Arquitectura de Software (75.73)

Trabajo Práctico Nº1

2do Cuatrimestre de 2023

Integrantes:

- Nazario Juarez, Ingrith Elizabeth - 99408
- Lovera López, Daniel Alejandro - 103442

Consigna

Crear un servicio HTTP utilizando la librería Express que a su vez consume de otras APIs para brindar la información necesaria a sus usuarios.

La API (nuestro servicio HTTP) será sometida a distintos escenarios de carga para poder analizar los datos y aplicar las diversas tácticas vistas en clase.

Análisis

La aplicación desarrollada actua como un API Gateway, redirige el tráfico proveniente del servidor de Ngnix hacia el servicio HTTP implementado en node con express. El Gateway envía datos a tres APIs con diferentes características cada una:

1. **Metars:** Es una API que se encarga de devolver información meteorológica de aeropuertos del mundo basada en su código de estación. Tiene el tiempo de respuesta medio más rapido con alrededor de 120 ms bajo un periodo de carga normal, no se encuentra limitada las requests por segundo pero si es capaz de denegar el servicio a un cliente cuando se ejecutan demasiadas llamadas durante un periodo de tiempo, es decir que internamente deben detectarlo como un intento de denegación de servicio. Adicionalmente requiere un mayor procesamiento por parte del Gateway para convertir los datos obtenidos a formato JSON.

2. **Spaceflight News:** Es una API que se encarga de devolver noticias sobre eventos del espacio. Tiene el tiempo de respuesta medio más alto con 1,8 segundos bajo un período de carga normal, se desconoce si es capaz de limitar las request o denegar el servicio porque el Gateway empieza a tener errores de conexión con la API externa como (ECONNRESET o desconexiones en la conexión de TCP antes de establecer la conexión de TSL) o ETIMEDOUT que es enviado por Nginx cuando una conexión tardó demasiado en responder.

3. **Quotes:** Es una API que se encarga de generar citas famosas aleatorias. Tiene un tiempo de respuesta de medio de 600 ms. El servicio ya cuenta con un rate limiter que impide enviar una cantidad mayor a 180 requests por segundo, lo cual es una limitante fuerte ya que la mayor cantidad de carga que se puede enviar por segundo es ese número.

Objetivo

Visualizar cómo los escenarios de carga y tacticas de implementación impactaran en los atributos de calidad de la aplicación para cada API planteada al ser comparadas contra el caso base, el cual es una aplicación sin réplicas, caching o rate limiters.

Desarrollo

Para cada caso se aplicará una prueba de rendimiento llamada Load Testing, la cual consiste tres etapas planteadas con los siguientes criterios:

1. **Etapa de calentamiento:** se simula un número de clientes que el servidor pueda soportar sin fallas por un breve período de tiempo.
2. **Etapa de carga:** se aumenta progresivamente el número de clientes por segundo para buscar el punto en el cual comienza a fallar el servidor.
3. **Etapa de enfriamiento:** se busca determinar en qué tiempo y con cuántos clientes el servidor puede recuperarse de la falla.

Esta prueba de rendimiento es utilizada para cada endpoint del Gateway a lo largo del trabajo práctico, en particular se realizaron modificaciones ligeras a la cantidad y forma de generar usuarios en cada una de las etapas planteadas basadas en las caracteristicas de las mismas.

Entre las tácticas que se emplearán para comprobar cómo responden los endpoints a los mismos escenarios de carga se encuentran:

1. **Caching**: Se utiliza Redis como base de datos para cachear la respuesta de cada servicio externo.
2. **Replication**: Se configura el archivo de docker compose para pedirle que replica el Gateway hasta 3 copias y modifica el archivo de configuracion de Nginx para convertirlo en un *Load Balancer* indicando el nombre de los contenedores réplica creados por docker compose.
3. **Rate Limiting**: Se utiliza un paquete de node llamado express-rate-limit para indicar la carga máxima que puede soportar cada endpoint del gateway.

Descripcion de los gráficos

Los graficos se representan a lo largo del tiempo con una resolucion de 1 punto cada 10 segundos con un máximo de tiempo apreciable en pantalla de 5 minutos.

- **Scenarios Launched:** Es el número de escenarios ejecutadas por artillery, cada cliente virtual ejecuta un escenario. En este caso cada escenario es un

único flujo (ir al endpoint indicado en los archivos de configuración) por lo cual representa la carga (requests) aplicada al servidor

- **Requests State:** Es el estado de las respuestas obtenidas del servidor en el tiempo
- **Response Time (Client):** Es el tiempo de respuesta percibido por cada respuesta
- **Response Time (Server):** Es el tiempo de respuesta total del servidor (tiempo de procesamiento de las requests + tiempo de respuesta de la API externa)
- **Response Time (External Server):** Es el tiempo de de respuesta de una API externa

API Metar

Para la carga planteada en el archivo load_metars.yaml se presentan los siguientes casos.

Base

Luego de un período de carga plana de aproximadamente 2400 requests cada 10 segundos comienza a fallar el servicio al obtener una respuesta de denegación, podemos suponer que la API entiende que es un intento de denegación de servicio afectando la disponibilidad de nuestra aplicación. El tiempo de respuesta se mantiene estable en el tiempo con algunas variaciones de máximos en algunos momentos.

