

75.73 Arquitectura de Software

Trabajo Práctico 1

Looney Tunes

Integrantes	Padrón	Email
Andres Zambrano	105500	azambrano@fi.uba.ar
Arian Jarmolinski	94727	ajarmolinski@fi.uba.ar
Mateo Cabrera Rodríguez	108118	mcabrerar@gmail.com
Adrián Romero	103371	adromero@fi.uba.ar

Introducción	3
Diagramas	4
Un nodo	4
Un nodo y caché	4
Tres nodos y load balancing	5
Pruebas realizadas	5
Ping	5
Escenarios creados	6
Estado de los requests	6
Tiempo de respuesta percibido	7
Uso de memoria y CPU	7
Ping - Rate limiting	8
Estado de los requests	8
Uso de memoria y CPU	8
Spaceflight News	8
Tiempo de respuesta percibido	9
Uso de memoria y CPU	11
Spaceflight News - Cache	11
Tiempo de respuesta percibido	12
Uso de memoria y CPU	13
Dictionary	13
Estado de los requests	13
Estado de los requests - Ramp	14
Tiempo de respuesta percibido	15
Quote	15
Estado de los requests	16
Estado de los requests - Ramp	16
Tiempo de respuesta percibido	17
Prueba con tres réplicas	18
Uso de memoria y CPU	18
Tiempo de respuesta percibido	19
Conclusiones	20

Introducción

El presente informe tiene como objetivo principal exponer una implementación de una API para luego realizar diferentes mediciones que nos permitan evaluar sus atributos de calidad.

Para llevar a cabo el desarrollo de la API, se utilizan las siguientes tecnologías:

- Node.js con Express
- Docker y Docker Compose
- Nginx, como proxy y load balancer
- Redis, como caché
- Artillery, como generador de carga

También se utilizan las siguientes herramientas para realizar mediciones y visualización de las mismas: cAdvisor, StatsD, Graphite y Grafana.

La API desarrollada se compone de cuatro servicios principales que se someterán a distintos escenarios de carga. Estos servicios son:

- Un servicio de ping como healthcheck
- Un servicio de diccionario
- Un servicio de noticias sobre actividad espacial
- Un servicio de citas aleatorias

En el trabajo práctico exploramos distintas optimizaciones, como el uso de Redis como caché, la replicación del servicio, la utilización de Nginx como load balancer y la implementación de rate limiting para el consumo de la API.

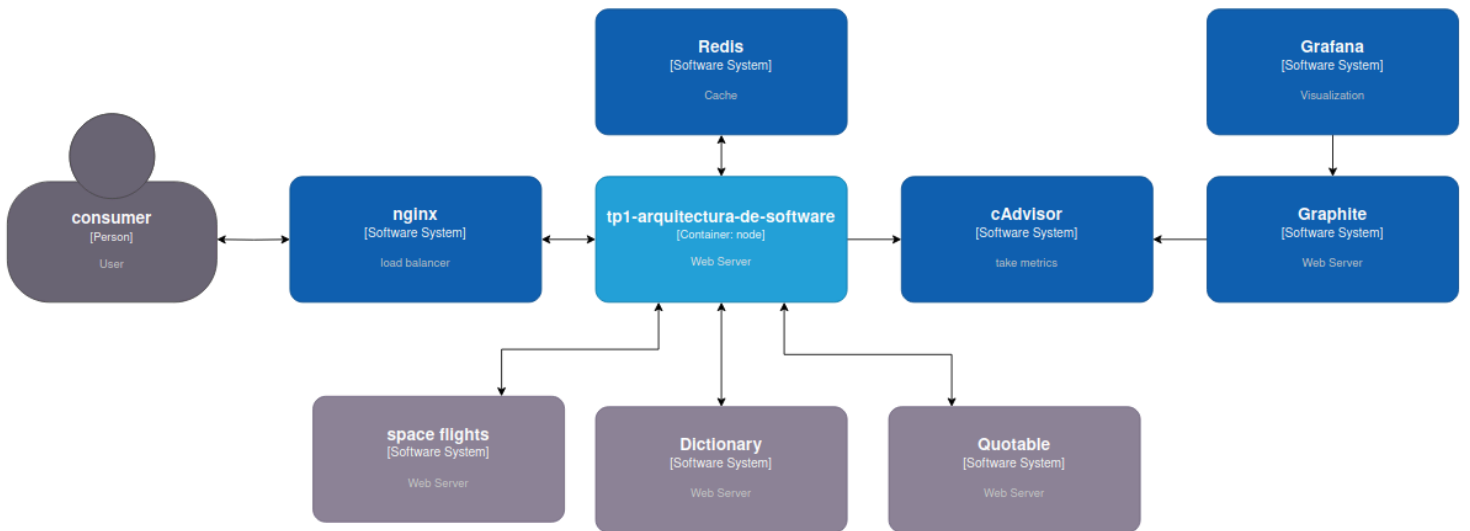
El informe incluirá también capturas de pantalla del dashboard de métricas para cada caso analizado, así como una vista Components & Connectors que represente la arquitectura del sistema para los distintos casos estudiados.

Diagramas

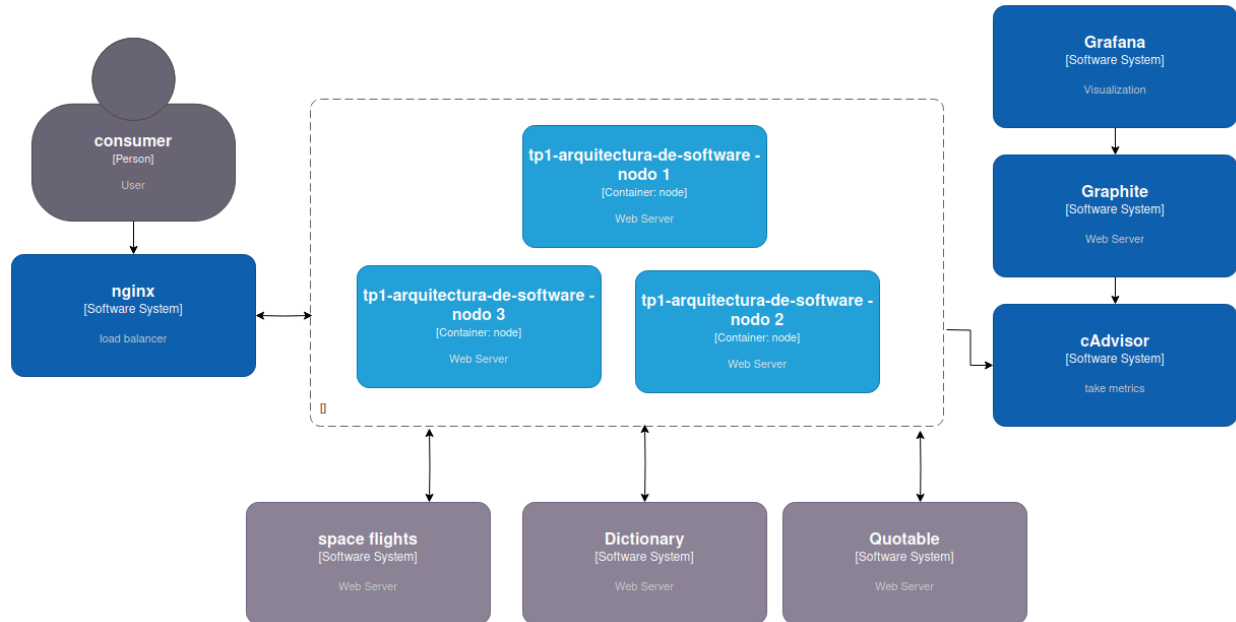
A continuación mostramos los diagramas components and connectors para las situaciones que evaluamos:

Un nodo

Un nodo y caché



Tres nodos y load balancing



Pruebas realizadas

A lo largo del trabajo práctico, vamos a exponer los distintos endpoints que conforman nuestra API a diferentes intensidades de requests enviados para evaluar y comparar su performance, teniendo en cuenta las siguientes métricas:

- Cantidad de requests respondidos por segundo
- Estado HTTP de los requests respondidos
- Tiempo de respuesta percibido por el usuario
- Tiempo de respuesta percibido por el servidor
- Uso de CPU & Memoria

Ping

Este endpoint (/ping) simplemente devuelve un string con el valor 'pong'. Debido a la simpleza de este endpoint, esperamos que el procesamiento de memoria y el uso de CPU sea bajo. También el tiempo de respuesta percibido por el usuario y por el servidor debería resultar bajo.

