **Nginx** que maneja las peticiones de los clientes y las deriva al endpoint correspondiente que corren en **NodeJs**. El componente de **Nginx** esta integrado con **cAdvisor**, **Graphite** y **Grafana**. A continuación, se detalla brevemente la funcionalidad de cada una de estas herramientas:

- **cAdvisor**: es una herramienta que ayuda a monitorear el uso de recursos y el rendimiento de contenedores en un entorno *contenerizado*, en este caso, *Docker*. Particularmente, recopila métricas de rendimiento para cada contenedor que se ejecuta en el *host*.

- **Graphite**: es una herramienta para monitorear y graficar métricas en tiempo real. Se utiliza para recopilar y almacenar datos de series de tiempo y proporcionar una forma de ver estos datos en un formato gráfico.
- **Grafana**: es una plataforma de visualización y monitoreo de datos. Se integra con **Graphite**.

El container de nginx es el que conecta al usuario con el servidor de express en el container de node. A su vez, el container de nginx se conecta con el container de cAdvisor, herramienta encargada de otorgar métricas de los containers. Este último también se conecta con el container de graphite donde almacena las métricas que luego va a leer el container de grafana y nos va a permitir ver las métricas a través de la interfaz de grafana.

3.2. Cache

La siguiente táctica que se utilizará para mejorar la performance es cache. Para ello, se agrega un container con ***redis*** dentro ya que se utilizará esa base de datos en memoria para implementar el cache. A continuación, se ilustra el nuevo diagrama de componentes en el cual se le agrega el container de ***redis*** al diagrama del caso base.

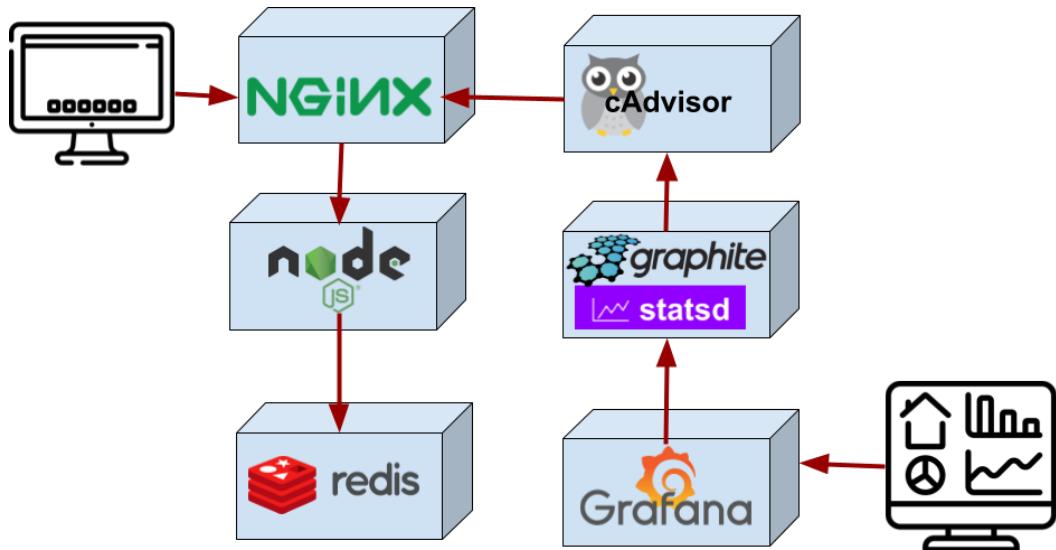


Figura 2: Diagrama de componentes de la táctica *cache*

A priori, las implicaciones de esta decisión de diseño son 2:

1. las solicitudes más frecuentes se pueden servir más rápidamente desde la caché en lugar de hacer la consulta completa al servidor externo. Entonces, mejora la performance.
2. aumenta la complejidad de la aplicación porque hay que configurar la aplicación para trabajar con caché y, entre otras cosas, establecer una estrategia de invalidación de caché adecuada para evitar que los datos obsoletos se almacenen en caché.

Caso 1: /metar Las características de la implementación de una cache para este endpoint son las siguientes:

- **Aplicación:** es *cacheable*. El valor no cambia por un por un tiempo considerablemente mas grande que el **TTL** (Time To Live) definido.
- **Tamaño:** no se logra llenar la memoria. En nuestro caso usamos **Ex: 30**, es decir, 30 segundos de tiempo de expiración.
- **Llenado:** se utiliza *lazy population*, que incorpora información cuando es solicitada por un primer cliente. Se elige esta opción porque hay varios aeropuertos y se quiere guardar la data de los aeropuertos estadísticamente mas significativos.
- **Tiempo de vida:** un dato puede permanecer por un tiempo prolongado ya que el estado del aeropuerto no se actualiza constantemente. Por lo tanto, para no persistir el dato eternamente, se define un **TTL** determinado
- **Vaciado:** cada ítem del caché permanece hasta que se finaliza su tiempo de vida o hasta que un nuevo dato lo reemplace (en caso de llenarse la cache)

Caso 2: /space_news Las características de la implementación de una cache para este endpoint son las siguientes:

- **Aplicación:** es *cacheable*. El valor no cambia por un tiempo considerablemente mas grande que el **TTL** definido.
- **Tamaño:** fijo de 5 noticias
- **Llenado:** se utiliza *active population* porque se desea obtener las 5 noticias mas actuales
- **Tiempo de vida:** un dato permanece hasta que sea actualizado
- **Vaciado:** cada ítem del caché permanece hasta que sea actualizado, independientemente de si se consulto o no

Caso 3: /fact No aplica utilizar la estrategia de cache ya que la información que devuelve es aleatoria (no se desean dos *facts* iguales continuos).

3.3. *Replication*

La segunda táctica que se analizará es la de replicación. En esta, se escala el servicio a 3 copias, convirtiendo a **nginx** en un **load balancer** como se ve en el siguiente diagrama de componentes:

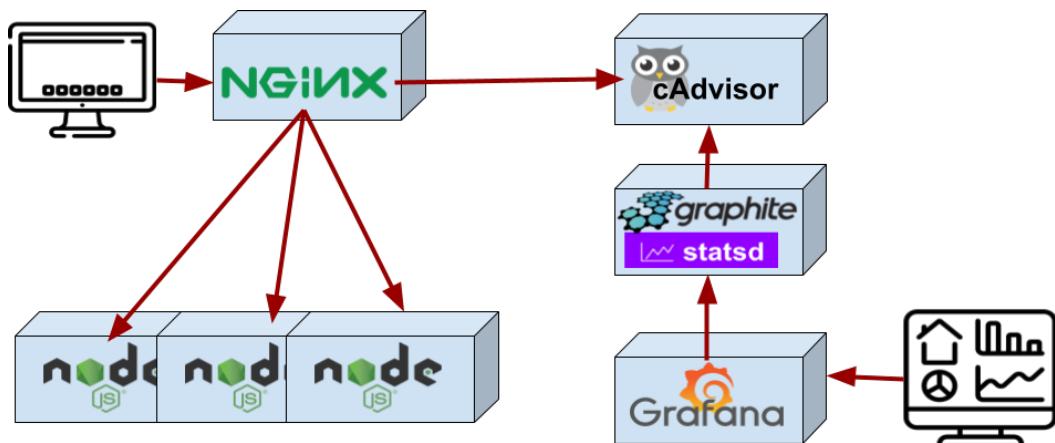


Figura 3: Diagrama de componentes de la táctica *replication*

Agregar réplicas con **Nginx** a la API en **NodeJS** tiene varias implicaciones en cuanto a la mejora del rendimiento y otros atributos de calidad de la aplicación, así también como un aumento de la complejidad.

En cuanto al *rendimiento*, agregar réplicas con **Nginx** puede mejorar la escalabilidad de la aplicación. Al tener varias instancias de la misma en ejecución, se distribuye la carga de las solicitudes entrantes entre ellas. Esto mejora la capacidad de la aplicación para manejar una mayor cantidad de solicitudes. Es decir que esta medida también afecta directamente a la *escalabilidad* de nuestra aplicación.

No obstante, agregar réplicas aumenta la complejidad puesto que se debe configurar **Nginx** para que actúe como *load balancer* y que la distribución de las peticiones sea adecuada.

3.4. Rate Limiting

Rate Limiting es una técnica utilizada para limitar la tasa de solicitudes que una aplicación web o API puede recibir en un determinado período de tiempo. En nuestro caso, definimos distintos parámetros para cada una de las pruebas.

Se definieron distintos valores para los parámetros que definirán nuestro *rate limiting*. Estos son:

- `windowMs`
- `max`

Es decir, se limita la cantidad de requests para cada IP en una “`max`” cantidad cada “`windowMs`” milisegundos.

