Trabajo Práctico 1

Arquitectura del Software [75.73/TB054]

1er Cuatrimestre 2024

Integrantes:

Emanuel Pelozo	99444
Agustina Fraccaro	103199
Nicolas Amigo	105832
Ezequiel Tosto Valenzuela	108321

Índice

1. Objetivo

El presente trabajo práctico tiene como objetivo implementar un servicio HTTP en Node.js-Express que representa una API que consume otras para dar información. Luego esta será sometida a distintas pruebas de carga para realizar mediciones y analizar los resultados de las mismas.

Se usarán las siguientes tácticas: caso base, caché, rate limiting y replicación.

2. Introducción

Para poder luego comparar las tácticas elegidas utilizaremos varios endpoints que nos permitirán resaltar las diferencias entre estas.

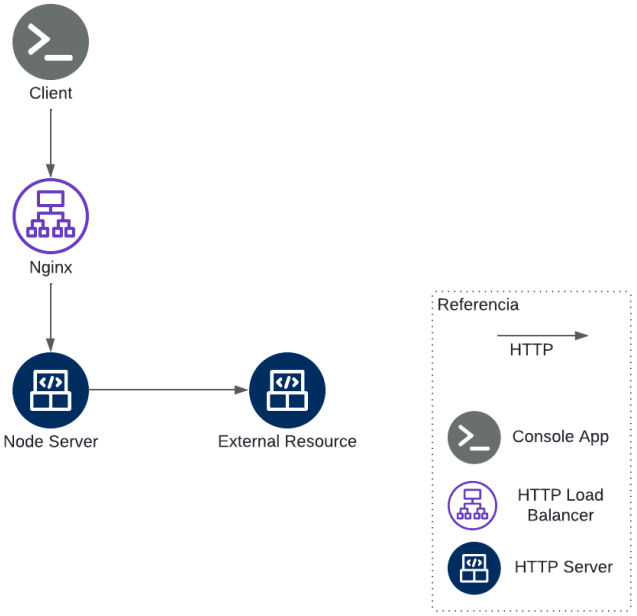
- **Dictionary:** Dada una palabra como parte del query devuelve la fonética y los significados de la palabra obtenidos del servicio Free Dictionary.

- **Spaceflight News:** Devuelve las 5 últimas noticias sobre actividad espacial obteniéndose del servicio Spaceflight News.

- **Random quote:** Devuelve una cita famosa con su autor. Se obtiene del servicio Quotable.

3. Arquitectura y tácticas utilizadas

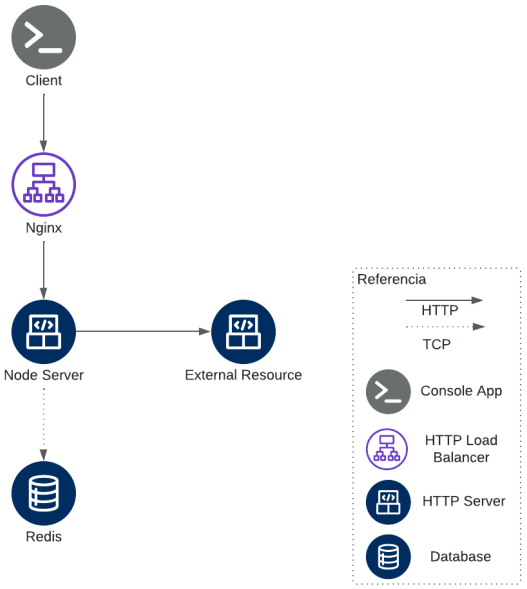
3.1 Arquitectura base



(Componentes en la arquitectura base)

Como base de proyecto implementamos una instancia de un servidor node. Este no recibe los requests directamente sino que se encuentra conectado a un load balancer. Este servidor node a su vez consume recursos de tres apis distintas. Para realizar los tests utilizamos la aplicación de consola artillery que produjo requests a nginx.

3.2 Cache



(Componentes utilizando la táctica de caché)

Al implementar la táctica de caché utilizamos distintas estrategias según el endpoint. Para todos los endpoints que decidimos cachear dejamos el tamaño default de redis (512 MiB según la documentación).

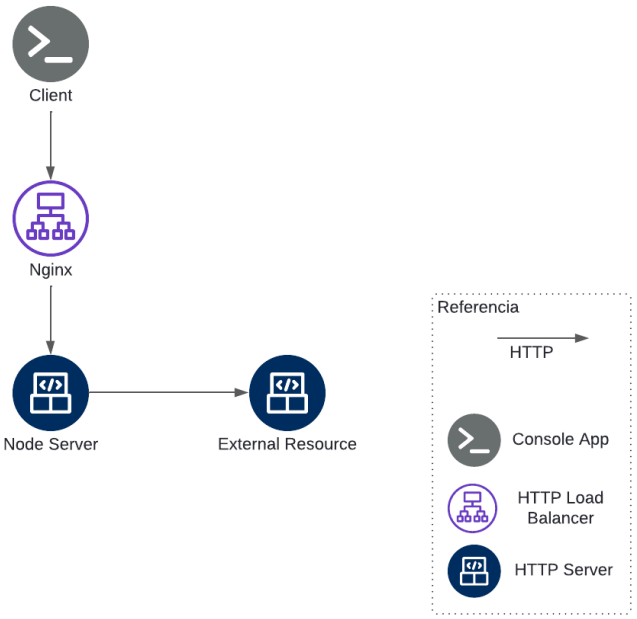
Endpoint Dictionary: para este endpoint decidimos que la información es cacheable debido a que un diccionario raramente cambia la información de las palabras. Para el llenado, dado que active population requeriría cachear todas las palabras del mundo (son muchas) y no tendríamos forma de saber qué tan requerida es una palabra como para poder cachear esas antes de que sean pedidas, elegimos lazy population, porque si un usuario busca una palabra podría querer buscarla de nuevo. Adicionalmente dado a que no conocemos cuántas veces se buscaría de nuevo una misma palabra utilizamos un tiempo de vida de la información de 30 minutos ya que es muy improbable que la información del caché quede desactualizada.

Endpoint Spaceflight News: investigando la API descubrimos que esta se actualiza con las noticias más recientes cada 10 minutos. Teniendo en cuenta este valor

decidimos cachear la información obtenida por 5 minutos. Esto lo realizamos de manera pasiva, guardando después de recibir un request de algún usuario.

Endpoint Quotes: para este endpoint decidimos que la información no sea cacheable debido a que la cita obtenida de la respuesta de la API externa es elegida al azar, por lo que no vimos necesario que se cachee.

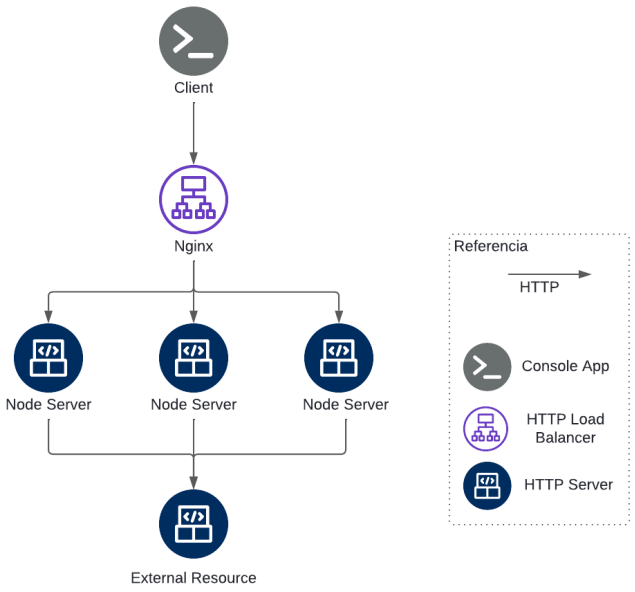
3.3 Rate limiting



(Componentes utilizando la táctica de rate limiting)

Otra táctica que utilizamos es la de Rate Limiting. Con esto la cantidad máxima de mensajes que puede enviar un usuario en un período de tiempo está limitado. Superado este límite, se devolverá error de HTTP 503. Para este proyecto lo dejamos como un máximo de 10 requests por segundo pero según el test se fue modificando según el rate limiting propio de los recursos externos.

3.4 Replicación



(Componentes utilizando la táctica de replicación con 3 instancias)

Finalmente, utilizamos la técnica de replicación. En este caso levantamos tres servidores node y nginx fue configurado para distribuir equitativamente la carga entre ellos.

4. Mediciones

4.1 Consideraciones de las mediciones:

Al continuación se presentarán los resultados de las mediciones de los diferentes escenarios, para estas mediciones es importante tener en cuenta las siguientes consideraciones:

- Time out del cliente (**Artillery**): 10 segundos.
- Mediciones de los tiempos de respuesta: Medidos en milisegundos.

- Escenarios a analizar: En todas los endpoints utilizamos dos escenarios, uno base y un pico sostenido en el tiempo con diferentes parámetros de configuración.
- Métricas a analizar en cada escenario:
    - Response time (total).
    - Response time de la dependencia.
    - Response time del lado del cliente.
    - Recursos.
    - Status Codes.
    - Throughput.
    - Cache Hits (en los casos de caché).
- Referencia de los status codes:
    - 200: Ok.
    - 424: Una falla no identificada en la dependencia.
    - Rate limit status code:
        - Originado en la dependencia: 429.
        - Originado de nuestro nginx: 503.

4.2 Dictionary

El endpoint dictionary es el encargado de, dada una palabra, regresar la fonética y los significados de la misma. Para lograr esto se utiliza la Free Dictionary API. Esta nos causó problemas al momento de medir ya que acepta un máximo de 1.5 requests por segundo. Debido a esto la cantidad de carga para testear este endpoint debió ser menor.

4.2.1 Escenario base

Para el escenario base implementamos el siguiente esquema de fases:

1. **Warm up**: Etapa plana inicial con una duración de 3 minutos y un arribo de 1 request cada 3 segundos.
2. **Elevación de carga**: Etapa de 5 minutos con una tasa de envío de 1 request cada segundo.

Utilizamos un set de palabras corto para no enviar siempre la misma palabra a la API externa.

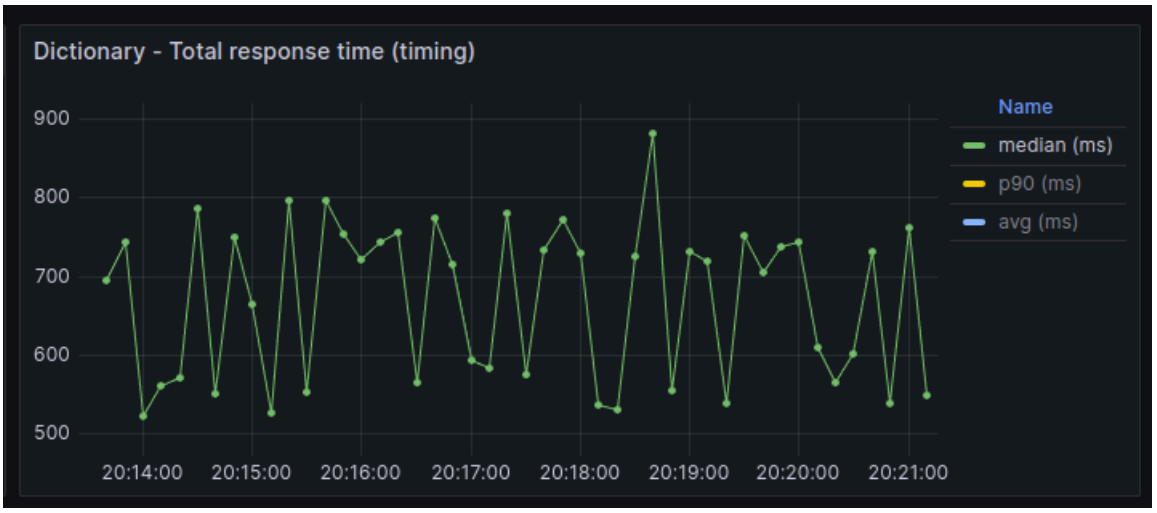
Throughput:

Throughput recibido en la ejecución de los escenarios:

El gráfico muestra un RPS de valor 1 constante pero figura de esta forma por cómo la librería agrupa las métricas cada 10 segundos, entonces no podemos ver el 0.3 RPS que realmente enviamos, si vemos los status codes se puede apreciar un poco más el cambio de fase.

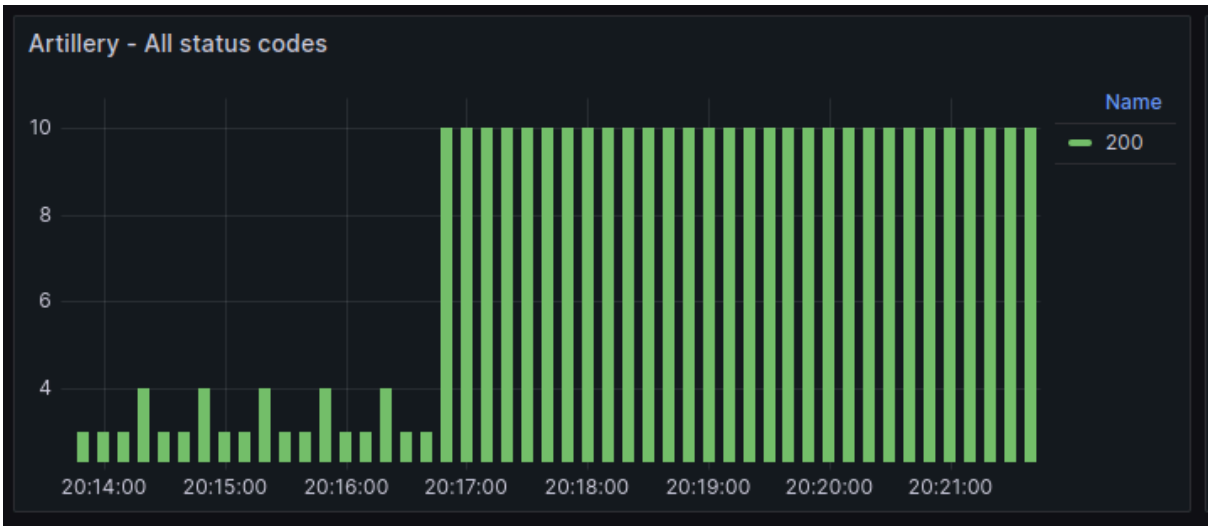
Response time:

Tiempo de respuesta total medido a través de la API:



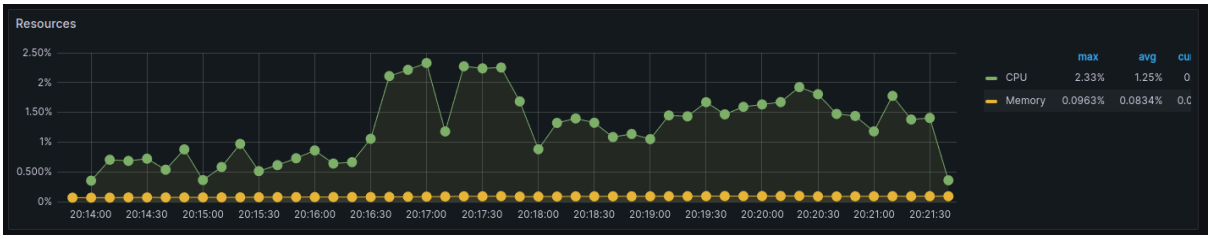
El tiempo de respuesta suele estar en los 700 MS.

Status codes recibidos desde el cliente:



Vemos que la aplicación responde sin problemas y se puede notar un cambio de fase al ver este gráfico, pero igualmente lo relevante es que no hubieron problemas durante la prueba.

Uso de recursos del server:



En cuanto a recursos no tuvo más del 2% del CPU y la memoria también tiene muy poco uso.

Tiempo de respuesta de la dependencia:

La dependencia tomó aproximadamente los mismos valores de la API. Ya que no hay más procesamiento que esta API call internamente tiene sentido que sean similares.

4.2.2 Escenario pico sostenido

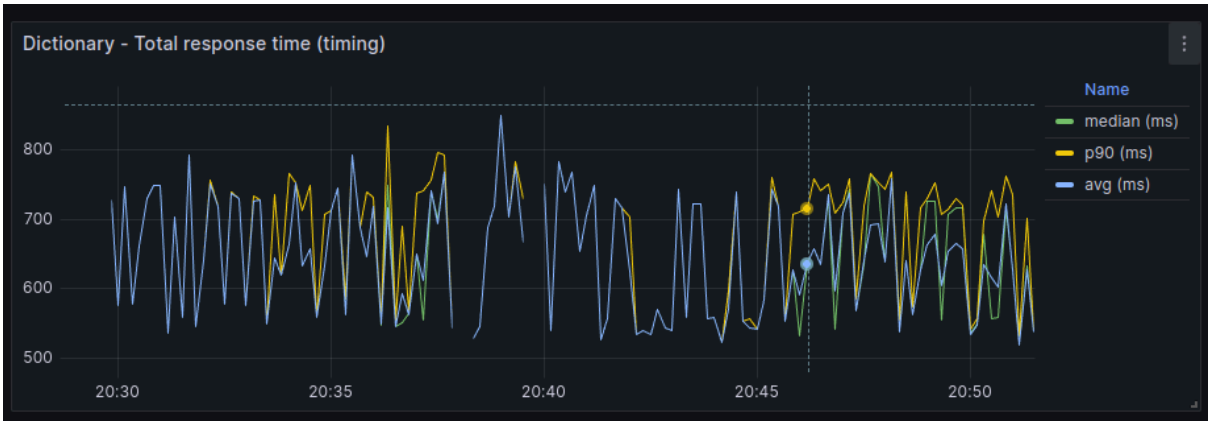
Para el escenario base implementamos el siguiente esquema de fases:

1. **Rampa Inicial:** Iniciamos con una rampa con una duración de 150 segundos, un arrival rate de 1 y un rampTo de 2.
2. **Plano Inicial:** Un primer plano con una duración de 150 segundos y un arribo constante de 2 request por segundo.
3. **Segunda Rampa:** Luego se incrementa durante 150 segundos con un arrival rate de 2 y un rampTo de 5.
4. **Segundo Plano:** Finalmente se obtiene una tasa constante de 5 requests por segundo y una duración de 150 segundos.

Throughput recibido en la ejecución de los escenarios:

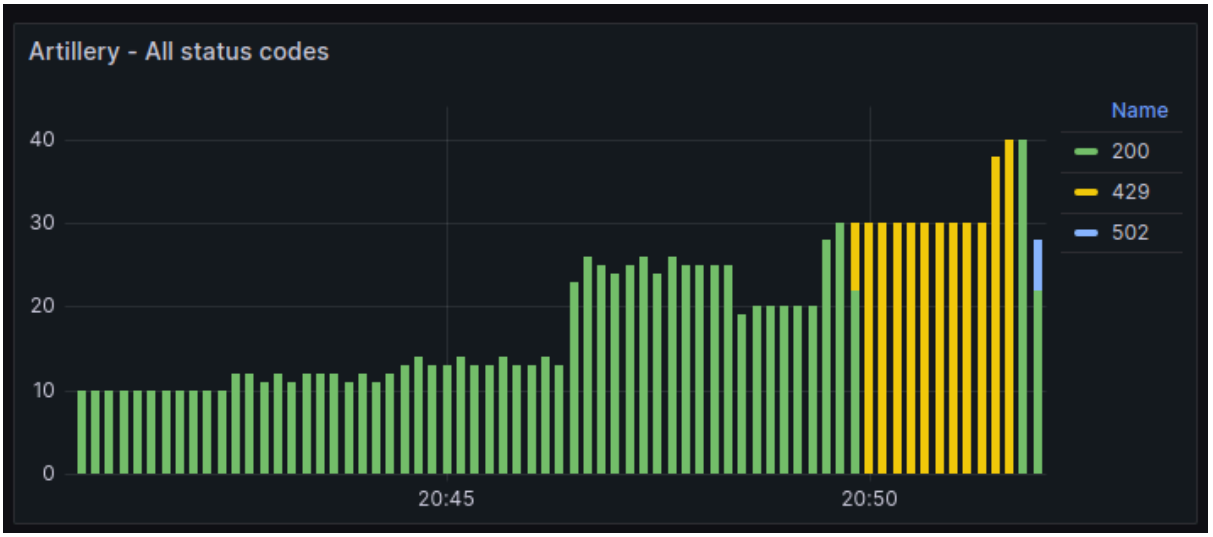
Al ser valores tan chicos en RPS no podemos apreciar tanto el incremento de las fases pero notamos que llegamos a 240 RPM que son 4 RPS.