El caso inicial bajo el cual evaluaremos este endpoint estará conformado por las siguientes fases:

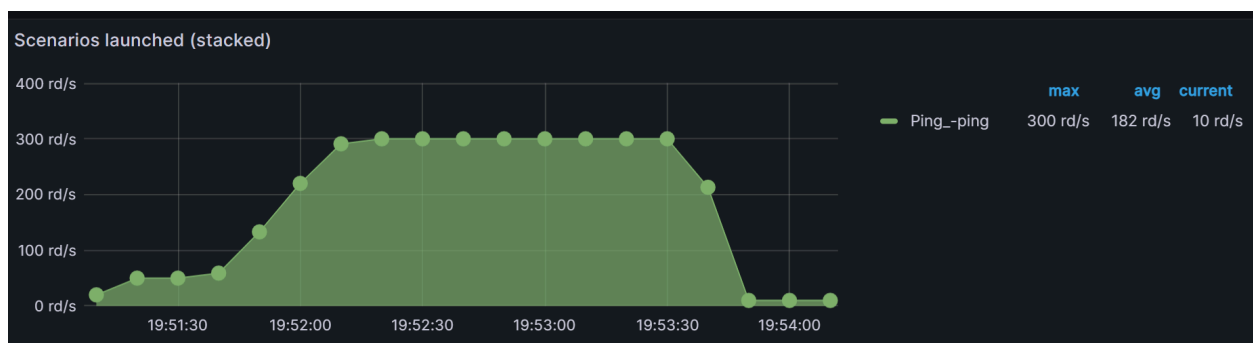
1. **Calentamiento:** por una duración de 30 segundos se envían requests a una tasa de 5 requests por segundo.
2. **Ramp up:** por una duración de 30 segundos se incrementa gradualmente la tasa de envíos a 30 requests por segundo.
3. **Plain:** por una duración de 90 segundos se envían requests a una tasa de 30 requests por segundo.

4. **End:** por una duración de 15 segundos se envían requests a una tasa de 1 request por segundo.

Escenarios creados

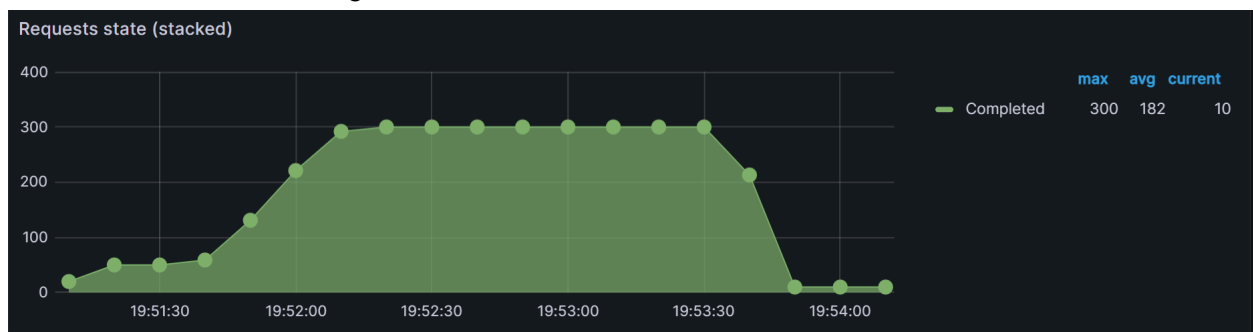
En este gráfico podemos ver en el eje Y la cantidad de requests recibidos por el servidor en los últimos 10 segundos, por este motivo vemos este eje escalado en un factor de 10 respecto de la cantidad requests enviados por segundo.

Podemos apreciar claramente las distintas fases del escenario propuesto: la fase de calentamiento donde se inician los requests a tasa de 5 por segundo, seguido del ramp gradual en el que se eleva la cantidad de requests por segundo a 30 y luego una meseta en la etapa plain, finalmente la tasa cae a 1 request por segundo en la última fase.



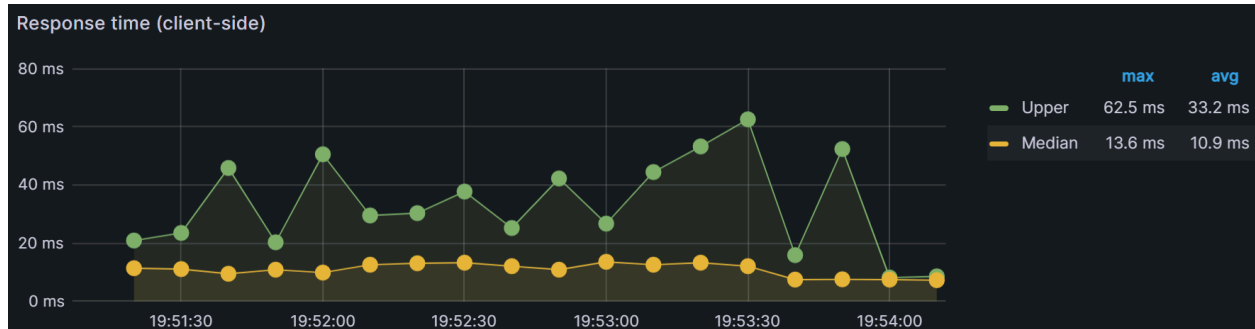
Estado de los requests

Respecto del estado de los requests, vemos que todos los requests fueron completados correctamente, con el código de estado HTTP 200.



Tiempo de respuesta percibido

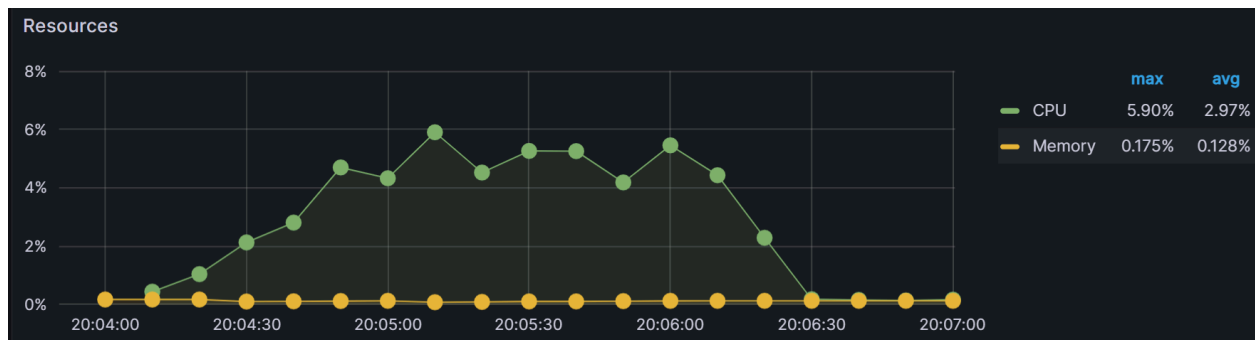
Esperamos que el tiempo de respuesta percibido por el usuario sea bajo, pues este endpoint no realiza procesamiento, ni debe esperar la respuesta de una api remota, como sí sucederá en los siguientes endpoints.