Caso 1: /ping Se limitaron las solicitudes a 1000 requests por minuto:

```
windowMs: 1 * 60 * 1000,
```

```
max: 1000,
```

Caso 2: /metar Se limitaron las solicitudes a 2000 requests por cada 30 segundos:

```
windowMs: 1 * 30 * 1000,  
max: 2000,
```

Caso 3: /space_news Se limitaron las solicitudes a 1000 requests por cada 30 segundos

```
windowMs: 1 * 30 * 1000,  
max: 1000,
```

Caso 4: /fact Se limitaron las solicitudes a 1000 requests por minuto:

```
windowMs: 1 * 60 * 1000,  
max: 1000,
```

4. Load tests

4.1. /ping

4.1.1. Caso base

Para este test, se definieron dos sub-fases: una fase **ramp** y otra **plain**. Las características de ambas sub-fases quedan especificadas según se muestra a continuación:

- name: Ramp
 - duration: 30
 - arrivalRate: 25
 - rampTo: 300
- name: Plain
 - duration: 60
 - arrivalRate: 300
- name: Clean
 - duration: 10
 - arrivalRate: 1

Los resultados obtenidos fueron los siguientes:

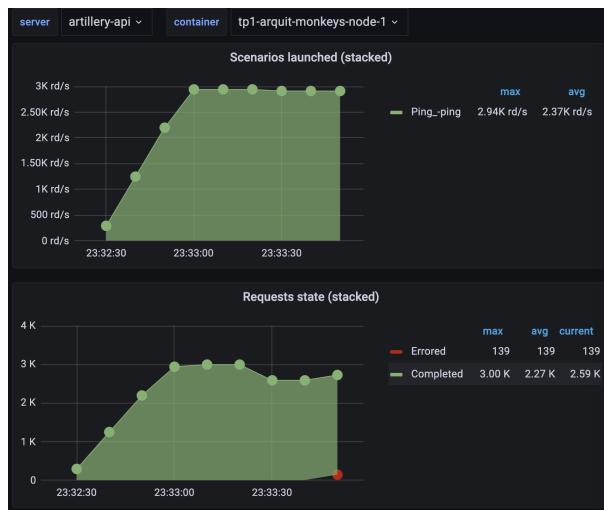


Figura 4: Diagrama de escenarios lanzados y estado de solicitudes del caso base de */ping* para el test de carga

En la primera fase, llamada Ramp, la carga aumenta gradualmente desde 0.25K solicitudes

por segundo hasta 3K solicitudes por segundo en un periodo de 30 segundos. En esta fase, se puede observar un incremento gradual de la cantidad de solicitudes en el segundo gráfico, lo cual indica que la aplicación está respondiendo a las solicitudes de manera efectiva y sin ningún tipo de limitación.

En la segunda fase, llamada Plain, se establece una carga constante de 3K solicitudes por segundo durante 60 segundos. En este caso, se observa un pico en la cantidad de solicitudes en el segundo gráfico al inicio de la fase, lo cual podría indicar que la aplicación necesita un tiempo de calentamiento para adaptarse a la carga constante. Sin embargo, después de este pico inicial, la cantidad de solicitudes se mantiene estable y en línea con la carga establecida.

En general, los dos gráficos muestran que la aplicación puede manejar de manera efectiva una carga gradual y constante de solicitudes, sin ningún tipo de limitación o problema de performance significativo. Sin embargo, se puede notar que en la segunda fase hubo un pequeño pico inicial que podría ser objeto de análisis para verificar si hay algún factor que pudiera estar afectando la respuesta de la aplicación.

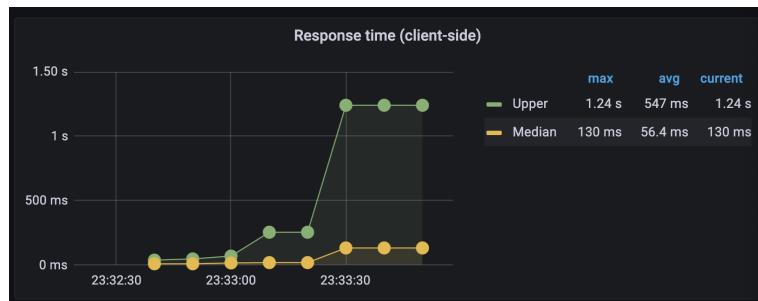


Figura 5: Diagrama de tiempos de respuesta de solicitudes del caso base de */ping* para el test de carga

Este gráfico muestra el tiempo de respuesta de la aplicación en el lado del cliente (es decir, el tiempo que tarda el cliente en recibir la respuesta después de enviar una solicitud) en función del tiempo. Hay algunas cosas que podemos observar en este gráfico:

Durante la fase de ramp, el tiempo de respuesta comienza en un valor relativamente alto, y luego disminuye lentamente a medida que aumenta el número de solicitudes por segundo. Esto es lo que se espera, ya que la aplicación no está optimizada aún.

Durante la fase de plain, el tiempo de respuesta se mantiene relativamente estable durante la primera mitad de la fase, pero luego comienza a aumentar lentamente a medida que se acerca al final. Esto podría ser un signo de que la aplicación no está escalando adecuadamente para manejar la carga de solicitudes entrantes.

Después de la fase de plain, el tiempo de respuesta comienza a disminuir rápidamente a medida que la carga de solicitudes se reduce durante la fase de limpieza. Esto también es lo que se espera, ya que la aplicación no está recibiendo tanta carga durante esta fase.

En resumen, este gráfico nos muestra que la aplicación puede manejar una cierta cantidad de carga de solicitudes sin problemas, pero comienza a tener problemas a medida que la

carga aumenta. Es posible que sea necesario optimizar la aplicación o agregar más recursos para manejar una mayor cantidad de solicitudes sin aumentar significativamente el tiempo de respuesta del cliente.

4.1.2. Replication

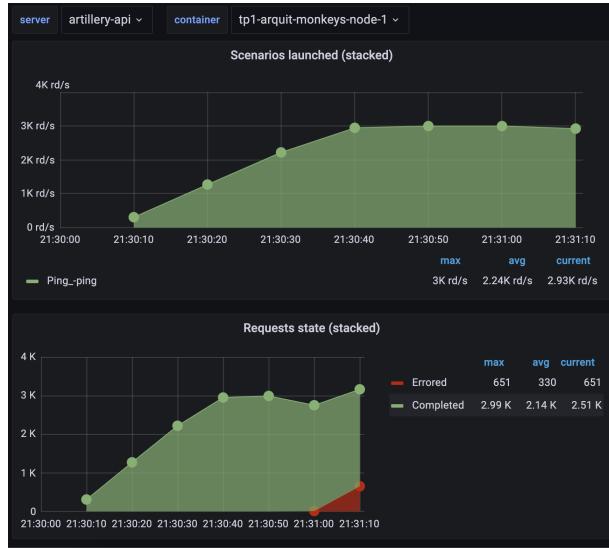


Figura 6: Diagrama de escenarios lanzados y estado de solicitudes de la táctica *replication* de */ping* para el test de carga

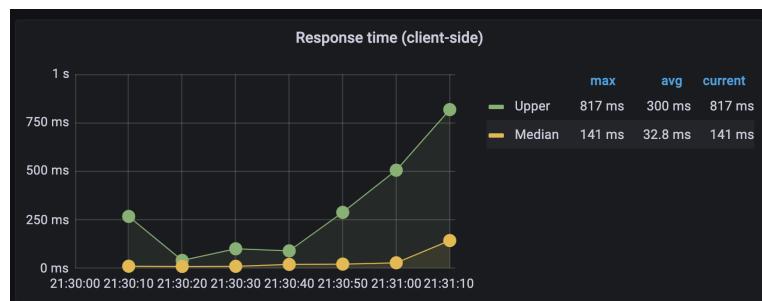


Figura 7: Diagrama de tiempos de respuesta de solicitudes de la táctica *replication* de */ping* para el test de carga

4.1.3. Rate limiting



Figura 8: Diagrama de escenarios lanzados y estado de solicitudes de la táctica *rate limiting* de */ping* para el test de carga

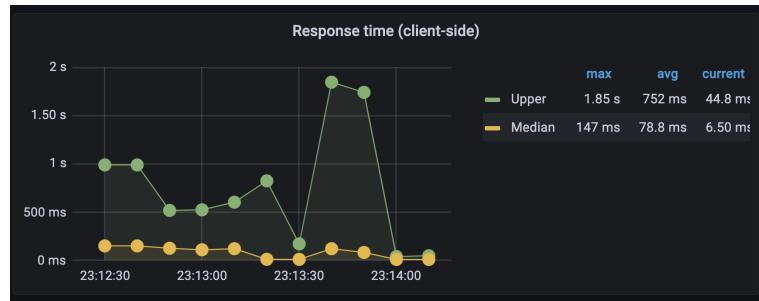


Figura 9: Diagrama de tiempos de respuesta de solicitudes de la táctica *rate limiting* de */ping* para el test de carga

4.2. /metar

4.2.1. Caso base

Los resultados de la prueba de carga quedan visibilizados en las siguientes graficas:

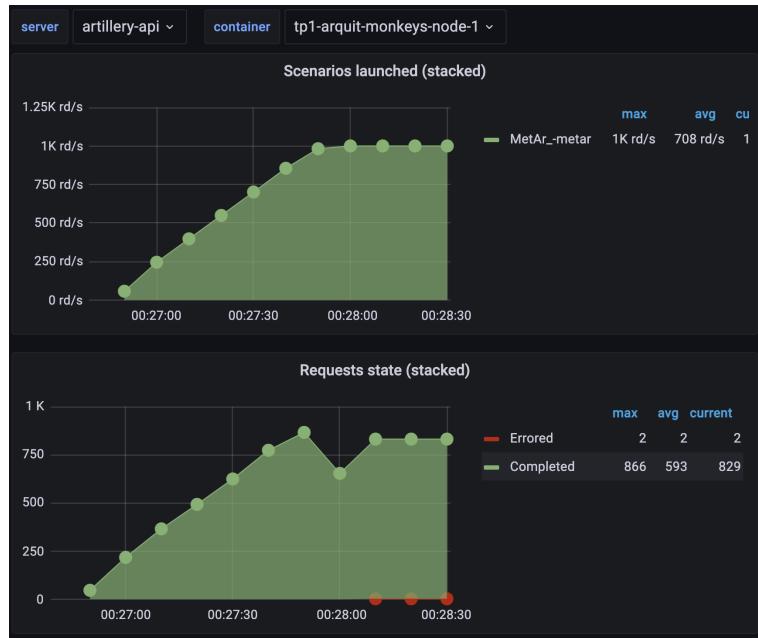


Figura 10: Diagrama de escenarios lanzados y estado de solicitudes del caso base de */metar* para el test de carga

Como indica el grafico, la cantidad de solicitudes comienza en cero y aumenta gradualmente a medida que avanza la prueba, lo que indica que la tasa de llegada de solicitudes también aumenta a medida que se acerca el pico de la prueba. Luego, la cantidad de solicitudes alcanza un pico de 866 solicitudes en un momento dado, antes de disminuir gradualmente a medida que la prueba llega a su fin.

