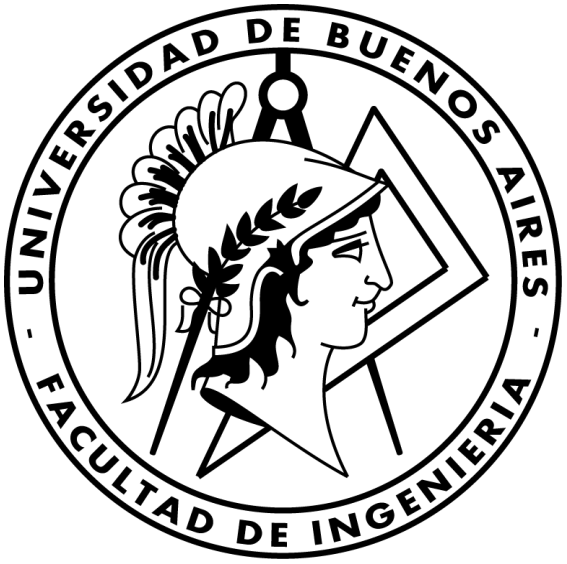
Trabajo Práctico 1

Arquitectura de Software (75.73)



Grupo: huanaro

Integrantes:

- Hugo Chavar
- Oscar Juárez
- Rodrigo Etchegaray Campisi
- Alejo Tomás Mariño

Fecha de entrega: 13/10/22

Tabla de contenidos

Tabla de contenidos **0**

Introducción **2**

Sección 1 **3**
1.1 Components & Connectors 3
1.1.1 Casos obligatorios 3
1.1.1.1 Un nodo 3
1.1.1.2 Replicado 4
1.2 Análisis de performance 5
1.2.1 Análisis de aprovechamiento de recursos 5
1.3 Análisis casos 7
1.3.1 Heavy 7
1.3.1.1 Comparación con Ping 7
1.3.1.2 Warm + ramp 8
1.3.1.2.1 Objetivo 8
1.3.2.2.2 Escenario 9
1.3.2.2.3 Ejecución de la prueba 9
1.3.2 Bbox 1 11
1.3.3 Bbox 2 13
1.4 Métrica propia 15
1.4.1 Blackbox1 15
1.4.2 Blackbox2 16
1.4.3 Ping y Heavy 17

Sección 2 **18**
2.1 Sincrónico / Asincrónico 18
2.2 Cantidad de workers (en el caso sincrónico) 19
2.3 Demora en responder 20

Sección 3 **21**
Caso de Estudio - Sistema de inscripciones **21**
3.1 Hipótesis 22
3.2 Configuraciones del sistema 22
3.3 Escenario a analizar 23
3.3.1 Resultados esperados 23
3.3.2 Ejecución 24

Conclusiones **24**

Introducción

El siguiente trabajo tiene como objetivo principal comparar diversas tecnologías para poder analizar cómo distintos aspectos impactan en los atributos de calidad y analizar qué cambios se podrían hacer para mejorarlos.

Un objetivo secundario del trabajo es el aprendizaje de herramientas útiles como:

- Node.js (+ Express)
- Docker
- Docker Compose
- Nginx
- Artillery + cAdvisor + StatsD + Graphite + Grafana

Sección 1

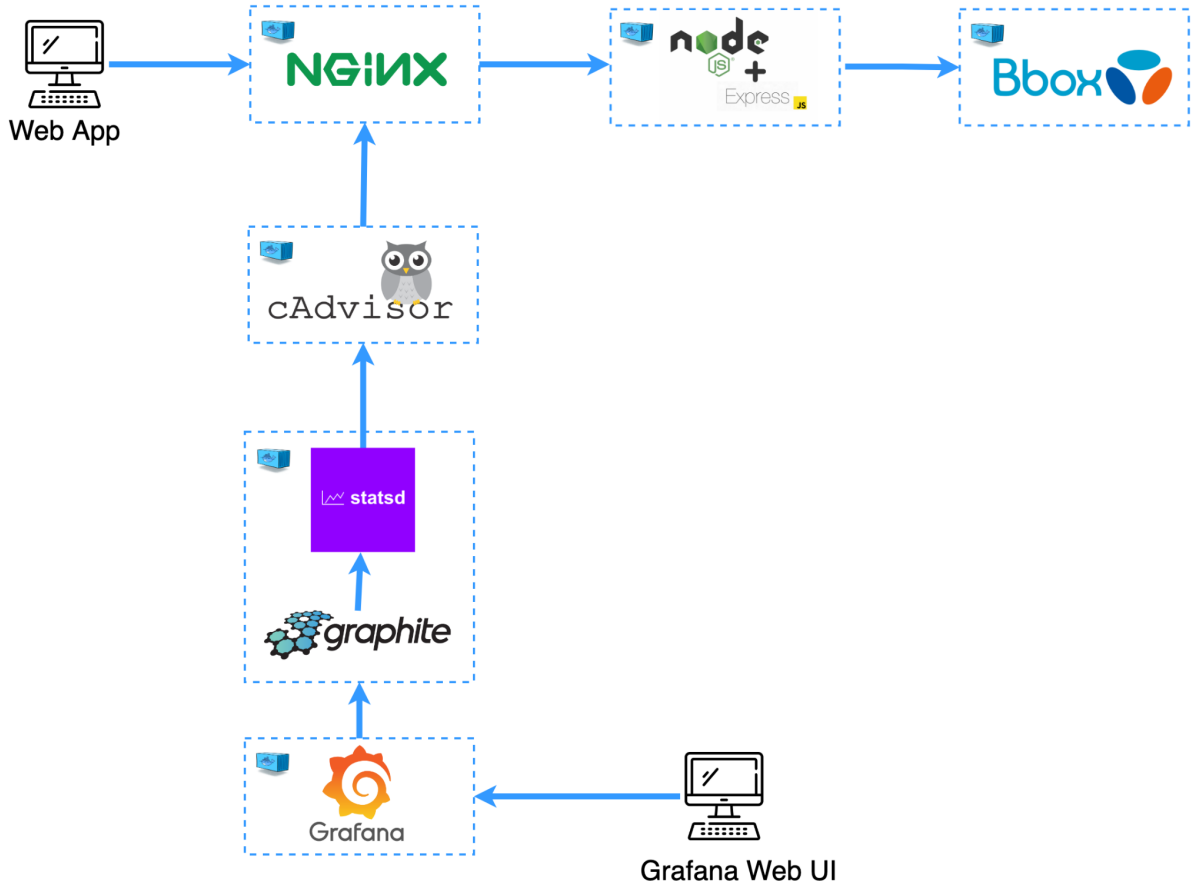
1.1 Components & Connectors

A continuación se visualizan distintas figuras en las que se muestran los componentes y conectores de distintos casos estudiados.[1]

1.1.1 Casos obligatorios

1.1.1.1 Un nodo

En este caso se representa el caso en el que hay un solo container con el servidor.



Documentos

[1] Link a gráficos

1.1: Diagrama de componentes y conectores del “caso nodo”

1.1.1.2 Replicado

En este caso se representan múltiples containers, con load balancing a nivel de nginx.

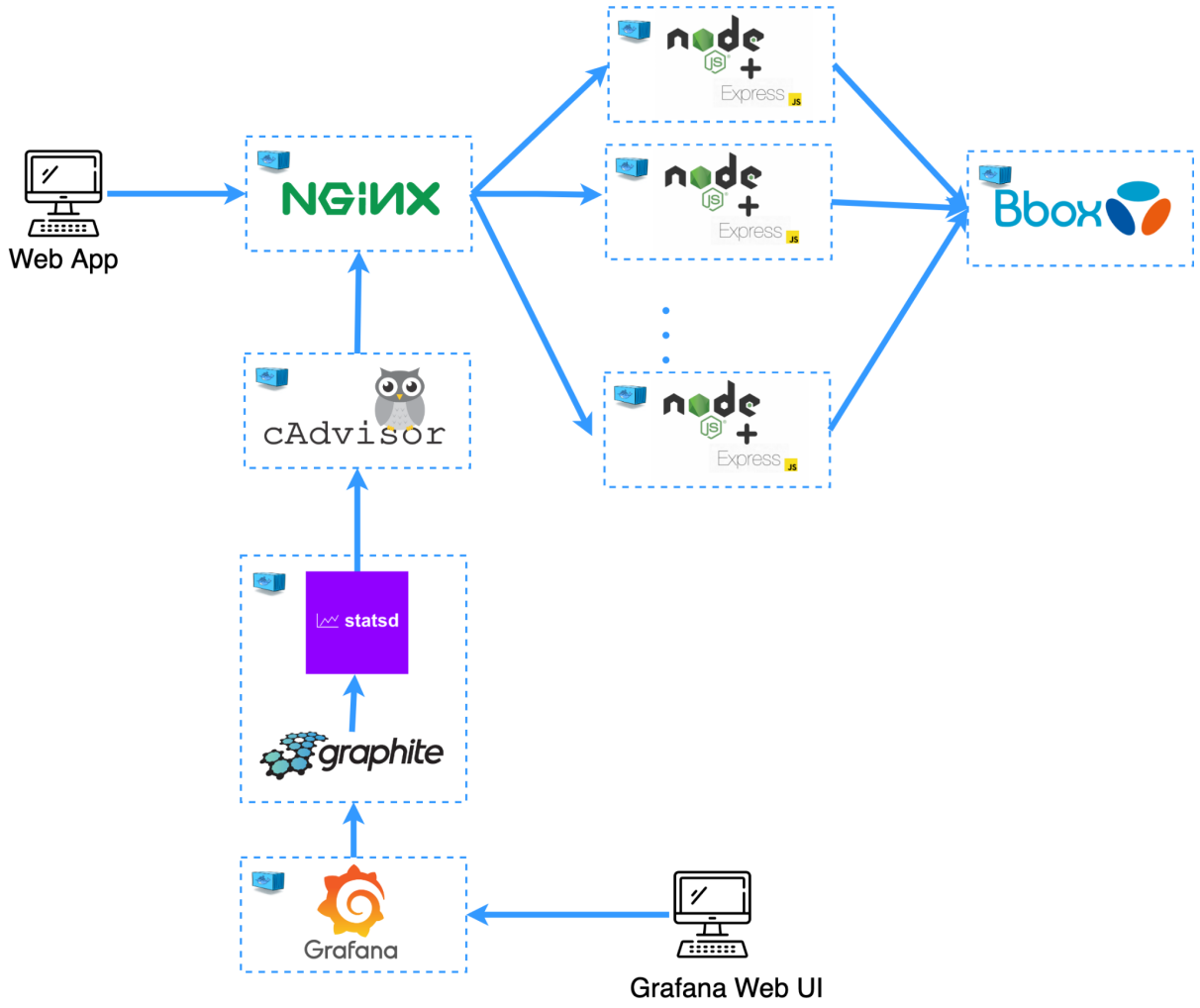


Figura 1.2: Diagrama de componentes y conectores del “caso replicado”