Tiempo de respuesta total medido a través de la API:

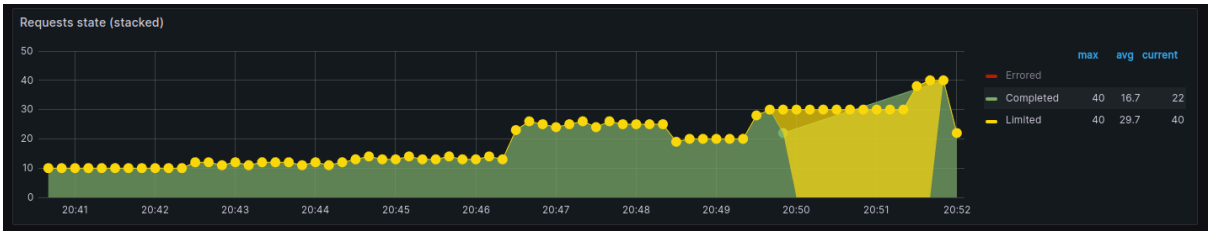


El tiempo de respuesta se mantuvo igual que el escenario anterior rondando los 700 MS de respuesta.

Status codes recibidos desde el cliente:

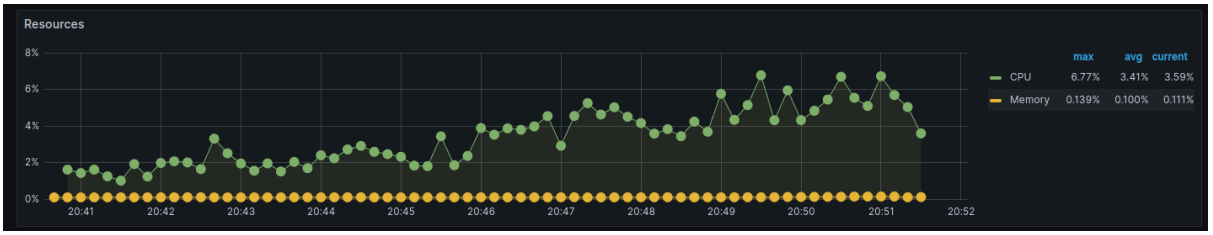


Resumen de resultados desde el cliente:



Lo que podemos notar es que en las fases finales de la prueba empezamos a recibir 429 de parte de la dependencia indicando que excedimos el número de requests que podemos hacerle.

Uso de recursos del server:



Los recursos también incrementan con respecto a la prueba anterior llegando a un máximo de 6% de consumo de CPU.

4.2.3 Escenario pico sostenido cacheado

Para solventar el problema de la prueba anterior podemos utilizar la estrategia de caché para no consumir tanto el servicio, aumentando la disponibilidad del mismo y el rendimiento de nuestra app. La caché cuenta con un TTL de 5 MS por lo que hay veces que no va a tener hits y hay veces que si (debido a las palabras que le estamos enviando).

Throughput recibido en la ejecución de los escenarios:

Tiempo de respuesta total medido a través de la api:

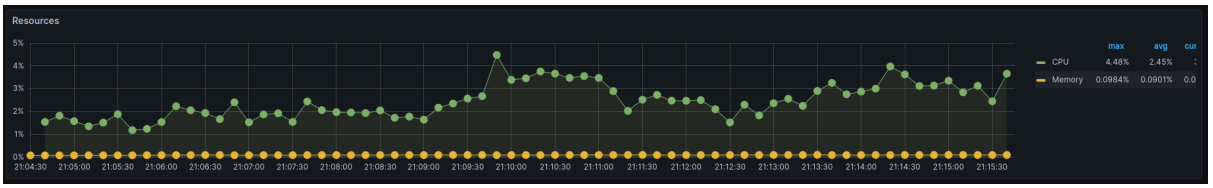
Cantidad de cache hits a través del tiempo:

Si vemos los tiempos de respuesta podemos notar que tenemos valores bajos y coinciden con los hits a la caché. El tiempo de respuesta cuando se utilizaba un valor en caché es de aproximadamente 5 MS.

Tiempo de respuesta de la dependencia:

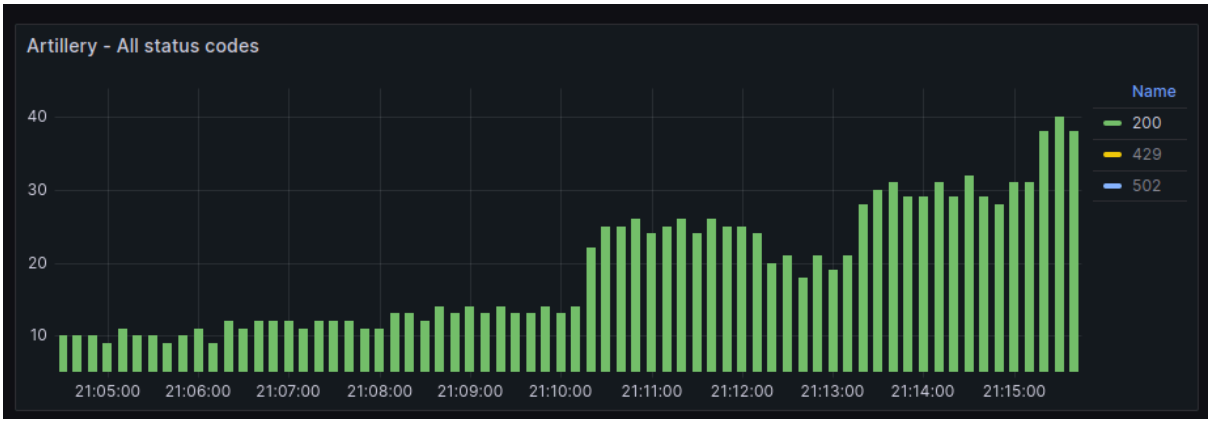
Mientras tanto la dependencia mantiene los mismos tiempos que esta vez son más altos que nuestra aplicación.

Uso de recursos del server:



Los recursos llegan más bajos con la caché alcanzando un máximo de 4.89%.

Status codes recibidos desde el cliente:



Lo que vemos es que efectivamente aumentamos el rendimiento y la disponibilidad de la aplicación ya que no tenemos errores y las pruebas corren sin problemas respondiendo todas las requests con 200.

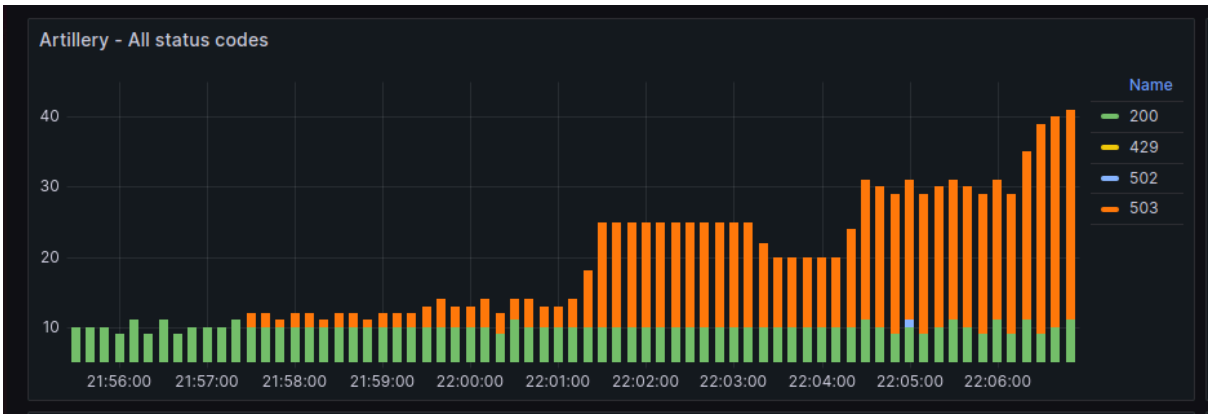
4.2.3 Escenario pico sostenido rate limited

En esta prueba tomamos el mismo caso que el 4.1.1 solo que aplicamos la estrategia de rate limiting. Si sabemos que nuestra aplicación no soporta más de 1 RPS por nuestra dependencia entonces limitamos a nuestros clientes para que no se caiga la misma y sigamos teniendo disponibilidad de la dependencia.

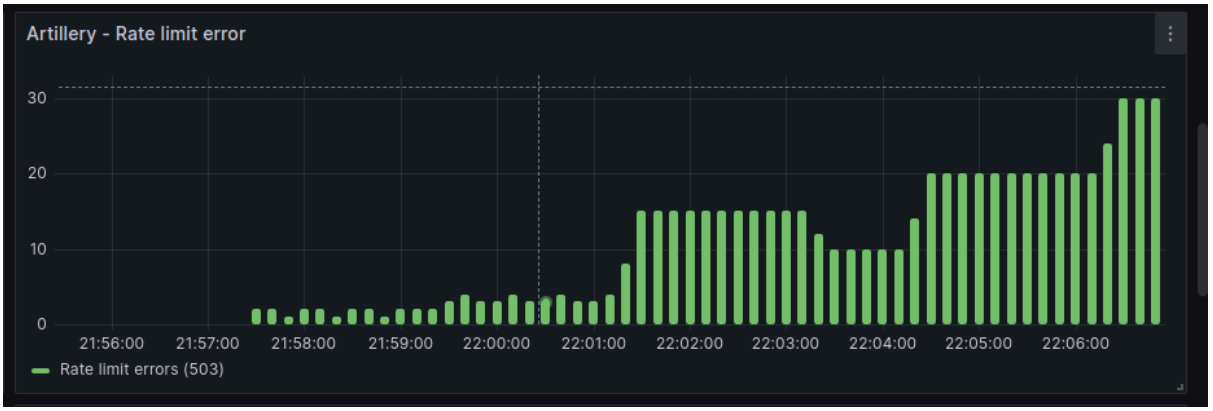
Throughput recibido en la ejecución de los escenarios:

Lo que vemos es que el throughput se mantiene constante durante toda la prueba dado que desde NGINX cortamos la conexión previa a llegar al container donde enviamos las métricas del throughput.

Status codes recibidos desde el cliente:

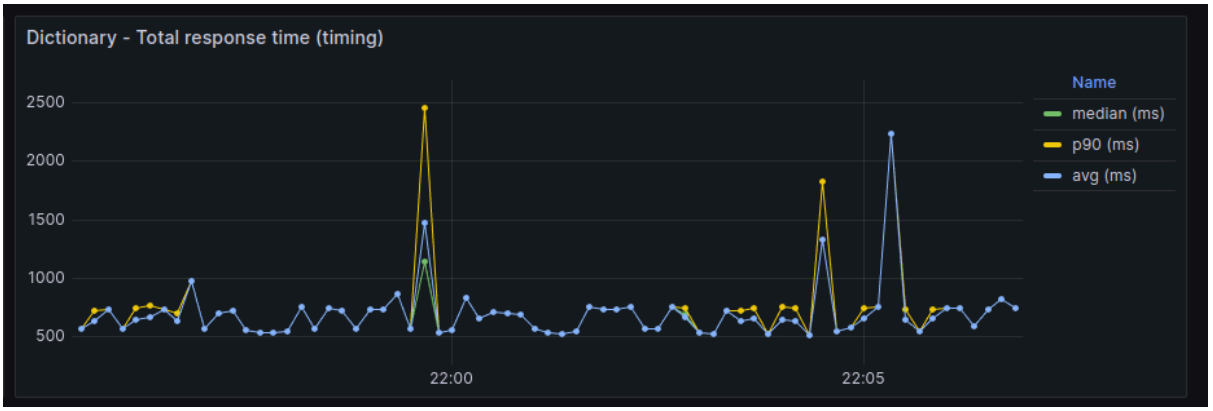


Errores por rate limit recibidos desde el cliente:

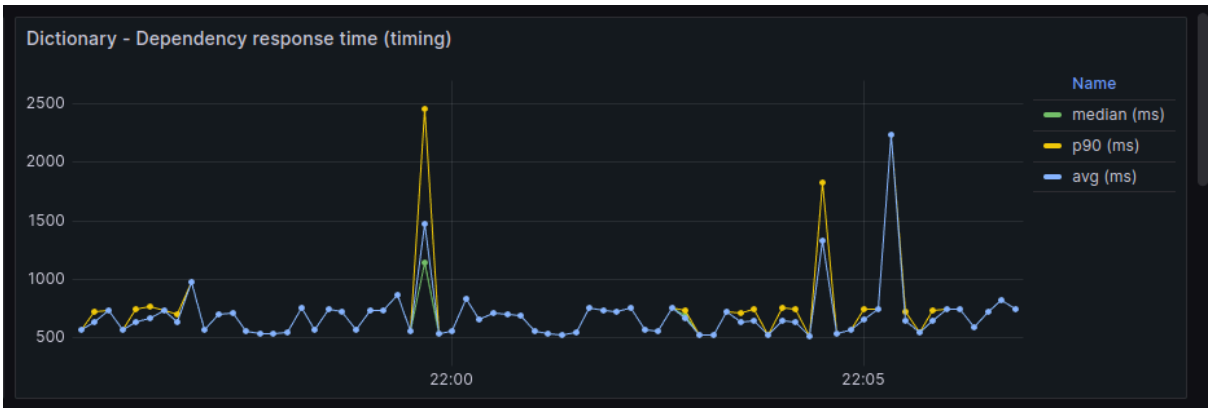


Como se notará en el gráfico de cara al cliente (artillery) llegan errores por rate limit, osea status code 503. Se ve cómo hay relación entre el incremento del throughput y los status 503 que mostramos al cliente.

Tiempo de respuesta total medido a través de la API:

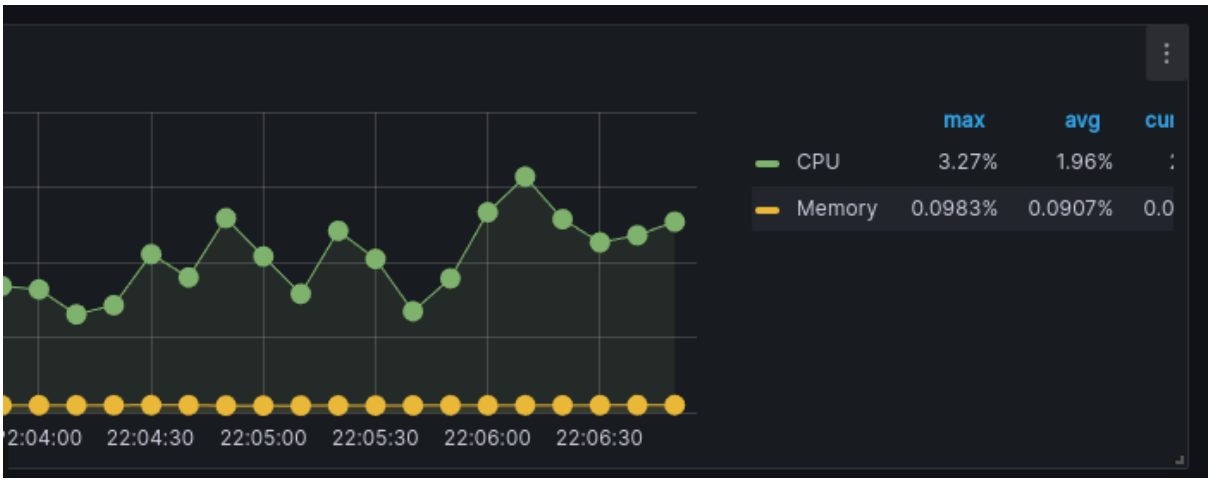


Tiempo de respuesta de la dependencia:



En cuanto a los tiempos de respuesta se muestran casi iguales que el caso de pico sostenido, teniendo unos picos de hasta 2s pero se mantiene en 700 MS por la mayoría de la prueba.

Uso de recursos del server:



Esto también reduce el consumo de CPU ya que no tiene que procesar tantas requests sino solo las que dejamos pasar a nuestra api.

4.3 Quotes

Este endpoint tiene el trabajo de entregar una frase famosa cada vez que se lo llama. Esta la obtenemos de la API Quotable. Al igual que dictionary, observamos que esta implementa en la implementación un rate limit propio de 3 requests por segundo. Esto fue tomado en cuenta para diseñar los tests de carga a realizar.

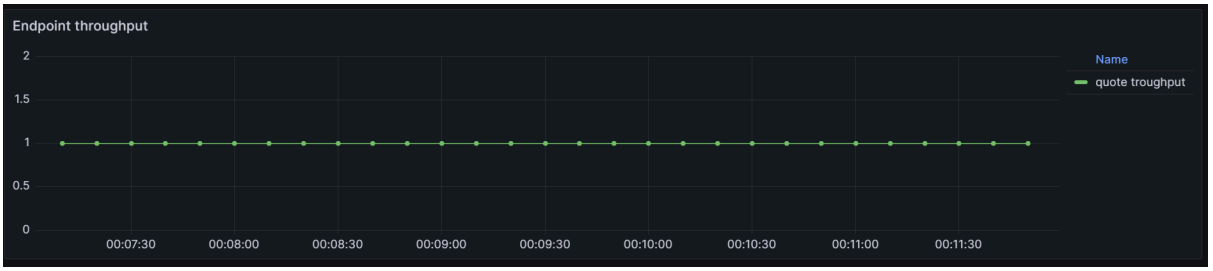
4.3.1 Escenario base

Para el escenario base implementamos el siguiente esquema de fases:

1. **Warm up**: Etapa plana inicial con una duración de 3 minutos y un arribo de 1 request cada 3 segundos.
2. **Elevación de carga**: Segunda etapa plana donde se eleva el arribo a 1 request cada 1.5 segundos (120 arribos en 180 segundos). Dura también 3 minutos.

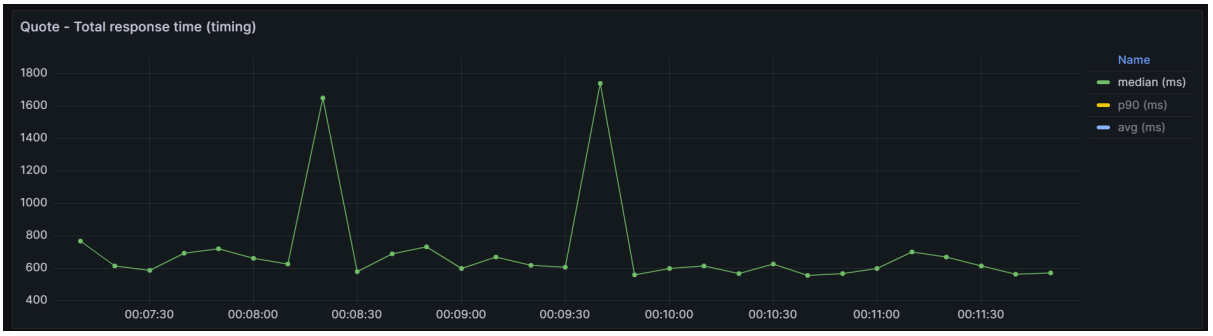
Throughput:

Throughput recibido en la ejecución de los escenarios:



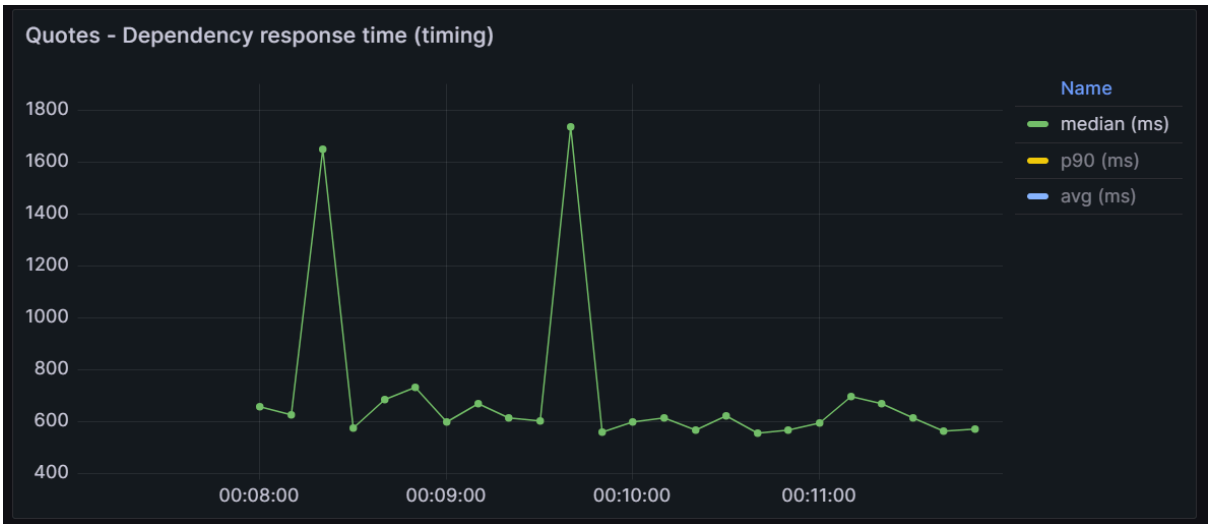
En este gráfico vemos que el throughput es constante, de 1 RPS, pero, como pasó también para el endpoint dictionary, figura de esta forma porque la librería agrupa las métricas cada 10 segundos y entonces no podemos ver el 0.3 RPS que realmente enviamos.