En cuanto a los diferentes estados de respuesta HTTP, podemos ver que la mayoría de las solicitudes tuvieron un estado de respuesta exitoso (200), mientras que una pequeña cantidad de solicitudes tuvieron errores de diferentes tipos (por ejemplo, 404 y 503). También se observa que algunas solicitudes estuvieron en espera y algunas estuvieron activas al mismo tiempo, lo que indica que el servidor estaba procesando múltiples solicitudes al mismo tiempo.

Entonces, este gráfico indica que el endpoint */metar* está siendo utilizado por un número creciente de usuarios a medida que avanza la prueba, lo que es una buena señal. También se pueden identificar algunos errores HTTP, lo que puede ser útil para el equipo de desarrollo para solucionar problemas y mejorar la experiencia del usuario.



Figura 11: Diagrama de tiempos de respuesta de solicitudes del caso base de `/metar` para el test de carga

En este caso, parece que el tiempo de respuesta es bastante estable en general, con algunos picos que se corresponden con momentos de mayor carga.

En la primera parte del gráfico, se ve que el tiempo de respuesta se mantiene relativamente estable y cercano a los 50ms durante los primeros 60 segundos. Luego, a partir de los 60 segundos, hay un pico de tiempo de respuesta que llega a superar los 2s, pero luego se estabiliza nuevamente en torno a los 4.5s. Es posible que este pico se deba a un aumento en la carga del endpoint, como puede apreciarse también en el gráfico de Request State.

Después de este pico, el tiempo de respuesta vuelve a ser relativamente estable, aunque se aprecian algunos picos adicionales, aunque no tan pronunciados como el primero. En general, el tiempo de respuesta parece estar dentro de los límites aceptables, aunque es posible que en momentos de mayor carga se deba monitorear el tiempo de respuesta para evitar que se degrade la experiencia del usuario.

4.2.2. Cache

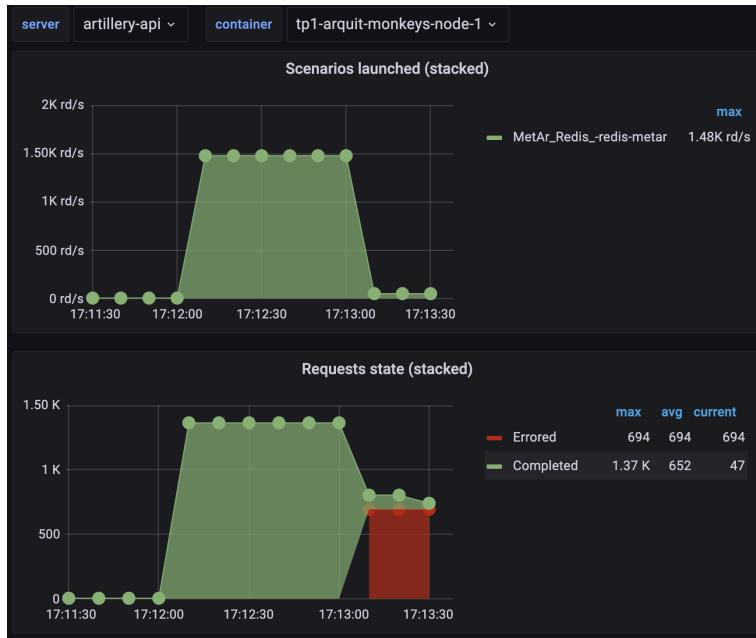


Figura 12: Diagrama de escenarios lanzados y estado de solicitudes de la táctica *cache* de */metar* para el test de carga

El gráfico muestra la evolución del estado de los requests del endpoint */metar* en el tiempo, mientras se utiliza la táctica de caché durante un test de carga.

El área verde indica la cantidad de requests enviadas (primer gráfico) y recibidas por el endpoint, mientras que el área roja representa las que fallaron en ser leídas.

Se puede observar que el endpoint recibe y responde de forma correcta a las requests a lo largo de la ventana de solicitudes, la tasa de éxito aparenta ser del 100 %, y finaliza con ciertos errores coincidentes con el aumento en el tiempo de respuesta, visible en el gráfico siguiente:



Figura 13: Diagrama de tiempo de respuesta de las solicitudes y uso de recursos de la táctica **cache** de */metar* para el test de carga

Este gráfico, al ser comparado con el del caso base de */metar* sugiere que la táctica de caché está funcionando y que el servicio está aprovechando la caché para mejorar su rendimiento, dado que los tiempos de respuesta son menores y se mantienen constantes a lo largo de la ventana de solicitudes. La subida del final no representa algo de carácter valioso para nuestro análisis ya que asumimos que la misma se debe a una caída del servidor.

Además, el segundo gráfico de recursos de esta última figura respalda con una pronunciada caída del uso de la CPU nuestra hipótesis: el caché nos está ahorrando operaciones.

4.2.3. Replication

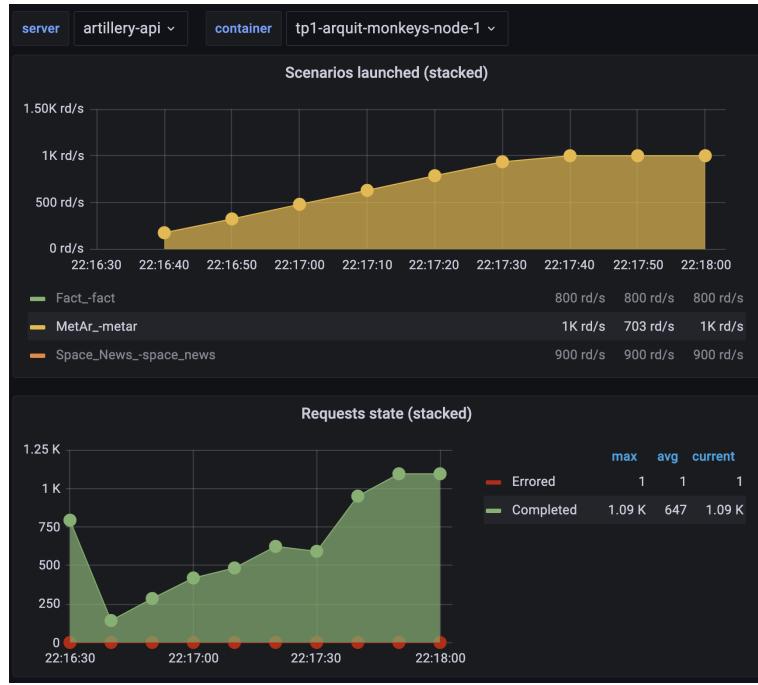


Figura 14: Diagrama de escenarios lanzados y estado de solicitudes de la táctica *replication* de */metar* para el test de carga

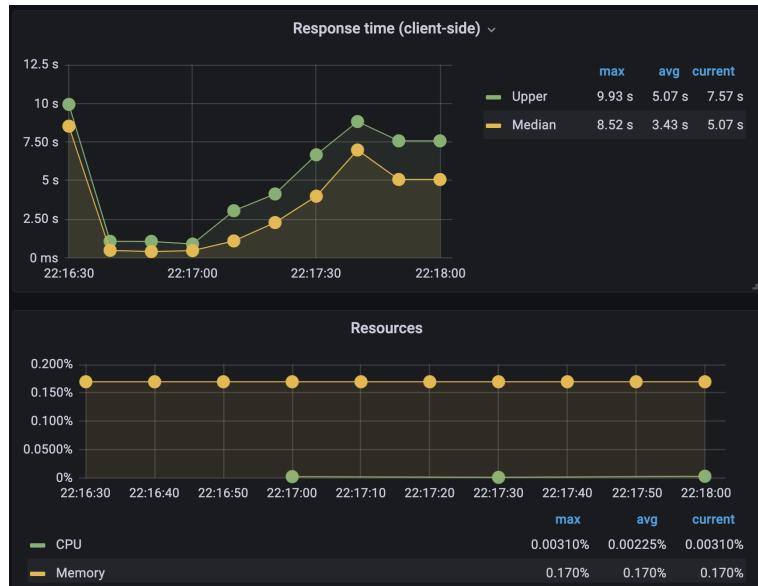


Figura 15: Diagrama de tiempos de respuesta de solicitudes y uso de recursos de la táctica *replication* de */metar* para el test de carga

4.2.4. Rate limiting



Figura 16: Diagrama de escenarios lanzados y estado de solicitudes de la táctica *rate limiting* de */metar* para el test de carga

Notamos que al principio del test como estamos por debajo de la cantidad de requests que se limitan no hay requests **Limited** (hay solo **Completed**).

Y, luego comienza a aumentar la cantidad de requests limitados.

4.3. /space_news

4.3.1. Caso base

Los resultados de esta prueba quedan condensados en las siguientes dos imágenes:

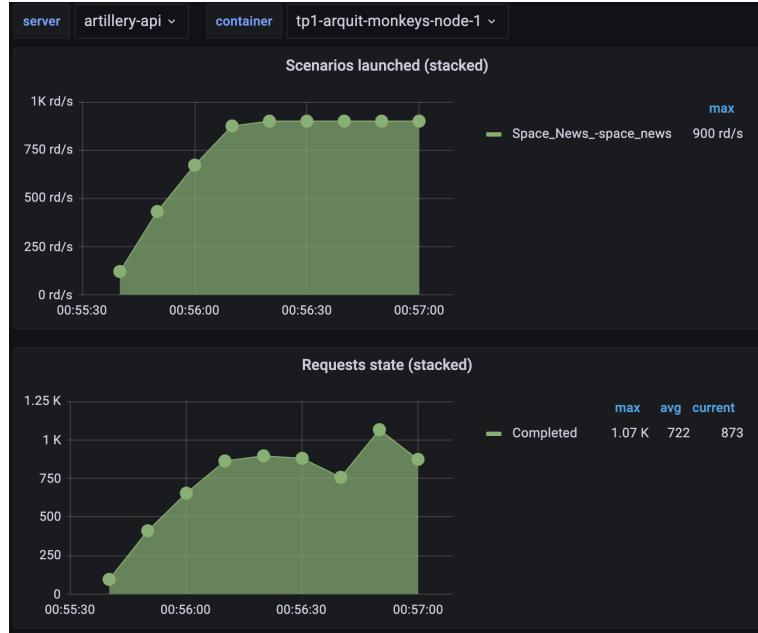


Figura 17: Diagrama de escenarios lanzados y estado de solicitudes del caso base de */space_news* para el test de carga

Podemos observar que el estado más común de las respuestas es 200 OK, que corresponde a una respuesta exitosa del servidor. También hay un número significativo de respuestas con estado 400 (posiblemente, Too Many Requests), lo que indica que el servidor está alcanzando su capacidad máxima y no puede manejar más solicitudes.