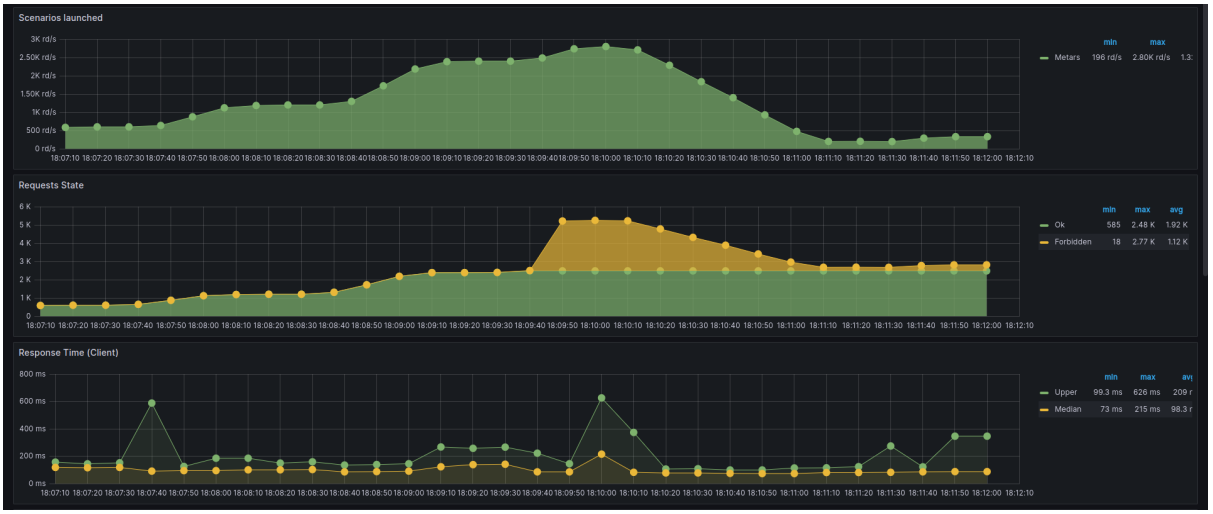

Imagen 1.0.1 Metricas relacionadas a los clientes para el endpoint de Metars

En general los tiempos de respuesta del servicio y de la API externa son muy similares, si bien el gateway tiene un mayor procesamiento que otras APIs analizadas este agrega décimas de milesimas de segundos al llamado externo. Se puede observar que incluso cuando la api de metars ya no está devolviendo una respuesta adecuada a los clientes esto no afecta la performance de la aplicación,

que sigue enviando y procesando al mismo ritmo las respuestas negadas del servidor

Imagen 1.0.2 Metricas relacionadas del servidor para el endpoint de Metars

En general se realizaron 42501 requests y todas fueron respondidas, aunque solo 25626 fueron exitosas, sabemos también que en promedio la API respondió cada 92.8 ms y que en estas condiciones podemos asegurar un p99 de 399,5 ms.

```
--------------------------------
Summary report @ 18:11:55(-0300)
--------------------------------

http.codes.200: ..................................................... 25626
http.codes.403: ..................................................... 16875
http.request_rate: .................................................. 105/sec
http.requests: ...................................................... 42501
http.response_time:
  min: .............................................................. 53
  max: .............................................................. 623
  median: ........................................................... 92.8
  p95: .............................................................. 194.4
  p99: .............................................................. 399.5
http.responses: ..................................................... 42501
```

Imagen 1.0.3 Resultados del escenario de carga sobre el endpoint de Metars

Caching

Con esta táctica observamos que ya dejamos de obtener denegaciones por parte de la API de metars, se mejoró la disponibilidad de la aplicación por completo para este escenario de carga. También se observa como disminuyó el tiempo de respuesta percibido por el cliente en promedio, a sólo 8 ms y se observan picos de máxima cada 30 segundos que es el tiempo en el cual se refresca el caché por lo tanto algunos requests completaron el flujo llamando nuevamente al servicio de metars. Dependiendo de los requerimientos de los usuarios finales y de la cantidad de veces que esta respuesta cambie en el tiempo desde la API se podria aumentar o disminuir este tiempo.

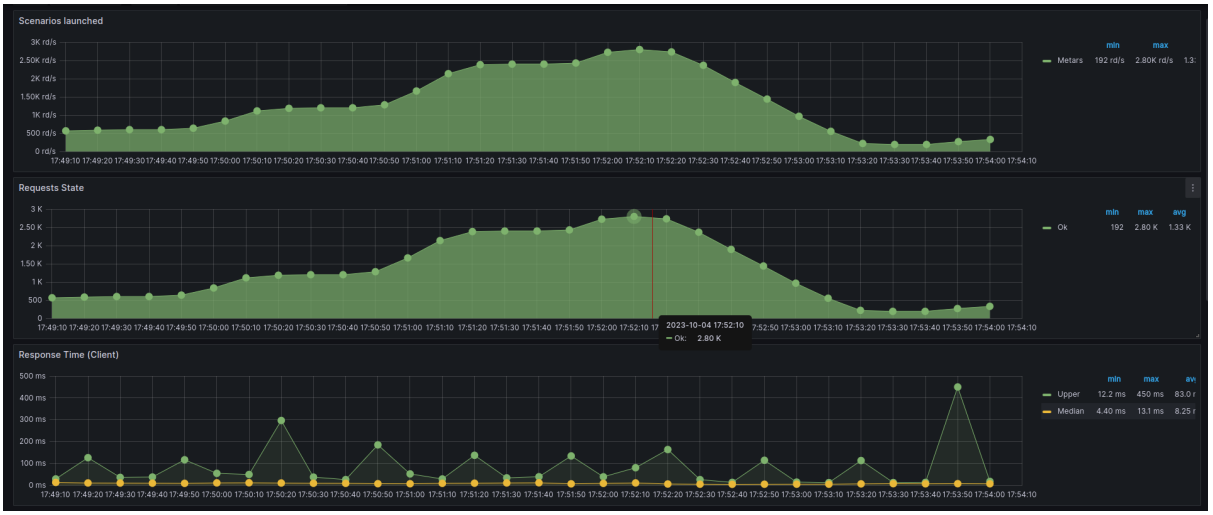

Imagen 1.1.1 Metricas relacionadas a los clientes para el endpoint de Metars con táctica de caching

Los tiempos de respuesta del servidor son mínimos, y no hay variaciones grandes por lo cual cada en un rango de 10 segundos no fueron muchas las requests que tuvieron un *miss* de caché y que terminaron solicitando datos a la API de metars. Eso también indirectamente es una mejora de performance en nuestra aplicación ya que este era mayor al tener que esperar las respuestas a llamados externos.

