Facultad de Ingeniería
Universidad de Buenos Aires

Arquitectura de Software 75.73

Alumnos

- Guido Bergman (104030)
- Matias Merlo (104094)
- Ramiro J Sanchez (104095)

Introducción **3**
Escenario evaluado 3
Tácticas utilizadas **4**
Rate limiting 4
Conclusión 6
Caché 6
Conclusión 9
Réplicas 9
Conclusión 12

Introducción

En este trabajo se busca comparar distintas tácticas aplicadas a servicios de Node.js involucrando el consumo de 4 APIs externas.
Las tecnologías que utilizaremos son:

- Docker
- Docker-compose
- Redis
- Nginx

Para las estadisticas y metricas:

- Grafana
- Graphite
- Artillery

Las APIs utilizadas son:

- Spaceflight News
- Useless facts
- METAR

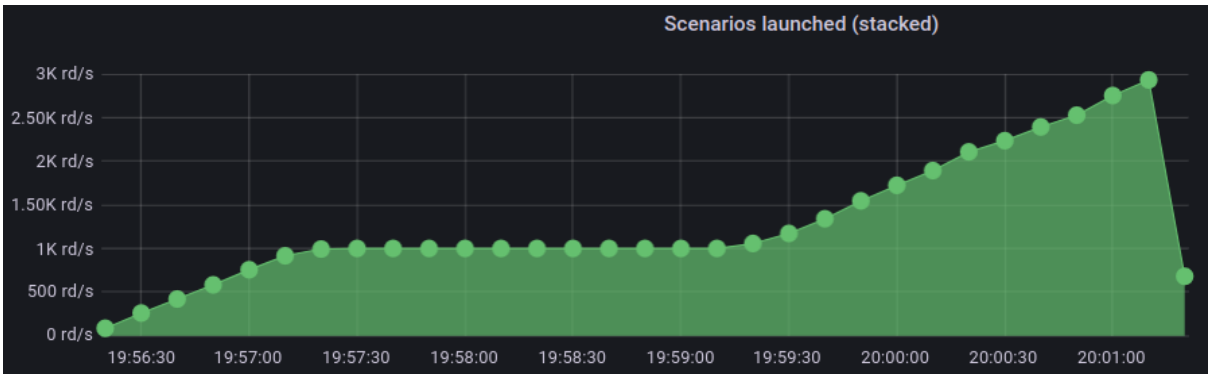
Escenario evaluado

Para evaluar el impacto en los atributos de calidad de las distintas tácticas implementadas, se usó *artillery*. Mediante está herramienta, se creó un único escenario, que fue utilizado para todas las pruebas. Las fases de este escenario son:

- Una primera fase de *ramp up*, que dura 1 minuto, en la que el *arrival rate* va de 5 a 100 clientes por segundo
- Una fase de 2 minutos, en la que el *arrival rate* se mantiene en 100 clientes por segundo
- Una segunda fase de *ramp up* que dura 2 minutos, en la que el *arrival rate* sube de 100 clientes por segundo a 300

Cabe aclarar que que se agregaron 2 fases de 30 segundos al inicio y al final para “limpiar” los gráficos ya que grafana persistía el valor de la última medición.

En la siguiente imagen, se pueden observar las distintas fases del escenario creado

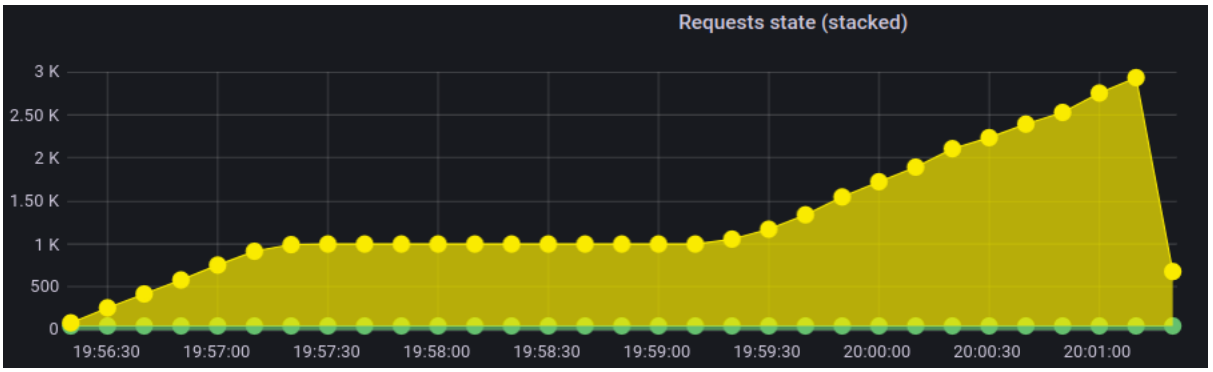


Tácticas utilizadas

Rate limiting

El *Rate Limiting* es una táctica que consiste en limitar la cantidad de *requests* que puede hacer un cliente. En nuestro servicio de *node* establecimos que un usuario puede realizar, como máximo 50 *requests* cada 15 segundos. Esto lo hicimos pensando en un caso real donde una dirección ip no debería mandar tantos requests en tan poco tiempo. (Asumiendo que la mayoría de nuestro tráfico no viene de una VPN u otro proxy)