Tiempo de respuesta total medido a través de la API:



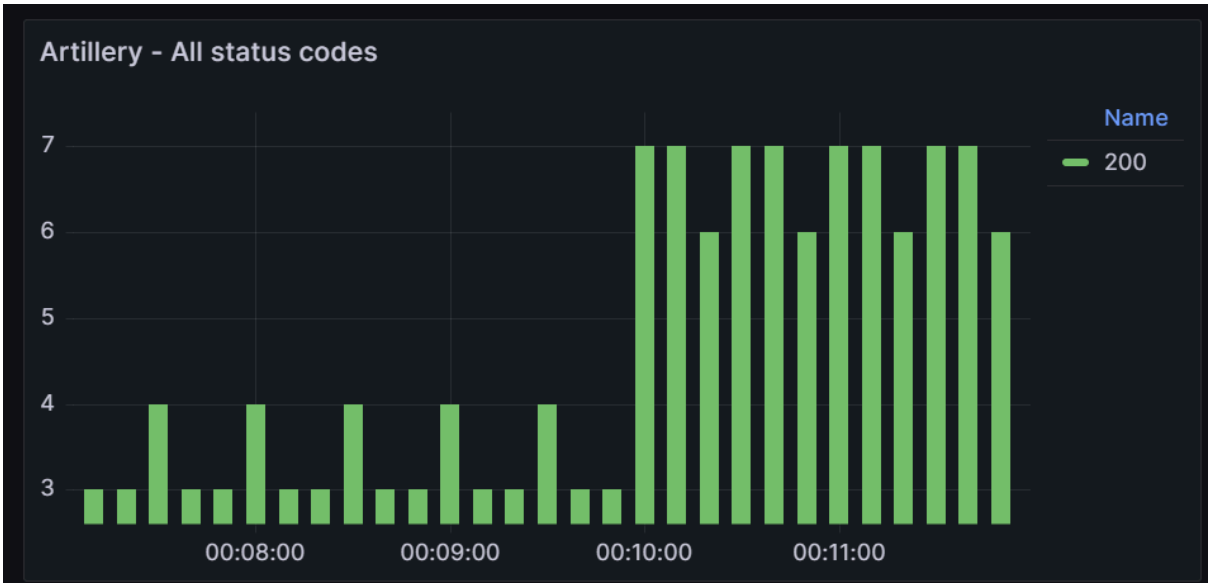
En este gráfico podemos ver que la media de tiempo de respuesta de nuestro endpoint es de alrededor de 700 MS.

Tiempo de respuesta de la dependencia:



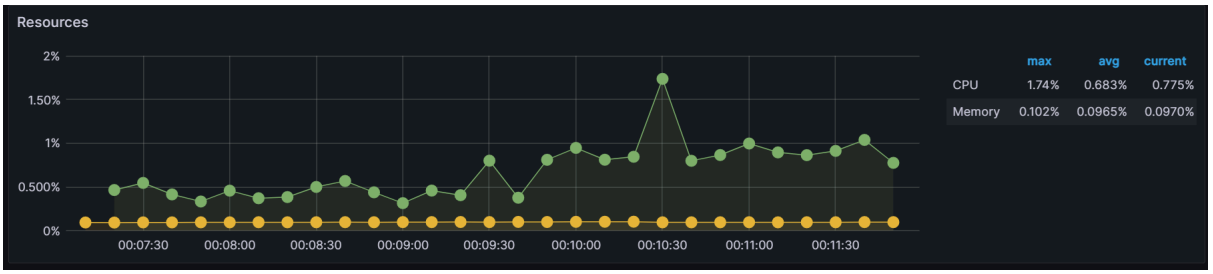
En este gráfico podemos ver que la media del tiempo de respuesta de la API de quotes es aproximadamente 700 MS, prácticamente igual a nuestra API.

Status codes recibidos desde el cliente:



En este gráfico podemos ver que el endpoint respondió con status code 200 todas las request, y se puede ver un cambio en la cantidad que se da debido al cambio de fase.

Uso de recursos del server:



Por último, podemos ver que no consume muchos recursos, el máximo de CPU no pasa el 2% y la memoria también son valores muy bajos los consumidos.

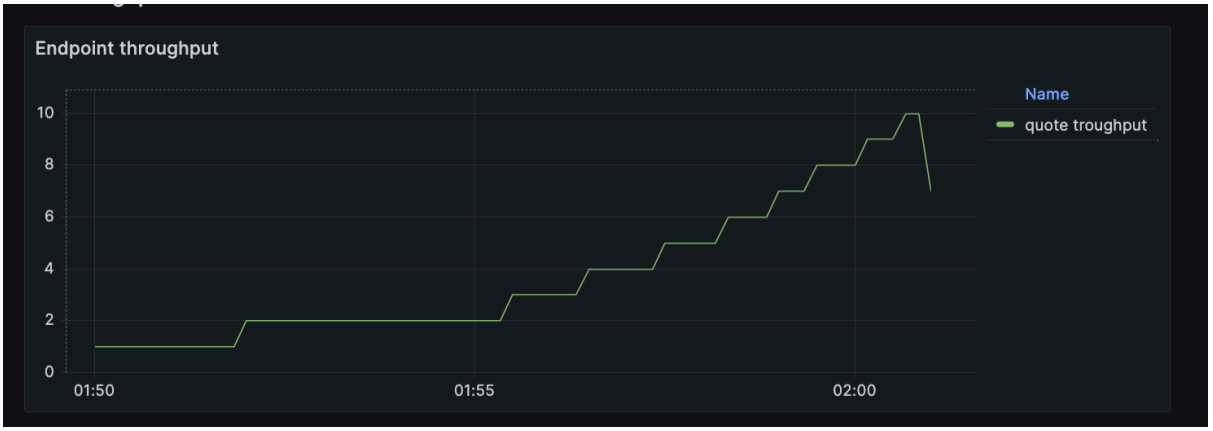
4.3.2 Escenario pico sostenido

Para el escenario base implementamos el siguiente esquema de fases:

1. **Warm up**: En esta etapa inicial iniciamos con 2 minutos con una tasa de arribos de 1 cada 2 segundos.
2. **Primera Fase plana**: Luego durante 1 minuto aumentamos a un ritmo de 1.5 arribos cada segundo.
3. **Segunda Fase Plana:** Durante 1 minuto incrementamos a un cantidad de 2 arribos por segundo.
4. **Tercera Fase Plana:** Con una duración de 1 minuto enviamos 2.5 requests por segundo.
5. **Primera Rampa:** Durante 2 minutos se utiliza un arrival Rate de 2 y un rampTo de 5.
6. **Segunda Rampa:** Durante 2 minutos se utiliza un arrival Rate de 5 y un rampTo de 10.

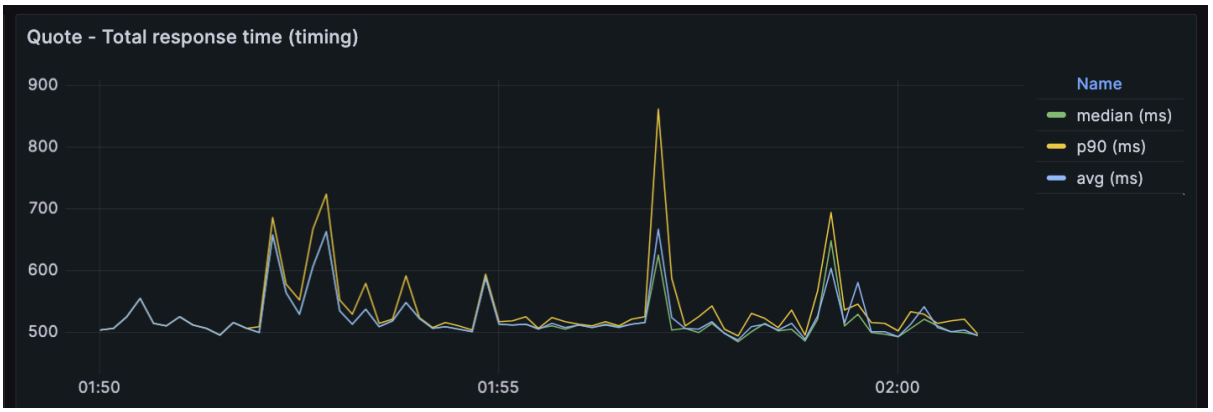
Throughput

Throughput recibido en la ejecución de los escenarios:



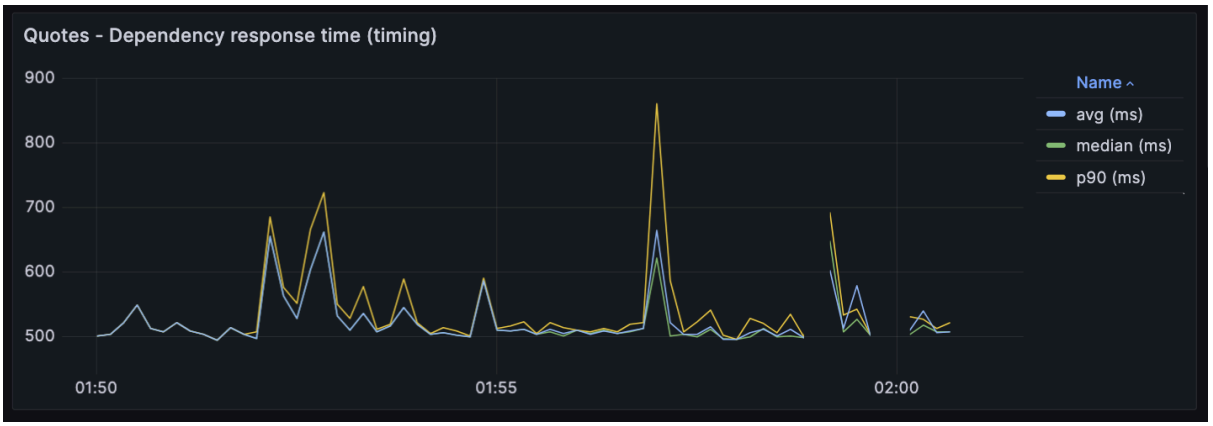
Aquí se puede ver lo descrito anteriormente. Se puede observar cómo se inicia en 1 RPS y se finaliza en 10 RPS.

Tiempo de respuesta total medido desde el cliente:



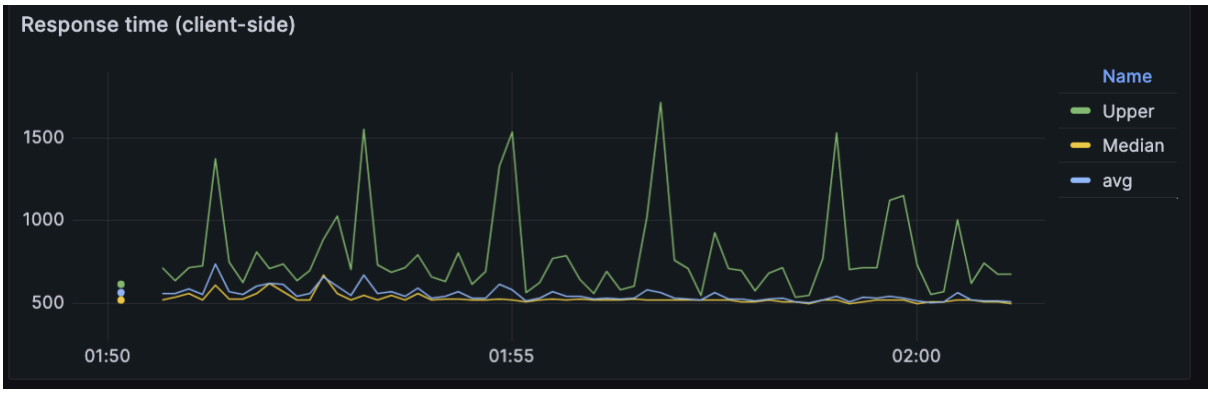
Podemos ver en este gráfico y en el siguiente que el response time promedio se encuentra alrededor de los 500 ms.

Tiempo de respuesta de la dependencia:



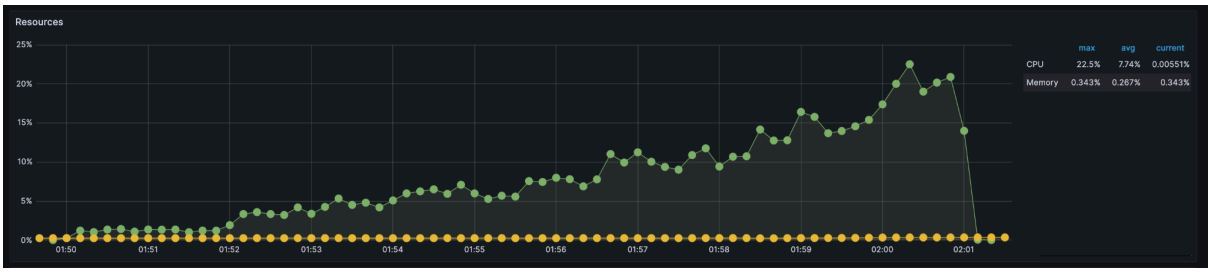
En este gráfico se puede observar saltos en la curva. Esto se debe a errores de rate limit provenientes de la API externa que no son incluídos en el cálculo de tiempo de respuesta.

Tiempo de respuesta total medido desde el cliente:



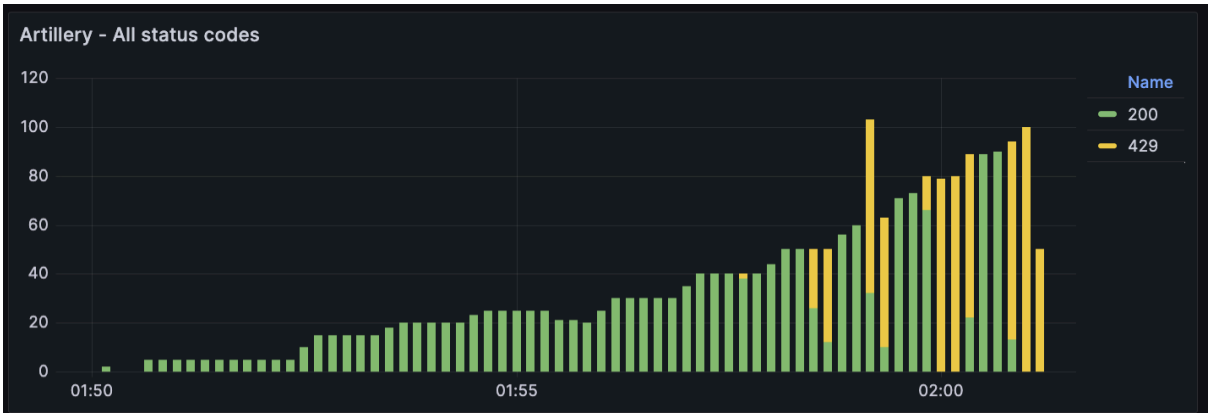
Viendo este gráfico podemos ver que artillery coincide con el tiempo promedio de respuesta establecido previamente.

Uso de recursos del server:



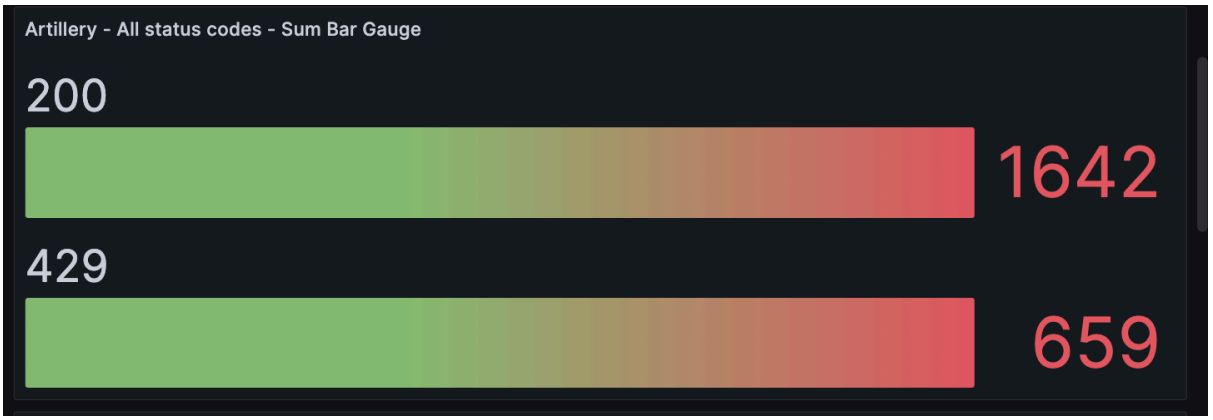
Como es esperable, aquí podemos ver cómo aumenta el uso del CPU a medida que aumenta la tasa de requests llegando a un valor máximo de 21%.

Status codes recibidos desde el cliente:



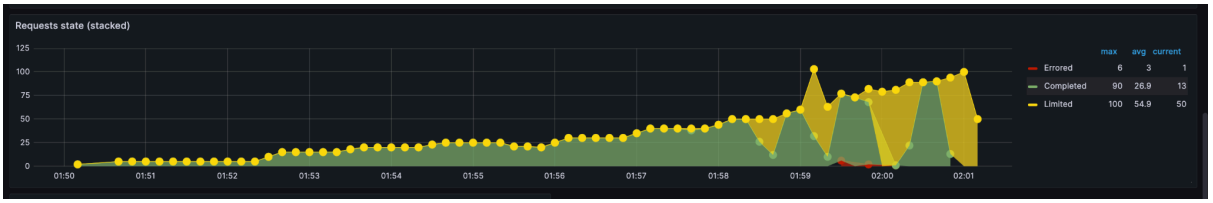
En este gráfico se puede observar bastante claro dónde se superó el rate limit de quotable.

Status codes totales:



En este gráfico podemos ver la cantidad de códigos de cada status. Utilizando además el gráfico anterior podemos ver cómo aproximadamente ¼ de los requests fallaron y eso se produjo en el último intervalo de los tests.

Status de requests de artillery



Otro gráfico para ver la cantidad relativa de mensajes enviados correctamente y errores de rate limit.

4.3.3 Escenario pico rate limited

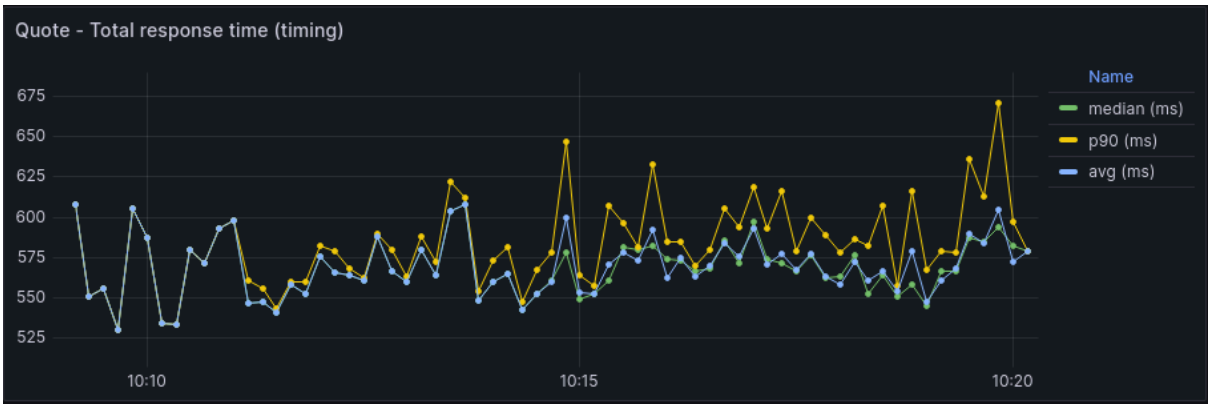
Para este endpoint decidimos utilizar una cantidad de requests por segundo máxima de 5 requests por segundo. Este se decidió a partir del límite de la API Quotable que soporta hasta tres requests por segundo antes de realizar rate limiting.