1.2 Análisis de performance

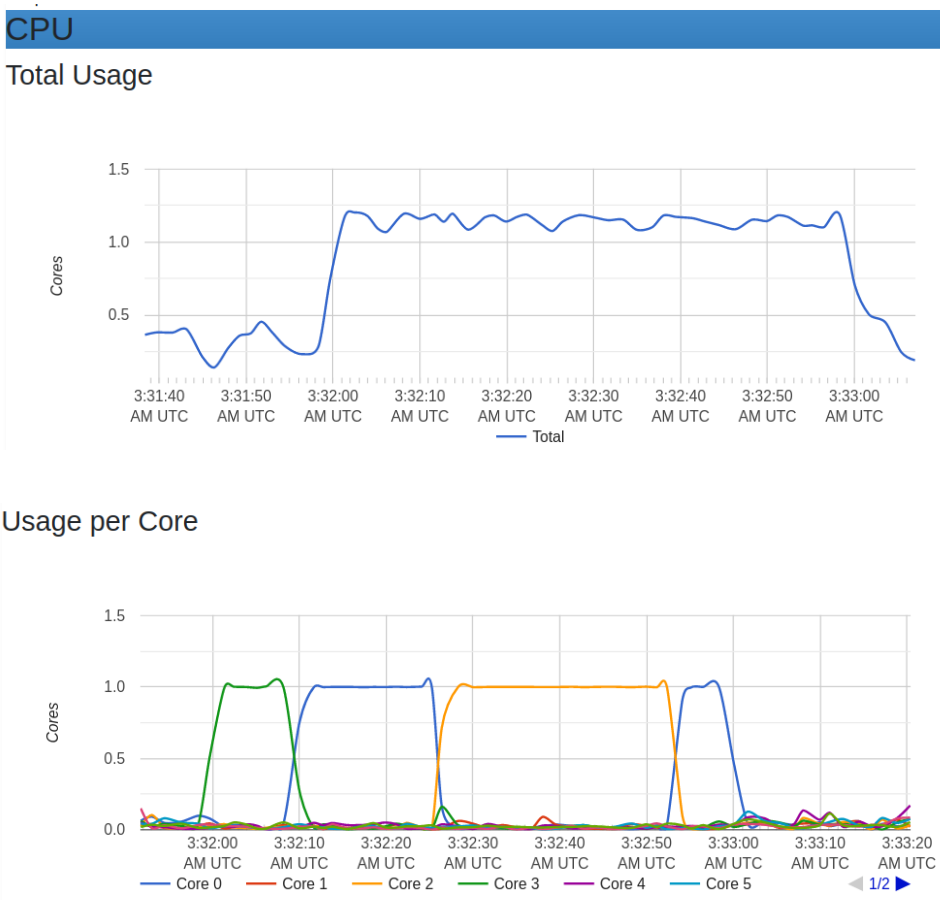
1.2.1 Análisis de aprovechamiento de recursos

En este apartado se presenta un análisis del aprovechamiento del uso de CPU.
Para ello se utiliza un endpoint de uso de CPU intensivo, suponiendo que dicho proceso no puede optimizarse a nivel código y es un servicio crítico. Entonces nos preguntamos si se puede mejorar algún atributo de calidad.

En el caso inicial disponemos de un solo nodo de la aplicación como se muestra en la sección 1.1.1.1. Con la ayuda de la herramienta *ab* se ejecutan 12 requests con 6 de ellos en paralelo, es decir se envían 6 requests juntos y después otros 6. Esto se logra con el siguiente comando:

```
ab -n 12 -c 6 http://localhost:5555/heavy
```

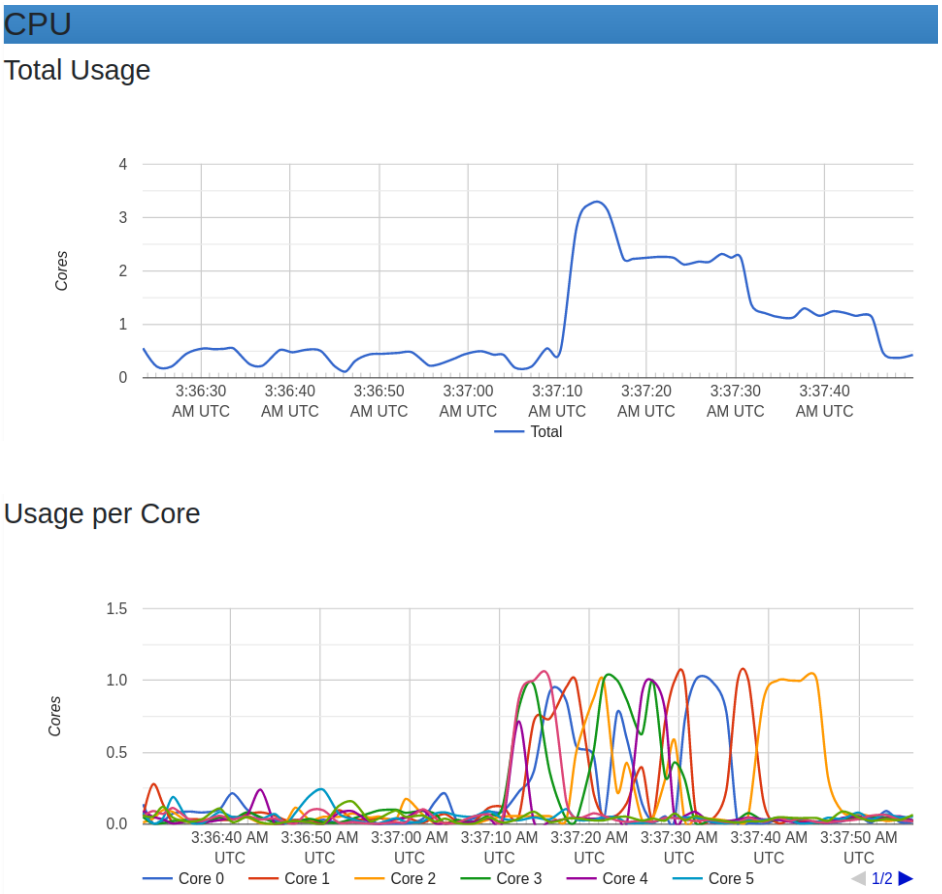
El análisis lo realizamos con la herramienta *cAdvisor* y obtuvimos lo siguiente:



La primera imagen muestra el uso total de la CPU, como puede apreciarse apenas por encima de un core está siendo utilizado simultáneamente. La segunda imagen muestra en distintos colores los distintos cores, si bien se alternan en el procesamiento no hay un aprovechamiento de paralelismo para maximizar el throughput.

Luego de observar el uso de CPU mostrado por cAdvisor notamos que solo un core era utilizado a la vez, estando los restantes 7 prácticamente inactivos mientras que los 6 requests entrantes eran encolados para ser atendidos por el único proceso uno a la vez.

Luego escalamos a 6 las instancias de node y realizamos el siguiente análisis:



En primer lugar vemos que casi no hay cores ociosos, el uso total mostrado en la primer imagen indica que aproximadamente más de 3 de ellos realizaron procesamiento en paralelo por un lapso de tiempo y como resultado se puede apreciar una disminución del tiempo de procesamiento total, esto también puede notarse en la salida del comando ab.

Un nodo

```
Time taken for tests:   60.093 seconds
```

6 nodos

```
Time taken for tests:   35.135 seconds
```

Como conclusión del análisis podemos decir que con herramientas como ab y cAdvisor se pueden detectar problemas de performance asociados a la mala utilización de recursos que sería tal vez difícil de notar de no contar con ellas. Se ha mejorado notablemente la performance y esto impactará en la disponibilidad de nuestro servicio.

