Arquitectura del Software

TP1

Berrotaran, Pablo. *Padrón Nro. 98.446*

pberrotaran@fi.uba.ar

Caliz Blanco, Alejo Martin Ezequiel. *Padrón Nro. 102.842*

acaliz@fi.uba.ar

Loscalzo, Melina. *Padrón Nro. 106.571*

mloscalzo@fi.uba.ar

Pernin, Alejandro. *Padrón Nro. 92.216*

apernin@fi.uba.ar

3 de Octubre de 2024

Índice

1. Introducción

El objetivo de este trabajo práctico es analizar de forma empírica como distintas estrategias de la arquitectura del software afectan en los atributos de calidad. Para tal fin dispondremos de un servidor web en NodeJS que expondrá una API con diversos endpoints, los cuales serán empleados para estresar el sistema.

Los resultados de algunos de los endpoints son provistos de forma directa por nuestro servidor y otros requieren de una consulta a una API externa. Este esquema, en conjunto con las estrategias nos permitirá apreciar diferencias entre el cliente final y nuestro servidor, así como el contraste entre llamadas que se ejecutan de forma puramente local, y aquellas que requiren de integración con endpoints externos.

Para la recopilación de métricas se emplearán diversas tecnologías como

- cAvdisor
- StatsD
- Graphite
- Graphana

Así como también Artillery para las pruebas de stress y otras tecnologías como NginX y Redis para las diversas estrategias.

1.1. Endpoints

El sistema expone los siguientes endpoints

- /ping: Simplemente para probar si el sistema responde, el servidor devolverá 'pong'
- /dictionary: El endpoint recibe una palabra, efectúa una consulta en una api externa que posee información de diccionario en Inglés, se procesa dicha respuesta y devuelve al cliente final la fonética y significado de la palabra
- /spaceflight_news: A través de otra api externa obtiene las últimas noticias referentes a la actividad espacial y devuelve las últimas 5.
- /quote: También mediante una API externa, se devuelve una cita aleatoria
- /random_fact: Similar al endpoint de citas, pero ésta devuelve un hecho / dato-curioso aleatorio.

1.2. Estrategias

Para el desarrollo de este trabajo se implementarán y medirán las siguientes estrategias

- Base: Un caso base ausente de estrategia particular, para servir de referencia
- Replicación: Tendremos múltiples servidores detrás de un load-balancer.
- Caché: Se implementará una caché para evitar procesar consultas duplicadas
- Rate Limiting: Se implementará un límite a la frecuencia de consultas

2. Ejecución

Para la ejecución de todos los sistemas necesarios se hace uso intensivo de **docker compose** tanto por su facilidad y versatilidad, como también su flexiblidad.

Existen las siguientes configuraciones posibles:

- Scaling: Define cuantas réplicas se emplearan para los servidores de la API
- Cache: Define si se habilita o no el uso de la caché
- Rate Limit: Define el rate-limit a emplear

que se pueden emplear de la siguiente forma

```
USE_CACHE=no RATE_LIMIT=10r/s docker compose up -d --scale node=3
```

Por defecto no se usa caché, ni existe rate limit (en rigor existe pero con un valor muy alto) y sólo se emplea una réplica

2.1. Artillery

Para las pruebas de stress mediante Artillery se definieron diversas entradas de make, por ejemplo

```
make artillery-run-ping
```

2.2. Grafana

Para la visualización de las métricas recolectadas se emplea un servidor de Grafana con un dashboard configurado.

Las métricas recolectadas son

- Requests Realizados: Muestra la cantidad de peticiones realizadas en el caso especifico que estemos ejecutando.
- Estados Requests: Los estados, exitosos o fallidos, de los requests hechos.
- Tiempo de Respuesta: El tiempo que llevó efectuar dicho request, esto desde la perspectiva del cliente.
- Tiempo de Respuesta del Servidor: El tiempo de respuesta desde la perspectiva del servidor, esto és, cuanto le lleva a él satisfacer el pedido.
- Recursos: Los recursos, CPU y memoria, utilizados por las replicas.

3. Pruebas

En esta sección se exhibirán las pruebas realizadas para los distintos endpoints empleando diversas estrategias

Para la ejercitación de los distintos endpoints empleando Artillery se implementó una curva de demanda de distintas etapas:

- Etapa Preliminar: Una etapa de demanda baja
- Etapa Incremental: Una etapa donde la demanda crece linealmente
- Etapa Constante: Se mantiene el punto de demanda máxima por un tiempo dado
- Etapa Decreciente: Se reduce la demanda hasta completar la prueba

3.1. Ping

3.1.1. Base

Este caso se puede ejecutar de la siguiente manera

```
docker compose up -d
make artillery-run-ping
```

Los resultados obtenidos fueron

En el el primer gráfico se observa la curva de demanda definida, asimismo cómo el estado de los requests (en este caso todos exitosos) copia a la perfeccion la demanda. Los tiempos de respuesta se mantienen bajos a lo largo de toda la prueba, consistente con la sencillez del endpoint