Throughput

Throughput recibido en la ejecución de los escenarios:

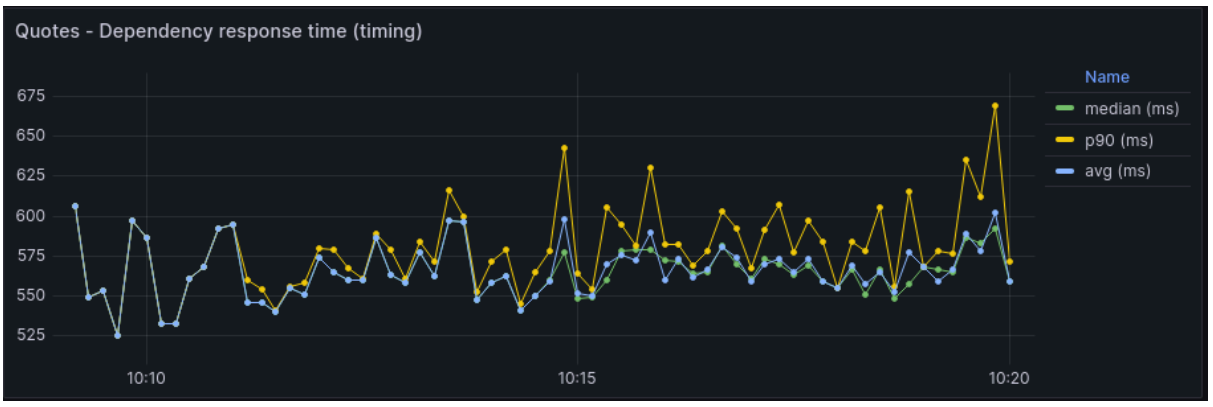
Aquí se puede ver claramente cómo el throughput se queda en 5 requests por segundo gracias a la configuración de un limitador de carga.

Tiempo de respuesta total medido a través de la api:

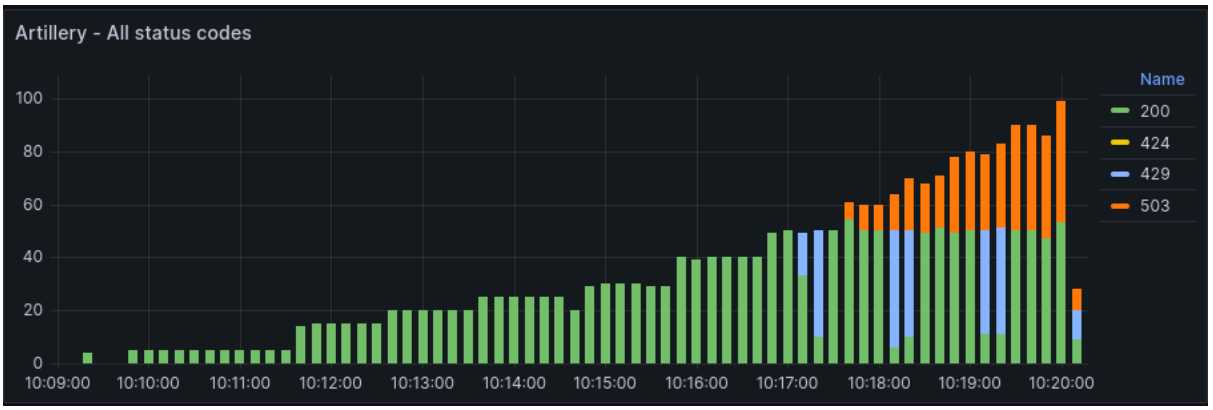


En este gráfico podemos ver el tiempo de respuesta del endpoint. Podemos ver que este se asemeja al tiempo testeado en el caso sin tácticas.

Tiempo de respuesta de la dependencia:



Status codes recibidos desde el cliente:



Aquí se puede observar claramente con los errores 503 dónde nginx realizó el trabajo de rate limiting.

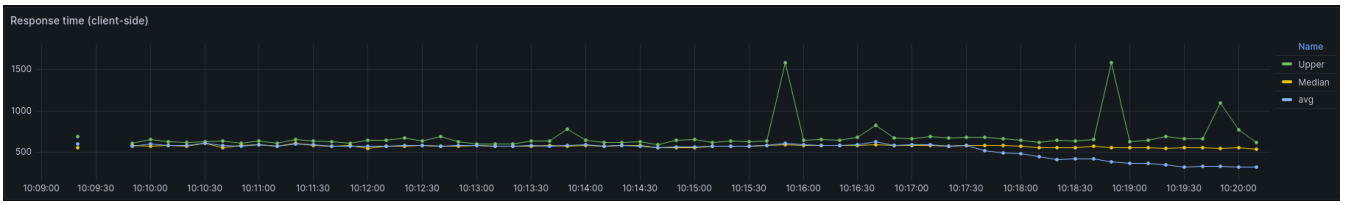
Errores por rate limit recibidos desde el cliente:

Puede verse aquí cómo la cantidad de errores 503 aumentaron en relación a la rampa establecida en los tests.

Status codes recibidos desde el cliente, mediciones gauge:

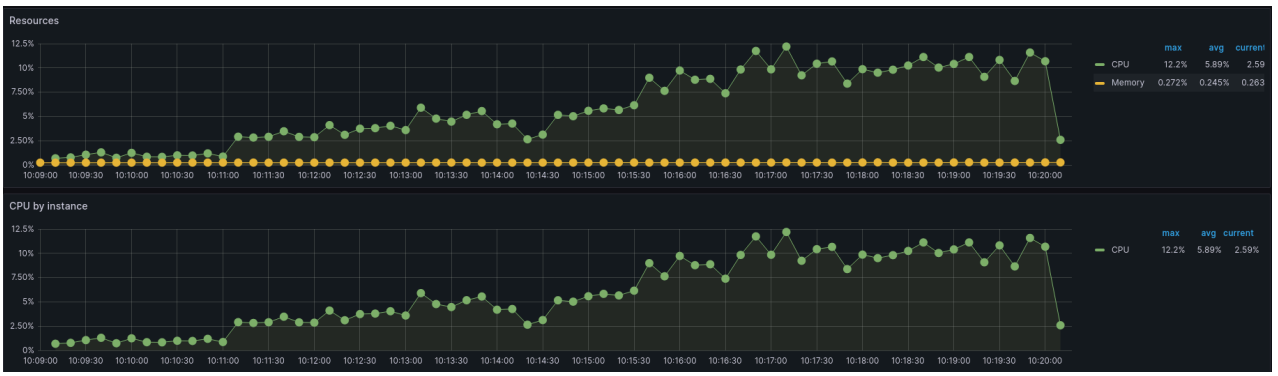
Aquí se puede ver que la cantidad de errores 429 y 503 se asemejan a la cantidad de errores 429 del caso sin rate limitado. Esto es porque nuestro máximo rate es mayor al de la API Quotable.

Tiempo de respuesta total medido desde el cliente:



Aquí podemos ver que el tiempo de respuesta promedio se mantiene constante con el caso sin táctica. El promedio disminuye al final debido a la cantidad de mensajes que retornan inmediatamente al ser limitados por nginx.

Uso de recursos del server:



Acá podemos ver cómo el uso de rate limiting redujo el uso máximo del cpu comparado con el caso sin tácticas.

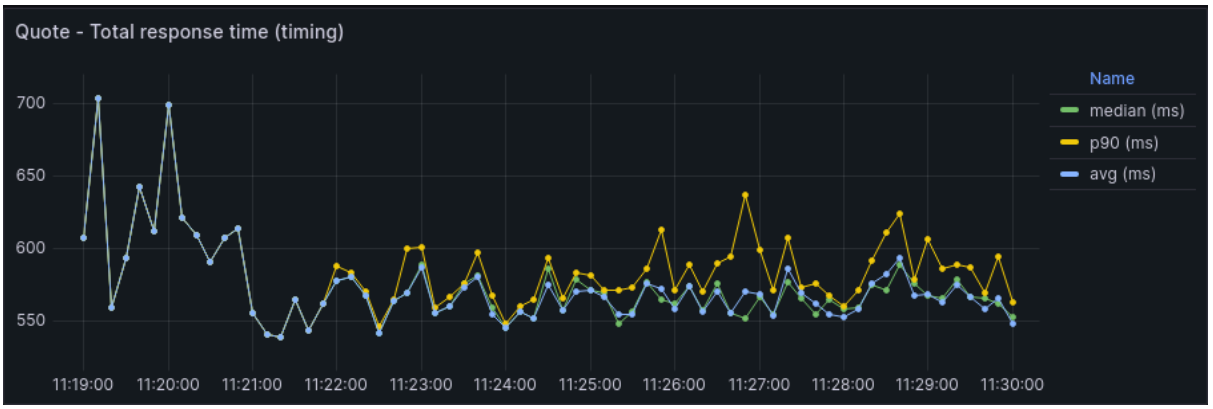
4.3.4 Escenario pico replicado

Throughput

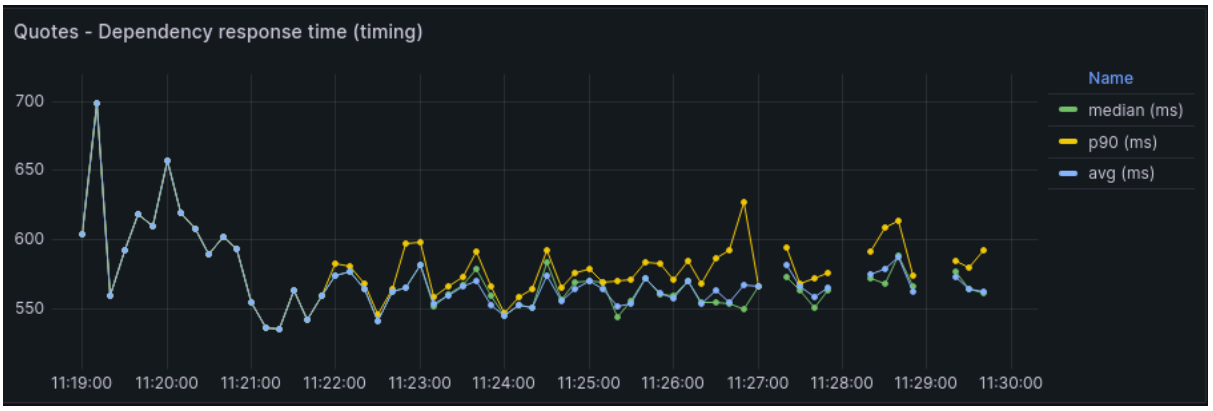
Throughput recibido en la ejecución de los escenarios:

En este gráfico vemos una curva similar a la establecida en el caso sin tácticas donde se observa una rampa en el throughput y se llega a un pico de 10 request por segundo.

Tiempo de respuesta total medido a través de la api:

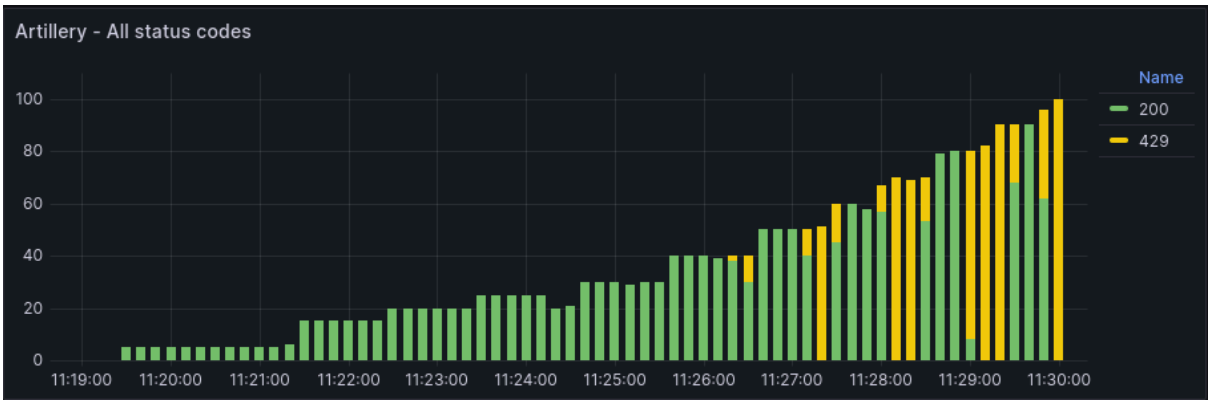


Tiempo de respuesta de la dependencia:



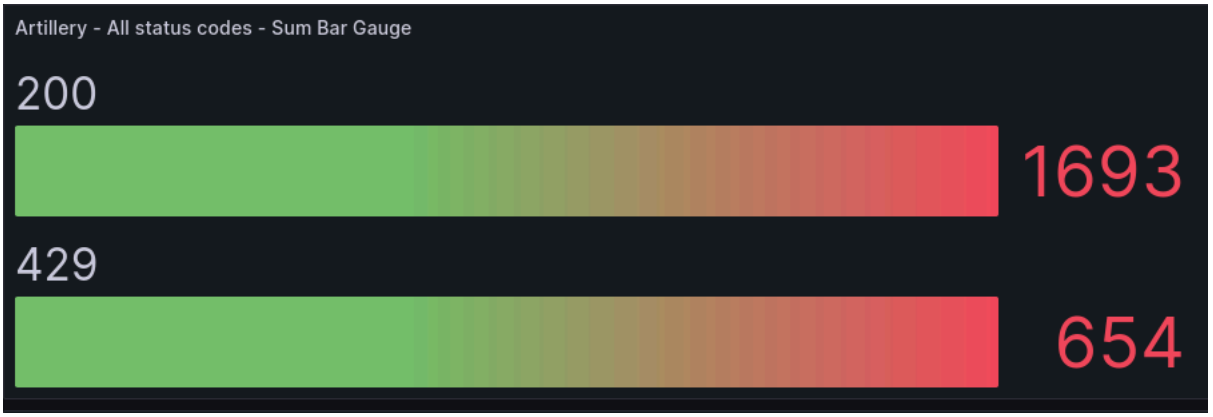
En ambos gráficos se puede ver cómo el response time se mantiene constante respecto a los tests anteriores y cómo se produce aquí también un corte en el gráfico debido al rate limit de Quotable.

Status codes recibidos desde el cliente:



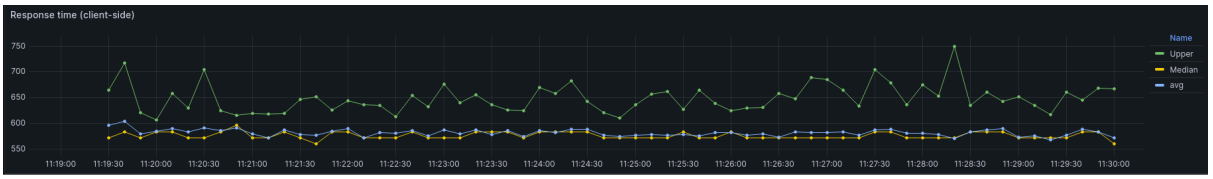
Aquí podemos ver un gráfico casi idéntico al caso sin tácticas que demuestra el rate limit de la dependencia.

Status Codes Sum and Count



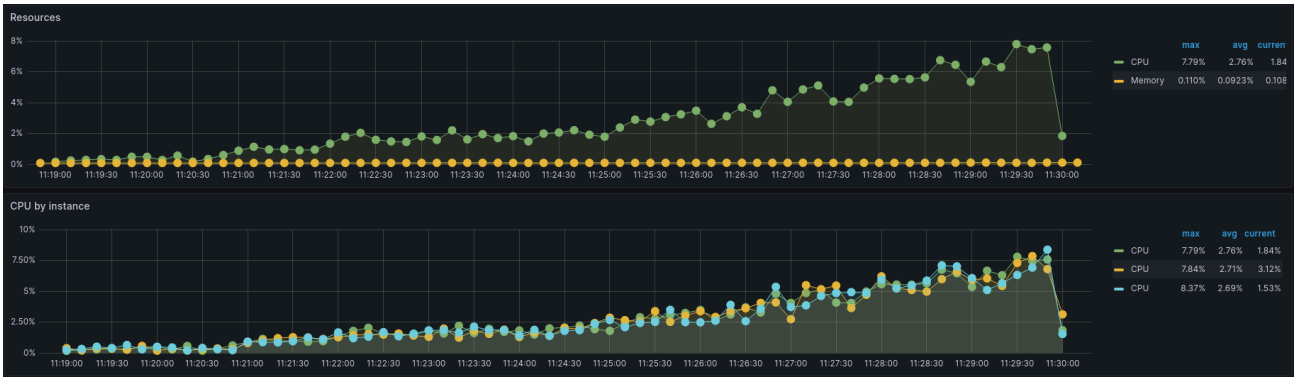
De la misma manera, podemos ver en este gráfico unos resultados similares a los testados anteriormente.

Tiempo de respuesta total medido desde el cliente:



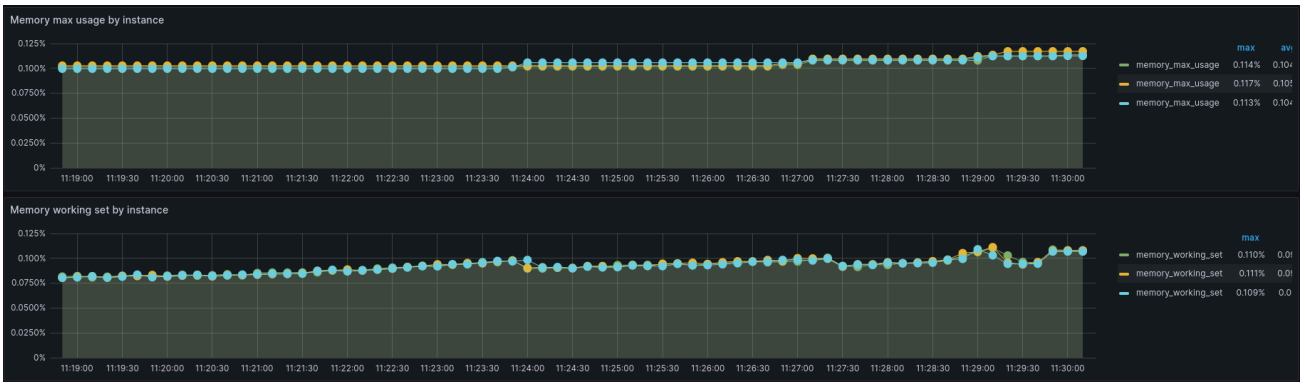
En este gráfico vemos nuevamente cómo el response time se mantuvo constante durante todos los escenarios.

Cpu por instancia



Aquí en el gráfico de recursos podemos claramente comparar el uso del cpu con el caso no replicado. Como es esperado, cada instancia de servidor utiliza menor cantidad del procesador teniendo en este caso en promedio 2.7% de uso.

Memoria por instancia



Igual que el gráfico anterior podemos ver cómo se divide cada instancia de la memoria y cómo esta se mantiene constante entre todas ellas.

4.4 Spaceflight news

El endpoint /spaceflight_news es el encargado de devolver los títulos de las últimas 5 noticias sobre la actividad espacial. Esta información es consumida a través de la Spaceflight News API, a través de su documentación se informa que la actualización de las noticias se hace cada 10 minutos. A través de la experimentación logramos ver que esta API soporta un nivel de carga superior al de las otras dependencias por lo que pudimos probar un conjunto de tácticas mayor que las del resto de endpoints ofrecidos.

A continuación veremos los resultados obtenidos de los escenarios elegidos para probar esta API.

4.4.1 Escenario base

Para el escenario base implementamos el siguiente esquema de fases:

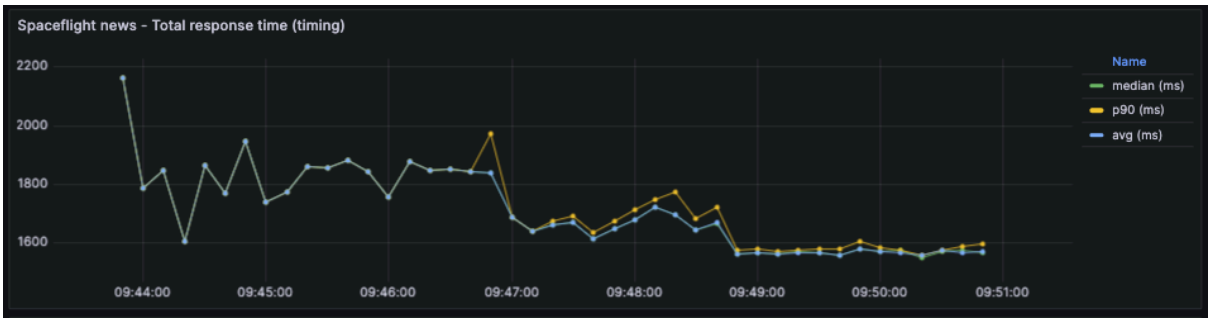
1. **Warm up**: Enfocado en precalentamiento inicial, con una duración de 3 minutos enviando 1 request cada 3 segundos.
2. **Elevación de carga**: Con una duración de 4 minutos, utilizando una rampa comenzando en 1 rps y elevándose hasta 5 rps.