Imagen 1.1.2 Metricas relacionadas del servidor para el endpoint de Metars con táctica de caching

En general se realizaron 42531 requests y todas fueron respondidas, lo cual es consistente con el caso base. Pero se observa que todas las respuestas fueron exitosas ya que se pudieron evitar las denegaciones del servicio por parte de la API de metars. Los tiempos de respuesta media e incluso el p99 mejoraron a sólo decenas de milésimas. Esta mejora está sujeta a los tiempos de refresco del caching y la necesidad de los usuarios.

```
-------------------------------
Summary report @ 17:54:06(-0300)
-------------------------------

http.codes.200: ..................................................................... 42531
http.request_rate: .................................................................. 105/sec
http.requests: ...................................................................... 42531
http.response_time:
  min: .............................................................................. 0
  max: .............................................................................. 872
  median: ........................................................................... 5
  p95: .............................................................................. 16
  p99: .............................................................................. 30.3
http.responses: ..................................................................... 42531
```

Imagen 1.1.3 Resultados del escenario de carga sobre el endpoint de Metars con táctica de caching

Replication

Con esta táctica se observa que no hubo mejora alguna, seguimos teniendo problemas de disponibilidad. Replicar los servicios puede mejorar la capacidad de recibir y procesar request del servidor en general pero el cuello de botella viene siendo la API de metars, por más que sigamos aumentando las réplicas no habría resultado. Se observa consistencia en esta ejecución con el caso base ya que aproximadamente en el mismo momento de ejecucion del escenario de carga se empiezan a obtener denegaciones del servicio. Al contrario con la disponibilidad podemos observar una pequeña mejora en la performance, ya que los tiempos de respuesta percibidos por el cliente en promedio disminuyeron 10ms, esto tiene sentido ya que son 3 servidores los que están respondiendo a pesar de la demora de la API de metars.

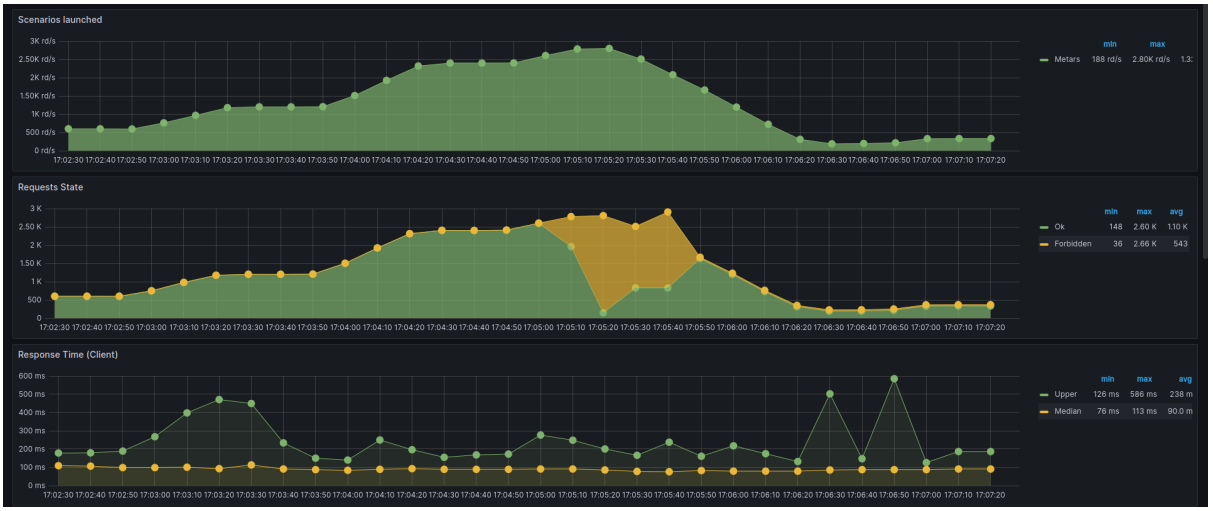

Imagen 1.2.1 Metricas relacionadas a los clientes para endpoint de Metars con táctica de replication

En este caso no contamos con una metrica de respuesta del servicio externo, pero esta no debería ser muy diferente de la respuesta total del servidor. Se observa un pequeño aumento en tiempo promedio con respecto al caso base ya que ahora es de 94 ms, pero podemos atribuirlo a las demoras de la red ya que no se espera que estos tiempos sean mejores en promedio por tener más réplicas, al ser réplicas las respuestas seran iguales en los 3 servidores.

Fe de errata: Request Time (Client) debería ser Request Time (External Server) por cuestiones de tiempo no se pudo corregir la corrida del caso.

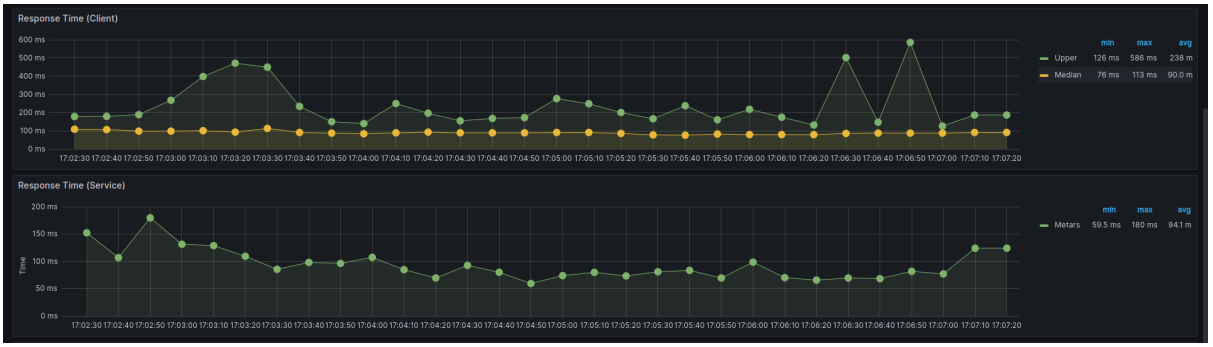

Imagen 1.2.2 Metricas relacionadas del servidor para el endpoint de Metars con táctica de replication

En general se realizaron 42547 requests con 7283 denegaciones por parte de la API de metars. Se observa que la cantidad de respuestas exitosas aumentó, aunque se desconoce ya que se encuentra asociado al servicio externo. Lo que sí pudo mejorar debido al aumento de réplicas son los tiempos de respuesta al cliente como observamos en la imagen 1.2.1, ya que en el caso base 1 servidor tenía que responder 3 requests mientras que ahora 3 servidores pueden responder esas 3 requests en el mismo momento (solapadas).

```
--------------------------------
Summary report @ 17:07:10(-0300)
--------------------------------

http.codes.200: ....................................................... 35264
http.codes.403: ....................................................... 7283
http.request_rate: .................................................... 106/sec
http.requests: ........................................................ 42547
http.response_time:
  min: ................................................................ 50
  max: ................................................................ 671
  median: ............................................................. 87.4
  p95: ................................................................ 133
  p99: ................................................................ 198.4
http.responses: ....................................................... 42547
```

Imagen 1.2.3 Resultados del escenario de carga sobre el endpoint de Metars con táctica de replication

Rate Limiting

En este caso, con límite de 8000 requests por minuto tampoco se encontraron mejoras con respecto al caso base, como se desconoce cómo la API de metars bloquea las solicitudes del servidor no se pudo establecer un límite que evitará llegar a las denegaciones del servicio y evitar el fallo, ya que en este punto habra que esperar un tiempo para desconocido para poder tener en funcionamiento nuevamente al servidor. Por lo tanto no se mejoró la disponibilidad de la aplicación.