El tiempo de respuesta del servidor se mantiene despreciable dado que el mismo no realiza

ninguna consulta externa y de forma análoga se aprecia el escaso uso de recursos

3.1.2. Caché

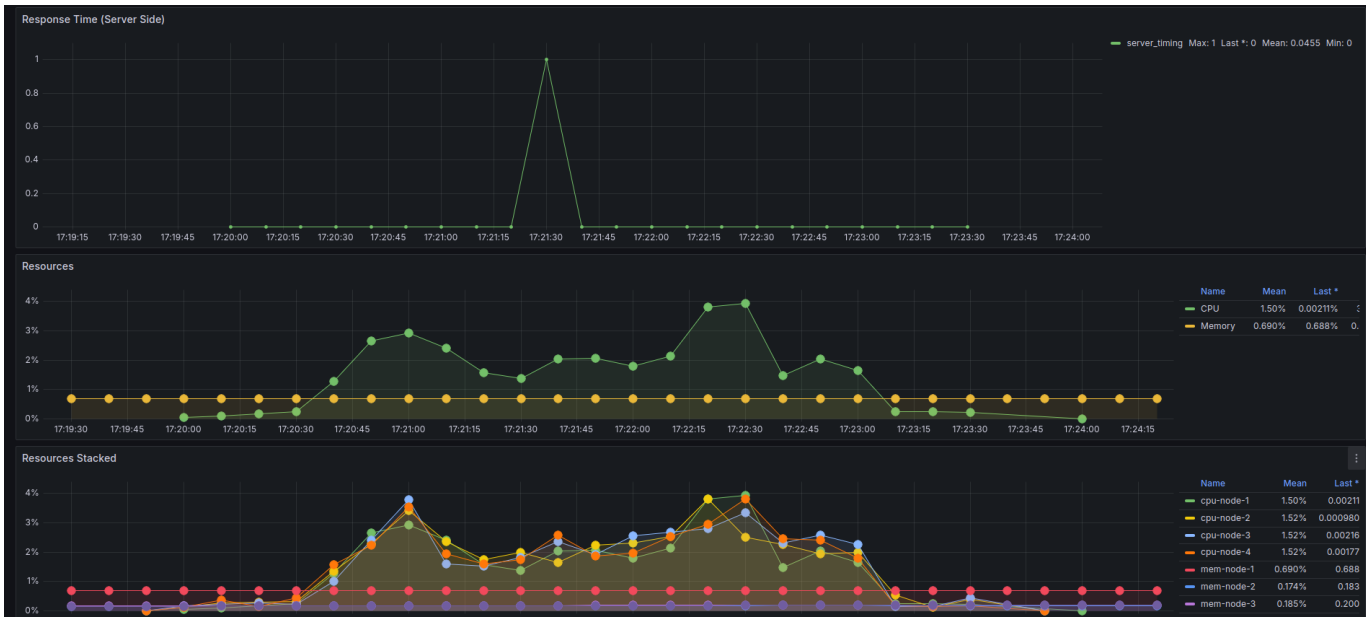
No se implementó caché en este endpoint dado que lo consideramos un sin-sentido, no existe variación en las respuestas ni consulta externa que amerite dicha implementación.

3.1.3. Replicación

Para este caso se emplearon cuatro replicas

```
docker compose up -d --scale node=4
make artillery-run-ping
```

obteniendo los siguientes resultados

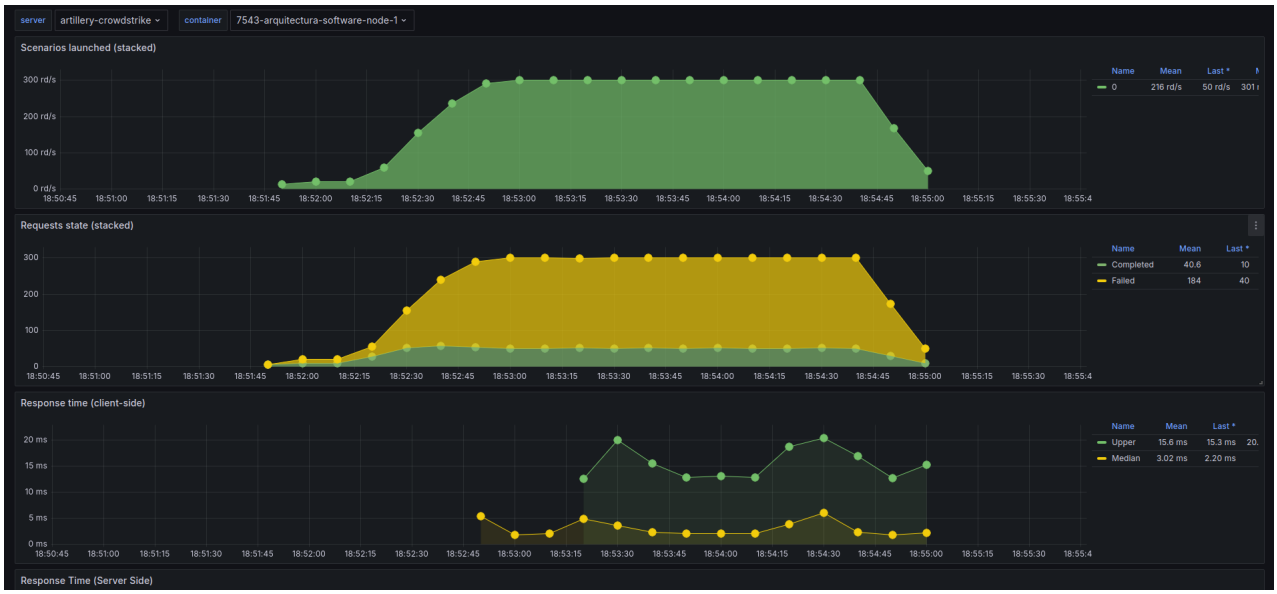


En esta oportunidad se puede ver que los resultados son muy similares al caso base, con la diferencia que en tiempo de respuesta del servidor se vé un valor fuera de la constante, una anomalía de un valor despreciable y que todas las réplicas se estresan de forma similar, con lo cual se puede afirmar que el balanceador de carga está distribuyendo la carga de forma pareja.