1.3 Análisis casos

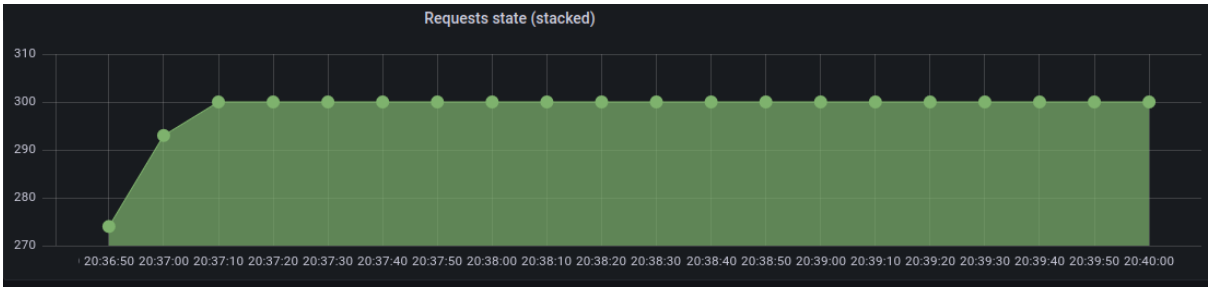
1.3.1 Heavy

Gracias al análisis de performance realizado en la sección 1.2 sabíamos que este endpoint podía empezar a fallar rápidamente dado que usando un único contenedor de node, se van a terminar encolando requests.
Como también se mencionó en la sección 1.2, habiendo escalado la cantidad de contenedores debería reducirse la cantidad de requests encolados.

1.3.1.1 Comparación con Ping

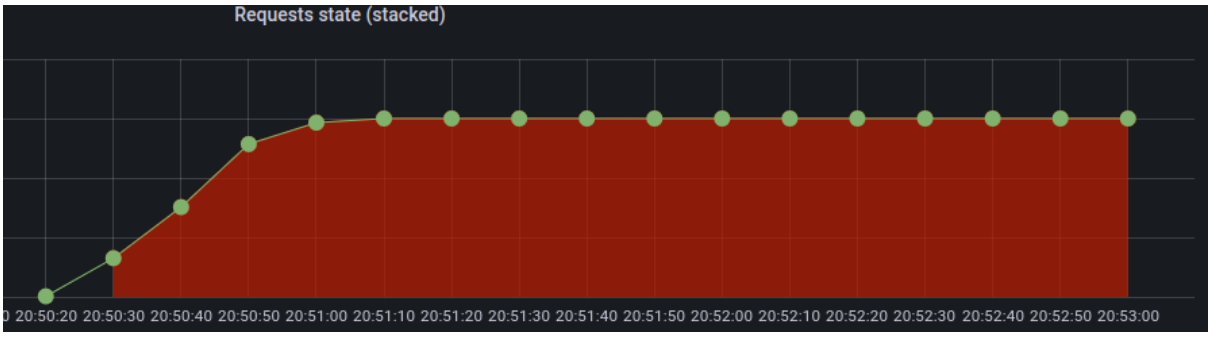
Para empezar, hicimos una comparación entre el endpoint de ping y el de heavy sobre el escenario de ejemplo provisto por la cátedra.
Esta comparación sólo tuvo como objetivo ver cómo se comportaba cada endpoint en un mismo escenario siendo uno un endpoint muy “simple”, es decir, con poco procesamiento y por otro lado un endpoint con trabajo pesado.

Para el endpoint de ping obtuvimos:




```
All VUs finished. Total time: 2 minutes, 33 seconds

------------------------------
Summary report @ 20:38:42(-0300)
------------------------------

http.codes.200: ..................................................................... 4184
http.request_rate: .................................................................. 30/sec
http.requests: ...................................................................... 4184
http.response_time:
  min: .............................................................................. 2
  max: .............................................................................. 78
  median: ........................................................................... 10.1
  p95: .............................................................................. 32.1
  p99: .............................................................................. 46.1
http.responses: ..................................................................... 4184
vusers.completed: ................................................................... 4184
vusers.created: ..................................................................... 4184
vusers.created_by_name.Root (/): .................................................... 4184
vusers.failed: ...................................................................... 0
vusers.session_length:
  min: .............................................................................. 3.1
  max: .............................................................................. 117.1
  median: ........................................................................... 14.7
  p95: .............................................................................. 40
  p99: .............................................................................. 58.6
```

Se puede observar claramente que el servidor puede manejar tranquilamente la cantidad de requests enviada, siendo así todos los requests exitosos.
Para el endpoint heavy podemos observar que:




```
All VUs finished. Total time: 2 minutes, 43 seconds

------------------------------
Summary report @ 20:52:33(-0300)
------------------------------

errors.ETIMEDOUT: ..................................................... 4156
http.codes.200: ....................................................... 1
http.request_rate: .................................................... 25/sec
http.requests: ........................................................ 4157
http.response_time:
  min: ................................................................ 5016
  max: ................................................................ 5016
  median: ............................................................. 5065.6
  p95: ................................................................ 5065.6
  p99: ................................................................ 5065.6
http.responses: ....................................................... 1
vusers.completed: ..................................................... 1
vusers.created: ....................................................... 4157
vusers.created_by_name.Heavy (/heavy): ................................ 4157
vusers.failed: ........................................................ 4156
vusers.session_length:
  min: ................................................................ 5039.6
  max: ................................................................ 5039.6
  median: ............................................................. 5065.6
  p95: ................................................................ 5065.6
  p99: ................................................................ 5065.6
```

Es el caso totalmente opuesto. Prácticamente todos los requests fallaron, excepto uno. Gracias a esta comparación inicial entendemos que los valores de carga para poder obtener un buen análisis de este endpoint deberán ser considerablemente menores.

1.3.1.2 Warm + ramp

1.3.1.2.1 Objetivo

Queríamos ver el comportamiento del endpoint heavy ante una situación de carga con pequeños incrementos. También, gracias a las pruebas iniciales, habíamos notado que cuando el endpoint empezaba a devolver errores, todos los requests subsiguientes terminaban en error. Por lo tanto, quisimos chequear la capacidad de recuperación del endpoint dejando una pausa intermedia.
Por último, buscamos ver si los puntos de falla de este escenario pueden ser mejorados aumentando la cantidad de instancias.

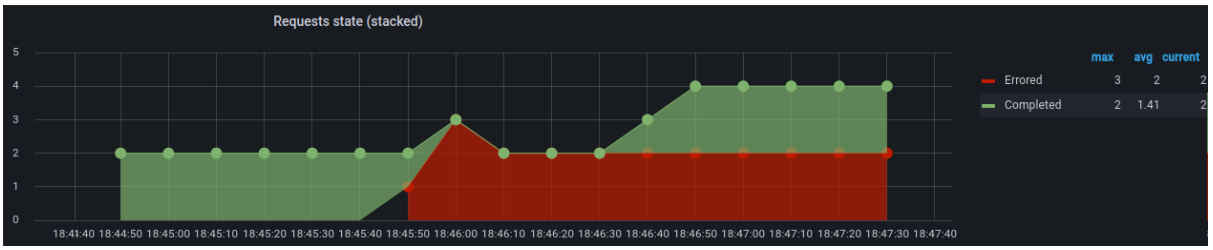
1.3.2.2.2 Escenario

```yaml
phases:
  - name: Warm-up - 1 usuario cada 6 segundos
    duration: 30
    arrivalCount: 5
    # rampTo: 2
  - name: Ramp - 1 usuario cada 4.6 segundos
    duration: 60      You, 31 minutes ago • U
    arrivalCount: 13
  - name: Ramp - 1 usuario cada 4.2 segundos
    duration: 30
    arrivalCount: 7
  - name: Pause
    pause: 20
  - name: Final - 1 usuario cada 6 segundos
    duration: 30
    arrivalCount: 5
```

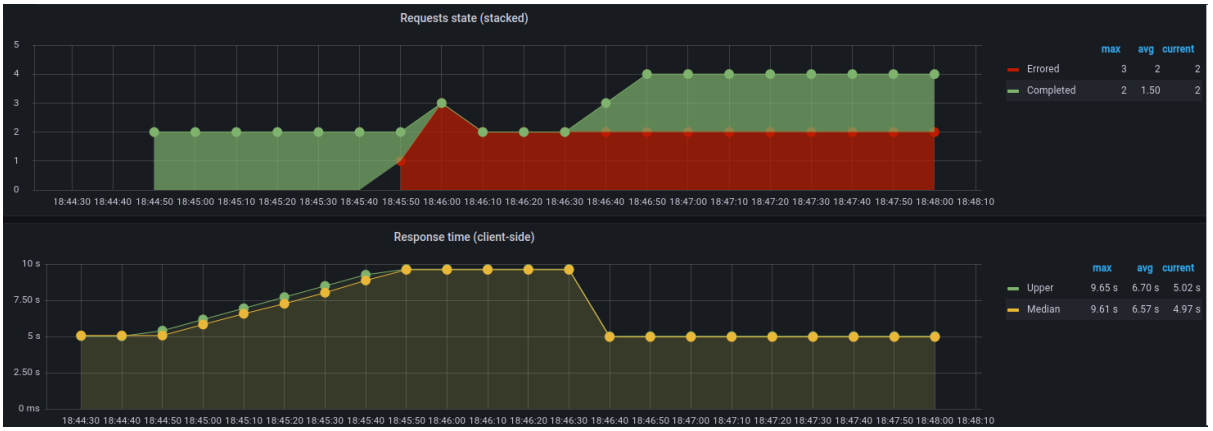
1.3.2.2.3 Ejecución de la prueba

```
All VUs finished. Total time: 2 minutes, 57 seconds

------------------------------
Summary report @ 18:46:52(-0300)
------------------------------

errors.ETIMEDOUT: ..................................................... 8
http.codes.200: ....................................................... 22
http.request_rate: .................................................... 0/sec
http.requests: ........................................................ 30
http.response_time:
  min: ................................................................ 5002
  max: ................................................................ 9645
  median: ............................................................. 5378.9
  p95: ................................................................ 8868.4
  p99: ................................................................ 9230.4
http.responses: ....................................................... 22
vusers.completed: ..................................................... 22
vusers.created: ....................................................... 30
vusers.created_by_name.Heavy (/): ..................................... 30
vusers.failed: ........................................................ 8
vusers.session_length:
  min: ................................................................ 5004.7
  max: ................................................................ 9647.1
  median: ............................................................. 5378.9
  p95: ................................................................ 8868.4
  p99: ................................................................ 9230.4
```

Más del 70% de los requests fueron exitosos.