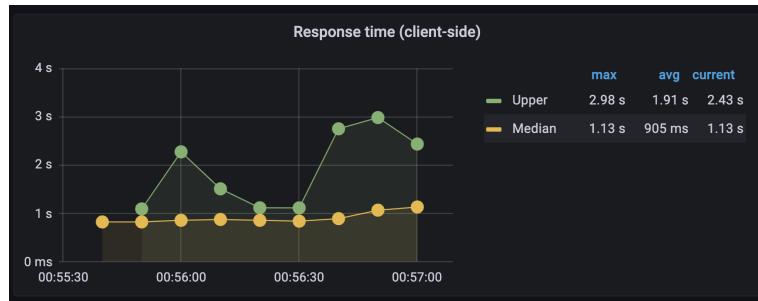


Figura 18: Diagrama de tiempos de respuesta de solicitudes del caso base de */space_news* para el test de carga

En este segundo gráfico se puede observar que el tiempo de respuesta (response time) comienza aumentando gradualmente desde el inicio de la prueba y alcanza un pico alrededor

de las 00:56:00, momento en el que se produce una caída abrupta en la cantidad de solicitudes de tipo *space_news*, tal como se puede ver en el primer gráfico.

Después de esta caída, se puede observar que el tiempo de respuesta comienza a disminuir rápidamente, alcanzando niveles más bajos que antes de la caída, y luego se mantiene en valores relativamente estables hasta el final de la prueba.

Este comportamiento sugiere que la caída en la cantidad de solicitudes de tipo *space_news* a partir de las 00:56:30 pudo haber sido causada por algún problema o cuello de botella en el servidor o en el proceso de procesamiento de las solicitudes, lo que llevó a un aumento en el tiempo de respuesta. Después de la caída, es posible que se haya realizado alguna intervención para solucionar el problema del lado del servicio externo, lo que llevó a una disminución en el tiempo de respuesta. Sin embargo, se necesitaría más información para confirmar estas hipótesis.

4.3.2. Cache

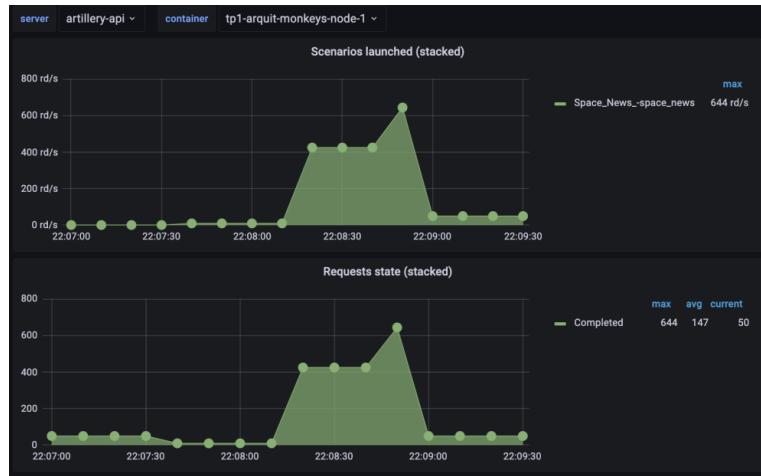


Figura 19: Diagrama de escenarios lanzados y estado de solicitudes de la táctica *cache* de */space_news* para el test de carga



Figura 20: Diagrama de tiempo de respuesta de las solicitudes y uso de recursos de la táctica *cache* de */space_news* para el test de carga

Se nota que el tiempo de respuesta medio es constante y muy bajo en comparacion al caso base mejorando la performance.

Se observan picos en el response time Upper es separaciones de tiempo de medio minuto como el **Time To Live** de lo guardado en el cache. Además, su response time ronda 1s como el promedio del caso base. Se puede inferir que ese aumento en el tiempo de respuesta representan los momentos en los que se liberan los datos de la cache y hay que reponerlos haciendo un request a la API externa.

4.3.3. Replication

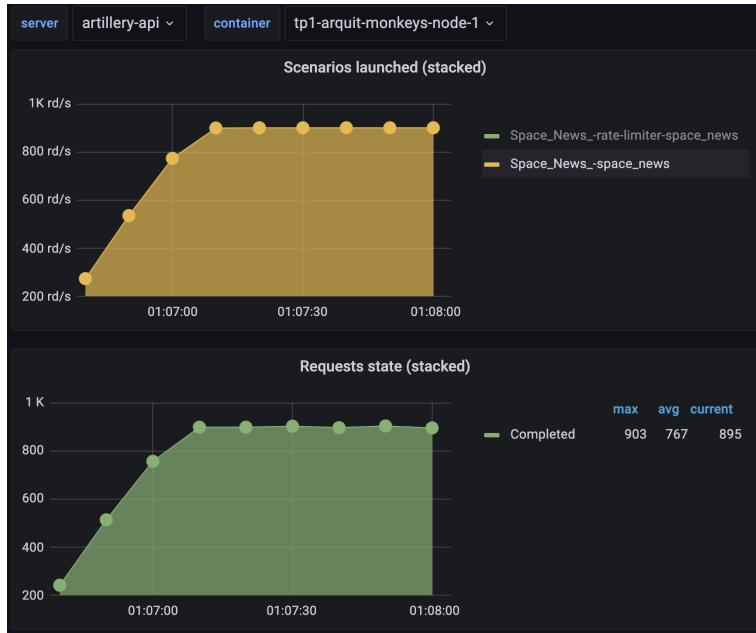


Figura 21: Diagrama de escenarios lanzados y estado de solicitudes de la táctica *replication* de */space_news* para el test de carga

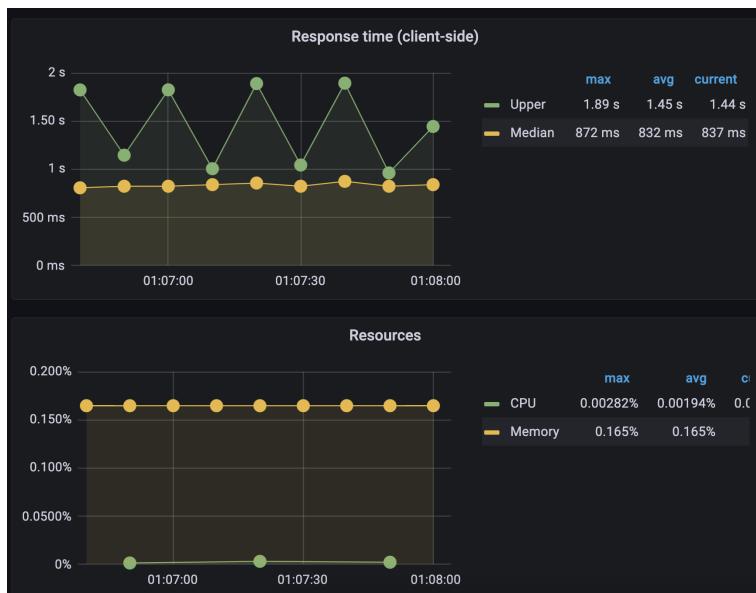


Figura 22: Diagrama de tiempos de respuesta de solicitudes y uso de recursos de la táctica *replication* de */space_news* para el test de carga

Se observa que se logra completar las requests uniformemente respecto a los escenarios lanzados con una muy leve mejora de performance.

4.3.4. Rate limiting

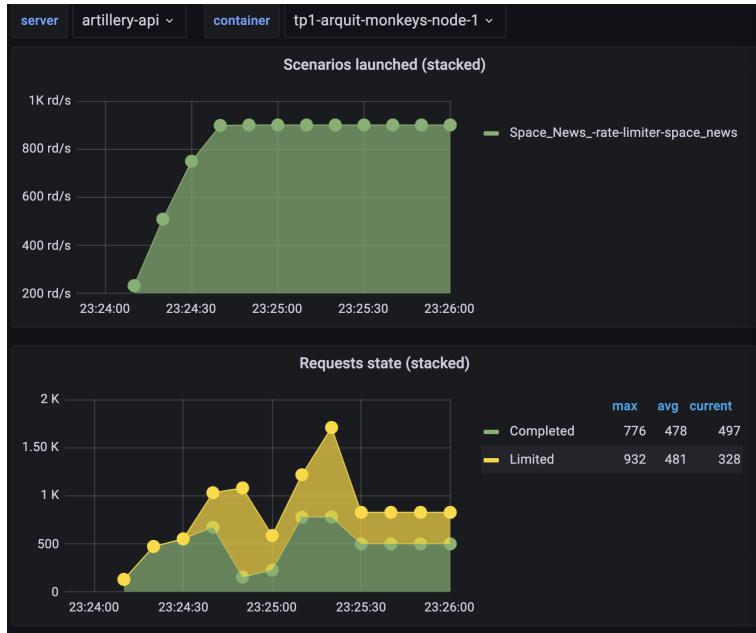


Figura 23: Diagrama de escenarios lanzados y estado de solicitudes de la táctica *rate limiting* de */metar* para el test de carga

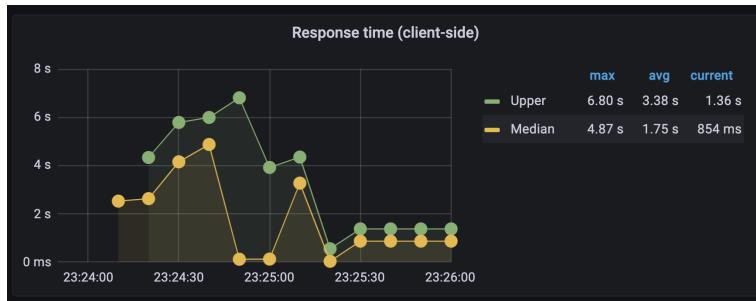


Figura 24: Diagrama de tiempos de respuesta de solicitudes de la táctica *rate limiting* de */space_news* para el test de carga

En este caso, si bien se limitan los requests impidiendo que se caiga el servidor, la performance empeora significativamente ya que no se logran completar varios de los requests,

4.4. /fact

4.4.1. Caso base

Los resultados de la corrida arrojaron los siguientes resultados:



Figura 25: Diagrama de escenarios lanzados y estado de solicitudes del caso base de */fact* para el test de carga

La curva verde representa las solicitudes completadas con éxito, mientras que la curva roja representa las solicitudes que arrojaron errores.