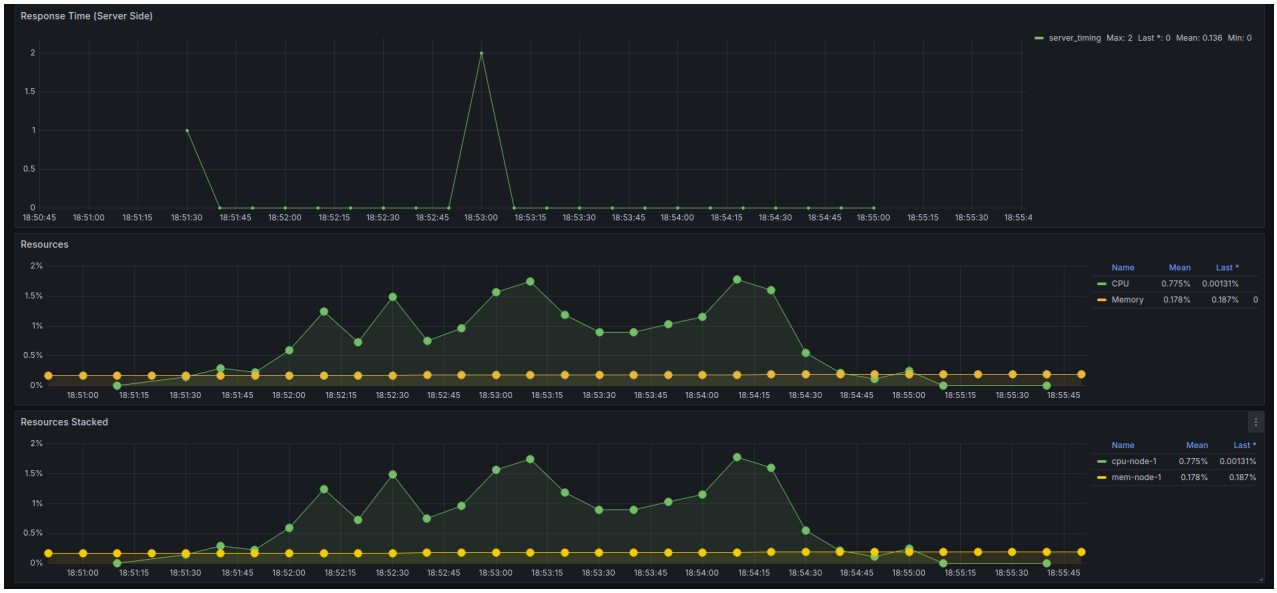
3.1.4. Rate Limiting

En los casos anteriores se pudo observar como el servidor responde de manera satisfactoria y en un tiempo despreciable, aquí veremos los efectos de limitar la cantidad de requests que se pueden efectuar. Se implementó un limite de 10 requests por segundo

```
RATE_LIMIT=10r/s docker compose up -d
make artillery-run-ping
```



Se puede distinguir como a comparación con el caso base, casi de inmediato comienzan a visualizarse fallas en los requests efectuados, manteniendose la cantidad de requests exitosos constante durante la casi totalidad de toda la prueba y lógicamente por debajo de la tasa definida.



No se observan diferencias sustanciales en relación a los tiempos de respuesta o recursos empleados

3.2. Dictionary

3.2.1. Base

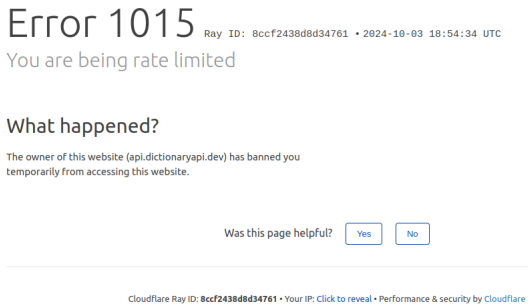
El caso base se ejecuta del a siguiente manera

```
docker compose up -d
make artillery-run-dict
```