Efectivamente, el tiempo de respuesta que observamos es muy bajo, con una mediana máxima de 13.6 ms.

Uso de memoria y CPU

Tal como esperábamos el uso de memoria y de CPU para este endpoint es muy bajo. Además, el uso de CPU parece seguir la forma que vimos en los primeros gráficos, lo que indica que, lógicamente a mayor cantidad de requests recibidos por el servidor, el uso de CPU fue mayor. Por otro lado, este endpoint no hace uso de memoria, por lo cual se mantuvo bajo.

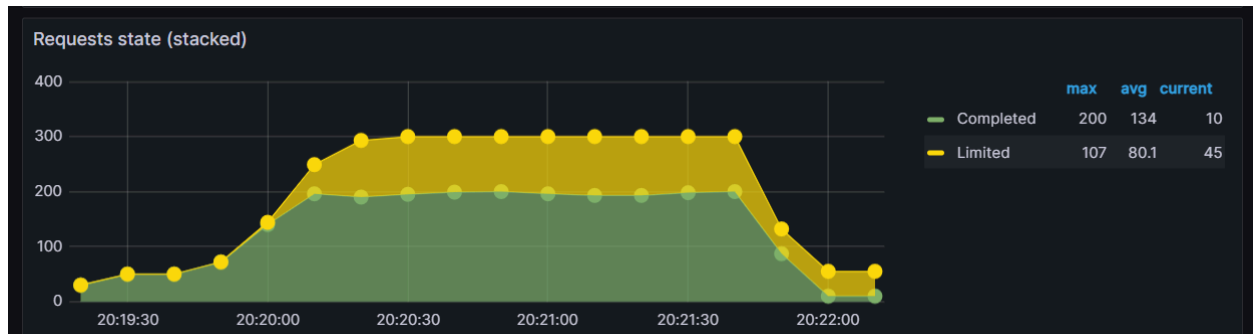


Ping - Rate limiting

Ahora vamos a testear el comportamiento del endpoint bajo el caso anterior pero incorporando un rate limiter que permita como tasa máxima 20 requests por segundo.

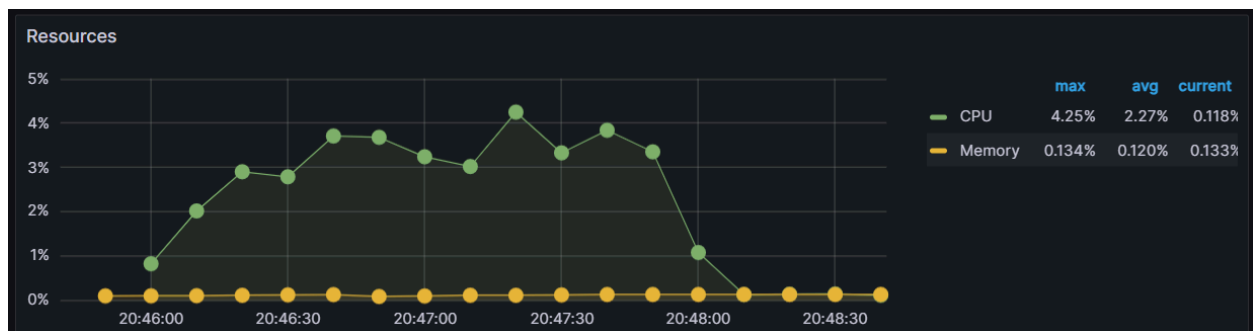
Estado de los requests

Respecto del estado de los requests, lo que esperamos ver es que hayan requests rechazados, debido a que en la etapa plain se alcanza una tasa de 30 requests por segundo. Podemos ver, como esperábamos, que un tercio de los requests son rechazados en la etapa plain.



Uso de memoria y CPU

Como esperábamos, el uso del CPU es menor en este caso. El máximo uso es de 4.25%, cuando sin rate limiting era de 6%. Esto es posible que se deba a que la cantidad de requests procesados es menor debido a que se rechazan requests.



sts.

Spaceflight News

En el endpoint (/spaceflight_news), devolveremos solo los títulos de las 5 últimas noticias sobre actividad espacial, obtenidas desde la [Spaceflight News API](#).

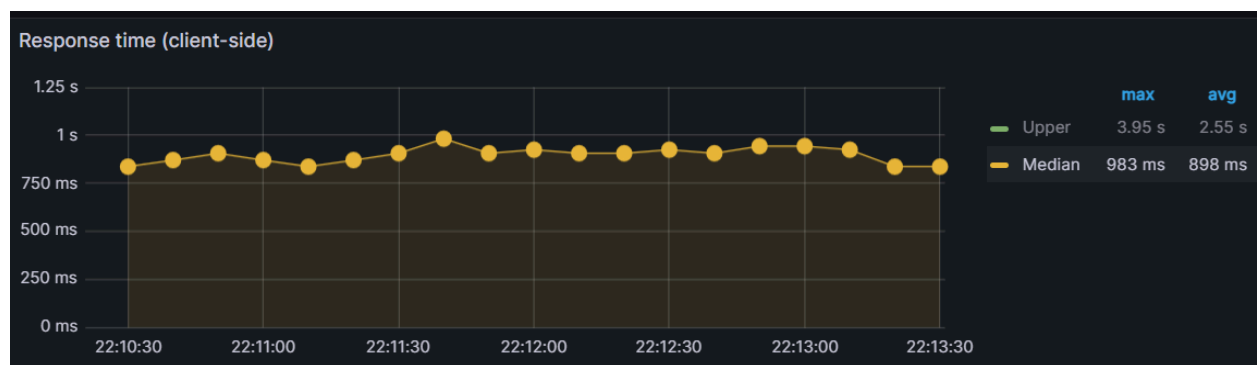
En este endpoint esperamos ver un mayor tiempo de respuesta percibido por el usuario y por el servidor, pues este endpoint consulta una API externa, por lo que se debe esperar a la respuesta de esta API externa antes de continuar con el procesamiento y realizar una

devolución al usuario. Además también esperamos ver un mayor uso de CPU y memoria debido a que se realiza una manipulación y formateo de los datos recibidos de la API externa.