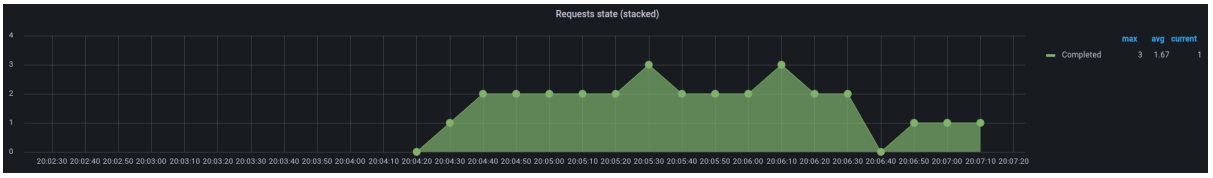


Podemos ver cómo el endpoint fue capaz de soportar la primera fase de warm up, pero comenzó a tener problemas con los ramp, no pudiendo devolver más de dos requests exitosos. Por último, vemos la recuperación del endpoint luego de la fase de pausa.

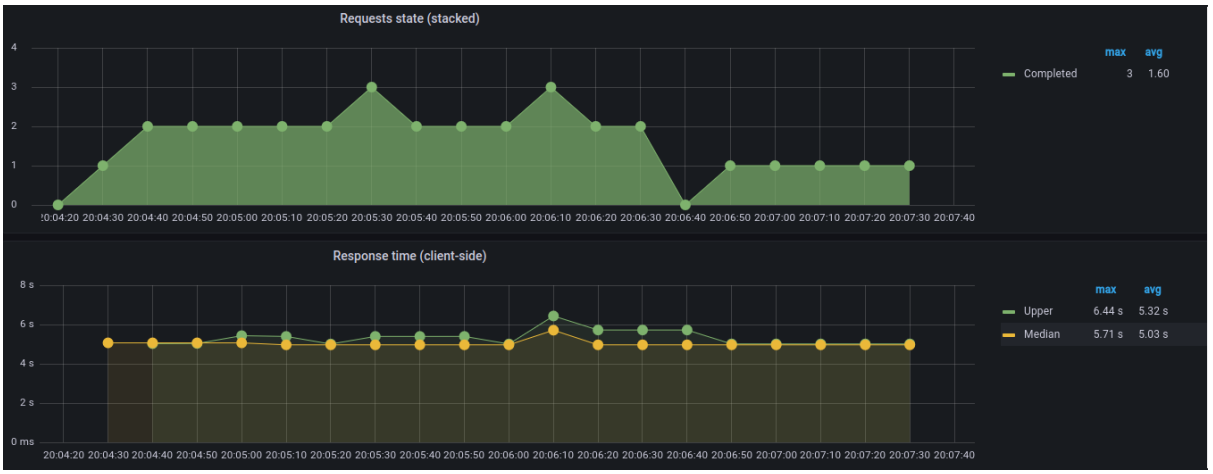


Algo importante a notar es la correlación entre los requests y el tiempo de respuesta. Podemos ver como el tiempo de respuesta fue aumentando hasta llegar a su máximo que coincidió con el instante en que empezaron a haber requests fallidos. A su vez, podemos ver cómo la pausa ayudó al endpoint a recuperarse y volver a un tiempo de respuesta menor, lo que se condijo con requests exitosos nuevamente.

Veamos qué sucede al aumentar la cantidad de instancias a 3:



Antes que nada, podemos ver cómo no hubo requests fallidos.



Volviendo a comparar con el tiempo de respuesta, podemos notar que ahora se mantuvo mucho más constante que antes.

Por lo tanto, podemos concluir que para este caso en particular, el escalamiento horizontal rinde sus frutos manteniendo un tiempo de respuesta mucho más estable y además no teniendo requests fallidos. También vale aclarar que dicho escalamiento dentro del mismo hardware está limitado a la cantidad de procesadores con los que cuenta el servidor, aunque se podría poner otro equipo físico y escalar todo lo que fuera necesario para un endpoint crítico que tenga estas características.

1.3.2 Bbox 1

Vimos anteriormente que el endpoint ping puede soportar 30 requests por segundo sin inconvenientes, y ahora tenemos la tarea de ver cómo se comporta al invocar a un proxy que desconocemos su implementación aunque sabemos que puede tratarse de uno sincrónico o asincrónico.

Haciendo requests individuales vemos un tiempo de respuesta de un poco menos de 1000 ms, entonces lo siguiente a realizar es determinar la carga que soporta sin degradar la performance y cuanta hasta degradar la disponibilidad.

Tomando el caso base en el cual tenemos una única instancia de nuestro servicio *bbox1* se ejecuta un escenario en forma de rampa lenta para determinar la latencia y los requests timeouts, y se obtuvo el siguiente resultado.

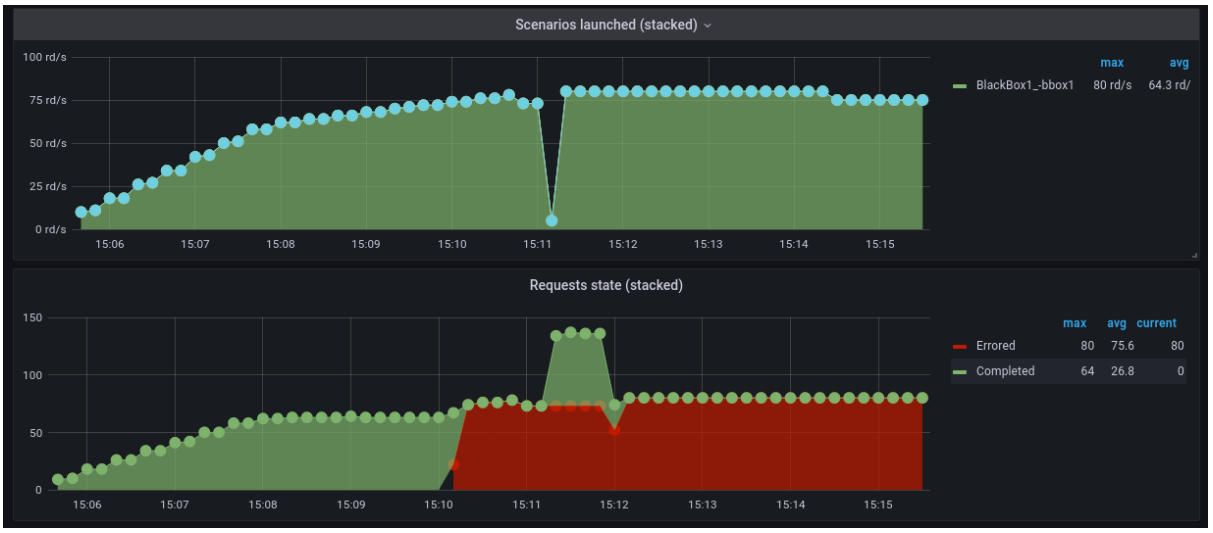
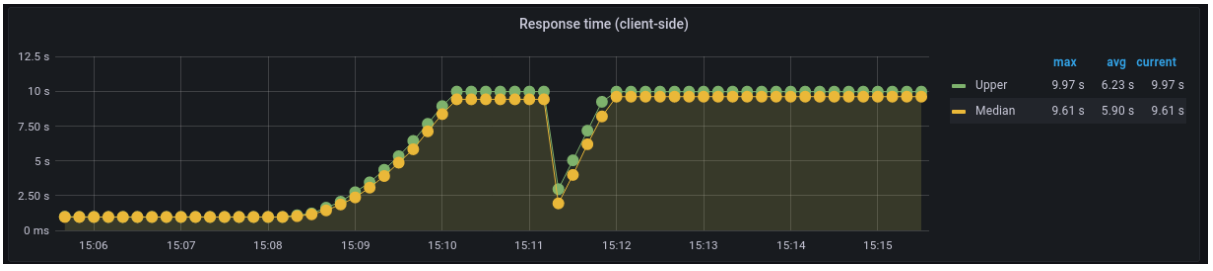


Gráfico 1 (Prueba de carga /bbox1 con 1 nodo)

El escenario consiste en gradualmente incrementar la carga por 5 minutos hasta llegar a 8 requests por segundo, se realiza una pausa de 20 segundos sin enviar requests y luego se envía una carga sostenida de 8 requests por segundo.