En el siguiente gráfico podemos observar los estados de las respuestas obtenidas al hacer *requests* al *endpoint* de *ping* utilizando *Rate Limiting*. En verde se observan los requests completados correctamente, es decir con estado 200 y en amarillo las respuestas con estado 429, es decir *Too Many Requests* (demasiados requests), lo que quiere decir que el *rate limit* había actuado.

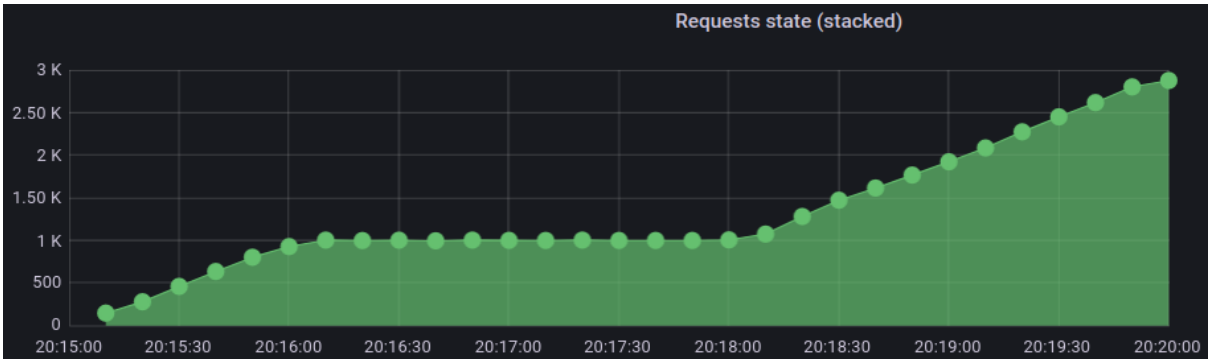


con rate limit

Como se puede ver, la mayoría de los requests no se completaron correctamente. Esto es entendible y se debe a que las pruebas se hicieron desde un mismo usuario (misma

dirección ip), y la frecuencia con la que llegaban los requests era ampliamente superior al *rate limit* fijado.

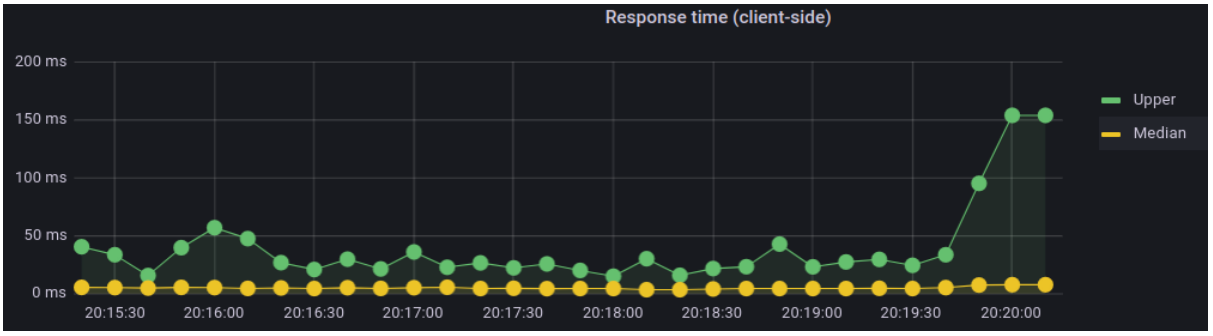
En el gráfico que se muestra a continuación, se puede observar el estado de las respuestas del *endpoint ping*, sin utilizar *rate limit*.