Corriendo el caso base para este endpoint los gráficos de Escenarios Creados & Estado de los requests son iguales a los primeros. Por este motivo, presentamos los resultados para el tiempo de respuesta percibido y el uso de memoria:

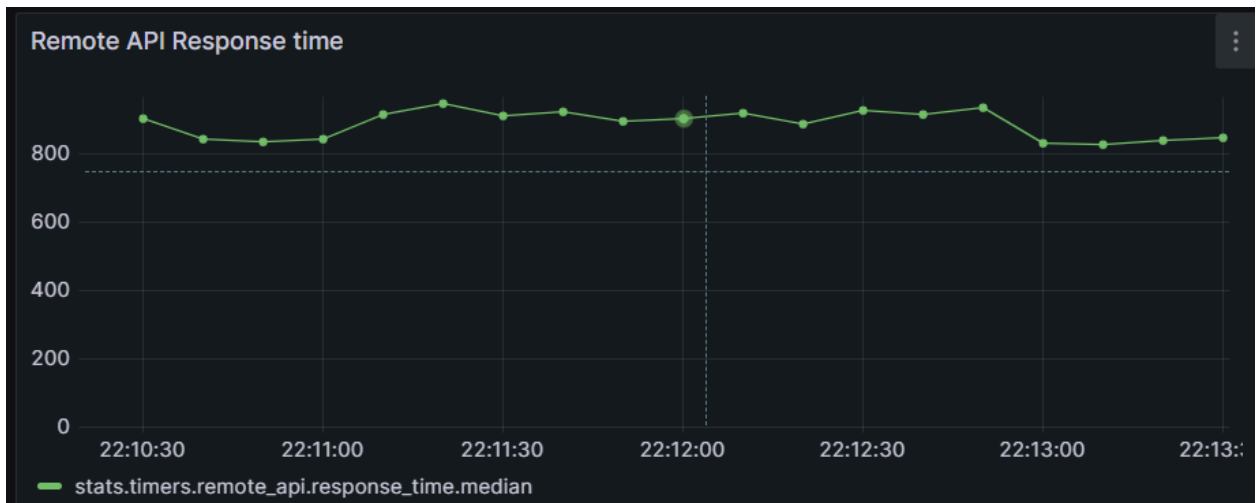
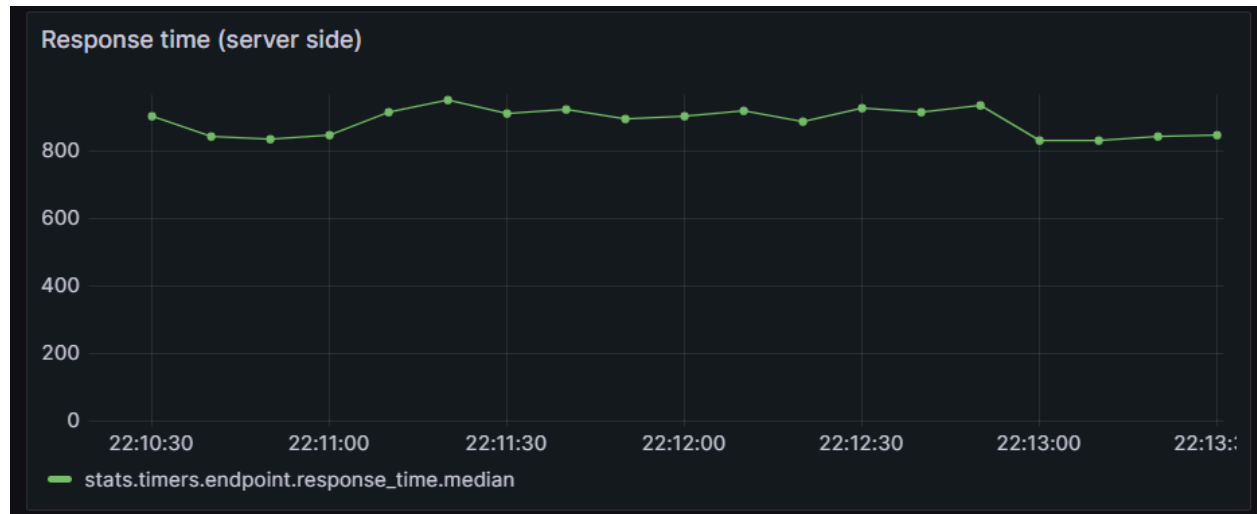
Tiempo de respuesta percibido

Si comparamos el tiempo de respuesta percibido por el cliente con el de /ping, vemos que es aproximadamente 100 veces mayor, esto tiene sentido pues este endpoint requiere de mayor procesamiento.

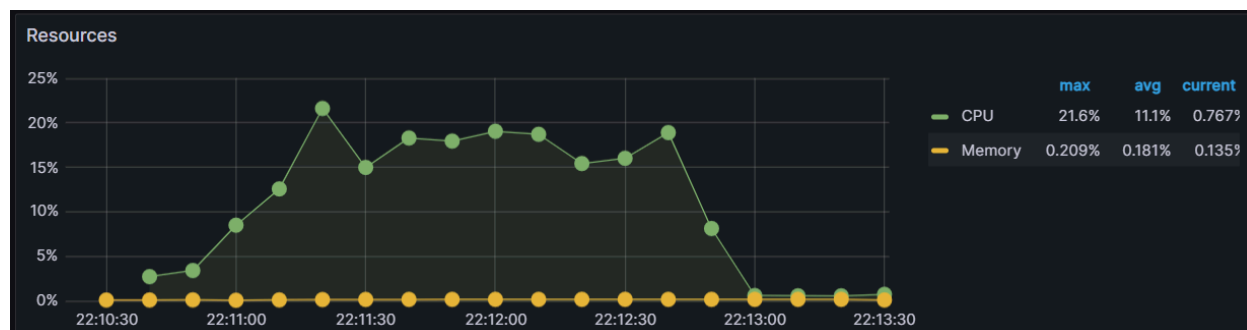


Además, si comparamos el tiempo de respuesta medido desde el lado del servidor con el tiempo de respuesta de la api remota, vemos que son muy similares. Esto es de esperarse, la mayor parte del tiempo de procesamiento está en esperar la respuesta de la api remota.

También podemos observar que el tiempo de respuesta percibido por el cliente y por el servidor no tienen mucha diferencia (aproximadamente 800ms, aunque el percibido por el servidor es ligeramente menor), esto parecería indicar que el tiempo en el que viaja la información desde el servidor al cliente es muy reducido. Posiblemente debido a que tanto cliente como servidor están en el mismo host local.



Uso de memoria y CPU



Claramente, comparando con el caso de /ping, el máximo uso de memoria es muchísimo mayor (21% vs 6%) lo cual es esperable debido a que este endpoint realiza un formateo de los datos provenientes de la API remota antes de responderle al cliente.

Spaceflight News - Cache

Debido a que este endpoint nos trae las últimas noticias sobre actividad espacial y no esperamos que estas sean actualizadas con gran frecuencia, decidimos realizar un cacheado de los datos.

Esperamos que esto disminuya el tiempo de respuesta percibido por el usuario, así como también el tiempo de respuesta percibido por el servidor, al no tener que ir a buscar los datos a la API remota y procesarlos, sino que los datos estarán cacheados en Redis.

Usar la caché debería también reducir el uso de CPU, pues los datos de respuesta no serán procesados por el servidor sino devueltos tal cual están cacheados.

Para estas pruebas decidimos cachear la respuesta de la API de manera lazy, es decir, que recién cuando el cliente le pega a nuestra endpoint existe la posibilidad de que nos traigamos los datos de la caché. Para este endpoint en particular tiene sentido también realizar un cacheado activo, para ya tener información disponible cuando el cliente quiere pegarle a nuestro endpoint.

Sobre el tiempo de vida de los datos cacheados decidimos ponerles 5 segundos, pues no encontramos documentación en la API de Spaceflight News acerca de con qué frecuencia se actualizan las últimas noticias.