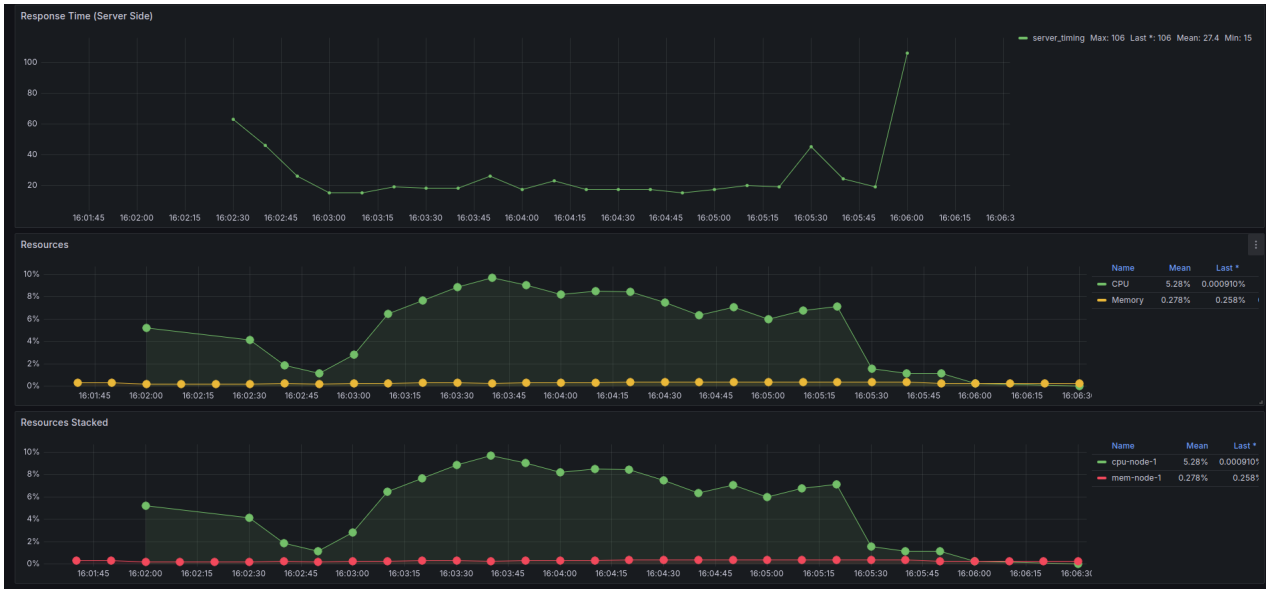
Para este conjunto de pruebas se emplea un wordlist de palabras en inglés el cuál se recorre de manera secuencial para realizar los requests

La primera observación que se puede hacer del caso base es que este escenario es muy similar al del ping con rate-limiting. Se puede apreciar como la gran mayoria de los requests no son exitosos y los que sí lo son se mantienen constantes con un valor muy bajo. Así como tambien se puede apreciar un aumento en los tiempos de respuesta

Esto se debe a que estamos siendo limitados por la API remota



Se observa también un tiempo de repuesta del servidor ligeramente mayor (al ping base), lo cuál es constitente siendo éste un endpoint que requiere un poco más de trabajo, hecho que también se aprecia en el uso de recursos



3.2.2. Caché

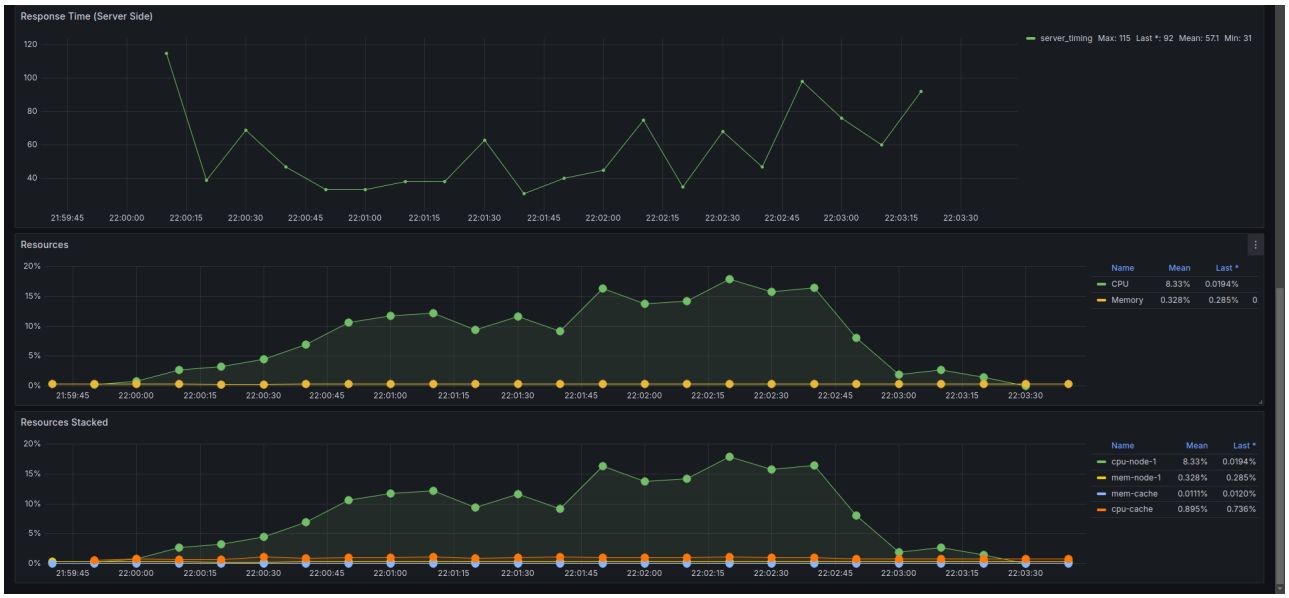
```
USE_CACHE=yes docker compose up -d
make artillery-run-dict
```

Partiendo de los resultados del caso base, podemos anticipar que los resultados empleando una caché serán similares, con la distinción que la caché misma prácticamente no será empleada por la baja cantidad de requests exitosos que habrán.