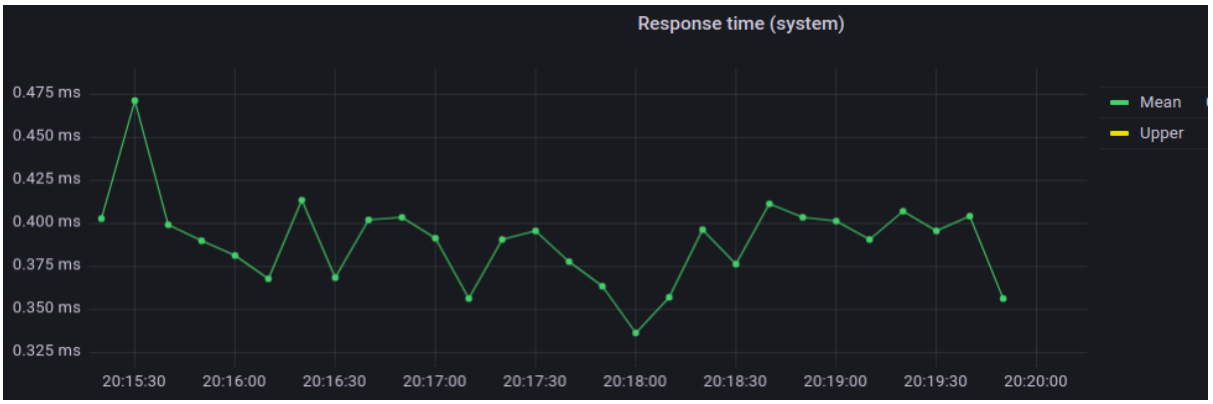
sin rate limit

Para este caso, todos los *requests* se pudieron completar correctamente. De esto se desprende que el *rate limit* estaba funcionando correctamente, por lo que ningún cliente podría realizar más de 50 *requests* cada 15 segundos. Esto mejora la **disponibilidad** ya que permite evitar ataques DDoS.

En los siguientes gráficos se pueden ver los tiempos de respuestas del *endpoint de ping*. Primero desde el punto de vista del cliente



y luego desde el punto de vista del servidor



Se puede observar que el tiempo que demora el servidor, que es siempre menor a un milisegundo, es mínimo respecto al tiempo de demora que percibe el cliente, que en

muchas fases llega a los 20 milisegundos. Esto se debe a que el del lado del servidor, el *ping* no realiza ningún procesamiento. Por está razón, podemos tomar al *ping* como la latencia de nuestro servicio en una red local.

Conclusión

Gracias a la táctica de rate limit, se evita que se hagan más peticiones por usuario que las que podría soportar el sistema. Esto mejora el atributo de calidad **disponibilidad** ya que ayuda a que el sistema no intente hacer un handle de más requests de los que puede soportar, es una gran forma de no saturar el sistema.