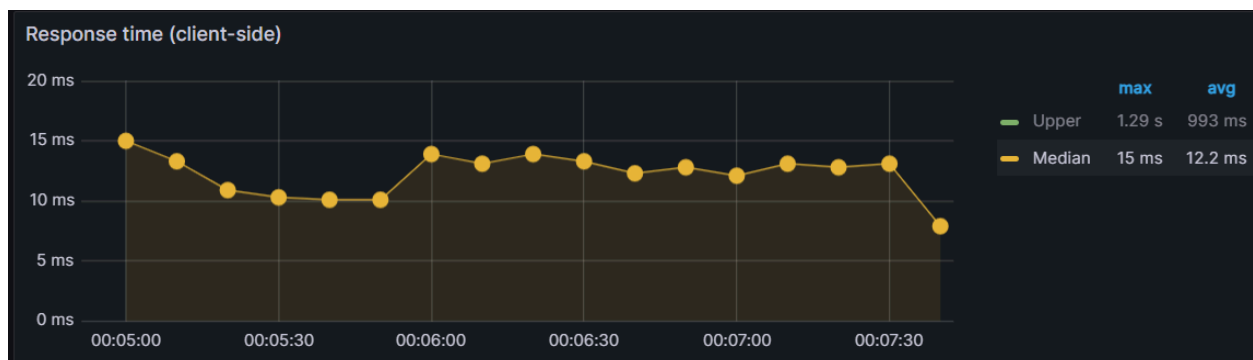
También, notamos que no sería necesario realizar un vaciado de la caché cuando un cliente sea respondido con el objeto almacenado en la caché. Esto es porque sería correcto que dos clientes distintos sean respondidos con el mismo objeto almacenado en la caché, dado que ambos deberían recibir las mismas últimas noticias.

Tiempo de respuesta percibido

Claramente el tiempo de respuesta de la API remota es el mismo que en el caso de no usar caché (aproximadamente 800ms), pues la API externa sigue tardando lo mismo en respondernos cada vez que le solicitamos los datos. Lo que cambia es la frecuencia con la que le solicitamos los datos a esta API.

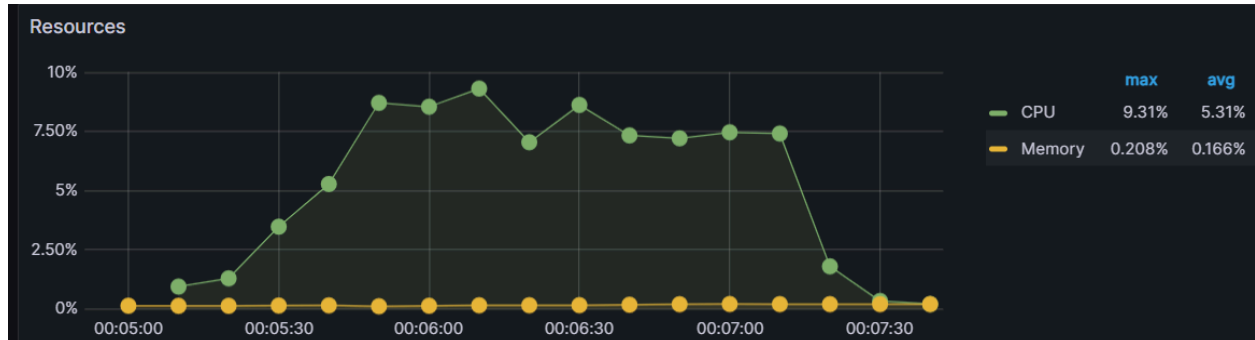


Por otro lado, los tiempos de respuesta percibidos tanto por cliente como por el servidor disminuyen drásticamente, pues para un gran porcentaje de las solicitudes, ya no nos quedamos esperando la respuesta de la API remota, sino que respondemos con el contenido de la caché. Esto se puede ver en los siguientes gráficos:



Uso de memoria y CPU

Los recursos utilizados también se redujeron, como esperábamos, pues el procesamiento es menor al utilizar los datos de la caché al no tener que procesar los datos provenientes de la API remota.



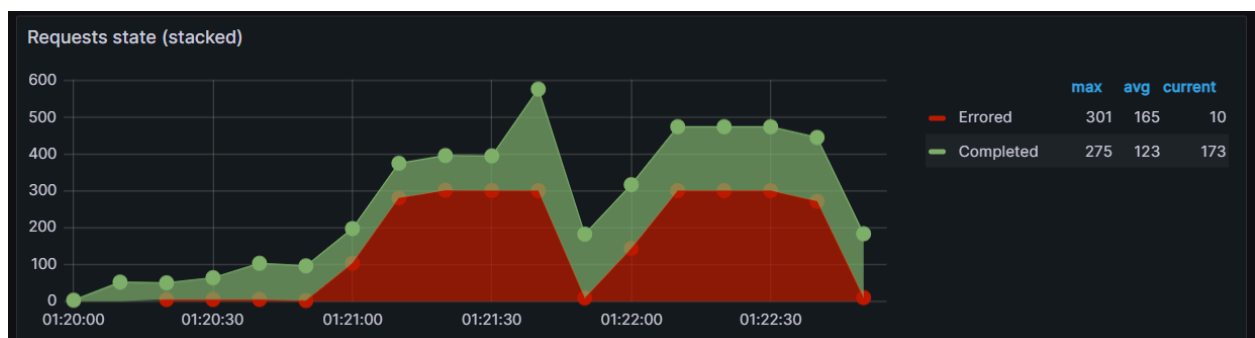
Dictionary

En el endpoint: `/dictionary?word=<word>`, devolveremos la fonética y los significados de palabras en inglés, consultando a la [Free Dictionary API](#). En términos de intensidad de procesamiento es similar al endpoint `spaceflight news`. Para este caso no vamos a cachear los datos provenientes del endpoint externo, porque consideramos que es poco probable que dos clientes consulten la misma palabra del diccionario.

Además para realizar las pruebas sobre este endpoint utilizaremos un archivo csv denominado `words.csv` en el cual incluimos un listado de palabras en inglés que se utilizan como parametro del endpoint al armar las requests con artillery en el archivo `dictionary.yaml`

Estado de los requests

Al realizar esta prueba y revisar el estado de las responses pudimos ver que muchos de los requests que se realizaron fallaron con status code 429 debido a que [Free Dictionary API](#) nos retornó `'TOO_MANY_REQUESTS'`:

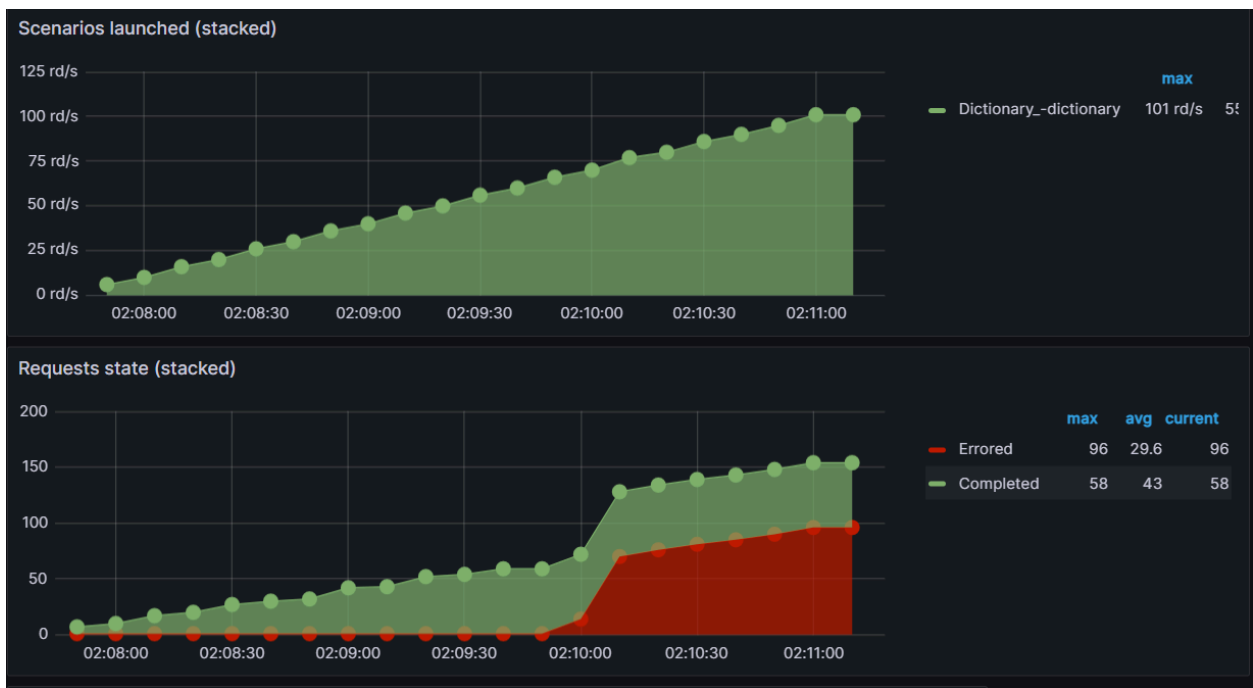


Estado de los requests - Ramp

Dado que pudimos ver requests rechazados a una tasa de 5 requests por segundo (tasa que utilizamos como warm-up), proponemos el siguiente escenario con la esperanza de que nos permita ver cuál es el rate limit:

- Cada 20 segundos incrementar la tasa en 1 requests por segundo, empezando en 1 y terminando en 10.

Realizando esta prueba pudimos ver que los errores recibidos se disparan cuando la cantidad de requests enviados por segundo alcanza el valor de 5.



Tiempo de respuesta percibido

Considerado el caso inicial, podemos ver que nuevamente la mayor parte del tiempo de la respuesta del servidor se le dedica a esperar la respuesta de la API remota. También pudimos observar que a partir de que empezamos a recibir requests fallidos, dejamos de recibir estas métricas que incorporamos con hot-shots.

Por otro lado vemos también que el tiempo de respuesta de esta API remota es considerablemente menor que el de Spaceflights News (500ms contra 900ms).



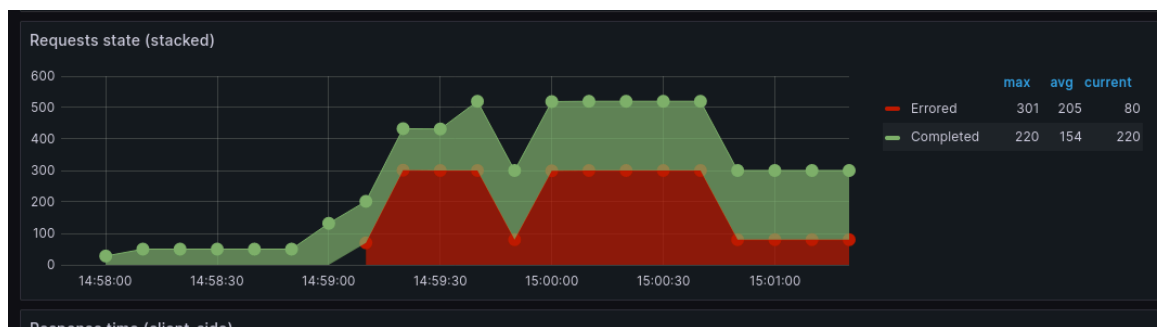
Quote

En el endpoint /quote devolveremos 1 cita famosa al azar, junto con su autor por cada invocación, tomada de [Quotable](#). Debe evitarse entregar la misma cita cada vez (salvo que la repita la API remota).

Para este caso tampoco utilizaremos caché para almacenar los datos del endpoint externo. Al ser frases al azar, el cache no mejorará los tiempos de respuesta.

Estado de los requests

Similar a los resultados obtenidos con la API de dictionary, observamos muchas requests que fallaron al superar cierto umbral.

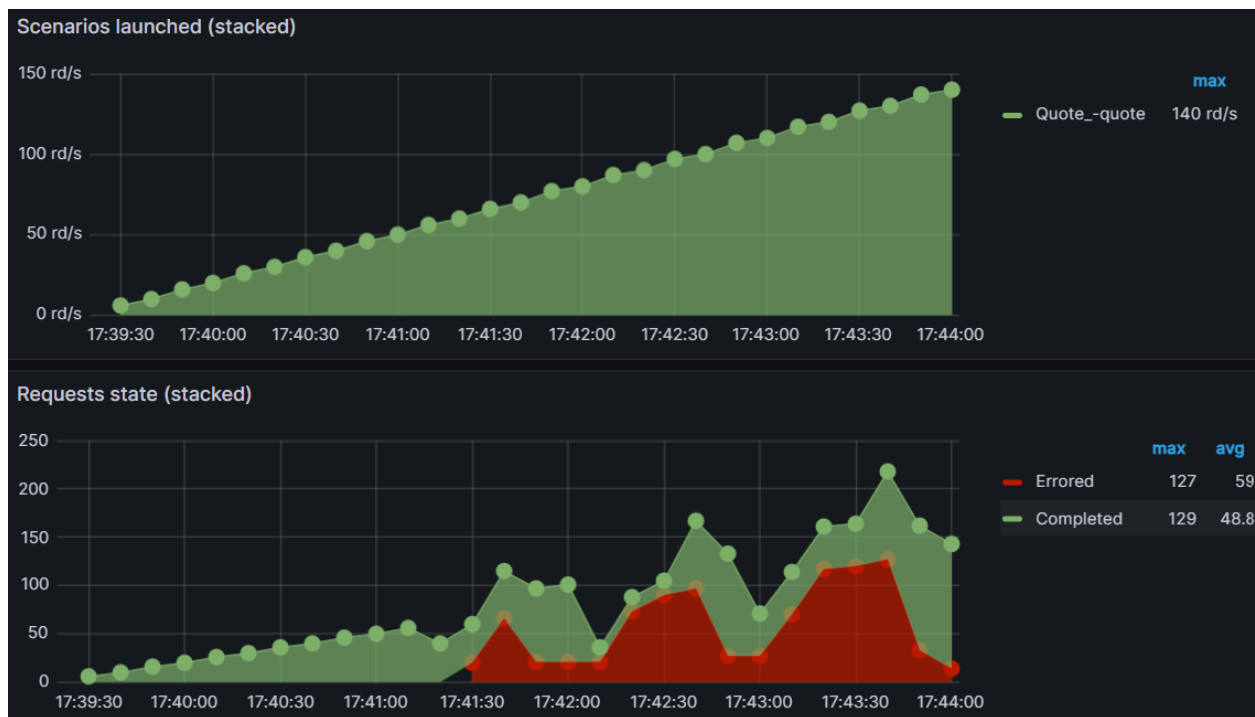


Estado de los requests - Ramp

Debido a que los errores los empezamos a observar en la fase de Ramp-up. Vamos a utilizar una táctica similar a lo que hicimos con Dictionary. Para conocer el rate-limit, proponemos el siguiente escenario:

- Cada 20 segundos incrementar la tasa en 1 requests por segundo, empezando en 1 y terminando en 14.