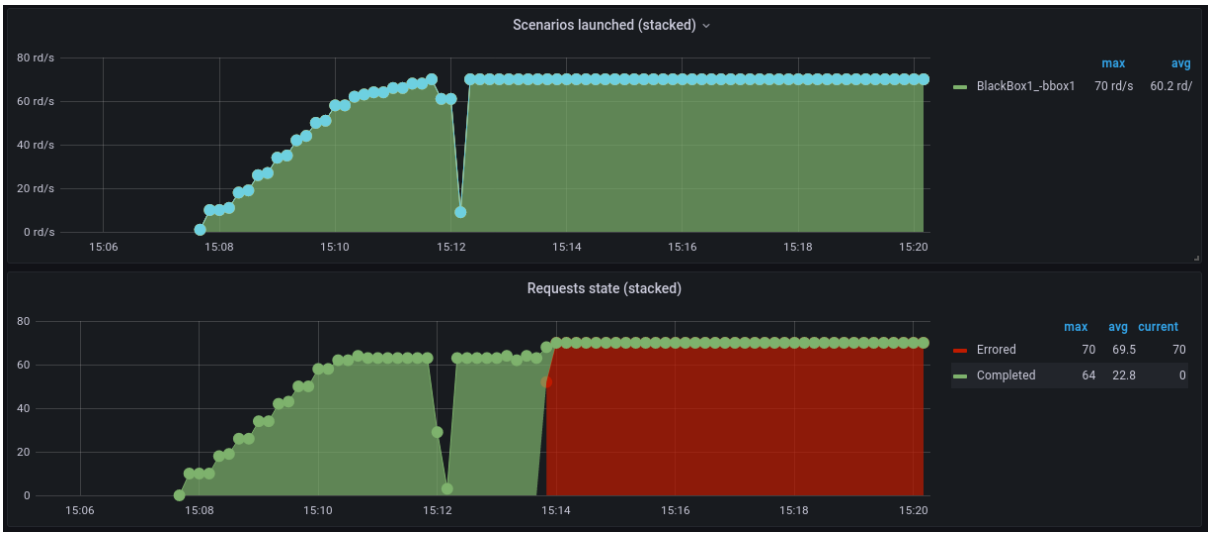
Analizando los datos obtenidos vemos que el servicio puede manejar sin perder requests hasta 63 requests cada 10 segundos, o sea 6.3 r/s. A tasas superiores se pierden la mayoría. También notamos que luego de la pausa de 20 segundos se vuelven a procesar unos 6.3 r/s por un lapso aproximado de 30 segundos y luego rápidamente se llega a la saturación perdiéndose todos los requests.

En el siguiente gráfico vemos la degradación de la performance a medida que aumenta la tasa de arribo, y notamos que la tasa límite antes de comenzar a degradar la performance es de 6.2 r/s, es decir por debajo de dicha tasa se mantiene el tiempo de respuesta alrededor de los 1000 ms.



Habiendo estudiado el comportamiento de nuestra configuración vamos a probar aumentar los nodos de nuestro endpoint para ver si hay alguna mejora. A priori el cuello de botella es el servicio bbox que estamos invocando pero veremos si obtenemos un indicio de cómo podremos escalar el servicio.

Luego de ejecutar el mismo escenario obtuvimos el siguiente resultado.



Nuevamente al alcanzar una tasa de 6.3 r/s se producen muchas pérdidas. La conclusión a la que arribamos es que este endpoint que en definitiva actúa como proxy de otro servicio no puede ser mejorado aumentando instancias de nuestro servicio y corresponde analizar posibles mejoras del servicio blackbox ya que se trata de un cuello de botella.

1.3.3 Bbox 2

En esta sección analizaremos el comportamiento de nuestras configuraciones a través del endpoint /bbox2, que al igual que en el caso anterior es difícil anticiparse a un resultado ya que desconocemos si se trata de un proxy sincrónico o asincrónico.

Haciendo requests individuales vemos un tiempo de respuesta de un poco menos de 1500 ms, y nuevamente intentaremos determinar la carga que soporta sin afectar la performance y la disponibilidad.

Se diseñó un test de carga con *artillery* intentando emular la prueba realizada en el endpoint /bbox1 de manera de poder comparar ambos. Las pruebas iniciales con 8 requests por segundo terminaron sin pérdidas tanto de performance como de requests. Por lo que se fue incrementando la carga para empezar encontrarnos con una performance disminuída. Mediante prueba y error llegamos a la configuración de tests explicada a continuación.

El escenario consiste en gradualmente incrementar la carga por 5 minutos desde 100 r/s hasta llegar a 700 r/s, se realiza una pausa de 20 segundos sin enviar requests y luego se envía una carga sostenida de 700 r/s.

Comenzamos analizando el caso base de un nodo node sin escalar. Analizando los datos obtenidos vemos que el servicio puede manejar sin perder requests hasta 5400 requests cada 10 segundos, o sea 540 r/s. A tasas superiores se observan pérdidas que reflejan la diferencia entre el máximo que puede soportar y la carga enviada, es decir las pérdidas son predecibles. También notamos que luego de la pausa de 20 segundos se vuelven a procesar unos 540 r/s por un lapso aproximado de 30 segundos y luego se vuelve a la situación mencionada antes, es decir realizar una pausa no afecta sensiblemente en el nivel de servicio provisto.

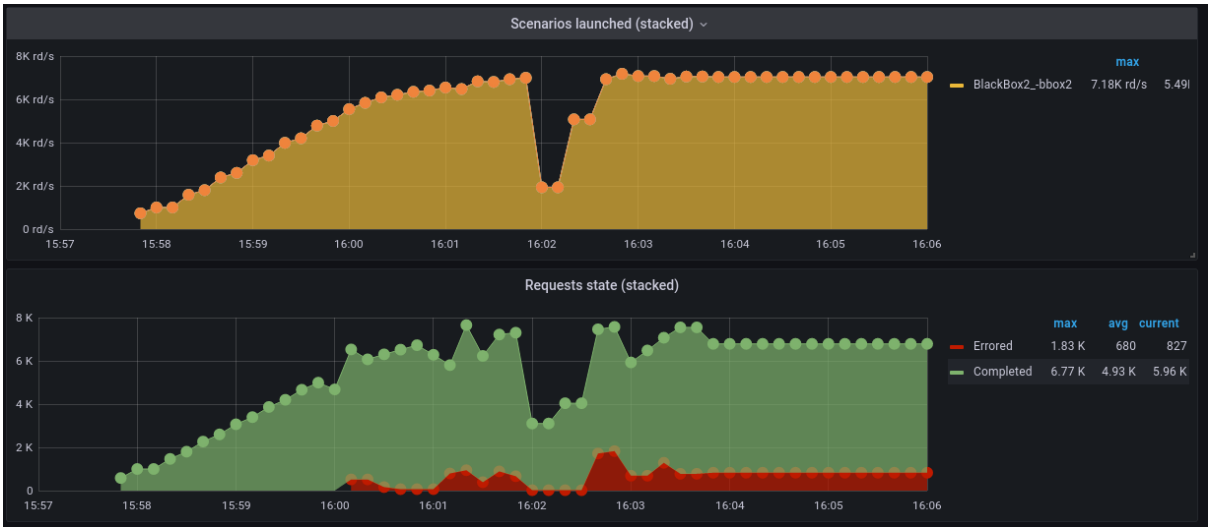
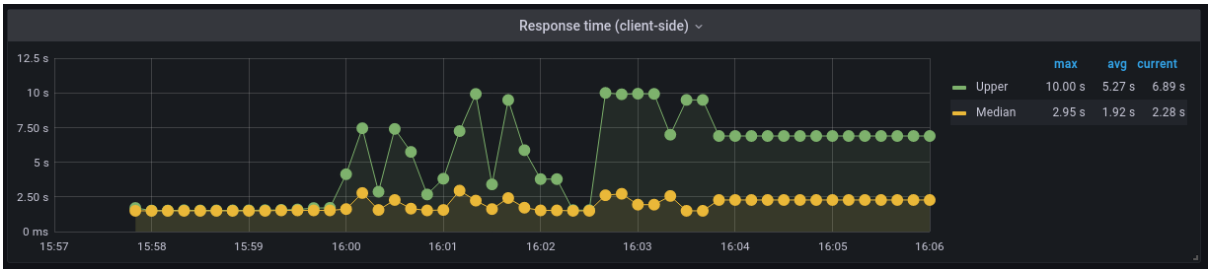
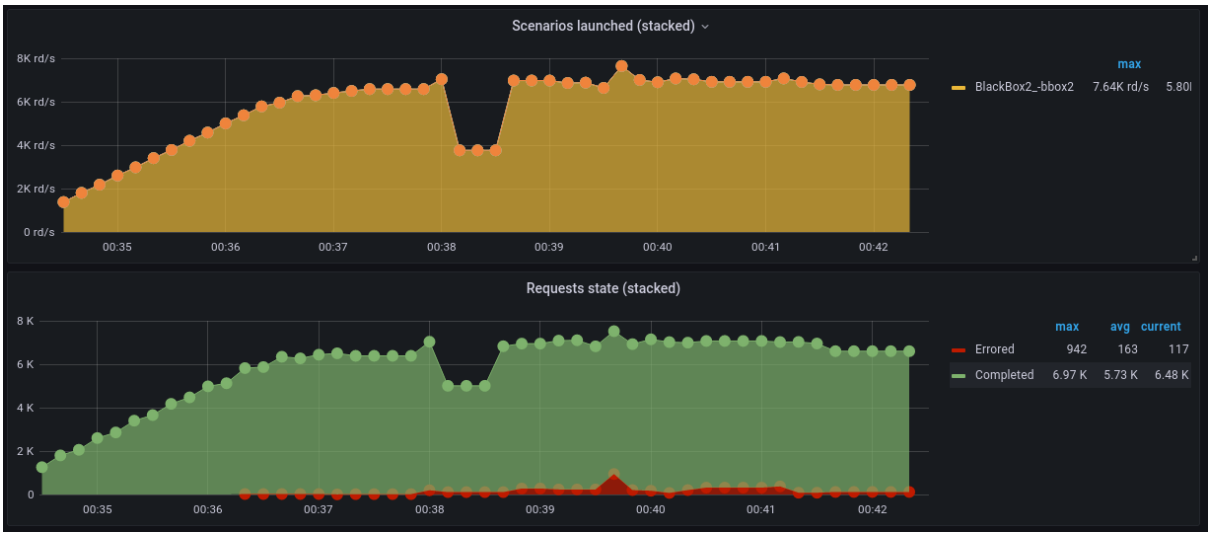


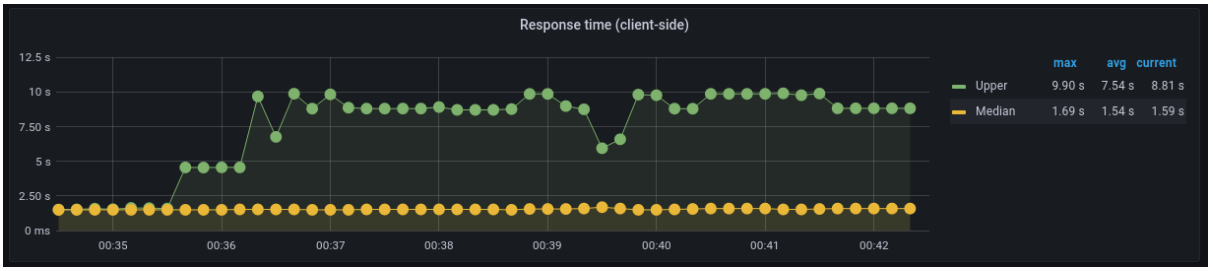
Gráfico 2 (Prueba de carga /bbox2 con 1 nodo)

En el siguiente gráfico vemos la degradación de la performance a medida que aumenta la tasa de arribo, y notamos que la tasa límite antes de comenzar a degradar la performance es de aproximadamente 520 r/s, es decir por debajo de dicha tasa se mantiene el tiempo de respuesta alrededor de los 1.5 s. Al llegar al tiempo de respuesta de 10 s sabemos del gráfico anterior que se están perdiendo requests, pero al estabilizarse la carga en 700 r/s el tiempo de respuesta también se estabiliza en 7.5 s.