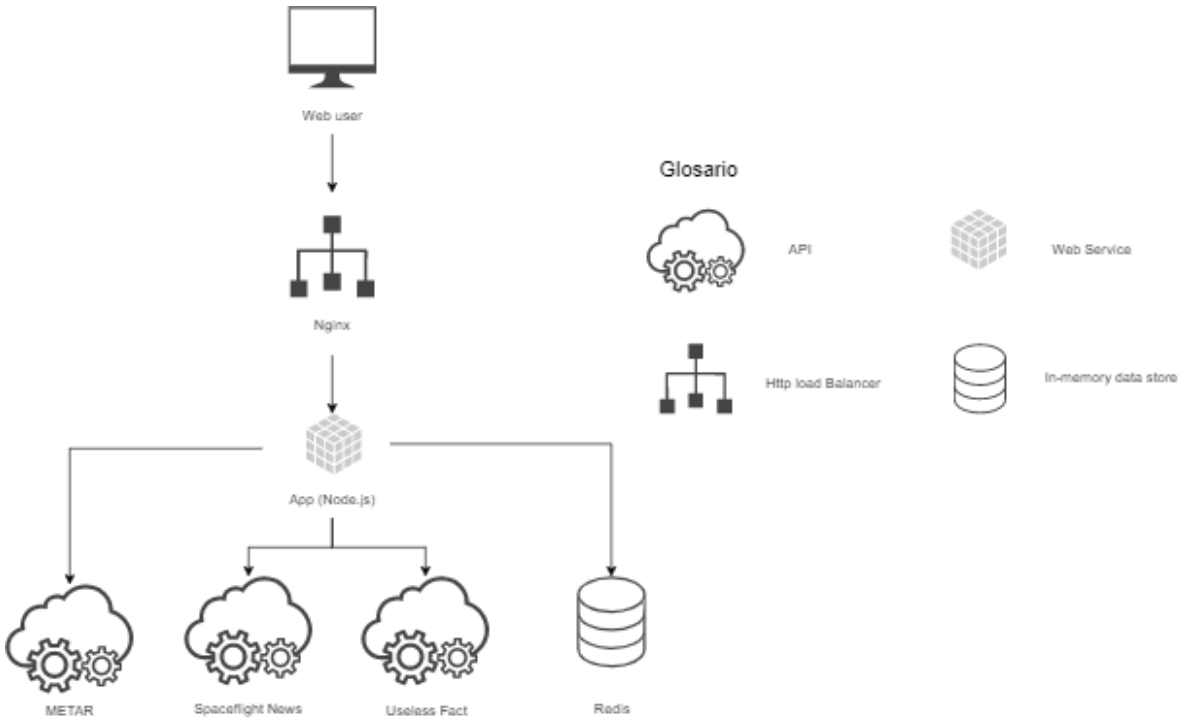
Caché

Una de las tácticas utilizadas para mejorar la **performance** fue el caché. Para esto, se dejaron guardados los datos en Redis, lo que permitía evitar consultas a una API remota. El único *endpoint* para el que se utilizó esta táctica fue el de *space news*. Esto se debe a que:

- El *endpoint* de *ping* no hace consultas a otra API remota, por lo que no habría nada para *cachear*
- El *endpoint* de *usless facts* devuelve los hechos de a uno, pero dado que se requería devolver uno distinto con cada llamado no podían quedar *cacheados*
- El *endpoint* de *Metar*, por la naturaleza de la información que devuelve se requiere que esté siempre actualizada. Esto se debe a que es información que usan los aviones para maniobrar, por lo que sería peligroso que utilicen información vieja.

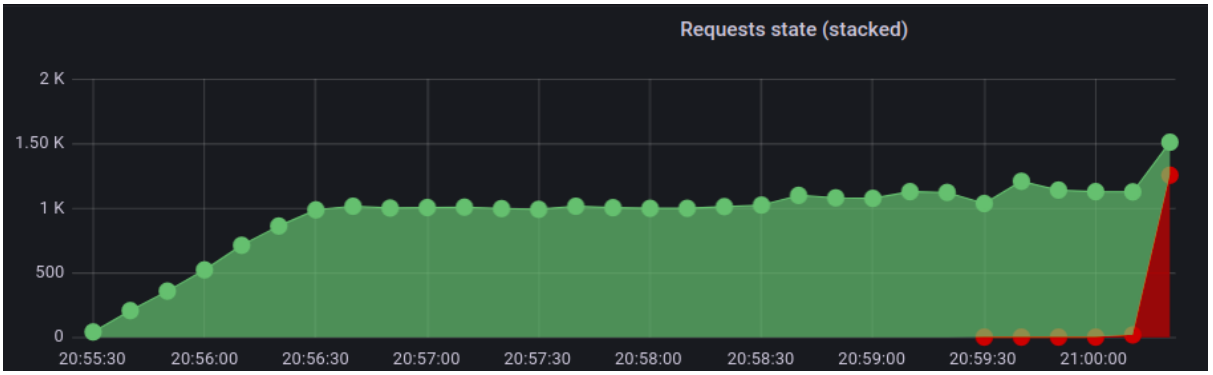
En el *endpoint* de *space_news*, en caso de que no haya notas en el caché, se guardan notas de 5 para devolverlas en la siguiente *request* recibida. Estas notas *cacheadas* se borran cada 5 segundos y no se borran al ser utilizadas ya que siguen siendo útiles para *requests* futuros.

Para esta técnica se puede observar el siguiente gráfico Components and Connectors:



Datos obtenido de las mediciones:

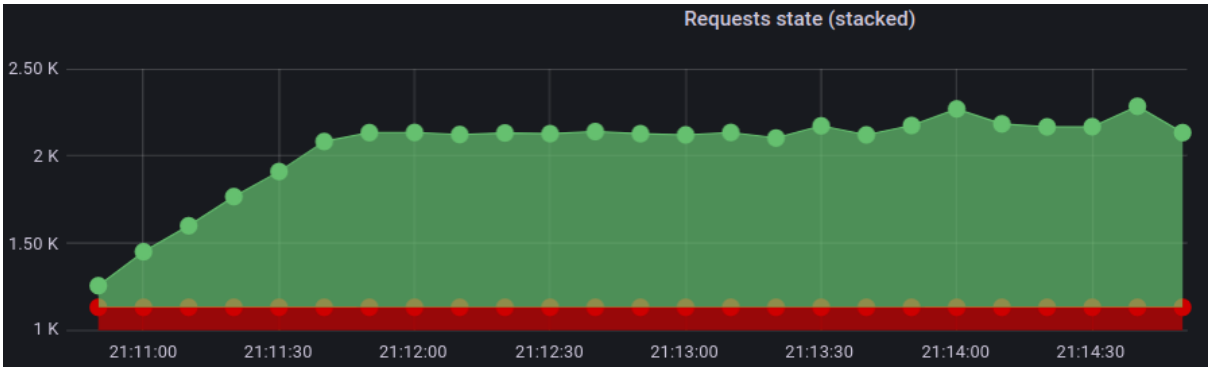
Primero, observamos el estado de las *responses* sin utilizar sin caché