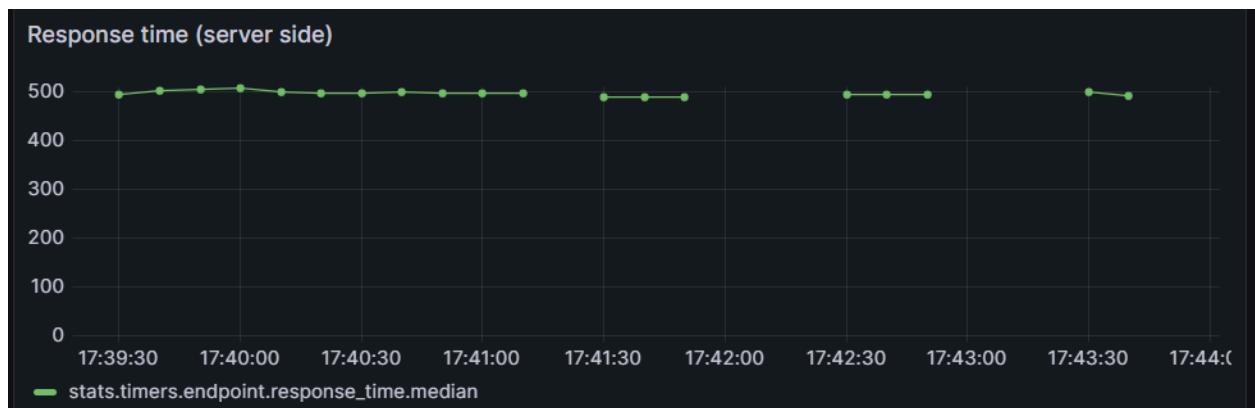
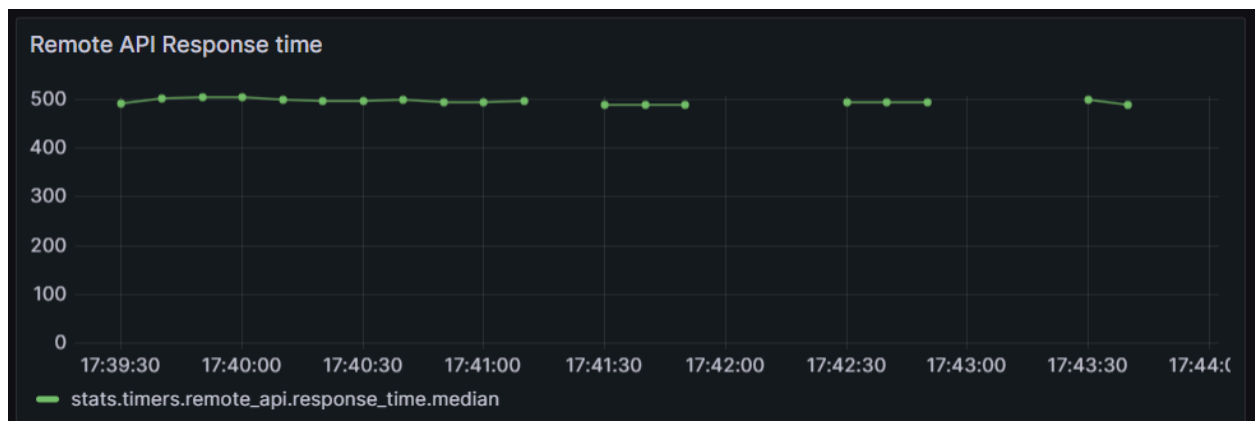
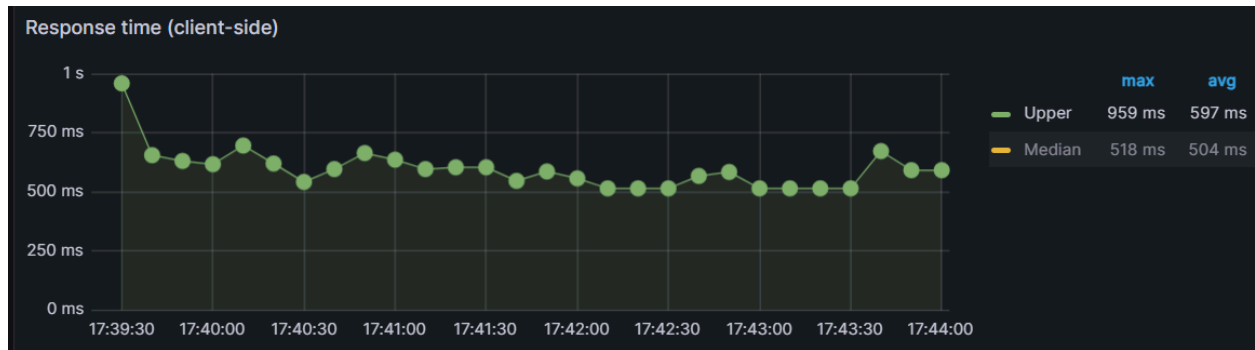
Realizando esta prueba pudimos ver que los errores recibidos se disparan cuando la cantidad de requests enviados por segundo se acerca a 5 o 6.



Tiempo de respuesta percibido

Nuevamente, por la lógica que sigue esta API, la mayor parte del tiempo de la respuesta del servidor, pasa esperando una respuesta de la API externa.

Los tiempos de respuesta de este endpoint son muy similares a los tiempos de respuesta de el de Dictionary, al igual que los tiempos de respuesta de sus respectivas APIS remotas.



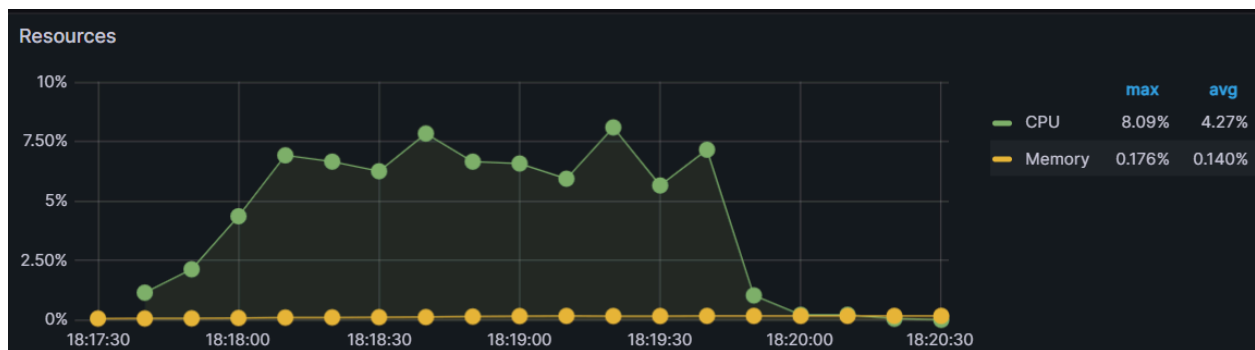
Prueba con tres réplicas

Para la siguiente prueba decidimos comparar el uso de recursos utilizando el endpoint `/spaceflight_news` sin caché, pero esta vez, distribuyendo la carga equitativamente en 3 nodos. Esto lo conseguimos utilizando nginx como loadbalancer. Esperamos que el uso de los recursos sea menor en cada nodo que al estar procesando todas las requests en un solo nodo.