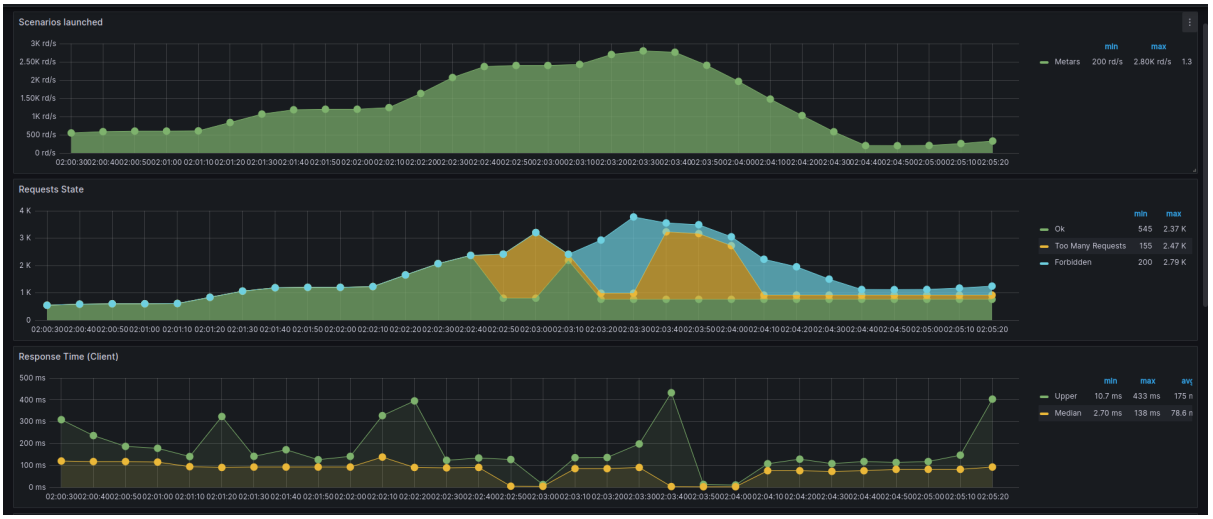

Imagen 1.3.1 Metricas relacionadas a los clientes para endpoint de Metars con táctica de rate limiting

En cuestión de performance tampoco hubo mejoras con esta técnica. En general tampoco se espera una mejora en el tiempo de procesamiento del servidor ya que está más vinculada a evitar consumir recursos y la cantidad de carga no es suficiente para agotarlos, tampoco podemos evitar el cuello de botella generado por la API Externa porque desconocemos cuándo frenar las requests para que este no nos bloquee.

Error detectado: Los puntos de respuesta que se mantienen constantes son incorrectos ya que se olvidó enviar metricas del servicio en el caso de rate limiting. Por lo tanto en el caso de limitar las requests, el tiempo de respuesta del servidor y sobretodo del external service deberian ser cercanos a cero porque no hay ningun tipo de procesamiento en el mismo, por lo cual hay valores que no representan la realidad.

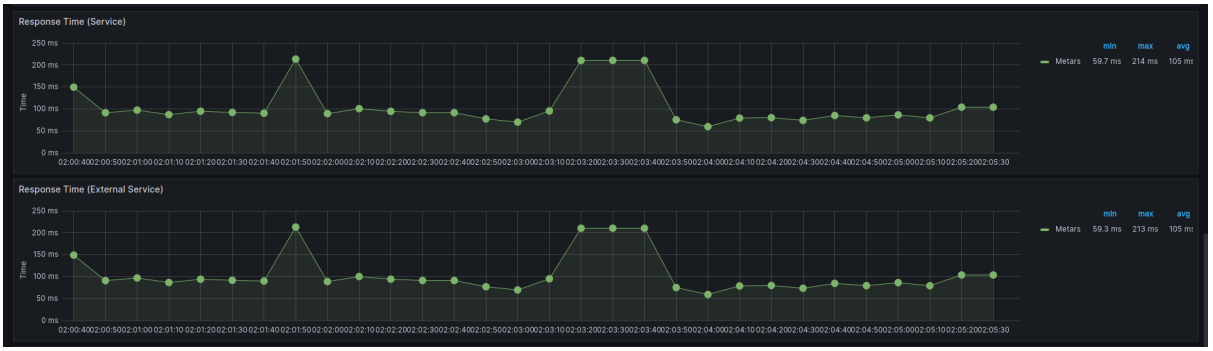

Imagen 1.3.2 Metricas relacionadas del servidor para el endpoint de Metars con táctica de rate limiting

En general se realizaron 42500 requests, con la diferencia de que tenemos muchas menos respuestas exitosas que antes. Con un límite de 8000 requests por minuto empezamos a rechazar respuestas muy pronto y al pasar esa restriccion y habilitar el envio de las requests a la API de metars está automáticamente nos niega el acceso (ver imagen 1.3.1), es decir, comenzamos a rechazar requests 30 segundos

antes de los esperado para evitar sobrecargar al servicio de metars pero al volver a habilitar el flujo nos bloquea rapidamente despues de algunas respuestas exitosas.

