

Trabajo Práctico 1

Arquitectura de Software (75.73)

2c 2022
Facultad de Ingeniería
Universidad de Buenos Aires

Alumno	Padrón
Franco Alejandro Primerano Lomba	106004
Nahuel Spiguelman	104644
Alejo Acevedo	99146
Juan Ignacio Reil Luz	106056

Sección 1	3
Endpoints de la aplicación	3
Infraestructura	3
Un solo nodo worker: La aplicación solo corre en un nodo.	3
Varios nodos workers replicados: La aplicación corre en 3 nodos.	4
Escenarios de prueba	4
Ping	4
Para un solo nodo:	4
Para nodos replicados:	6
Nodo 1:	7
Nodo 2:	7
Nodo 3:	8
Conclusión:	8
Heavy	8
Para un solo nodo:	9
Para nodos replicados:	11
Nodo 1:	12
Nodo 2:	12
Nodo 3:	12
Conclusión:	13
Bbox A	14
Para un solo nodo:	14
Para nodos replicados:	16
Nodo 1:	17
Nodo 2:	18
Nodo 3:	18
Conclusión:	18
Bbox B	19
Para un solo nodo:	19
Para nodos replicados:	21
Conclusión:	23
Sección 2	23
Analisis y caracterizacion:	23
Sincrónico/Asincronico:	23
Cantidad de workers:	24
Demora en responder:	25
Sección 3	25
Hipotesis:	26
Implementación:	26
Graficos:	27
Conclusion:	28