Para la ejecución con 3 instancias se obtuvieron los siguientes resultados.





En este caso la cantidad de pérdidas es un poco menor y más estable pero el tiempo de respuesta en general es más alto, es decir que se ganó levemente en disponibilidad a costa de la performance, pero la diferencia es poca.

Como podemos observar en ambas configuraciones de deployment se mantiene un comportamiento muy similar, concluimos que esto se debe a que el servicio bbox2 mantiene la misma configuración y por tanto genera un cuello de botella en el sistema a pesar de replicar los nodos, lo cual desfavorece la escalabilidad de nuestro servicio.

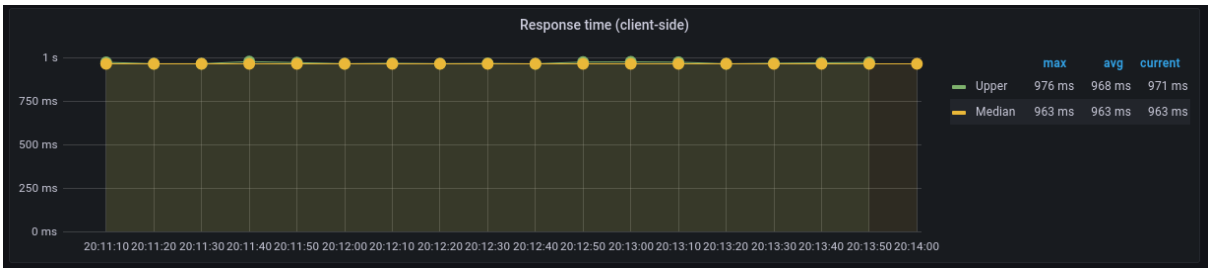
También se observa el hecho de que el response time tenga una tendencia a mantenerse constante en función de la cantidad de requests y a diferencia de /bbox1 este servicio responde más favorablemente a una mayor cantidad de requests esto nos da una pauta de que este servicio podría estar implementado de manera asincrónica.

1.4 Métrica propia

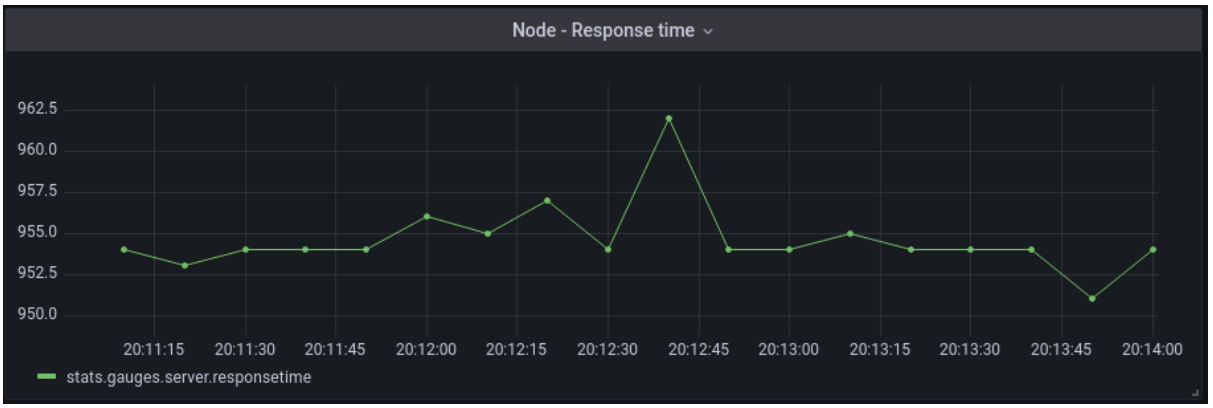
En este apartado mostramos los resultados obtenidos de la métrica de tiempo de respuesta desde Node, la cual mostramos en milisegundos para lograr una mejor precisión de la variación que tienen los endpoints. Es útil cuando la variación de tiempos es bastante estable, es decir cuando los endpoints no están en momentos de estrés.

1.4.1 Blackbox1

Se ejecuta un escenario al /bbox1 con 3.4 r/s obteniéndose los gráficos originales:

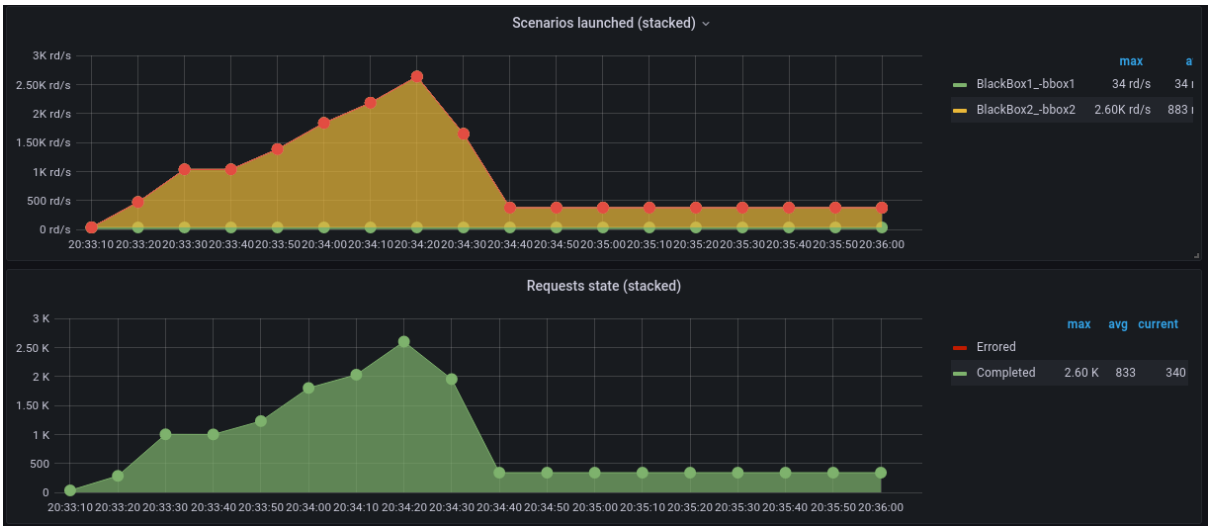


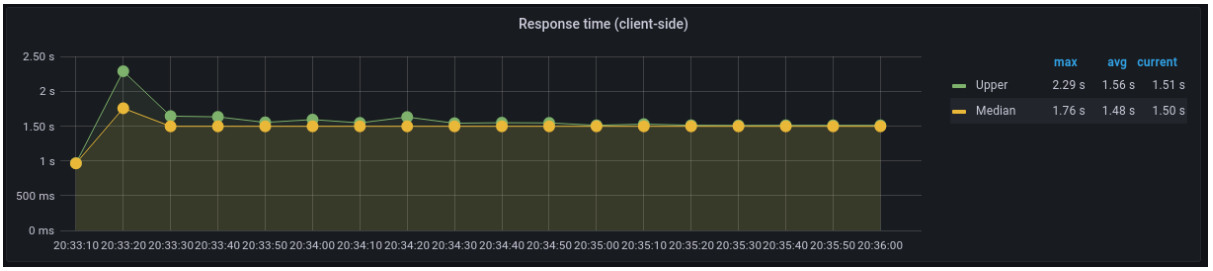
Y el resultado de nuestra métrica:



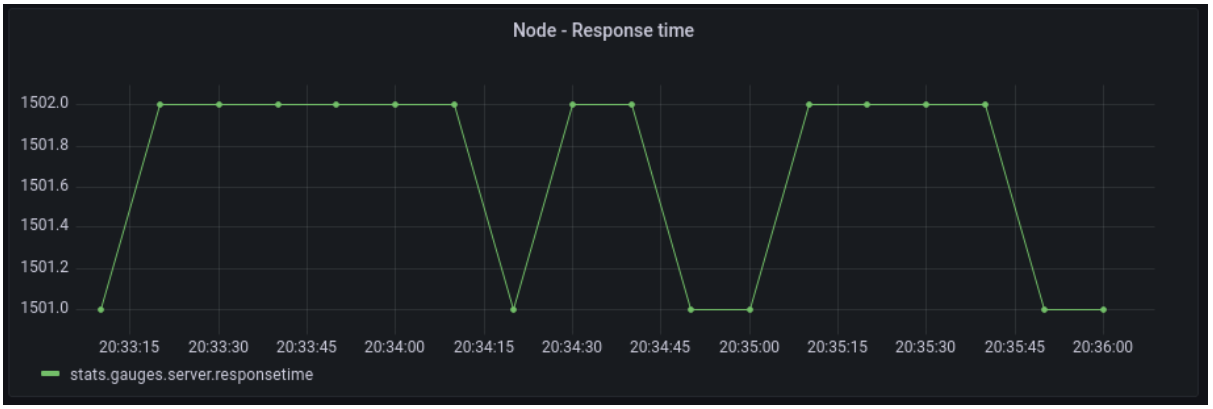
1.4.2 Blackbox2

En este caso ejecutamos un escenario de rampa para /bbox2 hasta los 340 r/s y en los gráficos originales se visualiza de la siguiente manera:





Y en nuestra métrica podemos apreciar los milisegundos de tiempo de respuesta.