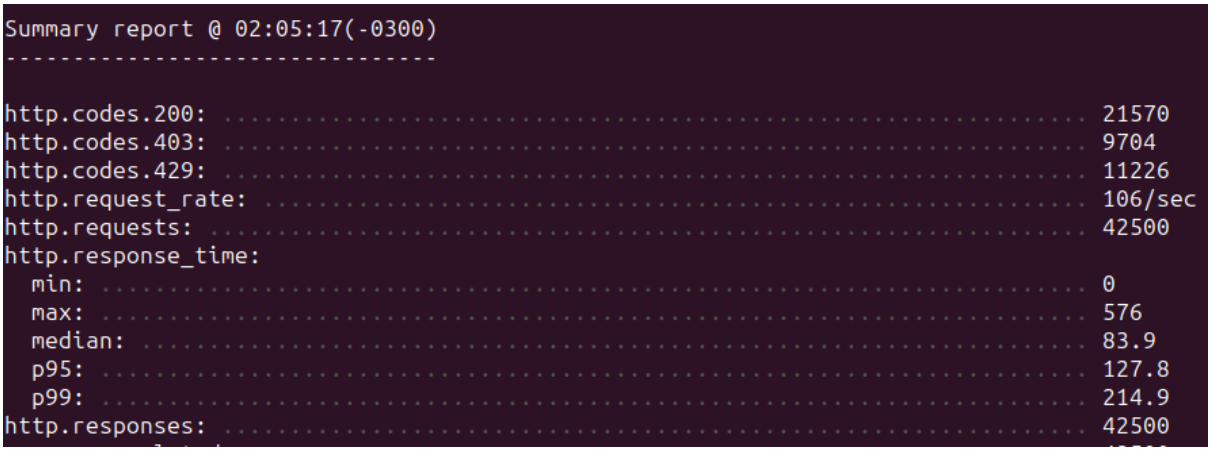

Imagen 1.3.3 Resultados del escenario de carga sobre el endpoint de Metars con táctica de rate limiting

API Spaceflights

Para la carga planteada en el archivo load_spaceflights.yaml se presentan los siguientes casos.

Base

Luego de un período de carga progresiva, cuando se alcanzó el máximo de 300 requests durante el segundo 30 de ejecución empezamos a tener problemas de disponibilidad. Está API como se mencionó en el análisis inicial tarda mucho en responder por lo cual empezamos a tener problemas en la conexión con Nginx (ETIMEDOUT) que se representan en la curva azul como "*connection errors*" y también tenemos problemas de conexión en el servidor representados con "*interval error*". Posiblemente esto ocurra por la cantidad de conexiones (sockets) tan grande que empezamos a tener abiertas dentro del servidor esperando por la respuesta de la API de spaceflights. Los tiempos de respuesta al cliente en promedio antes del fallo están en el orden del segundo, posteriormente cae el tiempo porque el servidor casi siempre falla antes de intentar llamar al servicio nuevamente.

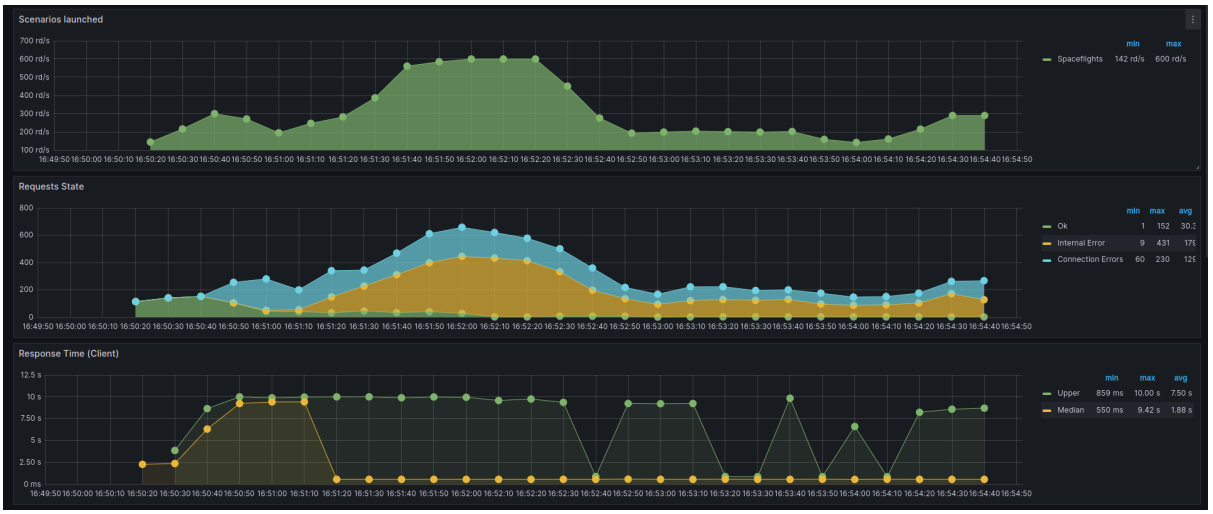

Imagen 2.0.1 Metricas relacionadas a los clientes para el endpoint de Spaceflights

Los tiempos de ejecución del servidor son muy elevados, es un caso extraño pero podría ser que con mayor carga la API externa de Spaceflights empieza a tener problemas de performance y las peticiones tardan mucho en ser respondidas, eso explicaría las picos tan altos y caidas repentinas que se registran en los tiempos de respuesta, si una rafaga de 100/200 requests viajará al mismo tiempo al servicio pudiera explicar los picos, y las caidas serian los errores de conexion internos como (ECONNRESET o desconexiones en la conexión de TCP antes de establecer la conexión de TSL)

Imagen 2.0.2 Metricas relacionadas del servidor para el endpoint de Spaceflights

En los resultados podemos observar que de 8335 peticiones solo 783 fueron correctas, hay 3325 errores de conexión generadas por Nginx y otras 4227 errores del gateway internamente.

```
--------------------------------
Summary report @ 16:54:46(-0300)
--------------------------------

errors.ETIMEDOUT: ..................................................... 3325
http.codes.200: ....................................................... 783
http.codes.500: ....................................................... 4227
http.request_rate: .................................................... 30/sec
http.requests: ........................................................ 8335
http.response_time:
  min: ................................................................ 529
  max: ................................................................ 9999
  median: ............................................................. 561.2
  p95: ................................................................ 9230.4
  p99: ................................................................ 9801.2
http.responses: ....................................................... 5010
```