Throughput

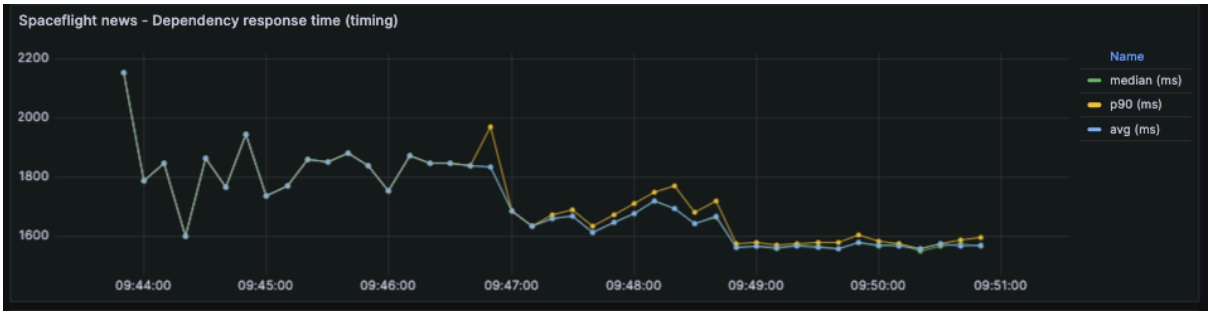
Throughput recibido en la ejecución de los escenarios:

Response time

Tiempo de respuesta total medido a través de la api:

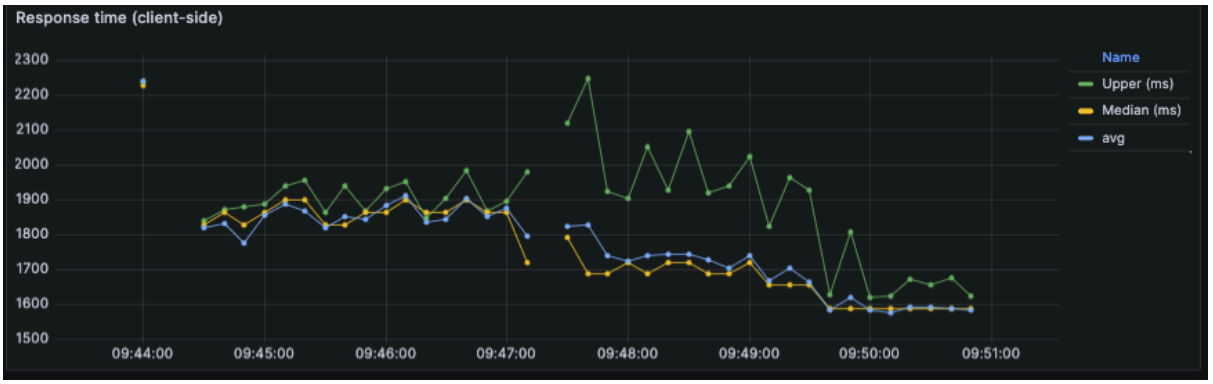


Tiempo de respuesta de la dependencia:



A través del análisis de los response times medidos desde la API, vemos que durante el período de Warm Up, el response contiene las estadísticas rondando los 1800 ms y luego desciende hacia 1600 ms.

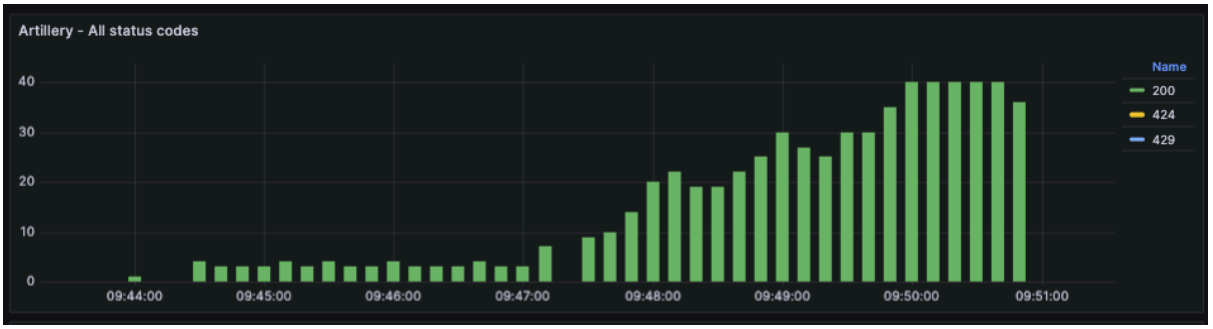
Tiempo de respuesta total medido desde el cliente:



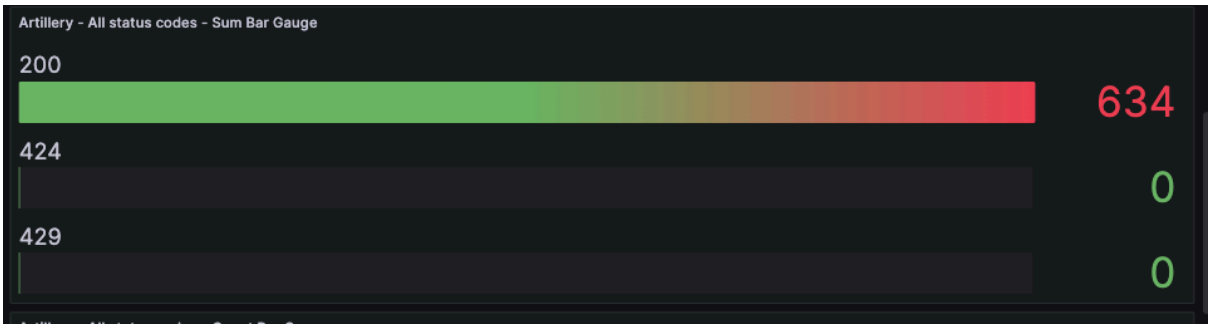
A la hora de observar las mediciones desde el cliente vemos que en general se mantienen cercanas a los tiempos de respuesta medidos desde el servidor. La excepción es el P90 que al comienzo del período de rampa se distancia sumando picos que van desde 200 a 800 ms al P90 medido desde el servidor.

Status codes

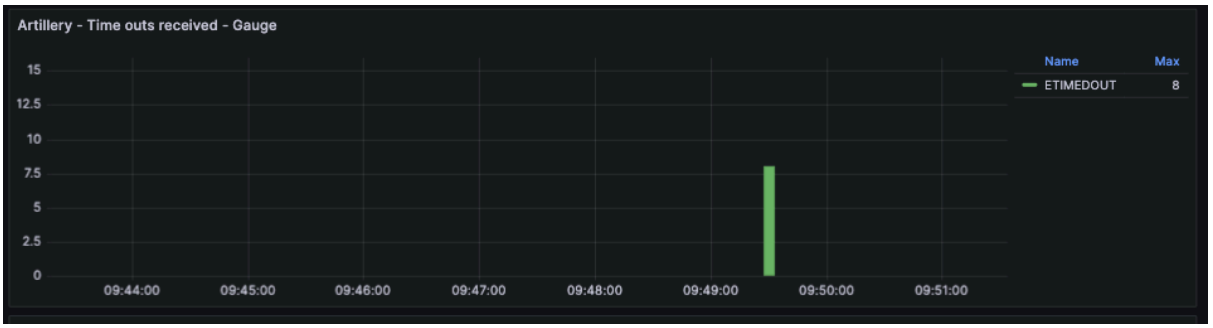
Status codes recibidos desde el cliente, gráfico de barras:



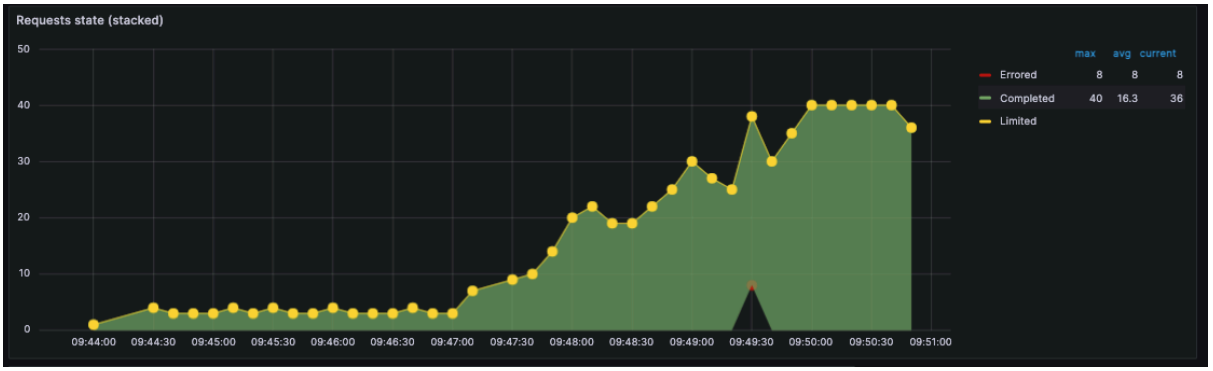
Status codes recibidos desde el cliente, mediciones gauge:



Time outs recibidos desde el cliente:



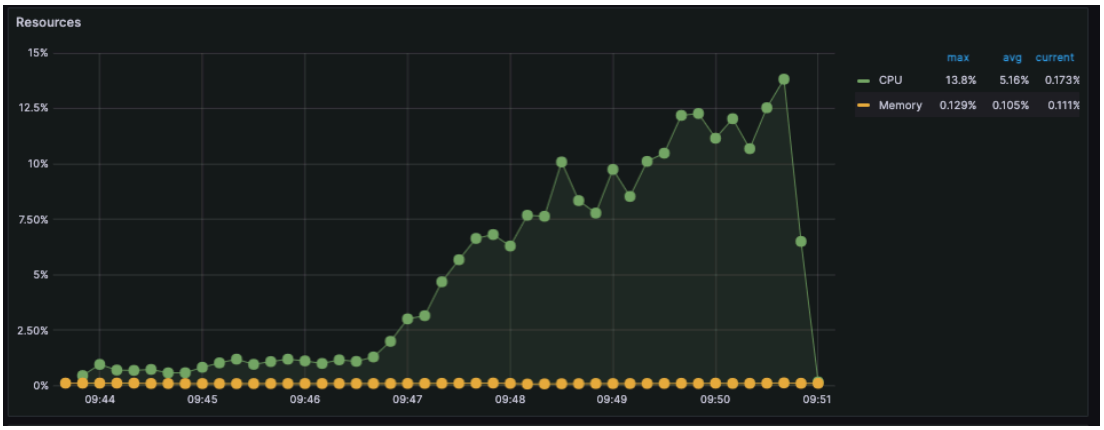
Resumen de resultados desde el cliente:



A través de las mediciones vemos que en este escenario la mayoría de las request fueron exitosas, a excepción de 8 casos que terminaron en un time out desde el cliente.

Recursos

Uso de recursos del server:



Cómo última métrica a analizar, podemos ver que la memoria no se ve afectada por la cantidad de tráfico recibido, en cambio la CPU sube a un porcentaje del 13% hacia el final de la prueba.

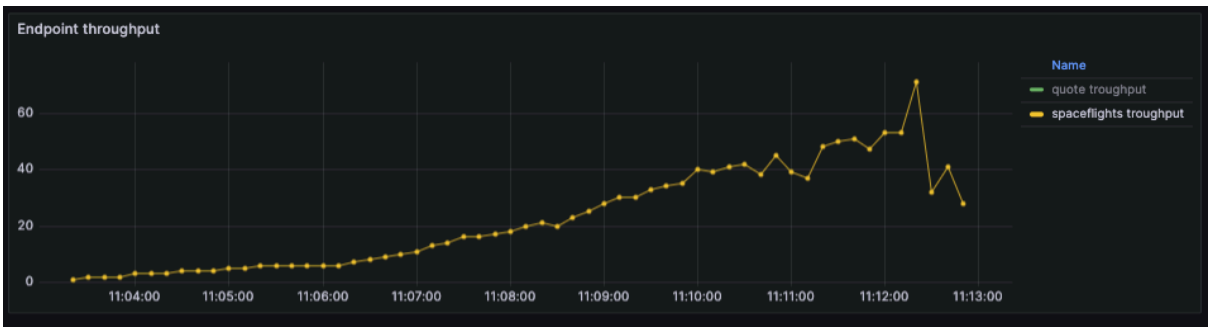
4.4.2 Escenario pico sostenido sin táctica

Para el escenario de pico sostenido implementamos el siguiente esquema de fases:

1. **Warm up**: Enfocado en precalentamiento inicial, con una duración de 2 minutos utilizando una rampa comenzando en 1 rps y elevándose hasta 5 rps.
2. **Fase plana de estabilización**: Con una duración de 1 minuto, enviando un tráfico constante de 6 rps.
3. **Primera elevación de carga:** Con una duración de 2 minutos, utilizando una rampa comenzando en 6 rps y elevándose hasta 20 rps.
4. **Segunda elevación de carga:** Con una duración de 2 minutos, utilizando una rampa comenzando en 20 rps y elevándose hasta 40 rps.
5. **Tercera elevación de carga:** Con una duración de 2 minutos, utilizando una rampa comenzando en 40 rps y elevándose hasta 55 rps.
6. **Carga plana final:** Con una duración de 1 minuto, enviando un tráfico constante de 60 rps.

Throughput

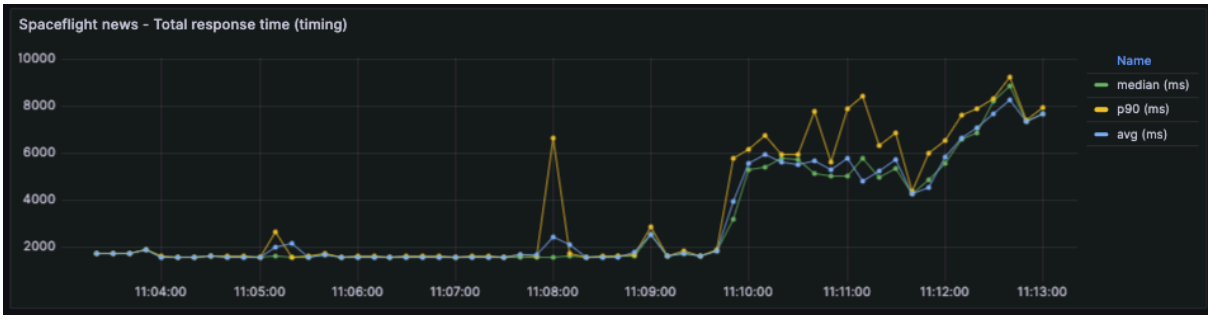
Throughput recibido en la ejecución de los escenarios:



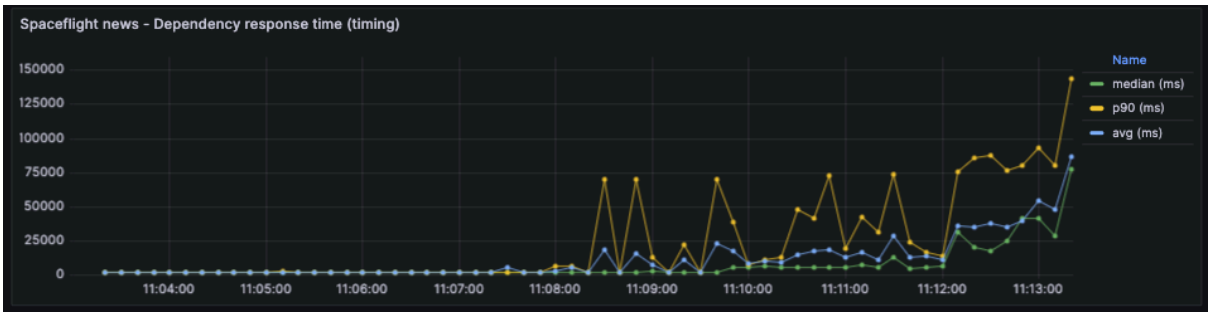
En las mediciones se puede ver el pico ascendente de tráfico recibido donde al final sufre una bajada, esto es debido a los problemas generados en el server por la cantidad de tráfico recibido, los cuales se verán en las próximas métricas.

Response time

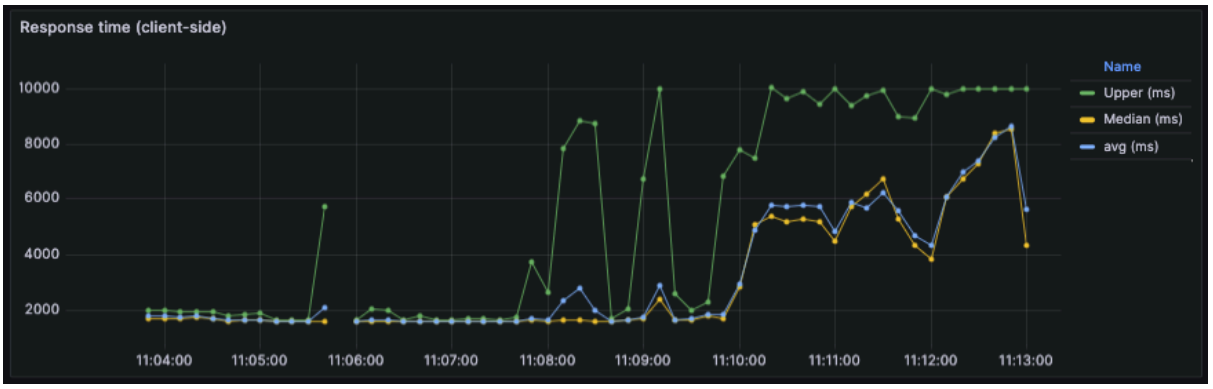
Tiempo de respuesta total medido a través de la api:



Tiempo de respuesta de la dependencia:



Tiempo de respuesta total medido desde el cliente:

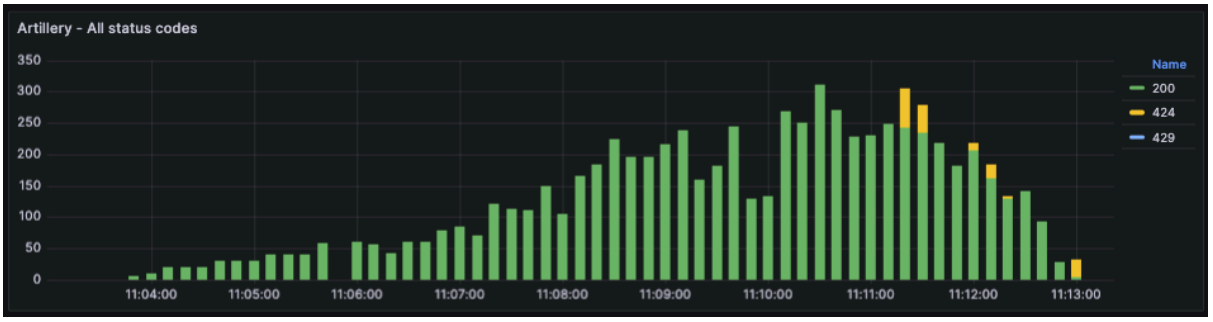


Al analizar los tiempos de respuesta podemos observar que durante la primera mitad de la carga los tiempos se mantienen estables rondando los 2 segundos para luego ir aumentando conforme aumenta la carga, en los tiempos del servidor todas las estadísticas se mantienen cercanas mientras que en los tiempos de la dependencia se puede ver cómo el P90 comienza a tener valores mucho más altos en órdenes de magnitud teniendo picos sobrepasando el minuto.

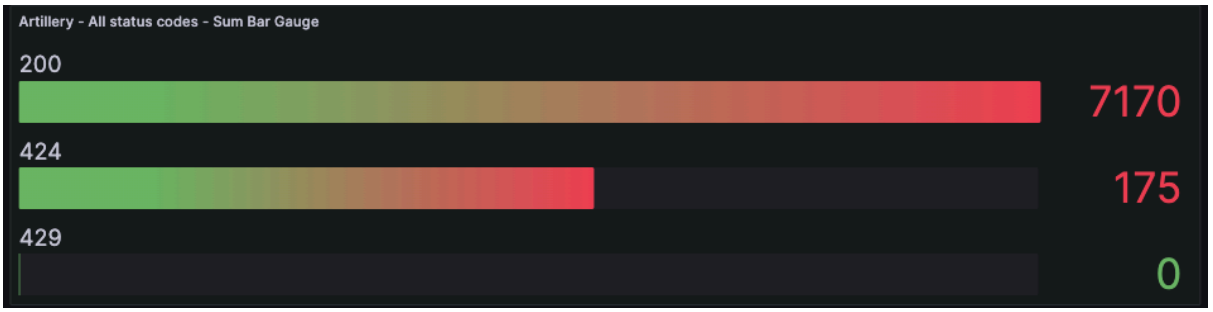
Desde el lado del cliente se puede ver que los tiempos de respuesta máximo se mantienen en 10 segundos debido al time out.