1.4.3 Ping y Heavy

No adjuntamos gráficos de los endpoints ping y heavy debido a que consideramos que no aportaban demasiado. Esto se debe a que para ambos casos los tiempos de respuesta desde el lado del server son prácticamente constantes. Para el caso de ping el tiempo arrojado era 0ms y es algo que tiene sentido debido a que no hay ningún tipo de procesamiento cuando se usa este endpoint. Para el caso de heavy el tiempo siempre era de 5s debido al procesamiento interno que realiza para simular el trabajo pesado.

Sección 2

En la siguiente sección se caracterizaron las dos "bbox" analizadas en la sección anterior.

Antes de comenzar con el análisis y la caracterización pedida para esta sección, vemos como algo fructífero la definición apropiada de ciertos términos utilizados en la consigna, de manera de mostrar el porqué abordamos esta sección como lo hicimos mostrando cómo entendimos los términos usados.
Para esta sección definimos las siguientes palabras:

- Servicio Sincrónico: Un servicio sincrónico habla de un servicio el cual en el momento que recibe una request, trabajará en ese momento para procesarla y devolver una respuesta (si hubiese previas requests aún no procesadas se añadiría a una cola de requests). El servicio en este caso está encargado de recibir y procesar procesar las requests entrantes para luego devolver la respuesta apropiada.

- Servicio Asincrónico: Un servicio asincrónico habla de un servicio el cual una vez que recibe una request, la redirecciona a otro componente de la arquitectura de un sistema para que lo procese, retorne o realice cualquier otra transformación necesaria. El servicio en este caso está únicamente encargado de recibir (y también encola requests si fuese necesario aunque para este caso, como el servicio no debe hacer ningún tipo de procesamiento propio las requests saldrán del servicio mucho más rápido, haciendo que las colas sean mucho más rápidas) requests y redireccionarlas a otro elemento de un sistema.

Con estas definiciones en pie, pasamos ahora a responder a la pregunta de qué tipo de servicio es cada bbox (/bbox1 y /bbox2).

Para hacer una comparación entre las bbox1 y bbox2 que tengan algún tipo de validez, asumimos que el servicio que ejecutan ambas es el mismo/es similar. Esta suposición la hacemos porque, por ejemplo un servicio asincrónico tarda más en promedio que uno sincrónico para resolver una request independiente por el hecho de que la request se pasa por distintas componentes antes de ser devuelta al cliente, pero si ahora decimos que el servicio sincrónico que contrapone a ese servicio requiere un procesamiento mucho más alto entonces es posible que el servicio sincronico tarde mas en responder a esas requests individuales.

2.1 Sincrónico / Asincrónico

Uno de los servicios se comportará de manera sincrónica, y el otro de manera asincrónica. Deberán detectar de qué tipo es cada uno.

Nos basaremos en la idea de que si el sincrónico debe no solo recibir requests (y encolarlas en caso de carga alta) sino que también debe procesarlas y finalmente encargarse de devolverlas, es claro que el servicio sincrónico se saturara más rápidamente que el servicio asincrónico, cuya única responsabilidad es recibir requests y redireccionarlas a otros componentes del sistema. Además un servicio sincrónico puede tener más oscilaciones en los tiempos de respuesta ya que depende de la disponibilidad de recursos en el momento de recibir un request.

Para llegar a la conclusión de cuál de los dos servicios es el sincrónico y cual el asincrónico nos prestaremos con los 2 siguientes gráficos de la Sección 1:
Gráfico 1 (en Sección 1.3.2): El cual muestra un test de carga sobre la bbox1 con los parámetros descritos en la sección apropiada.
Gráfico 2 (en Sección 1.3.3): El cual muestra un test de carga sobre la bbox2 con los parámetros similares a los del Gráfico 1, sólo con un mayor nivel de carga debido a que bbox2 soporta (muchas) más requests antes de perder servicio.

En el gráfico 1 vimos que /bbox1 comienza a presentar problemas cuando comienza a recibir 6.3 r/s y al superar dicho umbral se pierden la totalidad de los requests.

Por otro lado en el gráfico 2 mostramos que /bbox2 comienza a tener inconvenientes al llegar a una tasa de recepción de 540 r/s y que al superarla llegamos a pérdidas de 25% de los requests.

Además vimos en la sección 1.4 en nuestra métrica propia que el /bbox1 tiene tiempos de respuesta variables mientras que el /bbox2 es más estable.

Por el comportamiento analizado, vimos que bbox1 se satura antes que bbox2 y además tiene tiempos de respuesta más inestables, lo que nos permite inferir que bbox1 es el servicio sincrónico, el cual al recibir requests las procesa (por medio de una cantidad de workers la cual aún es una incógnita) para luego devolverlas, mientras que el bbox2 al ser asincrónico, una vez que recibe requests se encarga únicamente de redireccionarlas a otro componente, lo que permite que soporte un mayor número de requests por segundo.

Endpoint	Tipo
bbox1	sincrónico
bbox2	asincrónico

2.2 Cantidad de workers (en el caso sincrónico)

El servicio sincrónico está implementado con una cantidad de workers. Deberán buscar algún indicio sobre cuál es esta cantidad.

Habiendo determinado en la sección 1.3.2 que el /bbox1 comienza a sufrir problemas de performance al superar los 6 r/s y de la sección sabemos que se trata de un endpoint sincrónico. En este apartado intentaremos determinar la cantidad de workers que utiliza basándonos en lo que ya hemos analizado y en otras pruebas para confirmarlo.

El primer indicio encontrado es justamente que hasta los 6 r/s el tiempo de respuesta es el mismo, entonces realizamos pruebas con las herramientas time y ab para simular envíos en paralelo de requests y analizar los tiempos en los que terminan.

Utilizaremos el siguiente comando:

```
time ab -n 13 -c 7 http://localhost:5555/bbox1
```

Cambiando el parámetro n (cantidad de requests totales) y c (cantidad de requests en paralelo) hacemos las siguientes mediciones:

n	c	Tiempo total (s)
3	3	1
6	6	1
7	6	2
7	7	2
8	6	2
12	6	2
13	6	3
18	6	3

Vemos que el tiempo se amplía en un segundo cada vez que superamos una cantidad de requests múltiplo de 6, por lo que concluímos que el endpoint /bbox1 tiene 6 workers.

2.3 Demora en responder

Cada servicio demora un tiempo en responder, que puede ser igual o distinto entre ellos. Deberán obtener este valor para cada uno.

- */bbox1*: Como bien marcamos en la sección 1.3.2: “Haciendo requests individuales vemos un tiempo de respuesta de un poco menos de 1000 ms…”.
  Otra cuestión es la vista en el inciso 1 de esta sección donde usamos el tiempo de respuesta de cada servicio bajo un cierto nivel de estrés en los servicios para

determinar cuál era el sincrónico y cual el asincrónico, al ser un servicio sincronico, bbox1 tiene un tiempo mayor bajo carga que bbox2. También vimos en el apartado anterior que el tiempo de respuesta se mantiene en la medida que los requests no superen los 6 workers.

- */bbox2*: Como bien marcamos en la sección 1.3.3: “Haciendo requests individuales vemos un tiempo de respuesta de un poco menos de 1500 ms…”. De todas maneras, aunque para un request individual bbox2 responda a las requests con mayor latencia, es claro que al ser un servicio asíncrono, bbox2 tendrá una capacidad de carga para recibir requests mayor que bbox1. Esto quiere decir que si comenzamos a saturar bbox1 con una cantidad de requests, bbox2 comenzará a tener un mejor tiempo de respuesta relativo a bbox1.

Endpoint	**Demora en responder** (requests individuales)	**Requests en paralelo** (sin degradar la performance)
bbox1	~1000 ms	6
bbox2	~1500 ms	540

Sección 3

Caso de Estudio - Sistema de inscripciones

En esta sección simularemos el comportamiento bajo ciertas condiciones de carga de un sistema de inscripciones de materias de una facultad, tomando como base el sistema SIU Guaraní utilizado en FIUBA.

Según un reporte de 2021 la Facultad de Ingeniería de la Universidad de Buenos Aires cuenta con una base de aproximadamente 8200 alumnos regulares que se inscriben cada cuatrimestre y la cantidad total de inscripciones por periodo es alrededor de 31500, esto es en promedio 3,84 materias por alumno.
La facultad cuenta con un sistema de prioridades que van desde el 1 al 120. El sistema asigna a cada alumno un turno a partir del cual pueden iniciar su proceso de inscripción dentro del periodo de inscripciones hasta el final del mismo.
El periodo de inscripciones tiene una duración de 5 días generalmente de lunes a viernes y los alumnos se pueden inscribir desde las 9hs hasta las 21hs.
Se comienza a partir de los alumnos con prioridad 1 (la más alta) y cada media hora se habilita la inscripción a los alumnos de la prioridad siguiente.