request state space news sin cache

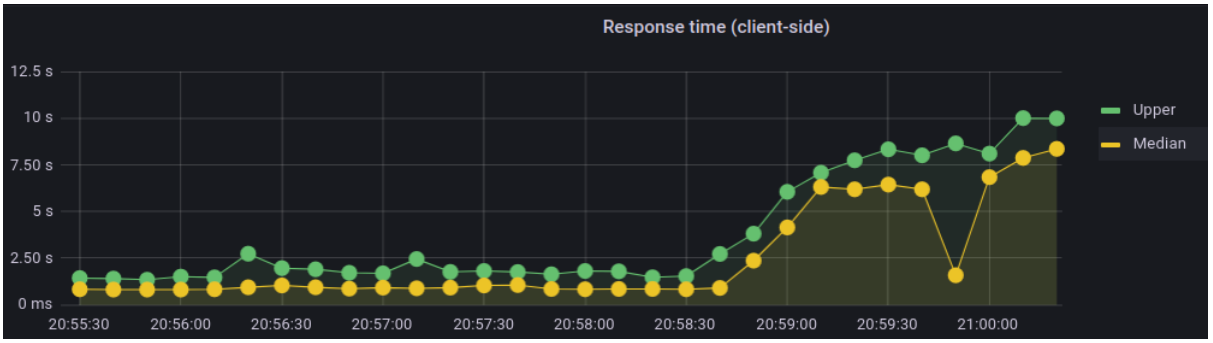
Como se puede ver, llegado al punto de stress, el servidor no pudo seguir manejando la cantidad de *requests*, por lo cual empieza a dar error.

Luego, mirando los estados de las *responses* utilizando *caché*

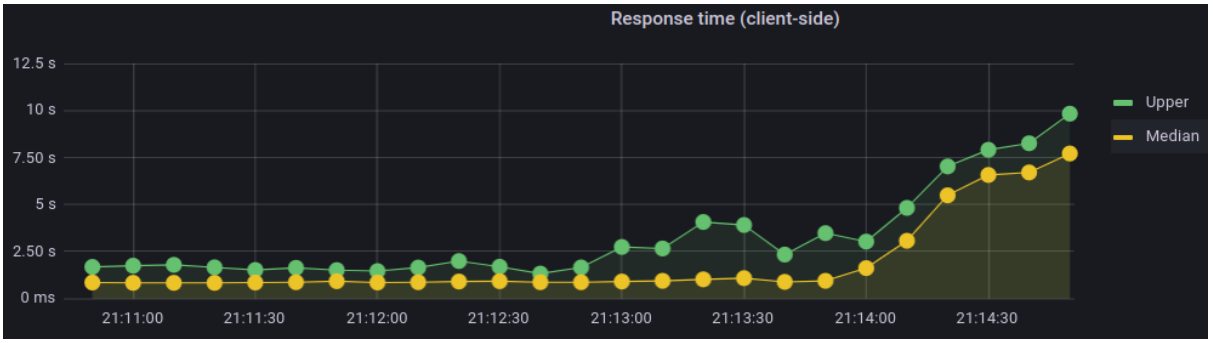


request state de space news con cache

Al utilizar caché, se observa que el servicio es capaz de manejar todas las *requests*. Cabe mencionar que las request que salen en rojo son los rastros de la corrida sin cache (graphite mantenía información de corridas pasadas), pero en este caso no se presentaron errores, por eso los errores se mantienen en un valor constante.