Imagen 2.0.3 Resultados del escenario de carga sobre el endpoint de Spaceflights

A continuación se presenta un caso extremo (puede tardar más o menos que esto) pero nos ayuda a concluir que la API de spaceflight tiene problemas de performance a mayor número de clientes concurrentes o simplemente los retrasa, lo que explicaría las variaciones en los tiempos de respuesta observados.

```
Concurrency Level:      200
Time taken for tests:   16.172 seconds
Complete requests:      200
Failed requests:        0
Total transferred:      380000 bytes
HTML transferred:       333600 bytes
Requests per second:    12.37 [#/sec] (mean)
Time per request:       16172.483 [ms] (mean)
Time per request:       80.862 [ms] (mean, across all concurrent requests)
Transfer rate:          22.95 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    3   0.8      3       5
Processing:  1705 8905 3352.9   9105   14466
Waiting:     1700 8905 3353.0   9105   14466
Total:       1705 8907 3352.5   9107   14468

Percentage of the requests served within a certain time (ms)
  50%   9107
  66%  10782
  75%  11768
  80%  12344
  90%  13371
  95%  13953
  98%  14314
  99%  14425
 100%  14468 (longest request)
```

Imagen 2.0.4 Ejecucion de ab -c 200 -n 200 localhost:5555/api/spaceflight_news en un OS Linux

Caching

Con esta táctica observamos que ya dejamos de tener problemas de conexión, con esto se mejoró la disponibilidad de la aplicación por completo para este escenario de carga. También se observa como disminuyó el tiempo de respuesta percibido por el cliente en promedio, a sólo 78 ms y se observan picos de máxima cada 30 segundos que es el tiempo en el cual se refresca el caché por lo tanto algunos requests completaron el flujo llamando nuevamente al servicio de spaceflights.

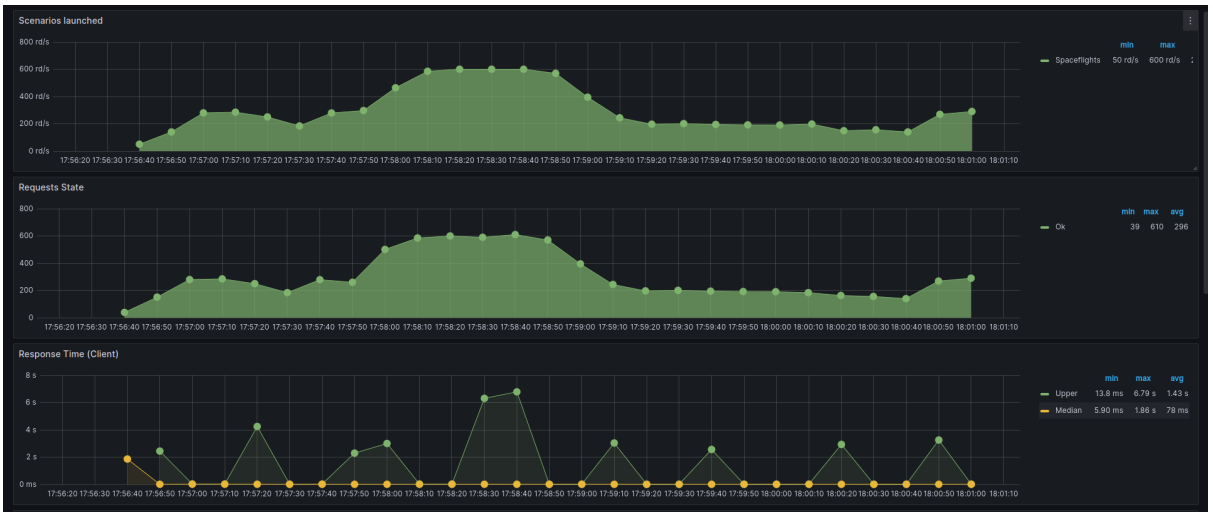

Imagen 2.1.1 Metricas relacionadas a los clientes para el endpoint de Spaceflights con táctica caching

En los tiempos de respuesta del servidor no hay mucho que explicar, se observa que casi todo va al cache, y los dos picos que se observan probablemente sean una cantidad significativa de peticiones que fueron a buscar datos al servicio en un rango de 10 segundos que aumentó el promedio del tiempo de resolucion final que es el mapeado en ese punto de resolución.

Imagen 2.1.2 Metricas relacionadas del servidor para el endpoint de Spaceflights con técticaca de caching

En los resultados podemos observar que de 8354 peticiones y todas fueron correctas, ya que la causa de falla principal de la aplicación es por supuesto el servicio externo.

```
--------------------------------
Summary report @ 18:01:03(-0300)
--------------------------------

http.codes.200: ........................................................... 8354
http.request_rate: ........................................................ 30/sec
http.requests: ............................................................ 8354
http.response_time:
  min: ..................................................................... 1
  max: ..................................................................... 6786
  median: .................................................................. 6
  p95: ..................................................................... 1978.7
  p99: ..................................................................... 3905.8
http.responses: ........................................................... 8354
```

Imagen 2.1.3 Resultados del escenario de carga sobre el endpoint de Spaceflights con técnica de caching

Replication

Con esta táctica se observa que no hubo mejora alguna, seguimos teniendo problemas de disponibilidad. Al igual que en el caso de la API de metars replicar los servicios puede mejorar la capacidad de recibir y procesar request del servidor en general pero el cuello de botella viene siendo la API de spaceflights. No sirve de nada esta táctica para este caso.