Inesperadamente encontramos que mejora la tasa de fallas, luego de analizar la ejecución encontramos a que esto se debe de una mala configuración en la prueba y donde se puede llegar a repetir las palabras solicitadas. Esto sería dado que los agentes de artillery no comparten sus estados [1].

```
node-1  | Searching for word:  aa
node-1  | Searching for word:  aa
node-1  | Searching for word:  aa
node-1  | Searching for word:  aa
node-1  | Searching for word:  aa
node-1  | Searching for word:  aardvark
node-1  | Searching for word:  aardvark
node-1  | Searching for word:  aardvark
node-1  | Searching for word:  aardvark
node-1  | Searching for word:  aardvark
node-1  | Searching for word:  aargh
node-1  | Searching for word:  aargh
node-1  | Searching for word:  aargh
node-1  | Searching for word:  aargh
node-1  | Searching for word:  aargh
```

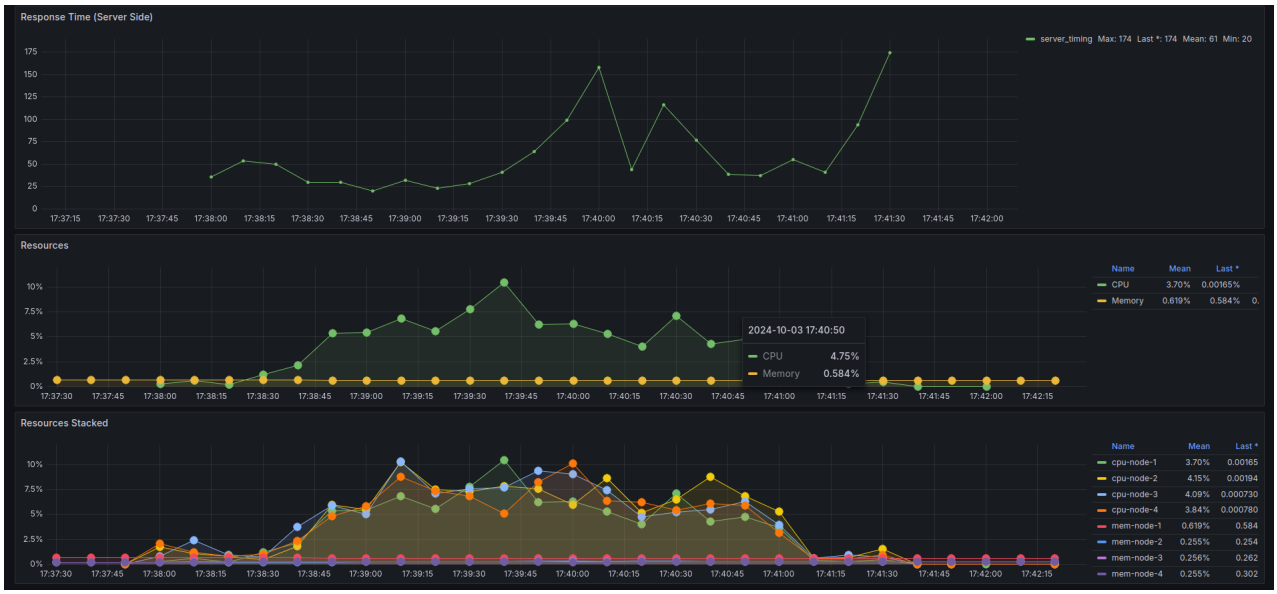
[1] https://stackoverflow.com/questions/69873961/artillery-repeats-same-line-from-payload-file