Al analizar el gráfico, podemos ver que la cantidad de solicitudes aumenta con el tiempo a medida que se produce el aumento de la tasa de llegada de usuarios durante la primera fase de rampa, y se mantiene alta durante la segunda fase de Plain donde se mantiene la tasa de llegada de usuarios.

La curva verde, que representa las solicitudes completadas con éxito, muestra un pico en el número de solicitudes completadas en torno al segundo 60. Esto se debe probablemente a una menor carga de usuarios en ese momento. Después de ese pico, la cantidad de solicitudes completadas con éxito se mantiene alta y estable a medida que la tasa de llegada de usuarios se mantiene alta.

La curva roja, que representa las solicitudes que arrojaron errores, muestra un pico de errores en torno al segundo 60, justo después del pico en la cantidad de solicitudes completadas con éxito. Esto podría deberse a que la capacidad del servidor se vio abrumada en ese momento, lo que llevó a un aumento en la cantidad de solicitudes que arrojaron errores. Después de ese pico, la cantidad de solicitudes que arrojan errores disminuye a medida que la tasa de llegada de usuarios se mantiene estable. En general, el número de solicitudes que arrojan errores se mantiene bajo durante todo el período de la prueba.



Figura 26: Diagrama de tiempos de respuesta de solicitudes y uso de recursos del caso base de `/fact` para el test de carga

En el primer gráfico, que mide el tiempo de respuesta en función del tiempo, podemos observar que la curva roja (correspondiente al tiempo de respuesta) comienza en un nivel relativamente bajo, aumenta drásticamente y luego fluctúa durante todo el período mostrado. El aumento drástico en el tiempo de respuesta podría indicar una sobrecarga del sistema o un aumento repentino en la demanda. A partir de entonces, la fluctuación podría sugerir problemas intermitentes con el rendimiento del sistema.

En el segundo gráfico, que mide el uso de recursos en función del tiempo, podemos ver que hay 2 curvas de diferentes colores que indican el uso de CPU y memoria, respectivamente. En el caso del uso de CPU, vemos un aumento de este recurso hacia el comienzo del período, seguido de un período de disminución gradual. Luego, el uso de CPU aumenta y disminuye de manera intermitente, lo que sugiere que hay picos intermitentes de actividad en el sistema que requieren más recursos. En lo referido a la memoria, el consumo es constante.

4.4.2. Replication

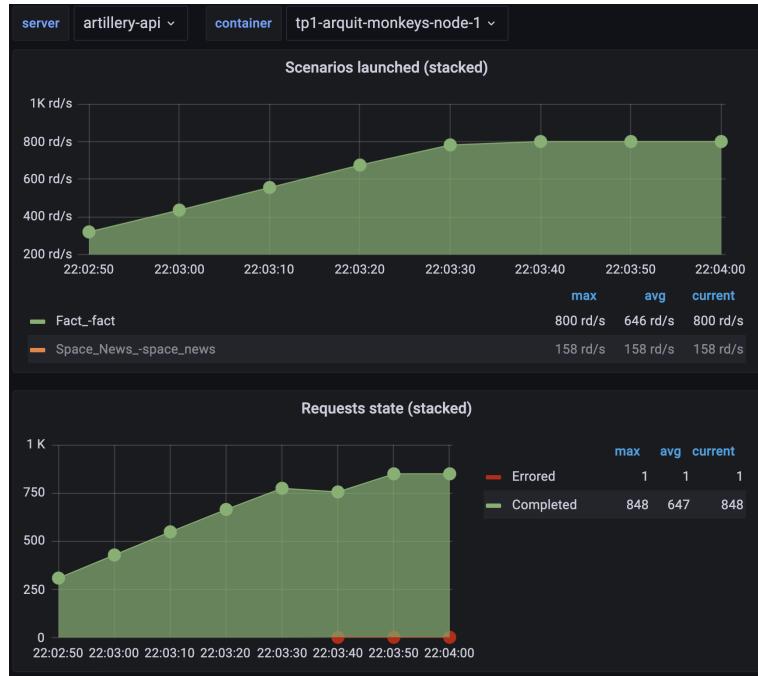


Figura 27: Diagrama de escenarios lanzados y estado de solicitudes de la táctica *replication* de */fact* para el test de carga



Figura 28: Diagrama de tiempos de respuesta de solicitudes y uso de recursos de la táctica *replication* de */fact* para el test de carga

4.4.3. Rate limiting



Figura 29: Diagrama de escenarios lanzados y estado de solicitudes de la tactica *rate limiting* de */fact* para el test de carga

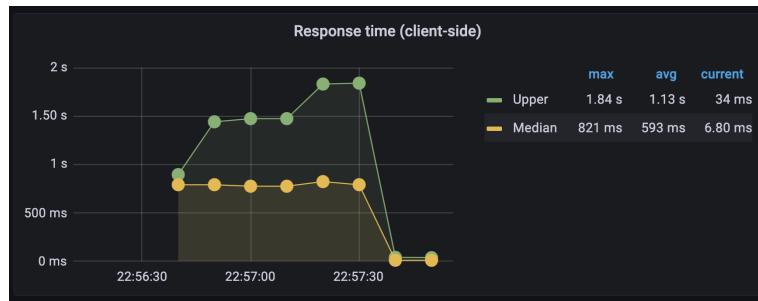


Figura 30: Diagrama de tiempos de respuesta de solicitudes de la táctica *rate limiting* de */fact* para el test de carga

Otra vez, se observa que el response time disminuye considerablemente cuando mas requests se encuentran limitadas.

5. Stress tests

5.1. /ping

5.1.1. Caso base



Figura 31: Diagrama de escenarios lanzados y estado de solicitudes del caso base de */ping* para el test de estrés



Figura 32: Diagrama de tiempos de respuesta de solicitudes y uso de recursos del caso base de */ping* para el test de estrés

5.1.2. Replication

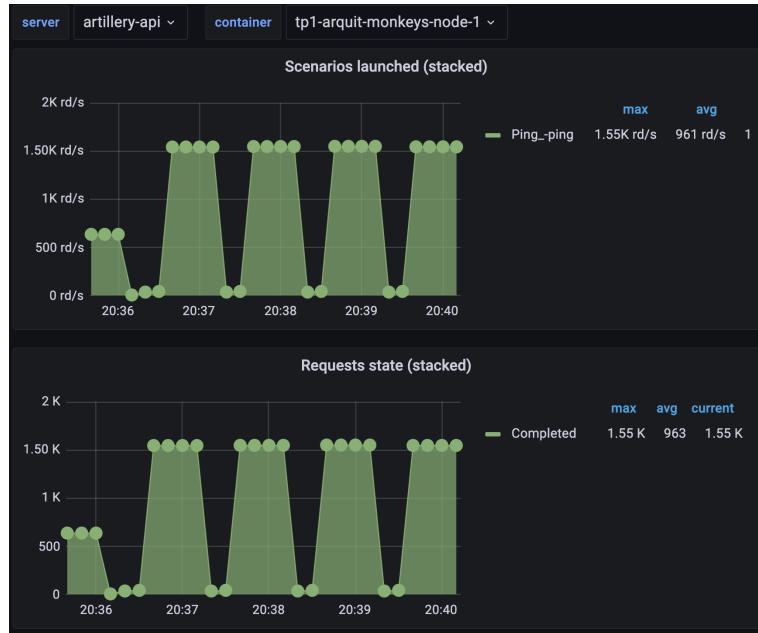


Figura 33: Diagrama de escenarios lanzados y estado de solicitudes de la táctica *replication* de */ping* para el test de estrés



Figura 34: Diagrama de tiempos de respuesta de solicitudes y uso de recursos de la táctica *replication* de */ping* para el test de estrés

5.1.3. Rate limiting



Figura 35: Diagrama de escenarios lanzados y estado de solicitudes de la táctica *rate limiting* de `/ping` para el test de estrés

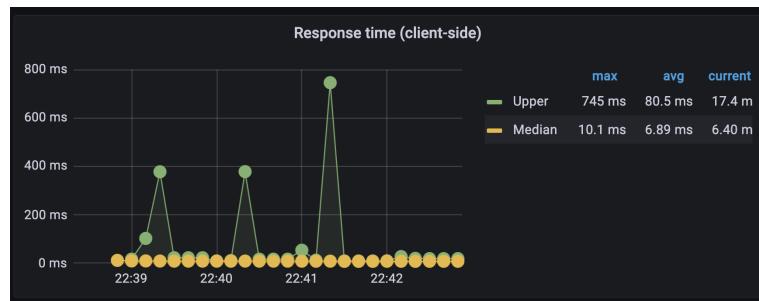


Figura 36: Diagrama de tiempos de respuesta de solicitudes de la táctica *rate limiting* de `/ping` para el stress test