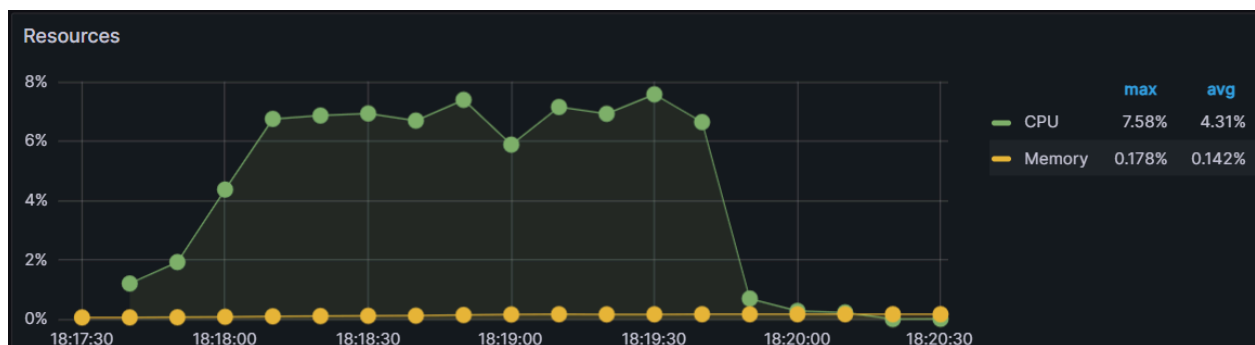
Los resultados que obtuvimos fueron los siguientes:

Uso de memoria y CPU

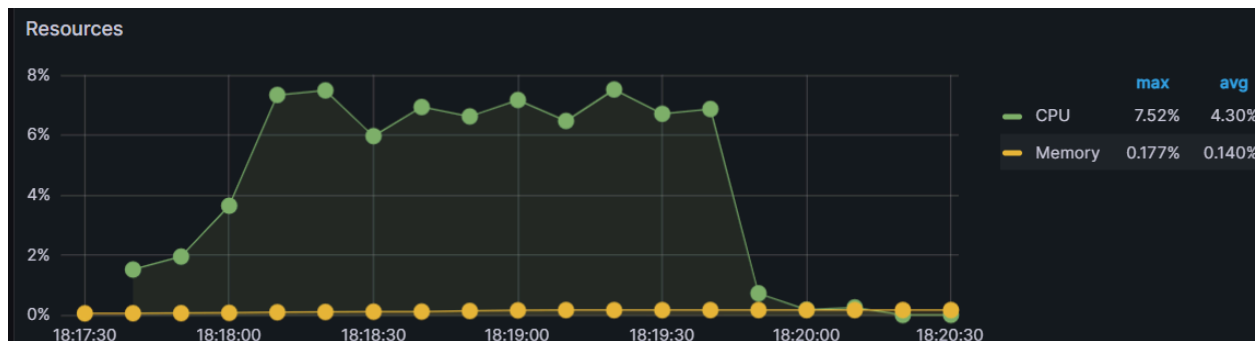
Nodo 1:



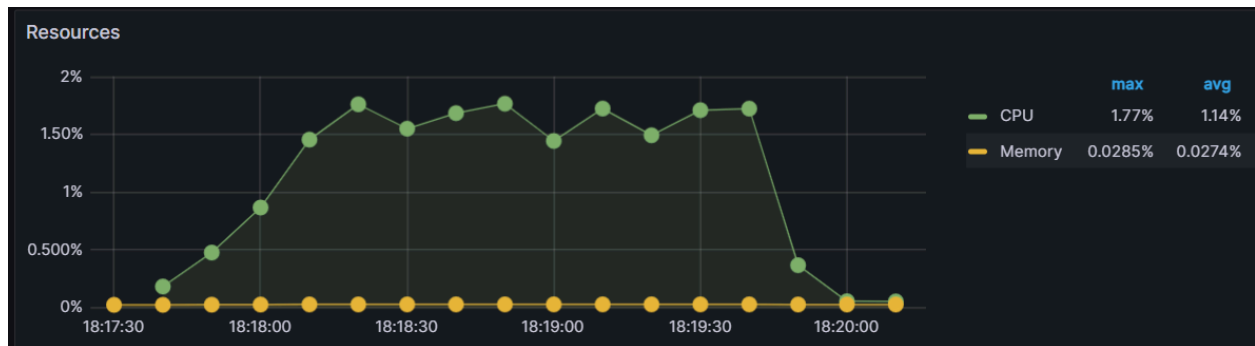
Nodo 2:



Nodo 3:



Nginx:

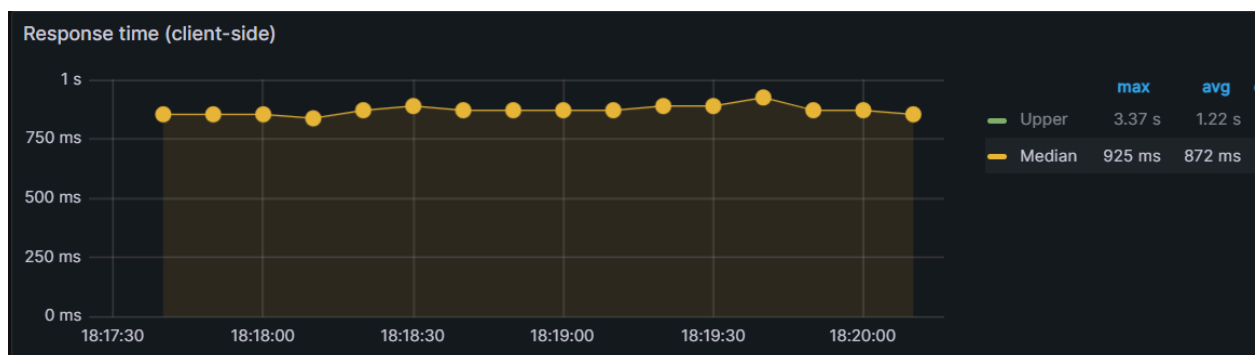


Podemos ver que el % de uso de CPU en los tres nodos es muy similar, esto indica que efectivamente el load balancer realiza un round robin en el cual todos los nodos tienen el mismo peso y por lo tanto la carga se disminuye de manera equitativa.

Además comparando con el caso de 1 solo nodo, el uso de CPU de estos 3 nodos fue menor (7,5% vs 21%) es decir casi un tercio.

Tiempo de respuesta percibido

El tiempo de respuesta percibido por el usuario parece disminuir levemente comparando con la contraparte de 1 solo nodo (980ms vs 925ms)



Conclusiones

Durante el desarrollo de este trabajo práctico, hemos tenido la oportunidad de utilizar una gran variedad de herramientas que permiten la observabilidad y medición de nuestros componentes, pudiendo comparar cómo responde un servidor ante diferentes situaciones de carga.

Logramos un entendimiento de las tecnologías utilizadas y llevamos a cabo un análisis exhaustivo para los diferentes escenarios de carga, examinando algunos atributos de calidad de nuestra API como el tiempo de respuesta y el uso de recursos.

Además pudimos explorar posibles optimizaciones como el uso de caché, rate limiting, la replicación y balanceo de carga, entendiendo cómo funcionan y cuándo aplicarlas aporta valor. Esto nos ha proporcionado una comprensión más profunda sobre la escalabilidad y la performance de las aplicaciones.