3.2.3. Replicación

```
docker compose up -d --scale node=4
```

```
make artillery-run-dict
```

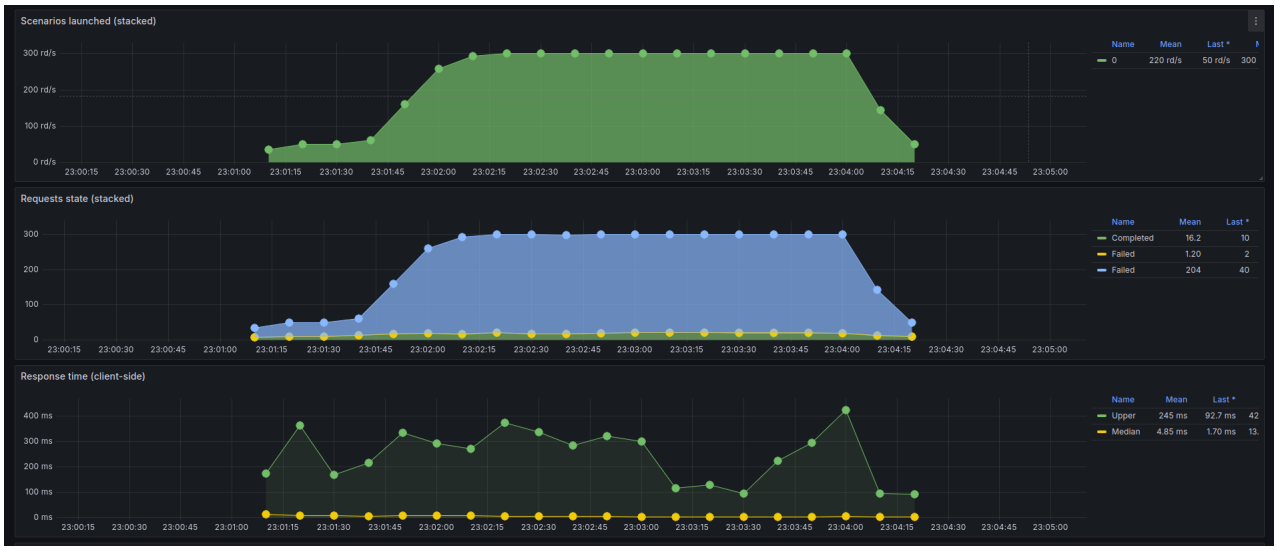


Como era de esperarse, desde la perspectiva del cliente los resultados son prácticamente idénticos al caso baso, con la distinción que es apreciable el balanceo de carga entre los distintos nodos

3.2.4. Rate Limit

```
RATE_LIMIT=2r/s docker compose up -d --scale node=4
make artillery-run-dict
```

Para este caso siendo que ya nos encontramos limitados por la API externa, consideramos que no encontraremos grandes diferencias respecto al caso base; con distinción de los tiempos de respuesta. Siendo que la limitación proveendrá del load-balancer local, el mismo deberá ser mucho mas rápido que tener que consultar a la API externa y ahí ser rechazado.



Obtenidos los datos podemos apreciar que se cumple nuestra afirmación y se observa una mejora sustancial del tiempo de respuesta

3.3. Spaceflight News

3.3.1. Base

El caso base se ejecuta de la siguiente manera

```
docker compose up -d
make artillery-run-spaceflight
```

De manera preliminar podremos suponer que al depender de una api externa, al igual que

en el caso del diccionario, rápidamente los requests fallarán

Presunción que rápidamente se confirma con los gráficos

Y donde no se aprecian diferencias significativas en comparación al caso del diccionario.

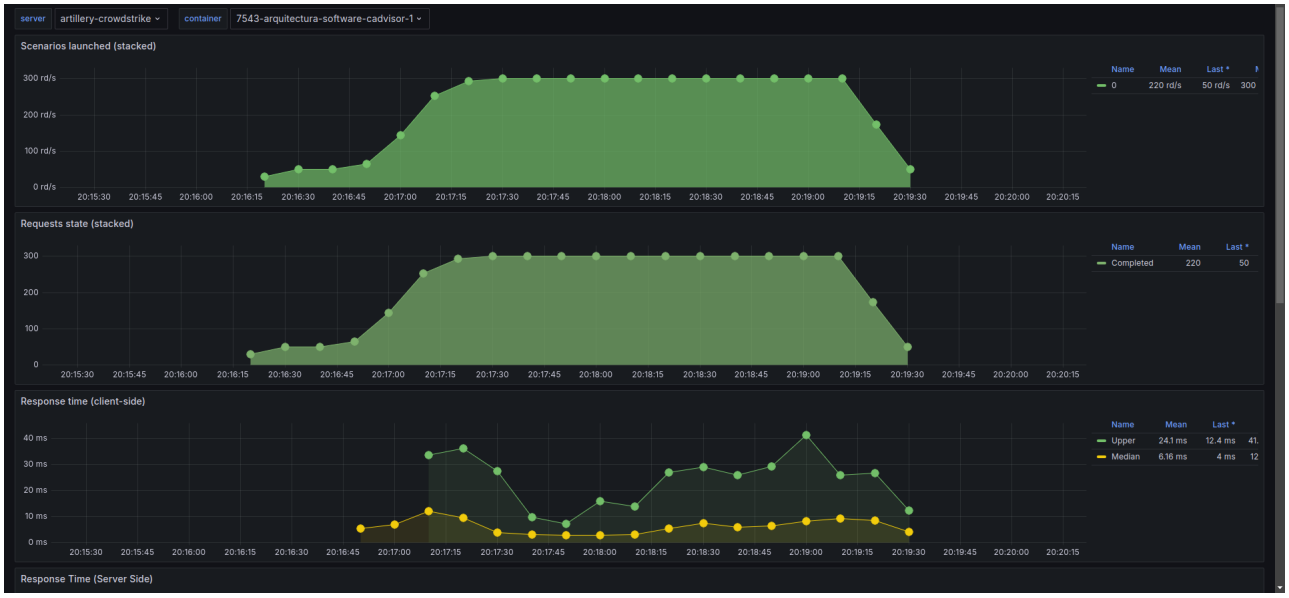
3.3.2. Caché

```
USE_CACHE=yes docker compose up -d
make artillery-run-spaceflight
```

Considerando el caso base donde rápidamente fallan los requests por una limitación de la api externa y contemplando la naturaleza de la información solicitada, podemos decir que éste

sera un caso ejemplo para la implementación de una caché

- Argumentos Inexistentes: No existe variación en los requests efectuados por el cliente que puedan alterarme la información a solicitar
- Baja probabilidad de expiración: La información es poco probable que sufra modificaciones repentinas y/o frecuentes, se trata de noticias aeroespaciales. No podriamos decir lo mismo si la información a obtener fuese la cotización de algun activo de la bolsa, que está sujeto a cambios rápidos.