response time percibido por el cliente en space new sin cache



response time percibido por el cliente en space news con cache

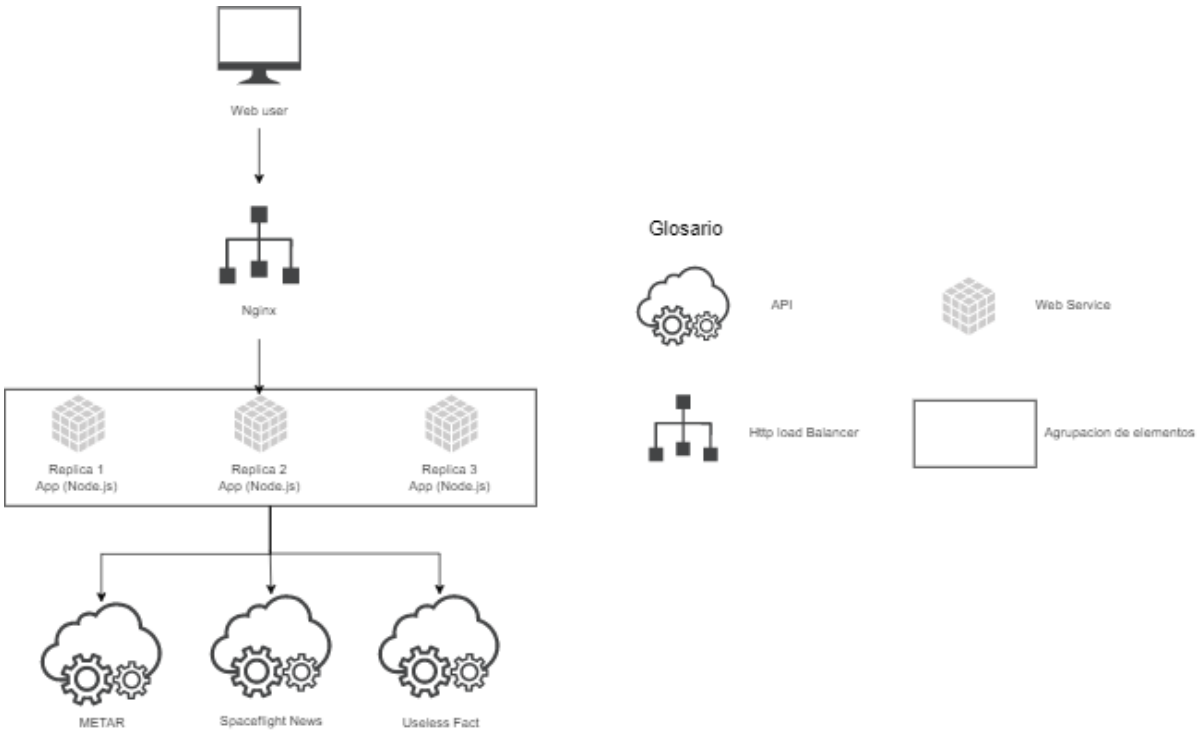
Nótese en cuanto a tiempos de respuesta, vimos que para el momento que empezó a fallar el sin cache tenía alrededor de 150 peticiones por segundo y con un tiempo de respuesta medio de 6,25 segundos. Mientras que con cache, a una carga similar los tiempos eran aproximadamente de 2.5s.

Conclusión

Gracias al cache se pudo reducir el tiempo de respuesta cuando se aumentaba considerablemente el número de requests. También fue gracias al cache que no se cometieron errores por timeout al aumentar las peticiones.
Por este motivo podemos decir que el cache aumenta la performance, en especial si aumentamos el tiempo de duración del cache. Además fue capaz de mejorar el atributo disponibilidad, ya que el sistema logró responder más request en comparación a cuando se usó sin cache.

Réplicas

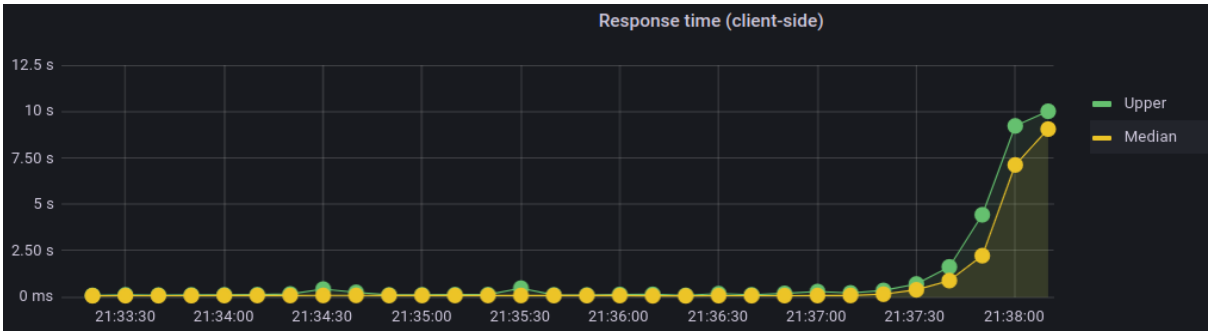
Está táctica se basa en tener distintas réplicas de un mismo nodo. En este caso, se usaron tres nodos. Esta táctica brinda fiabilidad ya que en el caso de que ocurra un fallo de uno de los nodos, todavía están los otros para encargarse de la carga. A continuación se puede ver el diagrama de *componentes & connectors,* para el caso en el que hay 3 réplicas.