Status codes

Status codes recibidos desde el cliente, gráfico de barras:

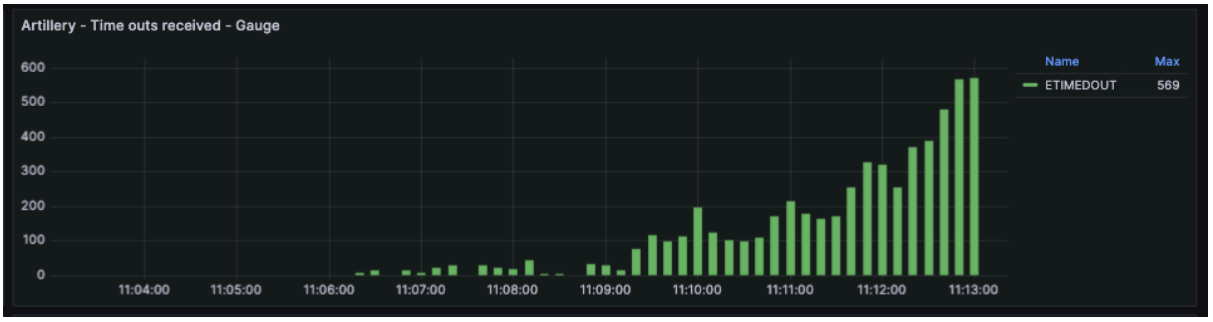


Status codes recibidos desde el cliente, mediciones gauge:

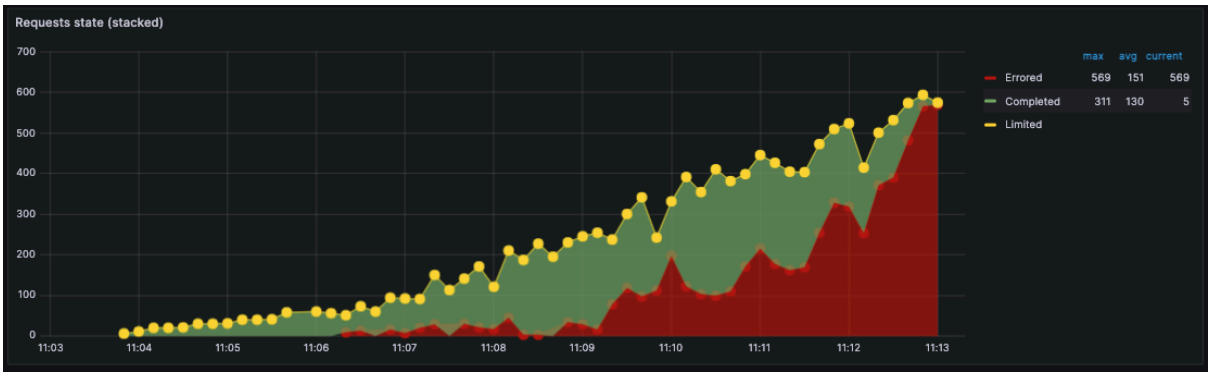


A través de los status codes recibidos correctamente desde el cliente podemos observar la ocurrencia de los 200 y los 424. Estos últimos son el status code elegido por nuestro servidor para informar que hubo un error en la solicitud hacia la dependencia. Observando sólo estos resultados podríamos pensar que el servidor soportó bastante bien la carga, pero a continuación vamos a hacer foco en demás métricas de error.

Time outs recibidos desde el cliente:



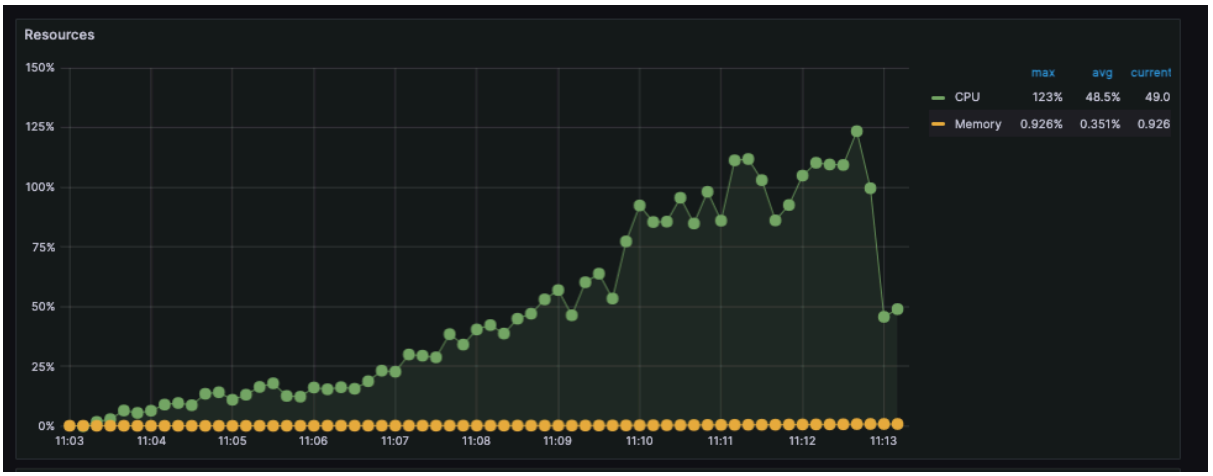
Resumen de resultados desde el cliente:



Observando estos últimos resultados podemos ver que la cantidad de errores totales registrados por el cliente fue mayor debido a la cantidad de solicitudes que terminaron con un time out, lo cual se evidenció en la medición de response times anteriormente mencionados.

Recursos

Uso de recursos del server:



Observando estos resultados podemos ver que en los últimos 4 minutos de ejecución de las pruebas el porcentaje CPU tuvo un fuerte incremento, lo cual puede asociarse claramente con la subida en tiempos de respuesta del servidor que terminaron en los time outs registrados por el cliente.

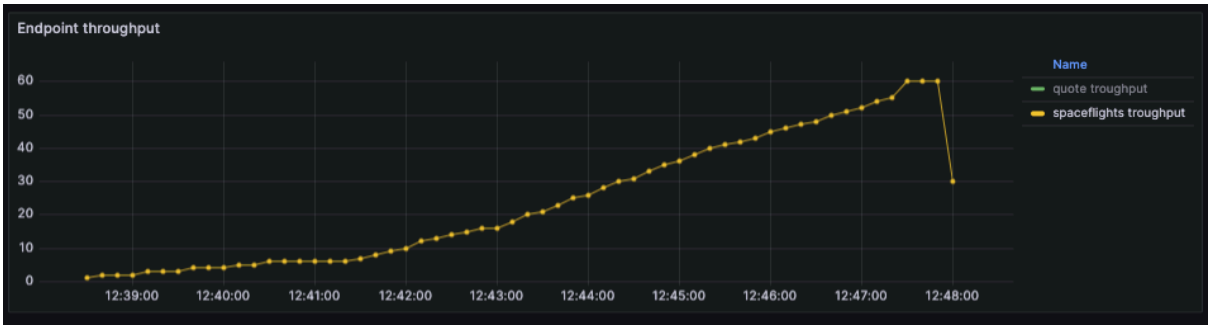
4.4.3 Escenario pico sostenido cacheado

En este escenario buscamos lograr una mejora en el soporte del pico final (*que en el escenario anterior causó los time outs*) a través del uso de una cache que nos evite tanto tráfico hacia la dependencia externa, evitando así sobrecargarla y resolver las peticiones con mayor velocidad y menor uso de CPU.

Para poder evidenciar esto de una manera más amigable en los gráficos, este escenario se probó utilizando un TTL para la caché de 30 segundos, de manera de poder diferenciar claramente la diferencia en los tiempos de respuesta.

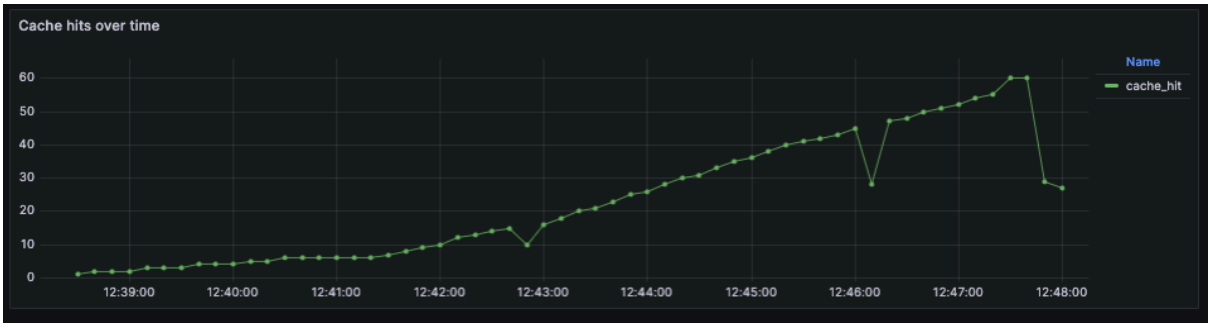
Throughput

Throughput recibido en la ejecución de los escenarios:



Cache

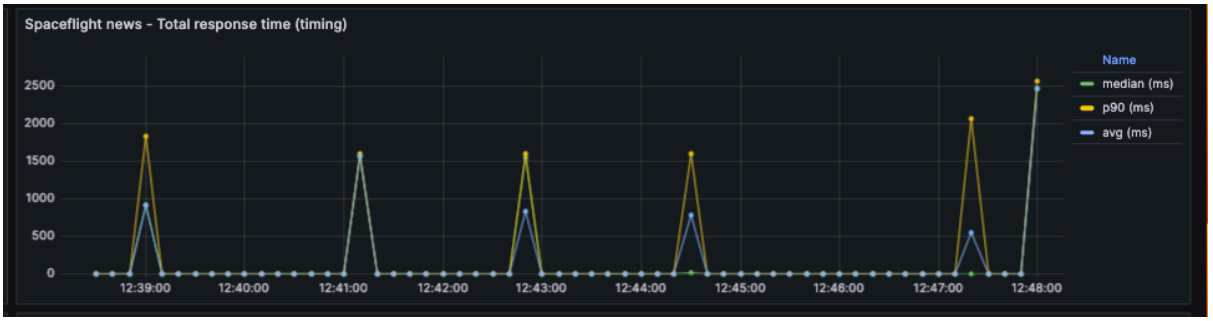
Cantidad de cache hits a través del tiempo:



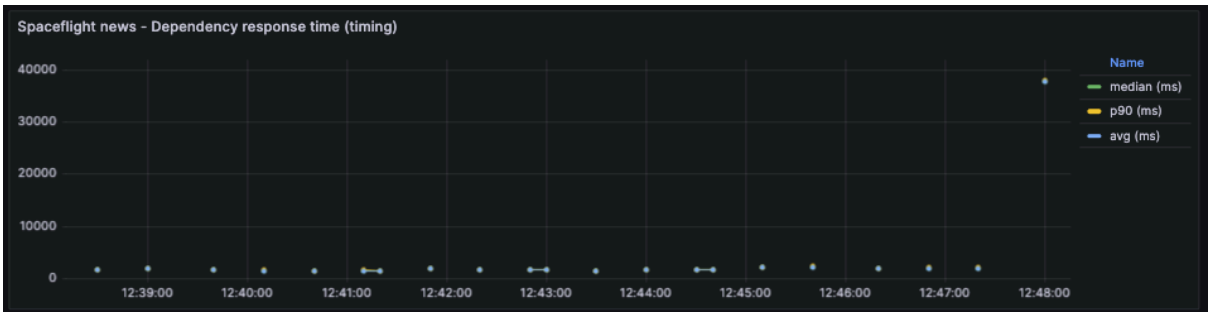
A través de esta medición podemos ver cómo hay una correlación entre el throughput recibido y la cantidad de veces que encontramos la información dentro de la caché.

Response time

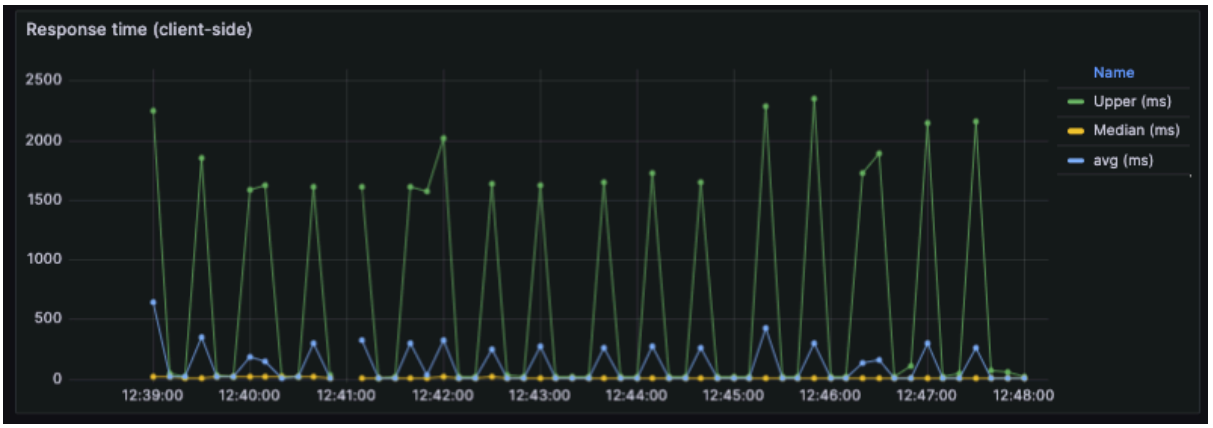
Tiempo de respuesta total medido a través de la api:



Tiempo de respuesta de la dependencia:



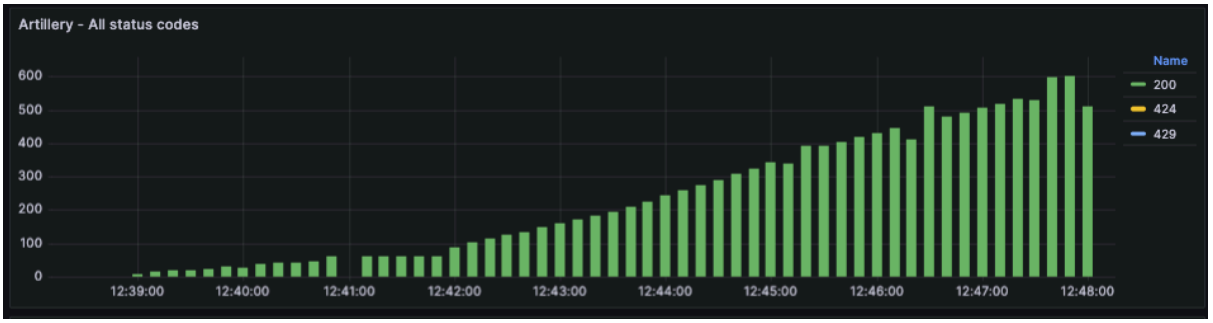
Tiempo de respuesta total medido desde el cliente:



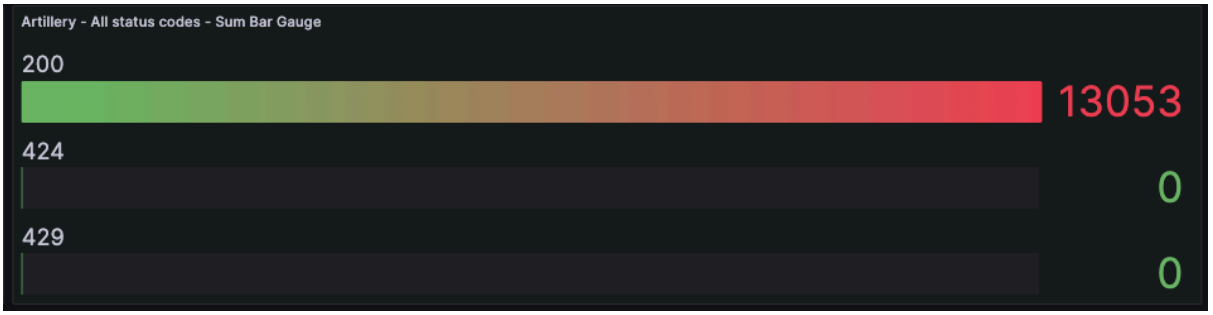
Al observar los tiempos de respuesta se puede observar claramente cómo mantienen un patrón constante por debajo de los 200 ms, en el P90 se puede ver las veces donde la caché no encontró la información solicitada por estar vencido el TTL y cómo aumenta el tiempo de respuesta al ir hacia la dependencia.

Status codes

Status codes recibidos desde el cliente, gráfico de barras:

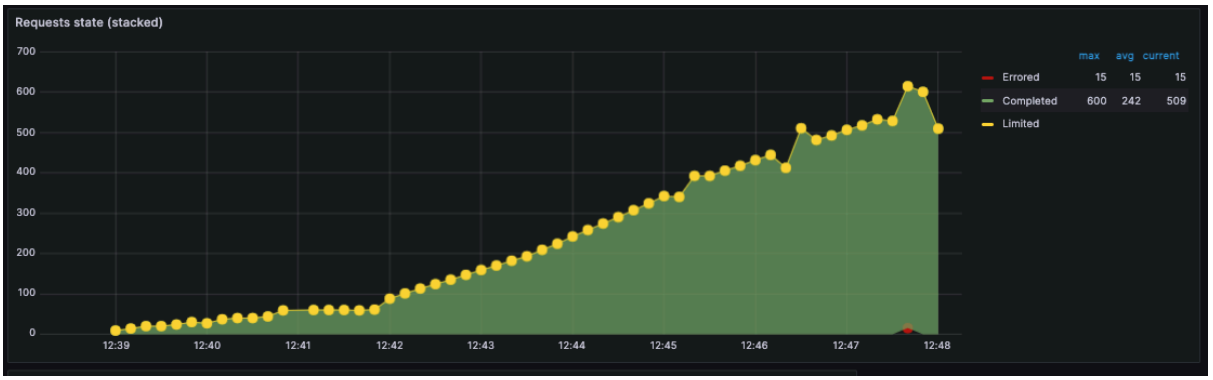


Status codes recibidos desde el cliente, mediciones gauge:



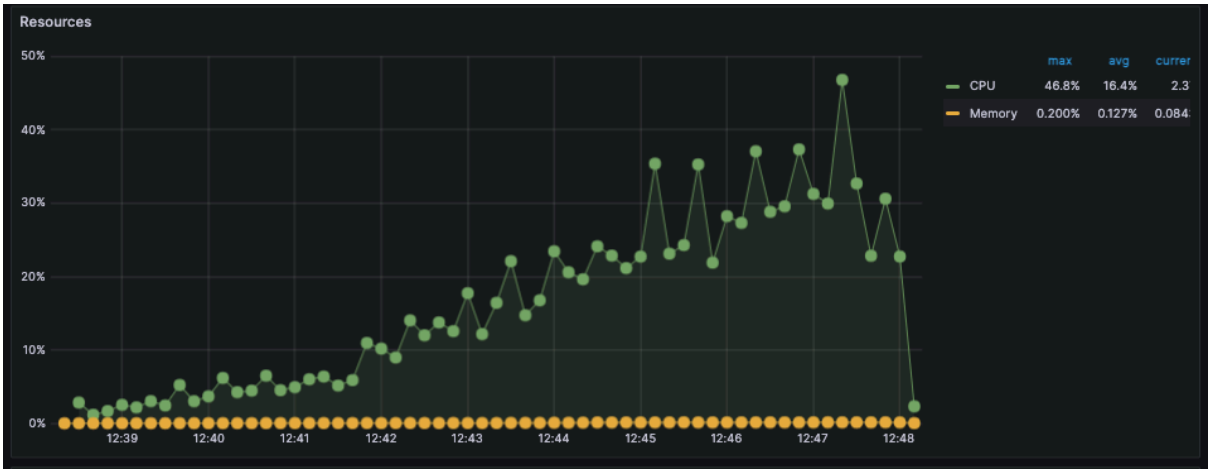
Time outs recibidos desde el cliente:

Resumen de resultados desde el cliente:



En el análisis de los resultados de las operaciones se puede ver claramente la influencia de la caché, evidenciando que las sobrecarga en las operaciones que se vieron en el escenario sin táctica no se ven al no derivar el tráfico de la misma manera hacia la dependencia.