5.2. /metar

5.2.1. Caso base



Figura 37: Diagrama de escenarios lanzados y estado de solicitudes del caso base de */metar* para el test de estrés



Figura 38: Diagrama de tiempos de respuesta de solicitudes y uso de recursos del caso base de */metar* para el test de estrés

5.2.2. Cache



Figura 39: Diagrama de escenarios lanzados y estado de solicitudes de la táctica *cache* de */metar* para el test de estrés

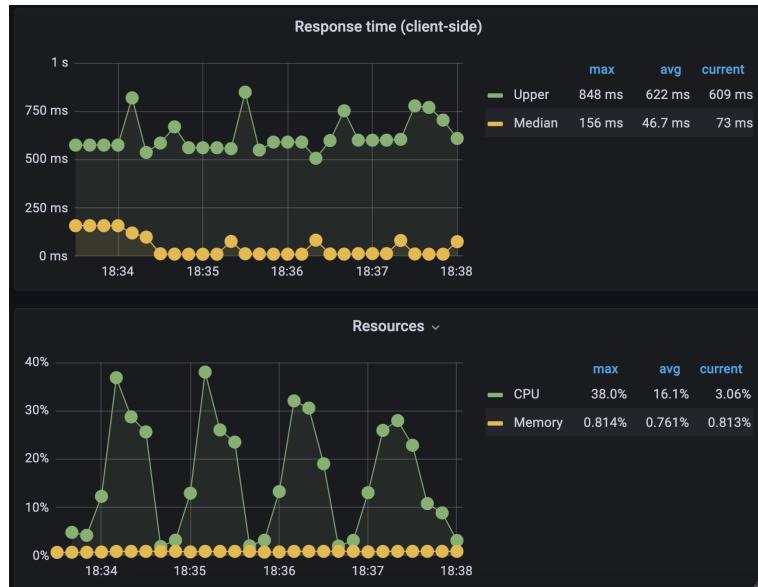


Figura 40: Diagrama de tiempo de respuesta de las solicitudes y uso de recursos de la táctica *cache* de */metar* para el test de estrés

Por un lado, se observa que el response time en el caso base las métricas de Upper y “Median” rondan por los mismos niveles, por lo que no hay gran fluctuación de valores de response time.

Mientras tanto, con la táctica de caché si hay una gran diferencia que al parecer se relaciona con que hay ciertos momentos en el que se debe consultar la API para guardar el caché cuando este no está. También se observa que en el caso base cada vez el response time aumenta mientras que con el método cache permanece más constante y con un valor promedio de mediana significativamente menor: 46.7 ms (aproximadamente cinco veces menor).

También se observa un mayor response time al principio del test cuando la caché estaba vacía y se debía consultar la API externa, esto se debe a que se utilizó la estrategia lazy population.

Por ende, aunque agregar caché aumenta el atributo de calidad de complejidad, si mejora significativamente la performance en comparación al caso base.

5.2.3. Replication

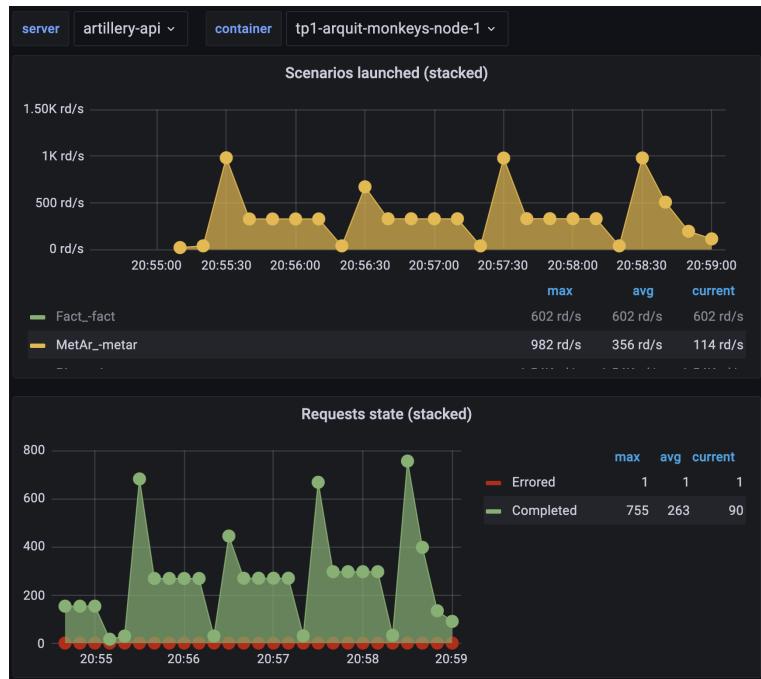


Figura 41: Diagrama de escenarios lanzados y estado de solicitudes de la táctica *replication* de */metar* para el test de estrés

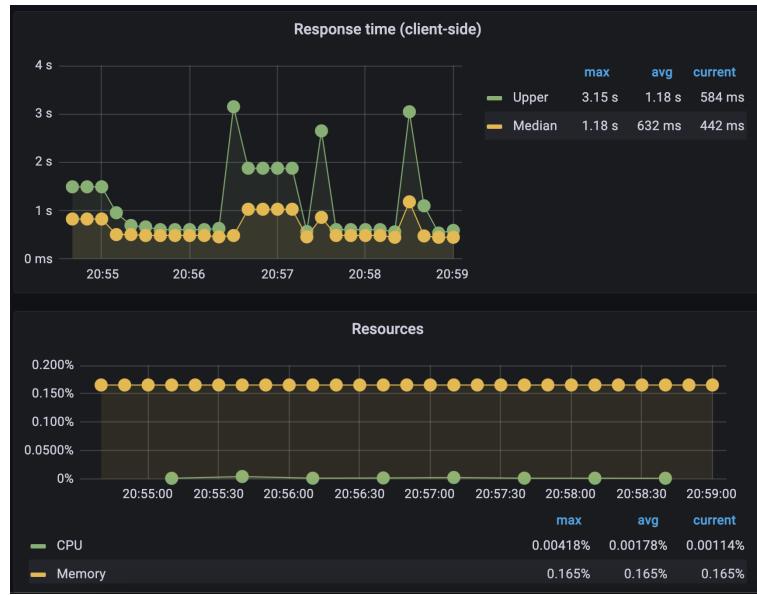


Figura 42: Diagrama de tiempos de respuesta de solicitudes y uso de recursos de la táctica *replication* de */metar* para el test de estrés

Por un lado, se observa que el response time en este caso tanto el Upper como el Median no varian tanto como en el caso de Cache y tiene la mitad de los valores que tiene el caso base. Por ende, se ve una mejora de performance.

Más aún, se nota que hay menos errores, se ve que hay más escalabilidad lo que se puede ver reflejado en contraposición al disminuido porcentaje de uso de CPU.

5.2.4. Rate limiting



Figura 43: Diagrama de escenarios lanzados y estado de solicitudes de la táctica *rate limiting* de */metar* para el test de estrés

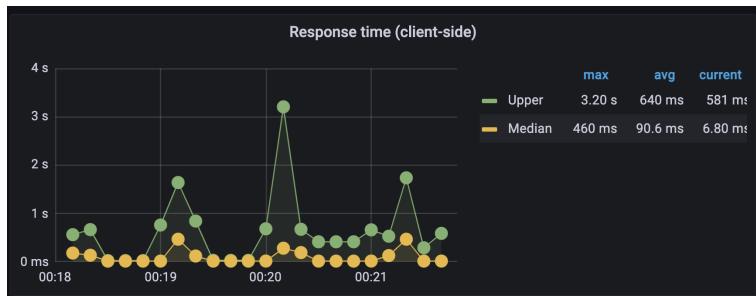


Figura 44: Diagrama de tiempos de respuesta de solicitudes de la táctica *rate limiting* de *metar* para el test de estrés

En este gráfico podemos ver que también el response time tampoco tienen grandes diferencias entre los valores Upper y Median. Además, vemos que la performance ya que el response time es aproximadamente 4 veces menor.

También, vemos que no hay requests con errores ya que se limita la cantidad de requests procesados.

También vemos en el gráfico de requisitos que hay requisitos de dos tipos: completados y limitados (por el rate limiting). Por un lado, vemos que si se limita de a medio minuto y que el máximo es 1000 requests.

5.3. /space_news

5.3.1. Caso base



Figura 45: Diagrama de escenarios lanzados y estado de solicitudes del caso base de */space_news* para el test de estrés



Figura 46: Diagrama de tiempos de respuesta de solicitudes y uso de recursos del caso base de */space_news* para el test de estrés

5.3.2. Cache



Figura 47: Diagrama de escenarios lanzados y estado de solicitudes de la táctica *cache* de */space_news* para el test de estrés

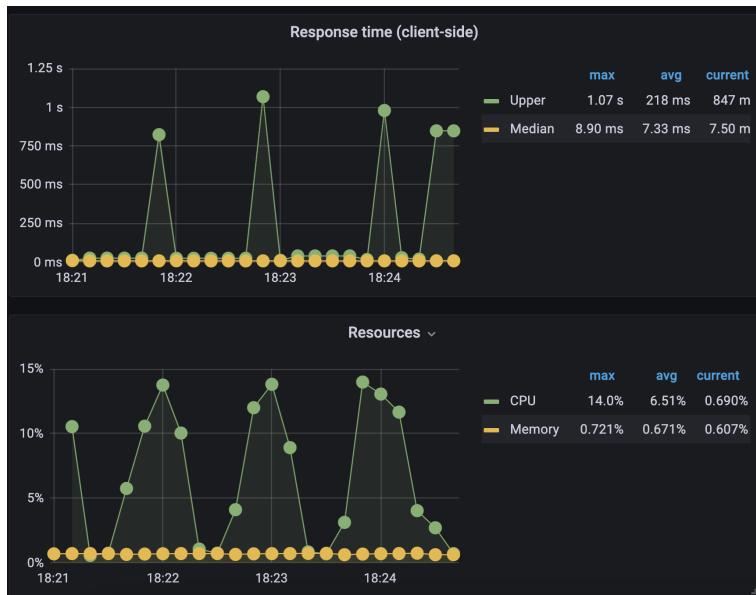


Figura 48: Diagrama de tiempo de respuesta de las solicitudes y uso de recursos de la táctica *cache* de */space_news* para el test de estrés

Análogamente al caso del endpoint *metar*, el endpoint *space news* tiene el response time Median es significativamente menor que el caso base. Además vemos que el response time

Upper tiene picos (cuando se limpia el caché y tengo que volver a guardar datos en la caché). En comparación al endpoint metar, el endpoint space news tiene subidas más cortas ya que el valor cacheado es 1 (las 5 noticias más recientes) mientras que en el endpoint metar se cachean distintos valores dependiendo del aeropuerto pasado por parámetro.