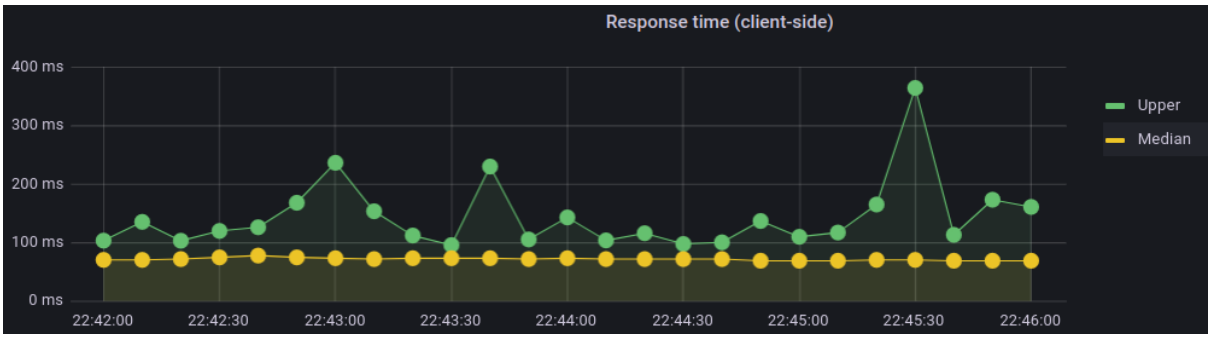
Components & connectors con 3 replicas

Como se puede ver, *nginx* actuará como un *load balancer* para el servicio.

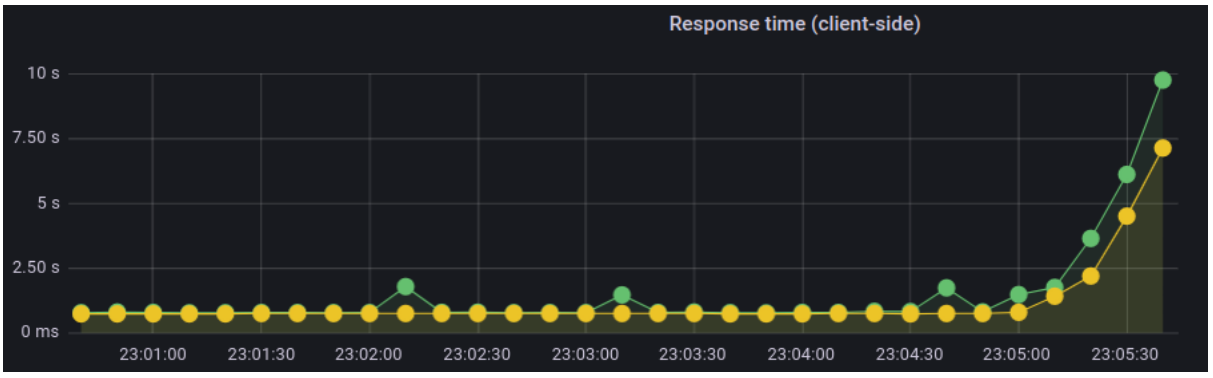
A continuación, compararemos los tiempos de demora percibidos por el cliente en el *endpoint* de *metar* y el de *facts* con y sin réplicas:



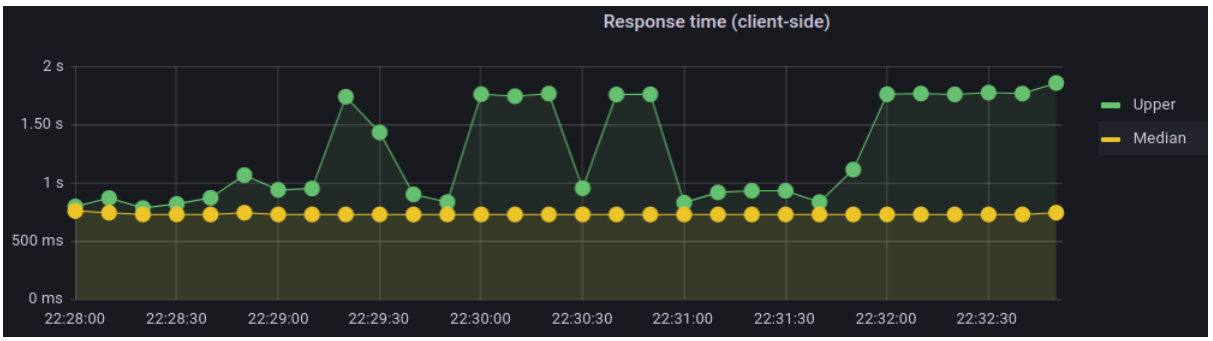
response time percibido por el cliente en metar sin réplica