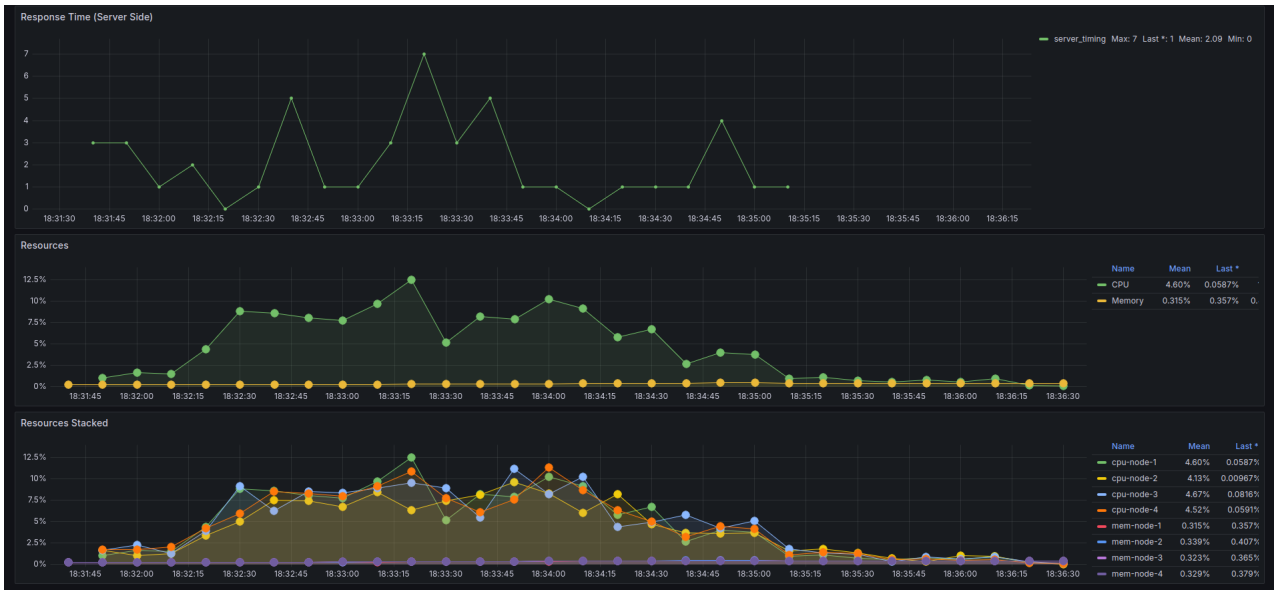
Lo explayado con anterioridad se puede apreciar a simple vista con este conjunto de métricas, donde la misma curva de demanda anteriormente producia inmediatamente fallas, en este caso no produce ninguna.

También vemos una mejora considerable en los tiempos de respuesta, dado que nuestro servidor carece de la necedidad de efectuar las requests externas, debido a la existencia de información cacheada

3.3.3. Replicas

```
docker compose up -d --scale node=4
make artillery-run-spaceflight
```

Al igual que el endpoint de diccionario, no esperamos diferencias con la implementación de replicación dado que el problema principal que posee el endpoint es la limitación de la API externa.



En efecto vemos la misma cantidad de errores, con la única diferencia que la carga se distribuye entre diversos nodos

3.3.4. Rate Limit

Dada la similitud con el endpoint de diccionario, prescindimos de realizar dicha prueba

3.4. Random Quote

A la hora de realizar pruebas de citas aleatorias, nos encontramos con que la API externa no era accesible, para lo cuál decidimos emplear otra api de similares características

3.5. Random Fact

3.5.1. Base

```
docker compose up -d
make artillery-run-random-fact
```

Este endpoint se podria considerar como uno similar al de spaceflight news, por su carencia de argumentos, pero con la gran diferencia de que en cada llamada deberá retornar un valor distinto.

En los gráficos se puede notar una diferencia sustancial en relación a nuestra hipótesis de que éste es un endpoint similar al de spaceflight news, la carencia de errores. Esto parece deberse a que la API externa o carece de un rate limit definido, o el mismo posee un valor mayor a la frecuencia que estamos empleando.