5.3.3. Replication

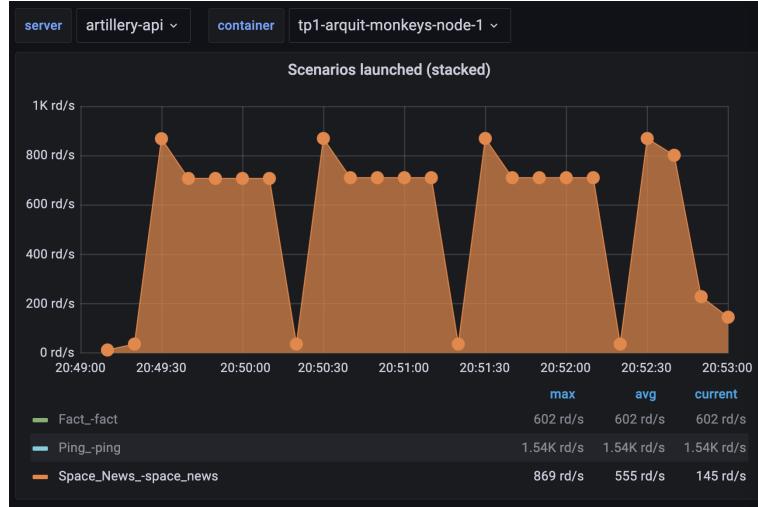


Figura 49: Diagrama de escenarios lanzados de la táctica *replication* de */space_news* para el test de estrés



Figura 50: Diagrama de estado de solicitudes y tiempos de respuesta de las solicitudes de la táctica *replication* de */space_news* para el test de estrés

Primero, vemos que tenemos menos errores.

Segundo vemos que la performance mejora aproximadamente 3 veces en base al response time.

Notamos además que hay una mayor separación entre los valores Upper y Median de response time.

Notamos que el response time en el caso base a medida que pasa el tiempo incrementa mientras que replicando nodos va disminuyendo.

5.3.4. Rate limiting



Figura 51: Diagrama de escenarios lanzados y estado de solicitudes de la táctica *rate limiting* de `/space_news` para el test de estrés

Se observa que cada 1 minuto se libera la cantidad de requests limitadas y, por ende, vemos picos periódicos de requests completadas respecto a las limitadas.



Figura 52: Diagrama de tiempos de respuesta de solicitudes de la táctica *rate limiting* de `/space_news` para el test de estrés

Observamos que el response time tiene una bajada en tiempo promedio cuando se llega al máximo de request permitidas y se comienza a limitar mejorando la performance ya que se procesan menos requests.

5.4. /fact

5.4.1. Caso base



Figura 53: Diagrama de escenarios lanzados y estado de solicitudes del caso base de */fact* para el test de estrés

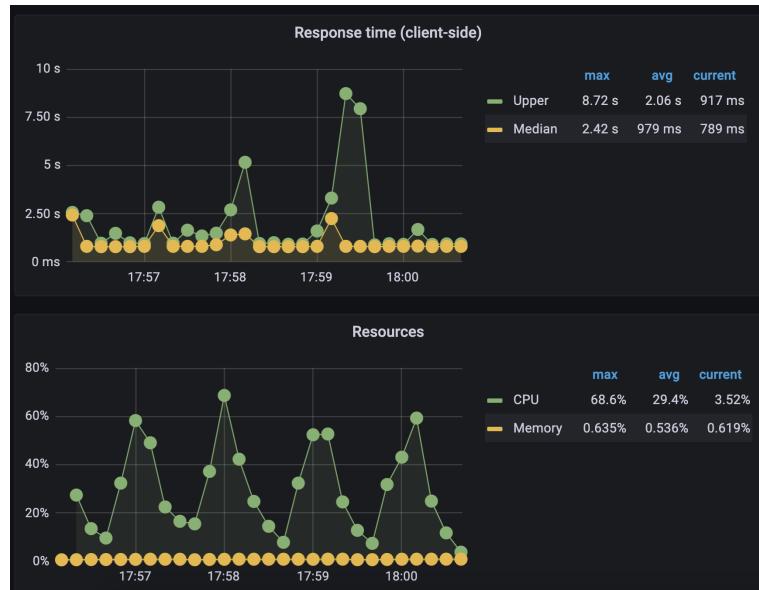


Figura 54: Diagrama de tiempos de respuesta de solicitudes y uso de recursos del caso base de `/fact` para el test de estrés

Se observa que en cada pico, el response time **Upper** va aumentando aproximadamente el doble pero el response time **Median** se mantiene aproximadamente constante.

5.4.2. Replication

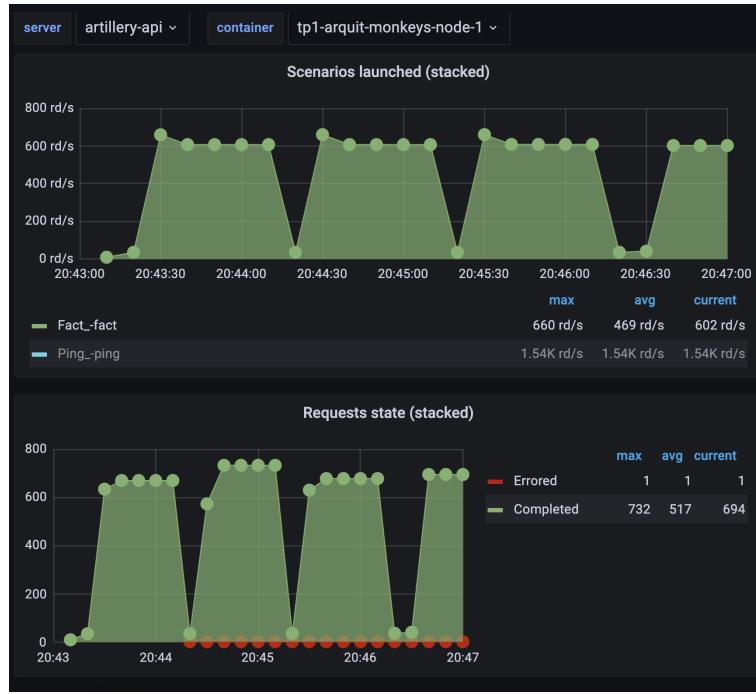


Figura 55: Diagrama de escenarios lanzados y estado de solicitudes de la táctica *replication* de */fact* para el test de estrés



Figura 56: Diagrama de tiempos de respuesta de solicitudes y uso de recursos de la táctica *replication* de */fact* para el test de estrés

Se observa que los escenarios lanzados no tienen los picos tan pronunciados como en otros tests. Esto es así ya que nuestro servidor puede aceptar mas requests en simultaneo ya que hay varias instancias del servidor listas para procesarlas. Se evitan cuellos de botella.

Mas aun, se observa que el tiempo de respuesta disminuye considerablemente a la mitad.

5.4.3. Rate limiting

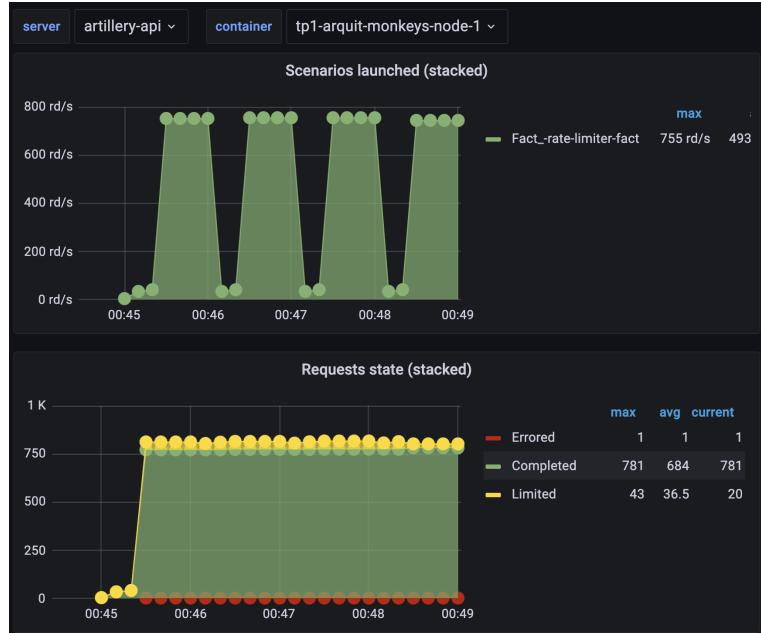


Figura 57: Diagrama de escenarios lanzados y estado de solicitudes de la táctica *rate limiting* de */fact* para el test de estrés



Figura 58: Diagrama de tiempos de respuesta de solicitudes de la táctica *rate limiting* de *fact* para el test de estrés

6. Conclusiones

Luego de analizar la aplicación de las distintas tácticas, podemos ver cómo se ven afectados de distinta forma los atributos de calidad. A continuación y a modo de conclusión se hace un análisis cualitativo de los mismos por cada táctica aplicada y siendo comparados con el caso base como referencia.