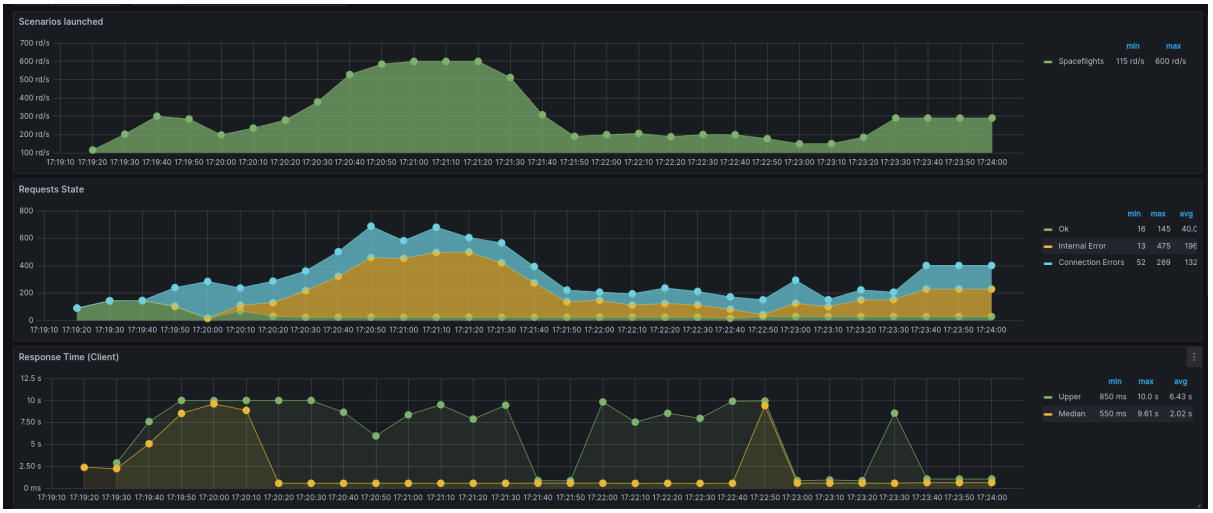

Imagen 2.2.1 Metricas relacionadas a los clientes para el endpoint de Spaceflights con táctica replication

Los tiempos de respuesta del servidor tampoco tienen mejoras con esta táctica

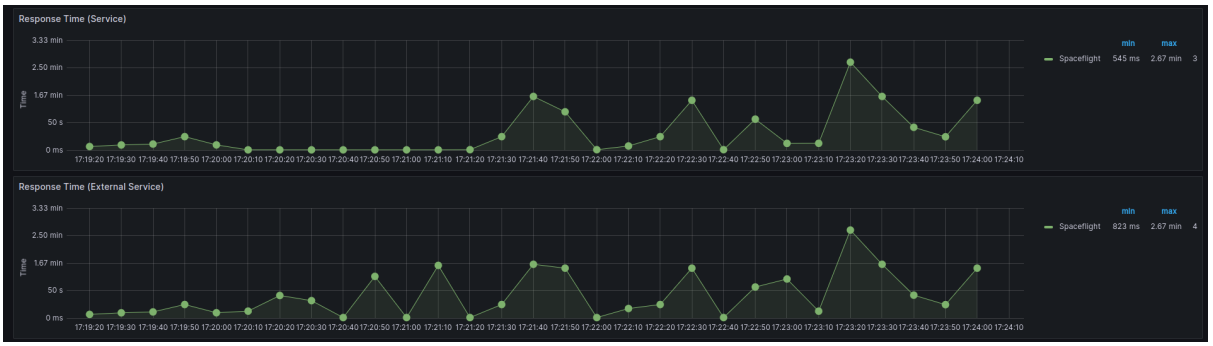

Imagen 2.2.2 Metricas relacionadas del servidor para el endpoint de Spaceflights con táctica replication

En los resultados podemos observar que de 8381 peticiones sólo 668 fueron exitosas, disminuyó con respecto al caso base, sin embargo no podemos concluir nada con estos resultados, simplemente no sirve replicar los servidores.

```
-------------------------------
Summary report @ 17:23:48(-0300)
-------------------------------

errors.ETIMEDOUT: ..... 3278
http.codes.200: ..... 668
http.codes.500: ..... 4435
http.request_rate: ..... 30/sec
http.requests: ..... 8381
http.response_time:
  min: ..... 527
  max: ..... 9998
  median: ..... 561.2
  p95: ..... 8520.7
  p99: ..... 9801.2
http.responses: ..... 5103
vusers.completed: ..... 5103
```

Imagen 2.2.3 Resultados del escenario de carga sobre el endpoint de Spaceflights con táctica de replication

Rate Limiting

En este caso se intentó limitar la cantidad de requests por minuto a 500 para disminuir la cantidad de problemas de conexion en el gateway, sin embargo seguimos obteniendo algunos, probablemente se podria intentar seguri bajando hasta encontrar un punto optimo en el cual bloquear las requests ayude a recuperar el servicio más rapido y no mantener tantas conexiones abiertas esperando a la API de spaceflights. Para el escenario planteado se alcanza rapidamente el límite por lo cual se limita el servicio bastante temprano en pro de proteger los recursos (sockets) del servidor y disminuir el tiempo que estamos bloqueados por estos problemas. Sin embargo 500 conexiones por minuto sigue siendo mucho en algunos casos si no se distribuyen bien en el tiempo y siguen habiendo problemas con las conexiones.

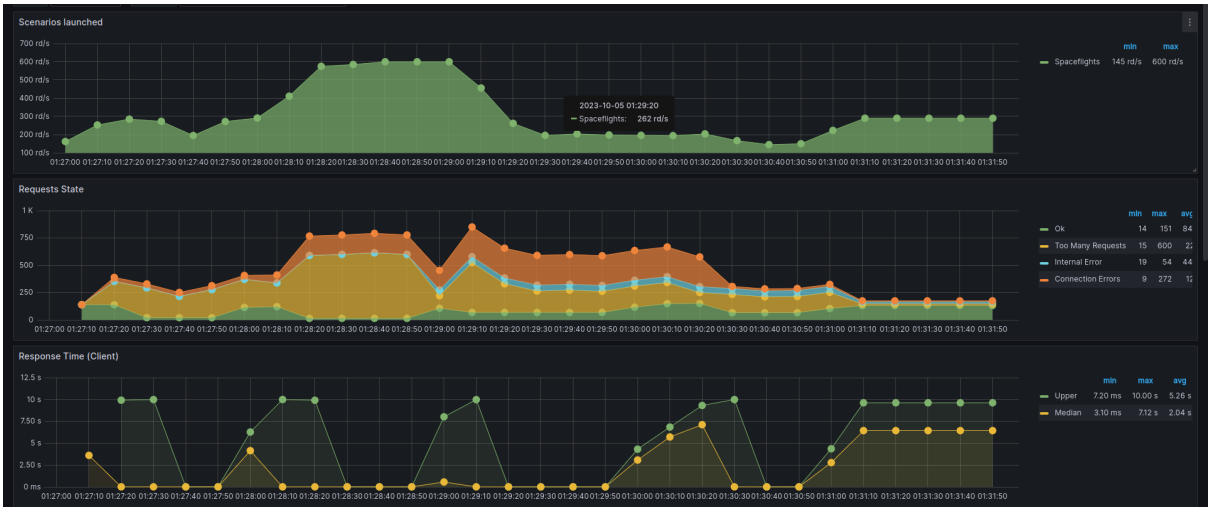

Imagen 2.2.1 Metricas relacionadas a los clientes para el endpoint de Spaceflights con táctica rate limiting