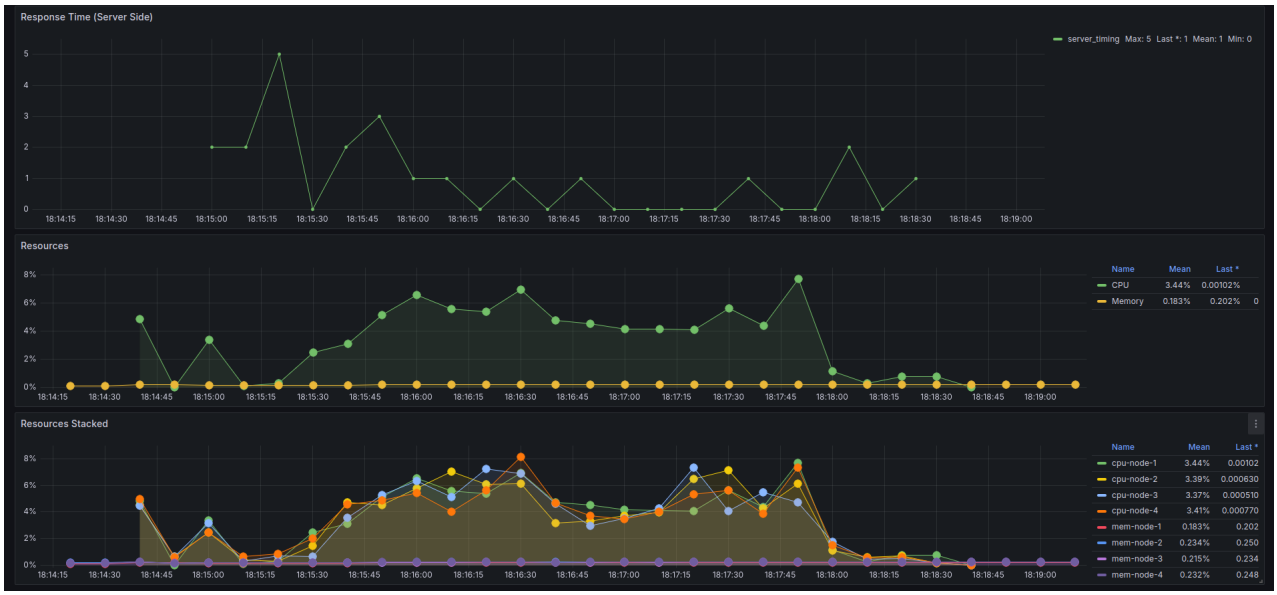
3.5.2. Cache

Siendo que cada request debe devolver un valor distinto, prescindimos de implementar una caché para evitar repeticiones. Una posible mejora a realizar en este caso sería el de implementar una caché activa, donde el servidor realiza una pre-carga de una cantidad de citas y al momento

de recibir un request en vez de solicitarla a la API externa, obtenerla de la caché interna. De esta manera se mejorarían los tiempos de respuesta, sacrificando quizá algo de randomización y/o cantidad de citas posibles.

3.5.3. Replicación

```
docker compose up -d --scale node=4
make artillery-run-random-fact
```



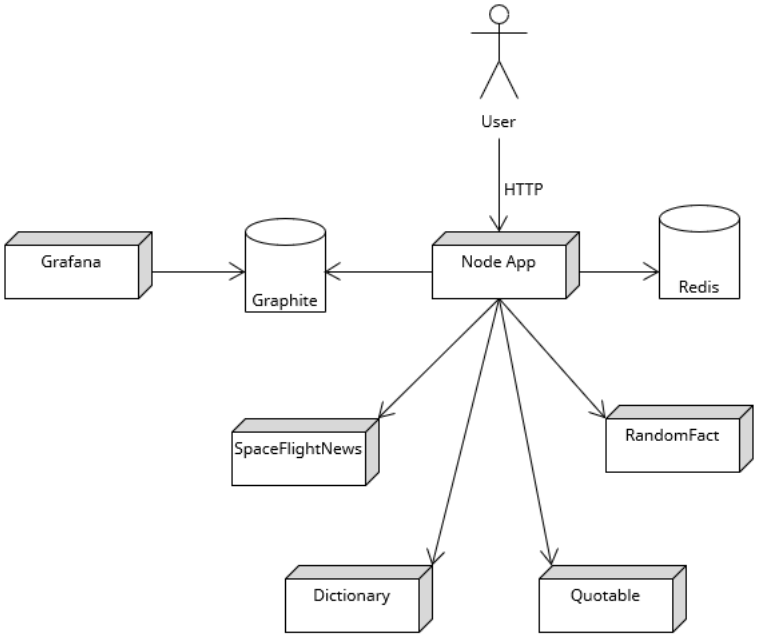
Nuevamente en un caso de replicación, no se aprecian diferencias significativas.

3.5.4. Rate Limit

Hemos prescindido de la implementación de un rate-limit en este endpoint dado que produciría resultados similares al observado para el ping; por lo cual no consideramos que sea de valor

4. Diagrama de componentes

En el siguiente diagrama se puede observar una vista de alto nivel de los componentes que interactúan en el funcionamiento de la aplicación de prueba



5. Potenciales Pruebas Adicionales

Un caso que consideramos valioso de probar es el del diccionario con una caché de tamaño limitado. En dicho caso jugará un papel esencial el tamaño de la caché, junto con la repetividad o no de las palabras que se solicitan.

Un caso donde no se repita ninguna palabra a solicitar, debería tener resultados similares a los de la ausencia de caché. Un caso similar podría ser que ante una caché de tamaño limitado, antes de que una palabra se repita (y haga uso de la caché) su entrada en la caché haya tenido que desalojarse en favor de otra palabra.

6. Conclusiones

El desarrollo de este trabajo nos ha servido para entender las herramientas disponibles para la obtención y análisis de métricas; como así su importancia a la hora de analizar estrategias. Hemos visto que las distintas estrategias poseen impactos diferentes dependiendo del dominio del problema y la relevancia de disponer de datos empíricos para respaldar suposiciones.