Por lo que podemos esperar que en un turno hayan 280 inscripciones que si se distribuye uniformemente en el turno equivaldría a 0.16 r/s, pero pueden ocurrir hasta 70 simultáneamente ya que todos los alumnos pueden hacerlo al mismo tiempo.

3.1 Hipótesis

En base a los datos anteriormente mencionados sobre el sistema de inscripciones de FIUBA y algunas mediciones de tiempos de inscripción se hacen las siguientes consideraciones sobre el sistema nuestro sistema.

- La cantidad de alumnos regulares que se inscriben en un periodo de inscripciones es de 8200
- Cada alumno se anota en 4 materias.
- Por cada turno de inscripción se espera un mínimo de 70 alumnos inscribiéndose a materias
- El tiempo que le toma al alumno desde terminar de loguearse hasta seleccionar una carrera es de 10 segundos
- El tiempo que le toma al alumno para ver las materias en que está inscripto es de 15 segundos.
- El tiempo utilizado para ver la lista de materias disponible e inscribirse toma en promedio de 70 segundos.
- El tiempo que le toma al alumno para cerrar la sesión luego de haberse inscripto a las materias es de 10 segundos.
- El flujo de un alumno inscribiéndose a las materias será el siguiente
    1. El alumno Inicia sesión en el sistema.
    2. Selecciona una carrera.
    3. Se Inscribe a materias siguiendo este flujo
        a. Ve la lista de materias en las que está inscripto
        b. Ve la lista de materias disponibles
        c. Se Inscribe en una materia.
    4. Cierra sesión.

3.2 Configuraciones del sistema

Para nuestro análisis vamos a tomar una configuración de un sistema que cuenta con un solo nodo (single worker) que procesa todas las peticiones de los usuarios.

Los endpoints disponibles para realizar el proceso de inscripcion seran los siguientes

Endpoint	**Descripción**
/login	Permite al usuario loguearse en el sistema
/logout	Termina la sesión del usuario

/materias	Devuelve el listado de materias en las que se encuentra inscripto el alumno
/materiasdisponibles	Devuelve el listado de materias disponibles para inscribirse
/inscribirse	Permite al alumno inscribirse en una materia
/carrera	Permite al alumno seleccionar la carrera

3.3 Escenario a analizar

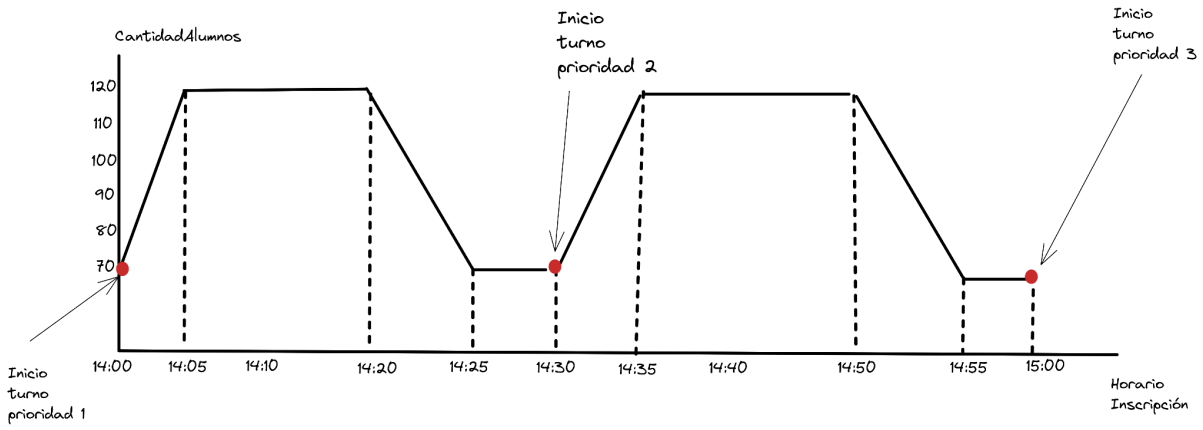
3.3.1 Resultados esperados

Para nuestro análisis definimos el siguiente escenario el cual posteriormente será ejecutado con Artillery.

Tomaremos en cuenta un periodo que va desde el inicio de la prioridad 1 pasando por la 2 y finalizando en el comienzo de la prioridad 3 lo que corresponde con un lapso de una hora de inscripción en el sistema.

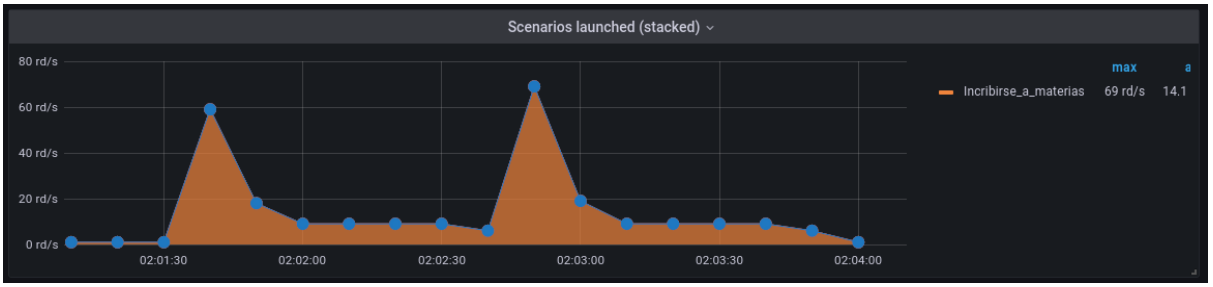
Vamos a partir de una base de 70 alumnos utilizando el sistema al momento del inicio del turno de la prioridad 1 y luego una carga gradual hasta llegar a los 120 inscribiéndose en simultáneo a las materias. Luego iniciará el turno de la siguiente prioridad aún habiendo alumnos de la prioridad 1 inscribiéndose, por lo tanto ingresan 80 nuevos alumnos, pero con prioridad 2 por lo que habrá procesamiento solapado.

Antes de iniciar el test de carga esperamos que se produzcan algunos picos en los momentos donde haya más cantidad de alumnos inscribiéndose que puedan llegar a saturar el sistema y afectar su performance.

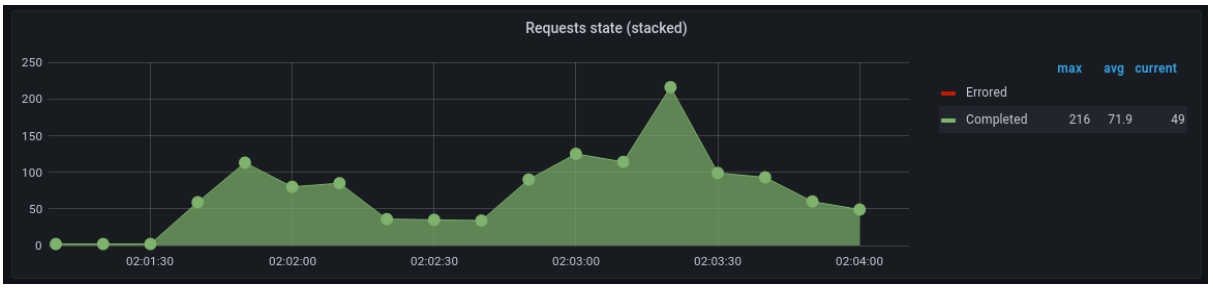


3.3.2 Ejecución

En cada inicio de turno de prioridad un gran porcentaje de alumnos ingresa en los primeros minutos generando un pico inicial y luego comienza a descender bruscamente hasta llegar al final del turno, unos pocos siguen ingresando a realizar su inscripción.



En la siguiente gráfica de los requests completados vemos que los picos se encuentran corridos respecto del arribo de usuarios ya que estos ejecutan varios endpoints con tiempos de espera en los que deciden entre las opciones. Esto lleva a que en el segundo turno se procesen más requests debido a los usuarios del primer turno que no han completado sus inscripciones y continúan haciéndolo. Esto explica porque la facultad decide separar los turnos por intervalos de 30 minutos, ya que a intervalos más cortos habrían en el sistema simultáneamente alumnos de varios turnos sometiendo al sistema a una carga no esperada.



Este escenario de prueba nos permite observar los “puntos débiles” del sistema actual y sus posibles puntos de mejora, entendiendo que no sólo es un tema de escalar horizontalmente los recursos sino también de cómo generar una correcta distribución de prioridades dado el comportamiento de los alumnos a la hora de inscribirse, para así tener una carga equilibrada a lo largo de todo el período de inscripciones.

Conclusiones

- Pudimos aprender sobre muchas tecnologías y herramientas nuevas para la mayoría de nosotros. Si bien alguna vez pudimos haber usado Docker o Nginx para algo, herramientas como cAdvisor, Graphite, Grafana y Artillery no las habíamos usado anteriormente. Además de entender el uso de cada una por separado, es importante haberlas utilizado juntas como en el caso de cAdvisor, Graphite y Grafana bastante acopladas y Artillery para generar carga simulando usuarios virtuales.
- Con todas las herramientas aprendidas, pudimos caracterizar un servicio web desde afuera, sin tener pistas de cómo está implementado internamente. Esto es muy poderoso de lograr, evaluando tiempos de respuestas y cómo responde ante distintos niveles de carga.

- En la sección 3 tuvimos la posibilidad de generar un escenario de prueba ficticio para un sistema que todos, como estudiantes, hemos sufrido como lo es el SIU Guaraní. Pudimos ponernos en el lugar de quienes mantienen este sistema y entender las complicaciones que genera el sistema de prioridades y su distribución para no generarle una carga excesiva al sistema.
- Las herramientas aprendidas nos permitirán analizar ciertos atributos de calidad de sistemas y darnos más información sobre qué se puede mejorar o dónde enfocar los esfuerzos según las necesidades del cliente.