response time percibido por el cliente en metar con réplica



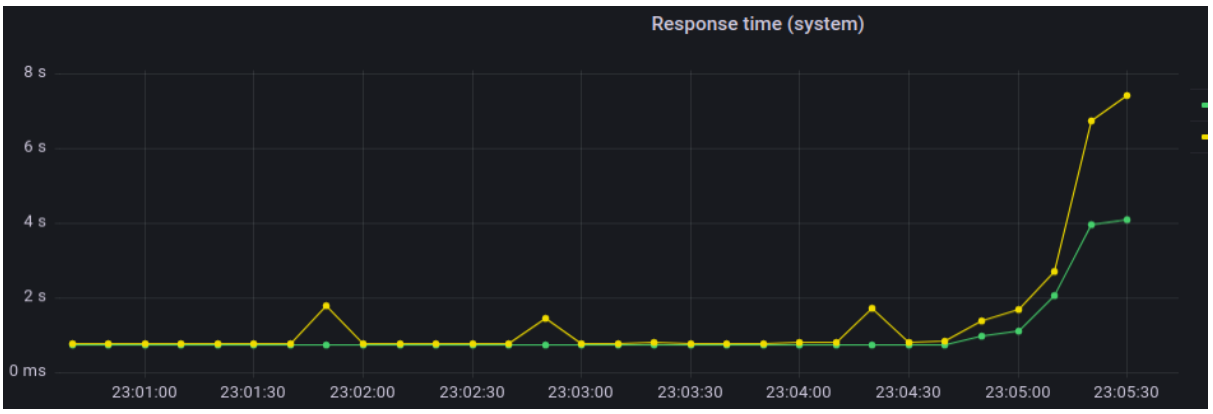
response time percibido por el cliente en facts con réplica



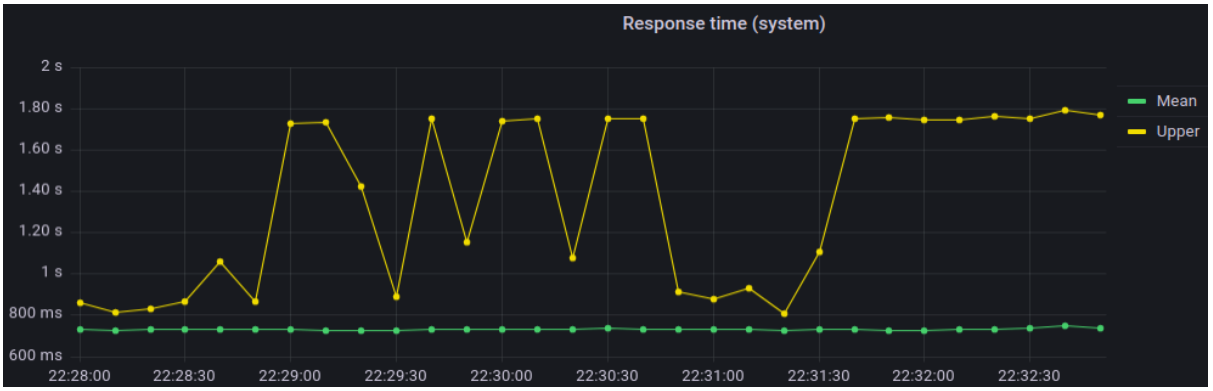
response time percibido por el cliente en facts con réplica

Se puede observar una muy clara diferencia entre los gráficos con y sin réplica: En el endpoint de metar sin utilizar réplicas, en las últimas etapas, cuando el *arrival rate* era más alto, el tiempo de respuesta llega a los 10 segundos. Sin embargo, al utilizar replicas nunca supero los 400 milisegundos. Algo similar ocurrió en el *endpoint* de *facts*, mientras que utilizando réplicas los tiempos de respuesta no superan los 2 segundos, al no utilizarlas llegaron a 10.

A continuación comparamos los tiempos de respuesta percibidos por nuestro servidor, con y sin réplicas para el *endpoint* de *facts:*



response time percibido por el servidor en facts sin réplica



response time percibido por el servidor en facts con réplica

Aquí se puede observar algo similar a lo que se puede ver en los gráficos anteriores: mientras que sin réplicas el tiempo de respuesta llega a un máximo de casi 8 segundos mientras que con las réplicas se tuvo picos de tan solo 1,8 segundos.

Conclusión

Analizando los gráficos anteriores, se puede concluir que la utilización de réplicas mejoró enormemente la **performance** cuando arriban aproximadamente entre 200 y 300 clientes por segundo. Tambien podemos observar que un atributo de calidad de este sistema es la **escalabilidad**, en particular la escalabilidad horizontal (scaling out) ya que se crearon nuevas unidades logicas para procesar los request. Debemos mencionar que esta fue una de las tacticas que mejores resultados dieron en este trabajo practico.