En este caso, considerando los puntos constantes como casi cero, en general el tiempo de respuesta mejoró con respecto al caso base, el tiempo máximo fue de apenas 32,9 s, el cual es un tiempo mucho mejor que los máximos registrados anteriormente en la Imagen 2.0.2 que eran del orden del minuto. Con esta táctica logramos estabilizar los tiempos de respuesta del servidor y el número de fallas que se generan al consumir sockets manteniendo las conexiones abiertas.

Error detectado: Los puntos de respuesta que se mantienen constantes son incorrectos ya que se olvidó enviar metricas del servicio en el caso de rate limiting. Por lo tanto en el caso de limitar las requests, el tiempo de respuesta del servidor y sobretodo del external service deberian ser cercanos a cero porque no hay ningun tipo de procesamiento en el mismo, por lo cual hay valores que no representan la realidad.

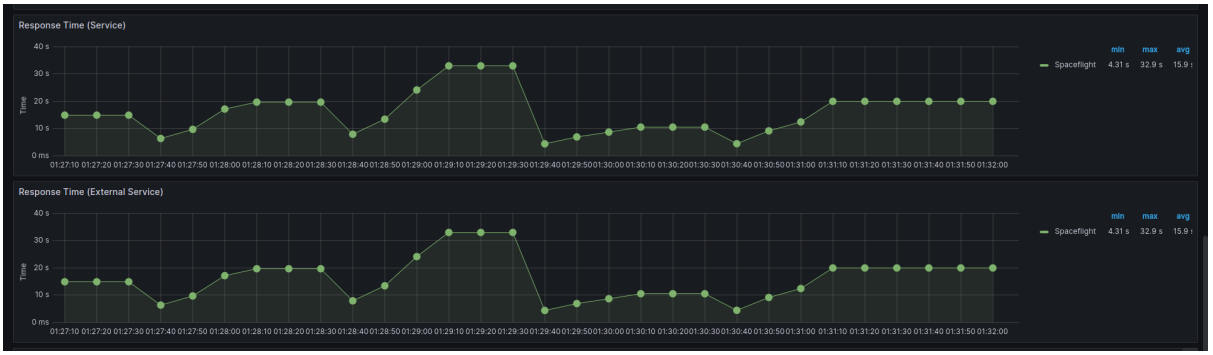

Imagen 2.2.2 Metricas relacionadas del servidor para el endpoint de Spaceflights con táctica rate limiting

En los resultados podemos observar que de 8347 peticiones 1575 fueron correctas, es decir aumentamos el número al doble con respecto al caso base. Esta táctica nos permitió disminuir la cantidad de errores de conexión protegiendo los recursos del servidor para conseguir un aumento en el número de respuestas exitosas.

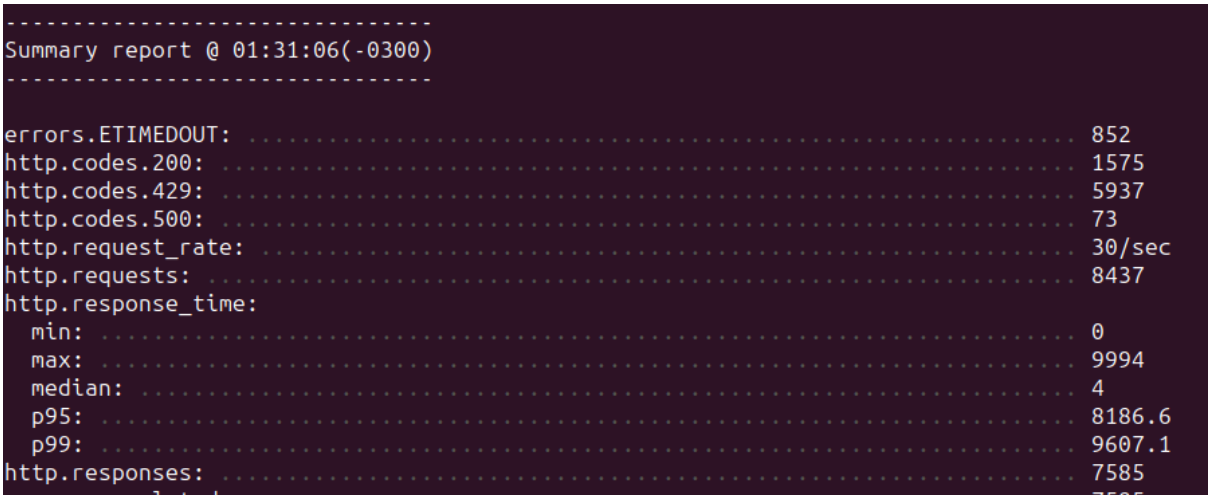

Imagen 2.2.3 Resultados del escenario de carga sobre el endpoint de Spaceflights con táctica de replication

Conclusión

Durante el trabajo se analizaron tres tipos de APIs, cada uno con características y limitaciones diferentes mencionadas en el análisis inicial. De los resultados obtenidos se puede concluir que la táctica más importante para mejorar la disponibilidad y performance de la aplicación es la de caching, esto ayuda a disminuir los tiempos de espera de las APIs externas y a sortear las limitaciones que tienen ofreciendo un mejor servicio a través del gateway. La peor táctica fue sin duda la replicación, es una táctica que en este tipo de sistemas no funciona, ya que el problema principal que tenía el gateway no era de rendimiento o problemas de escalabilidad sino el cuello de botella generado por las APIs externas. Por último, una técnica no tan efectiva pero válida es la de rate limiting, ya que puede ayudar a que el servicio se recupere más rapido del fallo si se consiguen optimizar bien los valores de peticiones por unidad de tiempo.