Cache

La implementación de **cache** significó una importante ganancia en **performance** para todos los casos en los que fue aplicada. Los gráficos muestran picos en los momentos que hay fallos en la caché, pero bajas mesetas comparadas con el caso base en el que el tiempo de respuesta promedio es mucho mayor (llegando a verse tres órdenes de magnitud de diferencia). Sin embargo, podemos afirmar que se perdió cierta **simplicidad** ya que en este caso debió ser usada la base de datos en memoria **Redis**.

Replicación

Replicar instancias permite tener mayor **escalabilidad**, es decir que sea sencillo dar soporte al aumento de la cantidad de usuarios. Se ve claramente, tanto en los tests de carga como de estrés, que en un principio el tiempo de respuesta es alto, cercano al caso base, y luego empieza a decrecer a medida que el tiempo pasa. Esto es debido a que la carga es distribuida entre las tres instancias, permitiendo mejorar estos tiempos de respuesta. Además, en caso de que una instancia se vea afectada tenemos la posibilidad de no dejar de prestar el servicio, aumentando la **confiabilidad** del sistema.

Rate limiting

Limitar la cantidad de requests por una ventana de tiempo definida significó en pocos casos mejoras en la performance, siempre teniendo en cuenta que no se estuvo dando respuesta a todos los pedidos. Por otro lado, esta táctica mejoraría la **seguridad y disponibilidad** (SLA) del sistema ya que estamos evitando sobrecargas en los recursos, lo cual incluso impacta en costos si se trabaja con un proveedor en la nube. Se vería afectada negativamente la **usabilidad y escalabilidad**, no permitiendo dar soporte a una crecida de usuarios, lo cual queda en evidencia en los gráficos analizados. Sería interesante aplicar ésta táctica como complemento de las otras, ya que favorece otros atributos de calidad.

7. Apéndice: métricas propias

7.0.1. Tiempo de respuesta local

En esta sección, se toma el endpoint `/metar` como ejemplo para comparar el tiempo de respuesta de nuestra implementación con los llamados direccionalos directamente a la API externa. Este análisis tiene el objetivo de demostrar la eficiencia de nuestro sistema al incorporar un cache Redis en el servidor Nginx. A priori, se espera que el uso de la cache permita reducir las consultas a la API externa, mejorando el tiempo de respuesta general. Los gráficos presentados ilustran claramente las ventajas de nuestra implementación en comparación con un sistema que siempre consulta a la API externa, es decir, sin cache.

Test de estrés

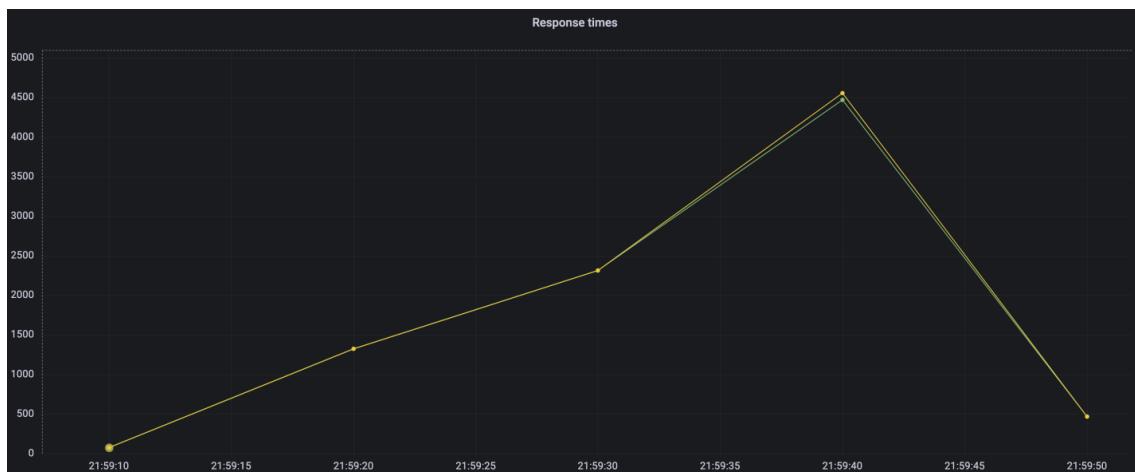


Figura 59: Tiempo propio vs. API externa sin cache

El gráfico presentado ilustra dos curvas distintas que representan el tiempo de respuesta al realizar una solicitud a la API externa. La curva verde muestra el tiempo de respuesta cuando se realiza la solicitud directamente al servicio externo. En contraposición, la curva amarilla representa el tiempo de respuesta al acceder al servicio externo a través de nuestra propia API. Cabe destacar que en ninguno de los casos se hace uso del recurso *cache*.

A pesar de que ambas curvas se comportan de manera muy similar, se puede apreciar una pequeña diferencia en favor de la curva verde, que muestra un tiempo de respuesta ligeramente más corto. Esta diferencia se debe al preprocesamiento que realiza nuestra API: una conversión de la respuesta XML a JSON.

Este preprocesamiento implica un aumento mínimo en el tiempo de respuesta, pero proporciona una estructura de datos más manejable y de mayor utilidad para nuestro sistema que prefiere el formato JSON (aporta una mejora significativa en términos de facilidad de manejo de datos).



Figura 60: Tiempo propio vs. API externa usando cache

Este gráfico presenta dos curvas distintas, representando cada una un método distinto de manejo de las consultas a la API externa.

La curva verde, que visualiza las consultas realizadas directamente al servicio externo sin hacer uso del cache de Redis, exhibe un tiempo de respuesta mayor. Esto es evidencia de la latencia inherente a cada consulta que requiere un contacto directo con el servicio externo.

Por otro lado, la curva amarilla representa el tiempo de respuesta de las consultas gestionadas por nuestra implementación, que incorpora una memoria cache de Redis. Esta configuración permite que, si una consulta ya ha sido realizada previamente, nuestra API resuelva la solicitud internamente, obviando la necesidad de realizar una nueva consulta al servicio externo. El resultado de esta estrategia es evidente: la curva muestra un tiempo de respuesta mucho menor en comparación con las consultas directas al servicio externo.

La diferencia marcada entre las dos curvas demuestra claramente cómo la incorporación de la memoria cache de Redis en nuestra implementación mejora significativamente la eficiencia global y reduce la latencia. De este modo, se proporciona un mejor rendimiento, a pesar de la carga adicional del preprocesamiento para la conversión XML a JSON.

Test de carga

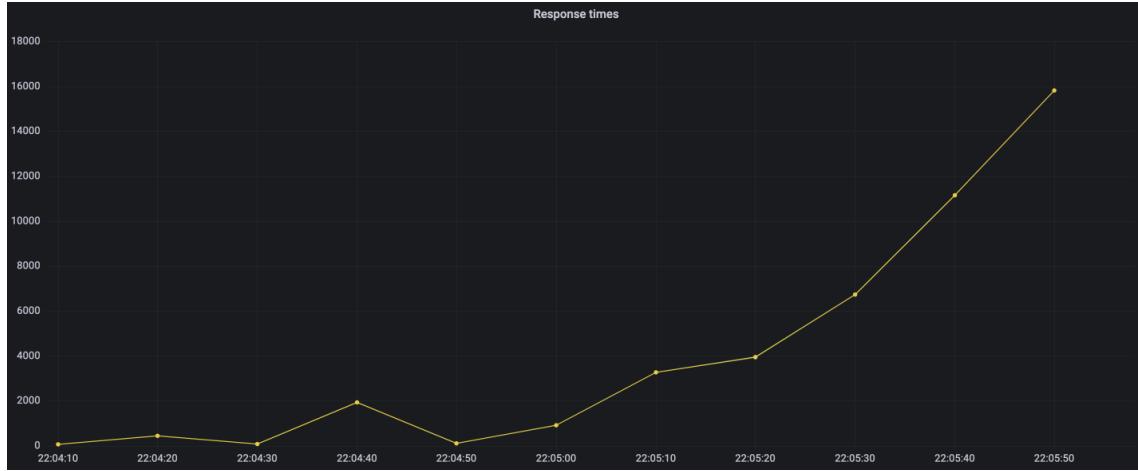


Figura 61: Gráfico de test de carga para tiempo propio vs. API externa sin uso de cache

Este gráfico muestra la relación entre el aumento de la cantidad de solicitudes al servidor externo y la carga resultante para ambos métodos de consulta sin cache: directamente y a través de nuestra API.

Tanto la curva que representa las solicitudes directas al servidor externo como la que refleja las consultas realizadas a través de nuestra API muestran un patrón similar: a medida que aumenta el número de solicitudes, la carga en el servidor externo incrementa en la misma medida. Esto indica que ambas modalidades de consulta generan una carga equivalente en el servidor externo.

Este comportamiento refuerza las conclusiones previas de nuestros análisis. Aunque nuestra API introduce un preprocesamiento, no introduce ninguna variación en la carga del servidor externo con respecto a las solicitudes directas, es decir que no se compromete la capacidad de carga del servidor externo.



Figura 62: Gráfico de test de carga para tiempo propio vs. API externa usando cache

En este último gráfico, se presentan dos curvas que representan distintos escenarios en la gestión de solicitudes al servidor externo.

La curva celeste muestra la carga sobre el sistema en el caso de las solicitudes directas al servidor externo, sin la implementación de cache. Este escenario refleja un mayor consumo de recursos, que se mantienen ocupados por más tiempo antes de ser liberados.

Por otro lado, la curva naranja ilustra las solicitudes gestionadas por nuestra API, que incorpora un cache de Redis. Este escenario presenta una rápida disminución de la carga en el sistema. En numerosos casos, la presencia del cache significa que el sistema externo no necesita ser consultado, lo que reduce aún más la carga.

El contraste entre estas dos curvas resalta claramente los beneficios de nuestra implementación. La incorporación del cache de Redis reduce significativamente la carga sobre el sistema externo, optimizando el consumo de recursos y liberándolos más rápidamente. Además, nuestra implementación evita en numerosas ocasiones la necesidad de contactar con el sistema externo, lo que previene la saturación de dicho sistema y mejora su rendimiento global.