Recursos



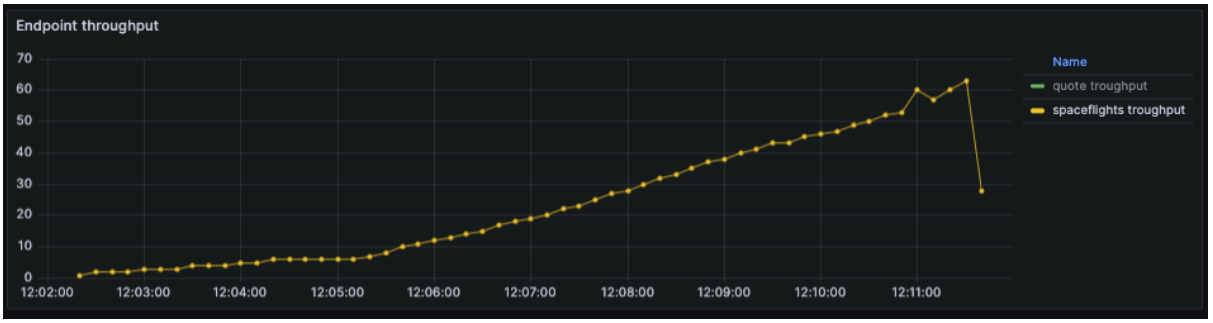
A nivel recursos se puede ver claramente que este escenario no llegó a consumir ni la mitad del porcentaje de CPU disponible, en concordancia con las métricas anteriores.

4.4.4 Escenario pico sostenido con replicación

En este escenario buscamos medir el impacto de disminuir la carga de CPU de nuestras instancias, cómo el nivel de tráfico termina siendo el mismo entendemos que esta táctica no debería aliviar significativamente los errores ligados a la sobrecarga de la dependencia.

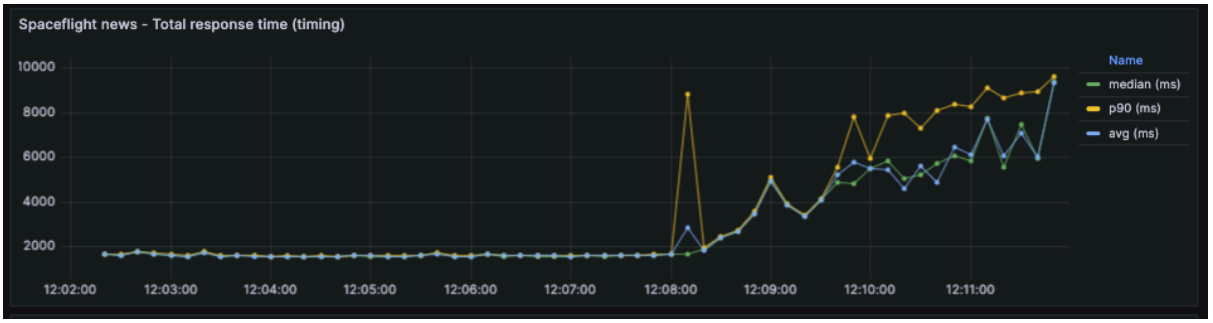
Throughput

Throughput recibido en la ejecución de los escenarios:

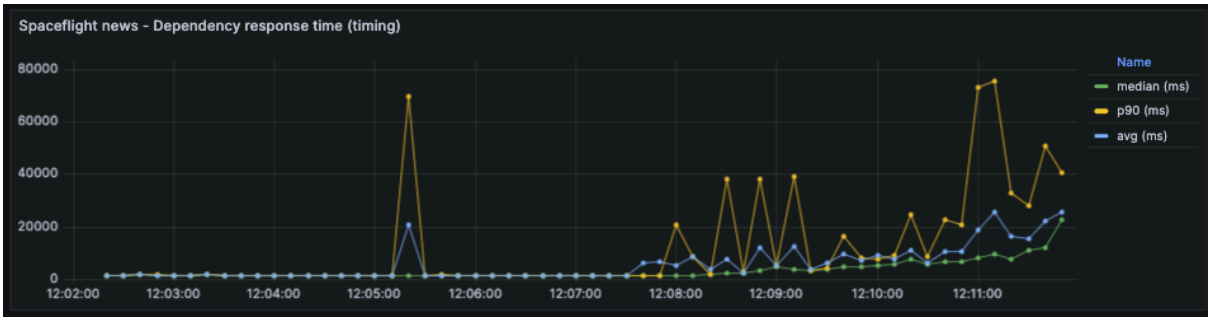


Response time:

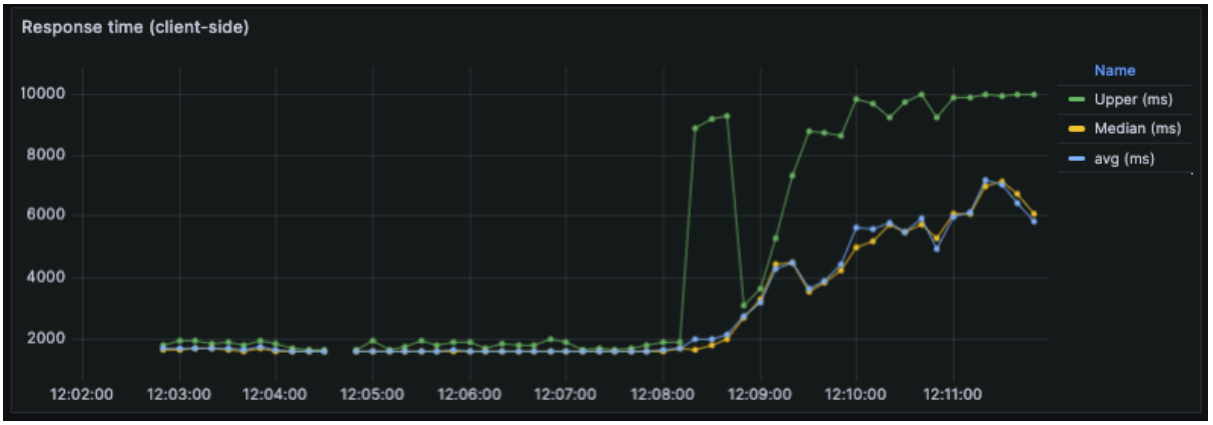
Tiempo de respuesta total medido a través de la api:



Tiempo de respuesta de la dependencia:



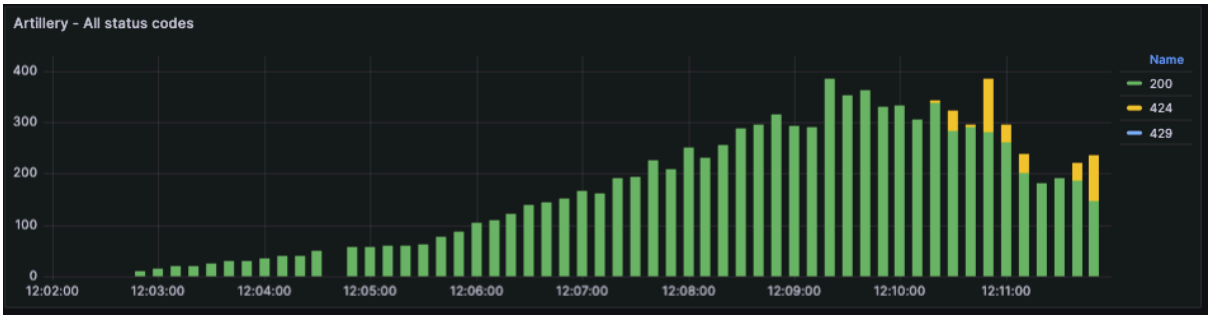
Tiempo de respuesta total medido desde el cliente:



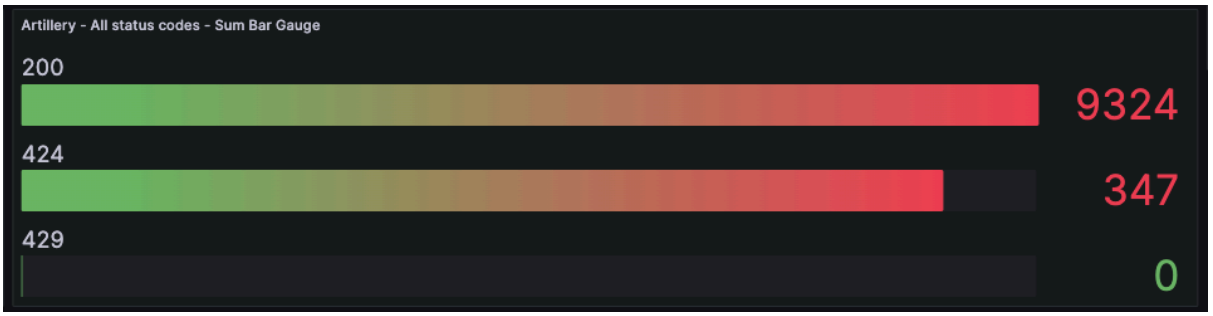
Analizando los tiempos de respuesta generales, no se observa una mejora realmente significativa al compararla con el escenario ejecutado sin ninguna táctica, se pueden observar picos de menor tamaño pero igualmente seguimos teniendo tiempos de respuesta altos.

Status codes:

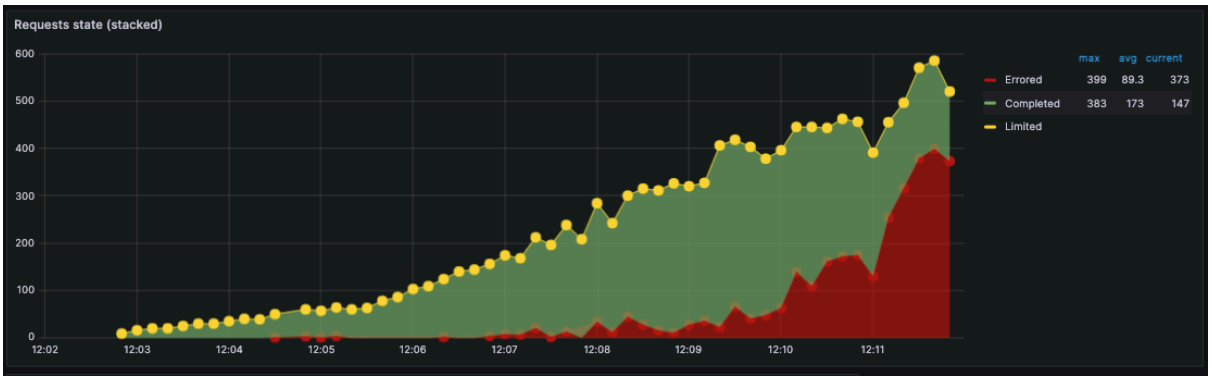
Status codes recibidos desde el cliente, gráfico de barras:



Status codes recibidos desde el cliente, mediciones gauge:



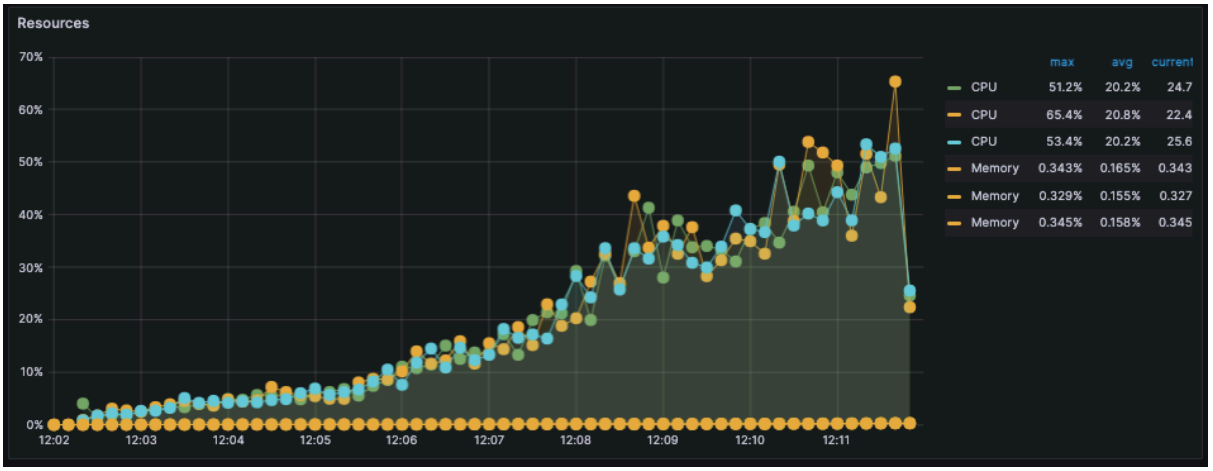
Resumen de resultados desde el cliente:



A la hora de analizar la cantidad total de respuestas correctamente recibidas desde el cliente podemos ver un aumento en la cantidad de códigos 200. En este escenario vemos aproximadamente 9 mil request exitosas en comparación a las 7 mil del escenario sin táctica.

Recursos

Uso de recursos del server:



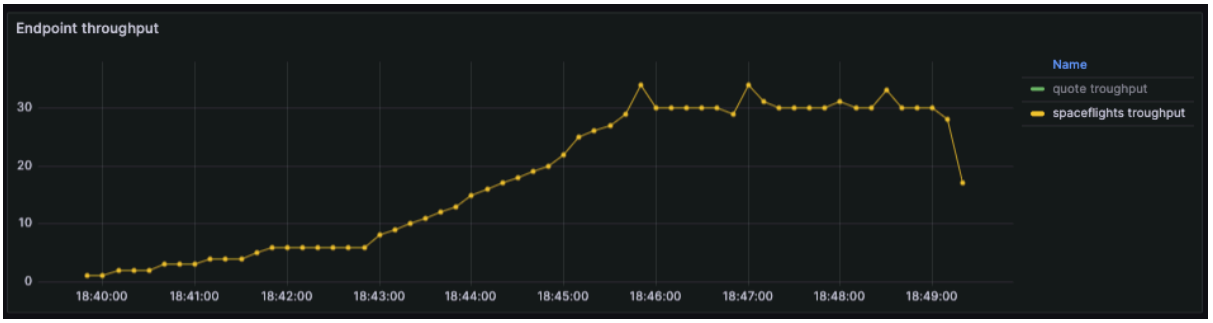
Al analizar el uso de recursos se puede ver la mejora que suponíamos al implementar la táctica, obteniendo un menor uso de porcentaje de CPU que el observado en el escenario de comparación. Se puede ver que el mayor pico alcanzado por todas las réplicas apenas sobrepasa el 50%.

4.4.5 Escenario pico sostenido con rate limit

Para la dependencia utilizada en este endpoint no encontramos documentación indicando que tenga un máximo de requests por segundo. A través de los anteriores escenarios analizados podemos observar que al sobrepasar el valor de 30 rps la dependencia termina ralentizando claramente y generando impactos en subida de CPU y tiempos de respuesta de nuestro servidor, por esto elegimos este valor para establecer nuestro propio rate limit y lograr disminuir las fallas por sobrecarga en nuestra dependencia.

Throughput

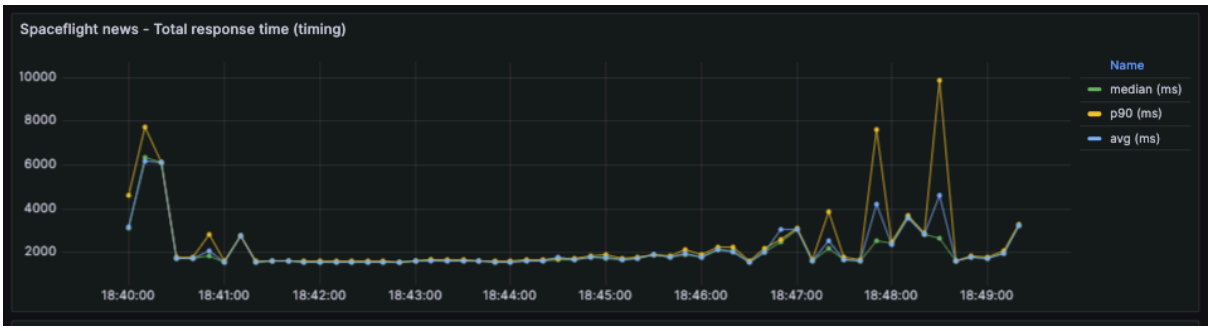
Throughput recibido en la ejecución de los escenarios:



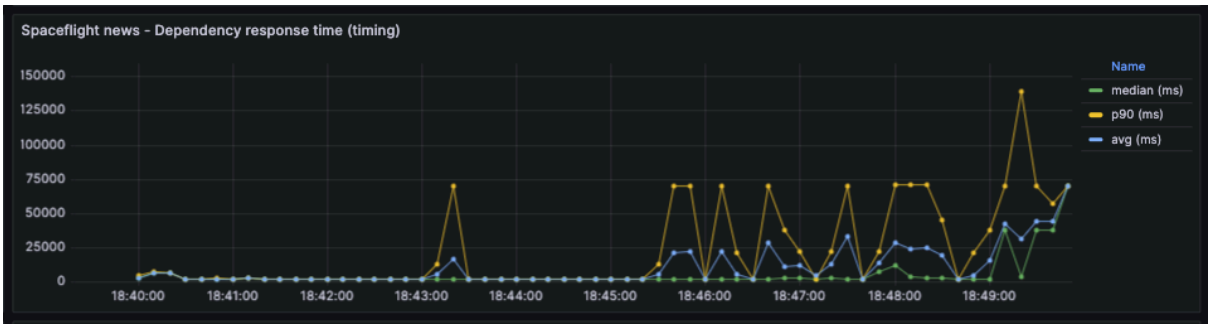
En la métrica medida desde la aplicación se puede ver actual al rate limit no permitiendo pasar más de 30 rps.

Response time

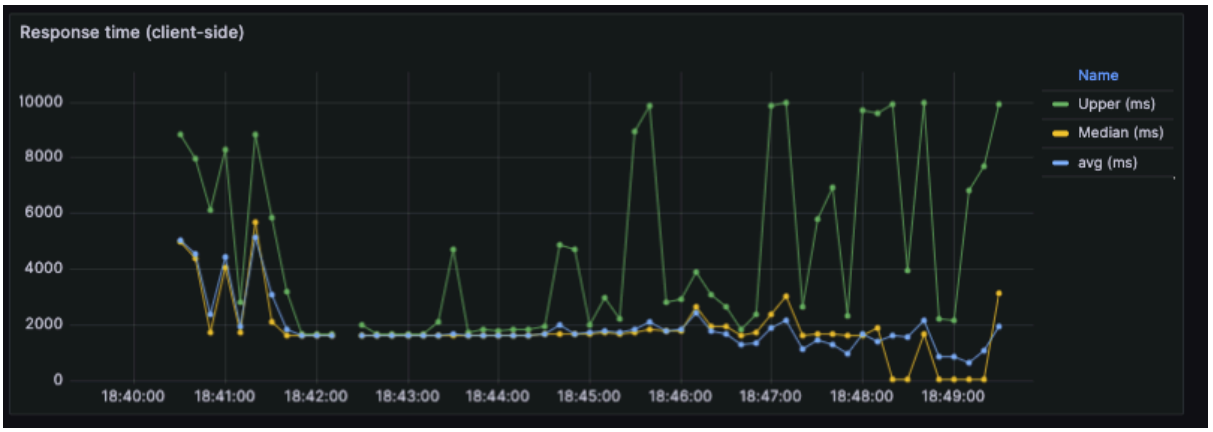
Tiempo de respuesta total medido a través de la api:



Tiempo de respuesta de la dependencia:



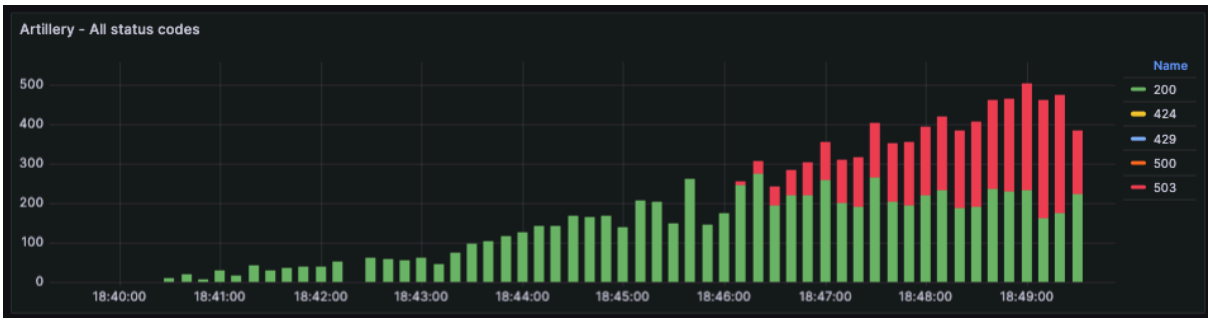
Tiempo de respuesta total medido desde el cliente:



Al analizar los tiempos de respuesta podemos ver que logramos menos picos en los tiempos de la dependencia, pero seguimos teniendo (*en menor cantidad que el caso sin tácticas*) escenarios de P90 muy altos medidos desde el cliente, lo cual debería corresponderse con time outs.

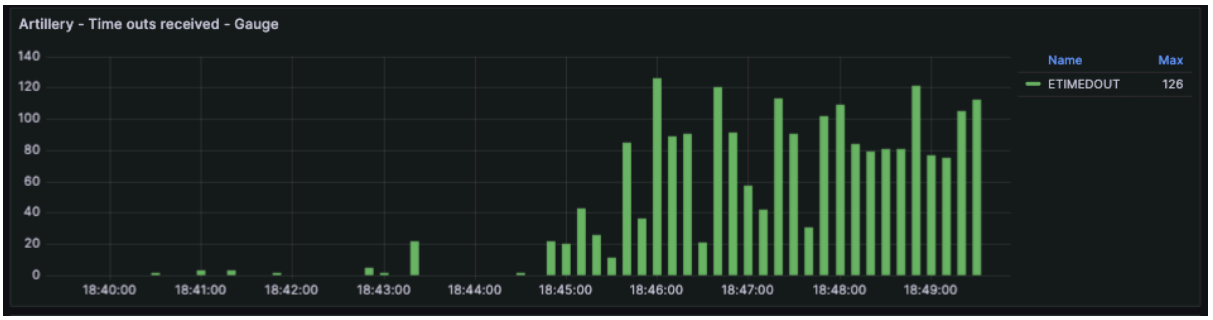
Status codes

Status codes recibidos desde el cliente, gráfico de barras:



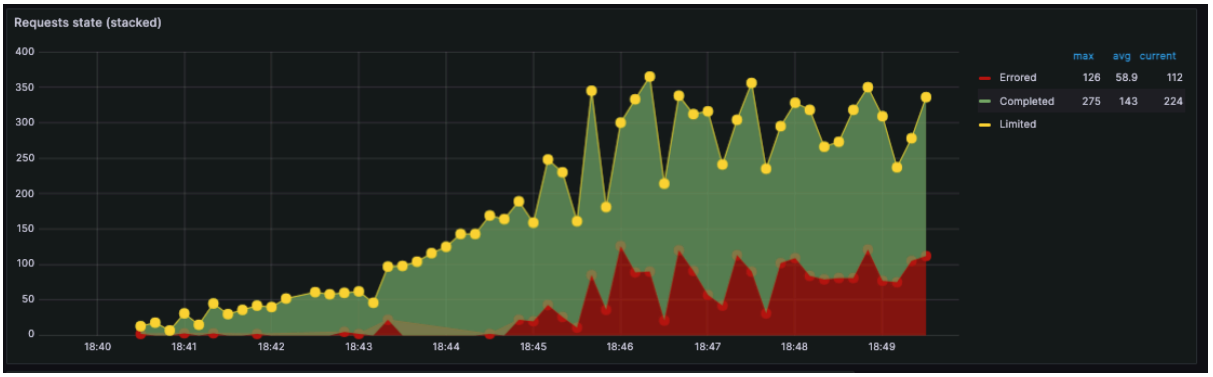
Errores por rate limit recibidos desde el cliente:

Time outs recibidos desde el cliente:



Status codes recibidos desde el cliente, mediciones gauge:

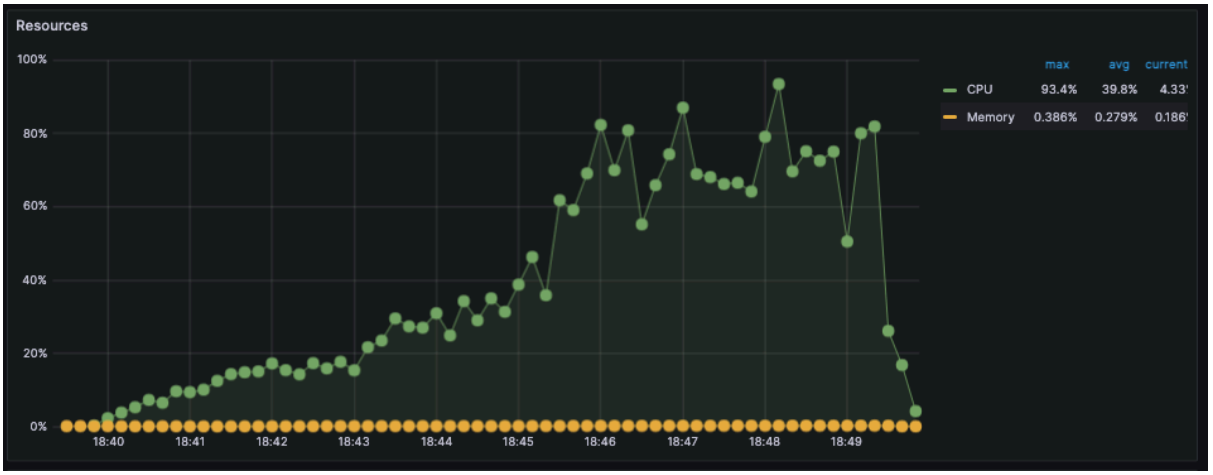
Resumen de resultados desde el cliente:



Al analizar los resultados obtenidos, podemos ver que dejamos de tener errores causados por fallas en la dependencia, esto lo atribuimos al rate limit establecido que evita sobrecargarla, cómo resultado final los tres casos que nos quedan son los 200, los 503 y los escenarios que dieron time outs.

Recursos:

Uso de recursos del server:



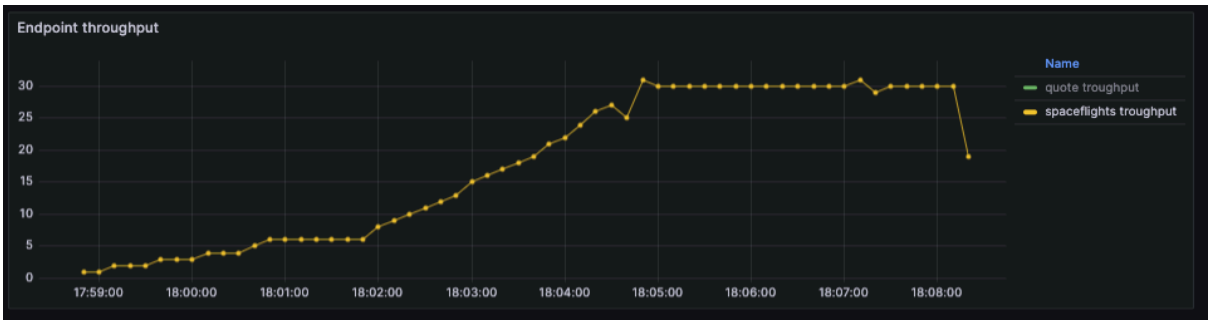
Como era de esperarse, el escenario de rate limit en mayor medida terminó protegiendo a la dependencia y si bien también protegió a nuestro server, el valor elegido de 30 RPS no nos deja tan cómodos a nivel CPU con una sola instancia del servidor, lo que apunta a ser la causa de la mayoría de errores de time outs obtenidos.

4.4.6 Escenario pico sostenido con rate limit y replicación

Finalmente, cómo último escenario se decidió probar una combinación de rate limit con la replicación, con el objetivo de lograr los beneficios de la replicación al disminuir el uso de CPU por instancia y a la vez el rate limit para evitar sobrecargar la dependencia externa.

Throughput

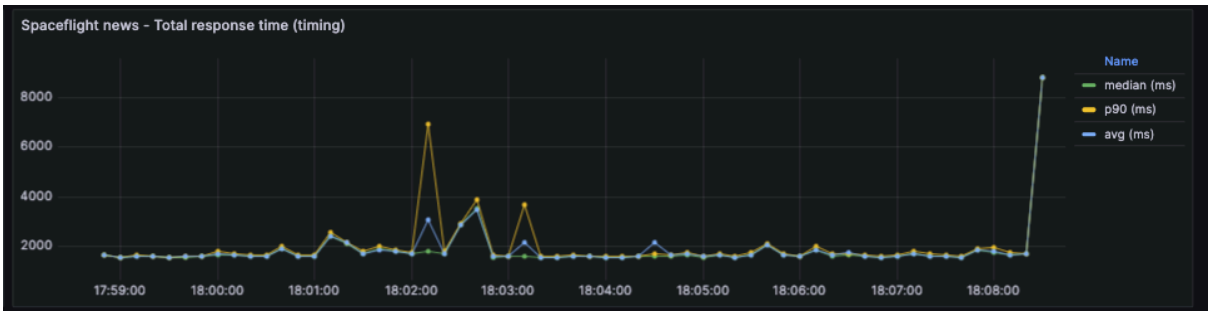
Throughput recibido en la ejecución de los escenarios:



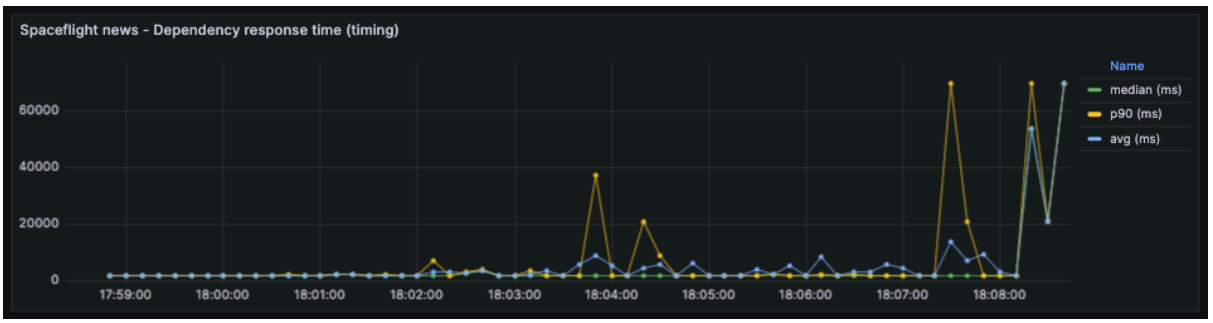
En la métrica medida desde la aplicación se puede ver actual al rate limit no permitiendo pasar más de 30 rps.

Response time

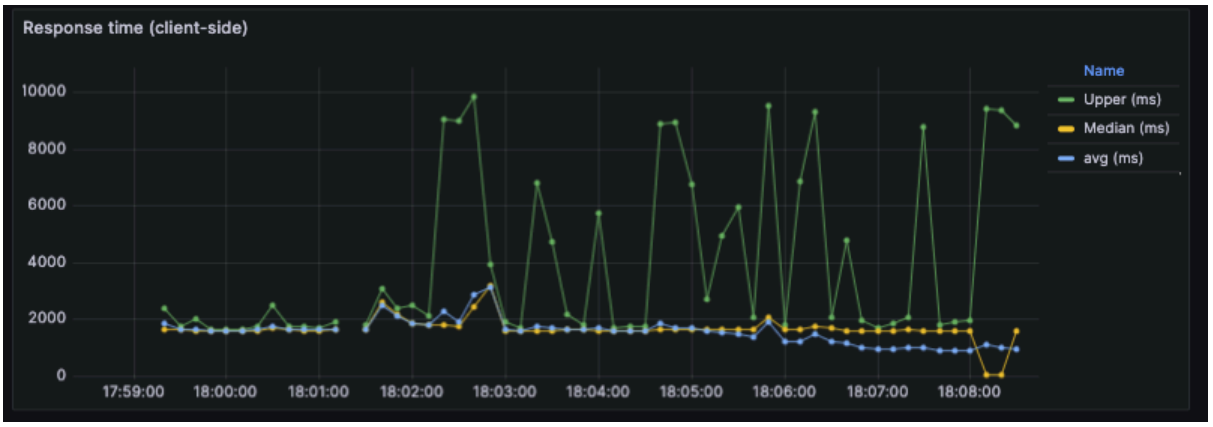
Tiempo de respuesta total medido a través de la api:



Tiempo de respuesta de la dependencia:



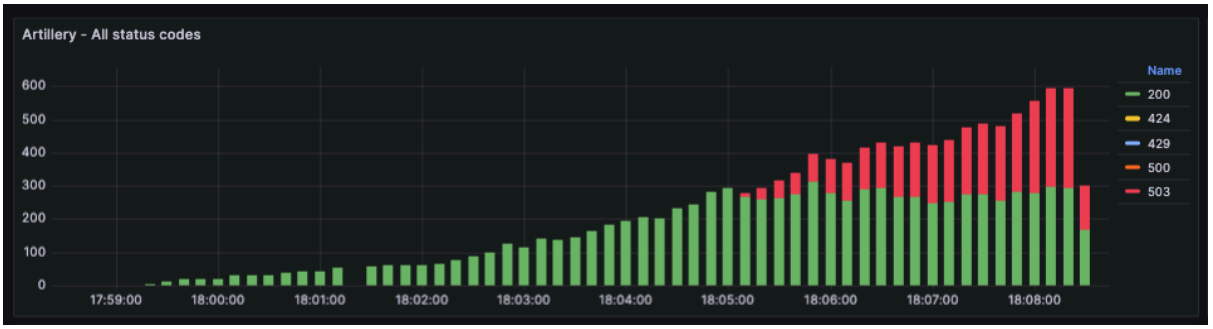
Tiempo de respuesta total medido desde el cliente:



Al observar los tiempos de respuesta, podemos ver cómo logramos una mejora muy considerable respecto al escenario ejecutado sin ninguna táctica, seguimos teniendo algunos picos en el P90 desde el lado del cliente, pero a nivel general los tiempos se mantienen mucho más estables.

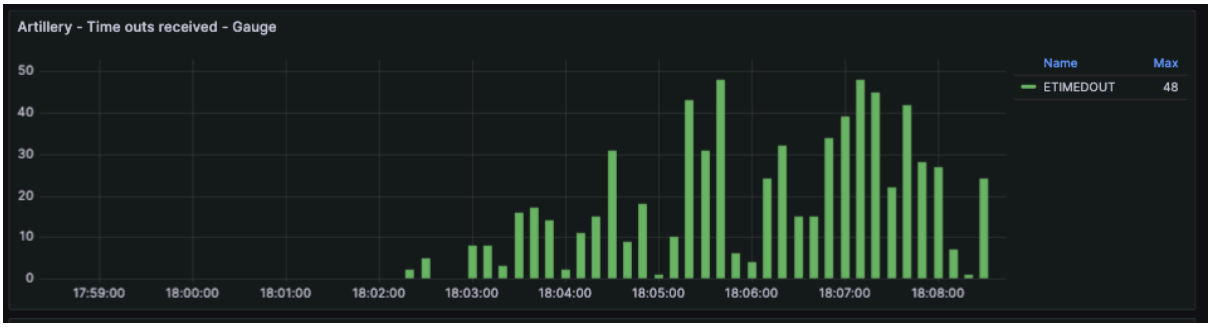
Status codes

Status codes recibidos desde el cliente, gráfico de barras:



Status codes recibidos desde el cliente, mediciones gauge:

Time outs recibidos desde el cliente:

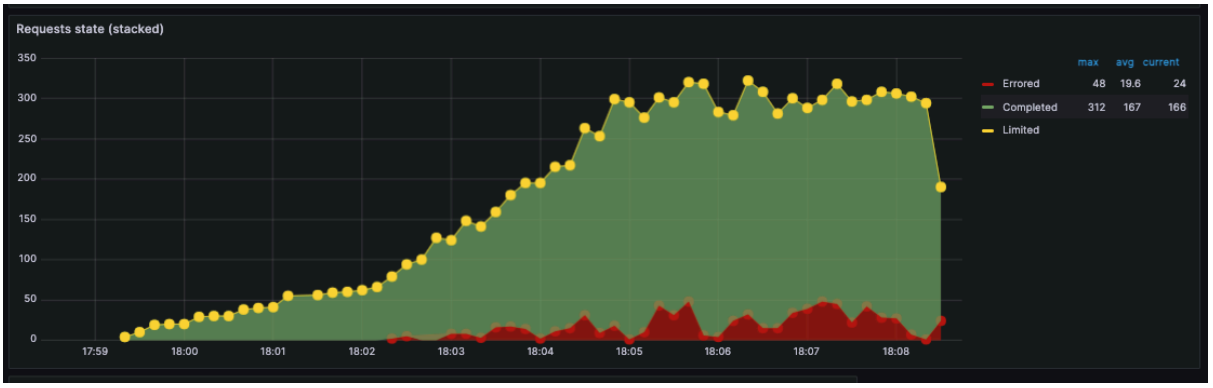


Errores por rate limit recibidos desde el cliente:

Al observar los status codes podemos ver que solo tenemos dos variantes, los 200 y los 503 causados por el rate limit, también vemos que tenemos una menor ocurrencia de time outs, solo 48 registrados. No tenemos ningún 424 lo que nos indica que no

recibimos un error inesperado desde la dependencia lo cual se puede asociar a que recibió una menor carga debido a nuestro rate limit.

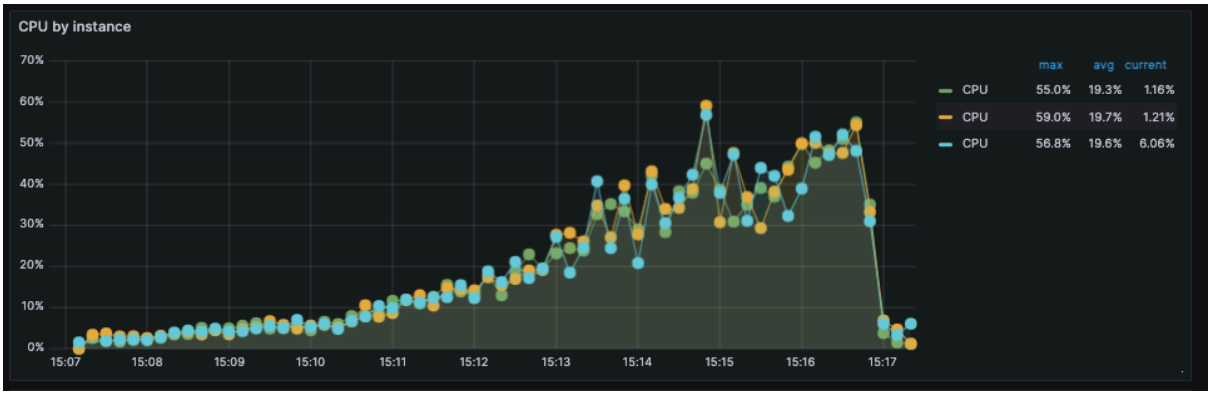
Resumen de resultados desde el cliente:



Al observar los resultados finales desde el cliente se ve una increíble mejora en la cantidad de escenarios correctamente ejecutados en comparación de los errores por time out.

Recursos

Uso de recursos del server:



Al igual que en el escenario de solo replicación se puede ver que las instancias mantienen picos de porcentaje de CPU cercano al 50% , pero manteniéndose por debajo de ese valor en la mayoría del tiempo durante el pico.

5. Conclusiones

Como análisis final podemos notar que los diferentes endpoints tenían características distintas ligadas a la dependencia que consumían, permitiéndonos, en algunos casos como el de Spaceflight, aplicar una mayor cantidad de carga y en otros como el de Dictionary muy poco espacio de mejora.

Poniendo en común los resultados observados podemos ver los beneficios que tuvieron ciertas tácticas en algunos aspectos de nuestra arquitectura, algunos a mencionar:

- **Cache**: Pudimos ver que al utilizarla en aquellos endpoints que consideramos adecuados logramos una reducción de los tiempos de respuesta. Spaceflight news produjo una diferencia más destacada que Dictionary pero ambos obtuvieron menor response time que el caso sin caché.
- **Rate Limiting:** Al implementar esta táctica llegamos a observar el efecto drástico que tiene la limitación en en la cantidad de requests que puede enviar un usuario en un periodo de tiempo determinado y cómo influye en la “protección” de la performance en escenarios donde una cantidad de carga no es soportada y se decide limitarla.
- **Replicación**: En general, fue la estrategia que mejores resultados generó para soportar un nivel alto de carga sin la utilización de una caché. Al comparar los resultados obtenidos con rate limiting (especialmente en el caso de Spaceflight News) pudimos ver como la replicación ayuda a reducir el costo de recursos de cada servidor individual ayudando con la escalabilidad del sistema.