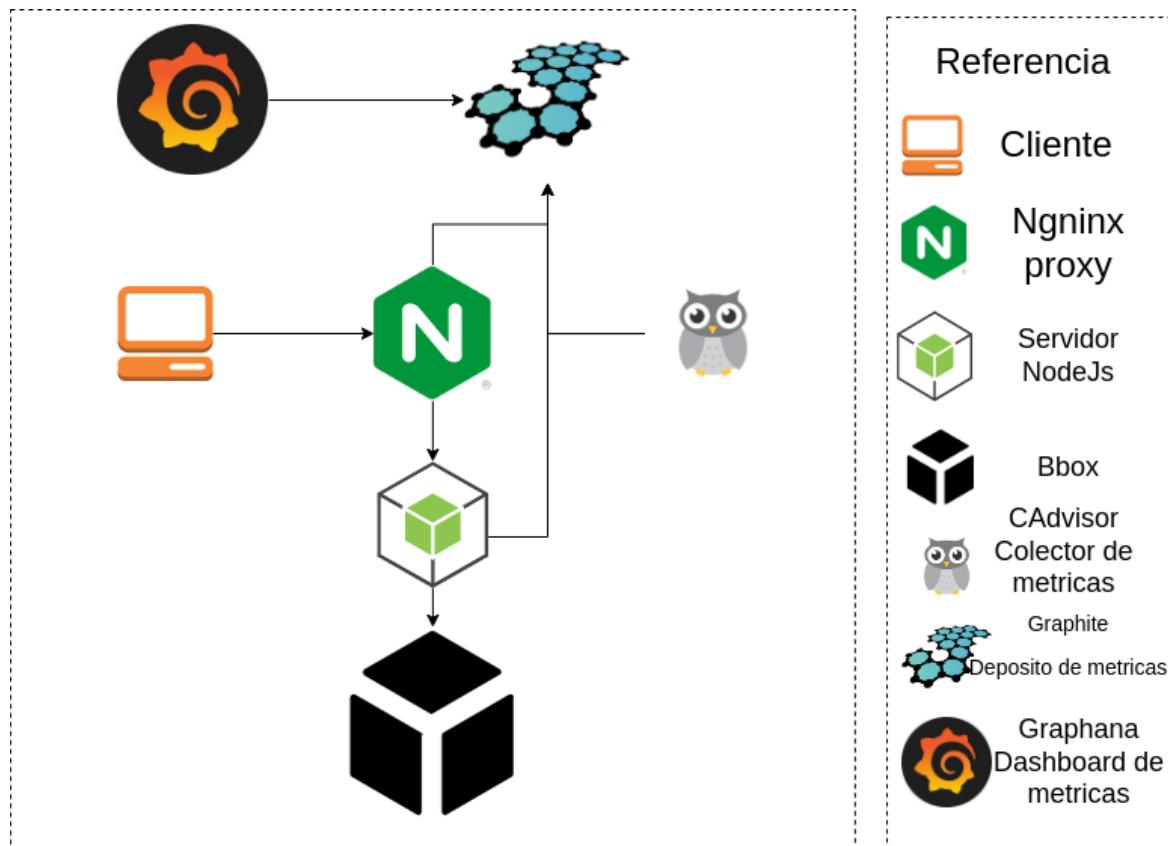
1. Sección 1

a. Endpoints de la aplicación

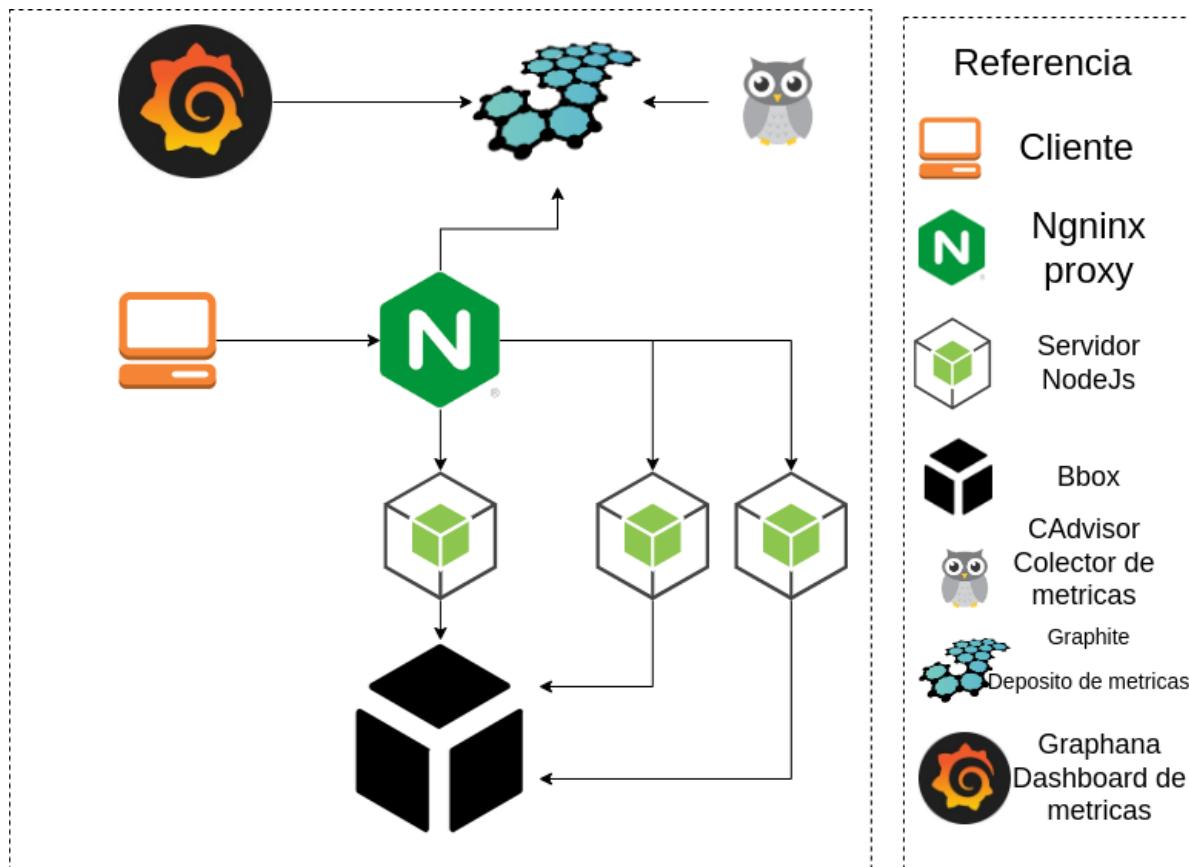
/ping	Respuesta simple HTTP status 200
/heavy	Loop de cierto tiempo (lento y de alto procesamiento)
/bbox/a	Respuesta al servicio 1 provisto por la cátedra
/bbox/b	Respuesta al servicio 2 provisto por la cátedra

b. Infraestructura

- Un solo nodo worker: La aplicación solo corre en un nodo.



- Varios nodos workers replicados: La aplicación corre en 3 nodos.



c. Escenarios de prueba

Los escenarios de prueba que utilizaremos para comparar ambas infraestructuras son los siguientes:

- **Spike Test:** En este escenario enviaremos pocas requests por períodos alternandolas con períodos de alta cantidad de requests.
- **Load Test:** Para este escenario enviaremos una cantidad creciente de requests en un período de tiempo determinado.

Ping

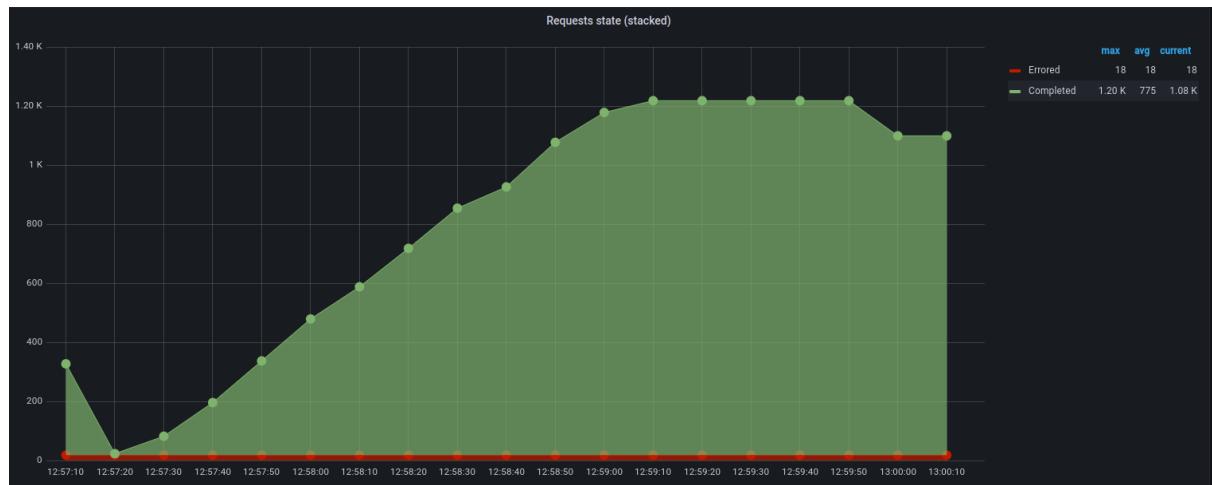
Para comprar los servidores a través de este endpoint utilizaremos el escenario de prueba Load, empezando en 1 r/s y estableciendo el ramp To de artillery a 10000 r/s.

Load testing para un solo nodo:

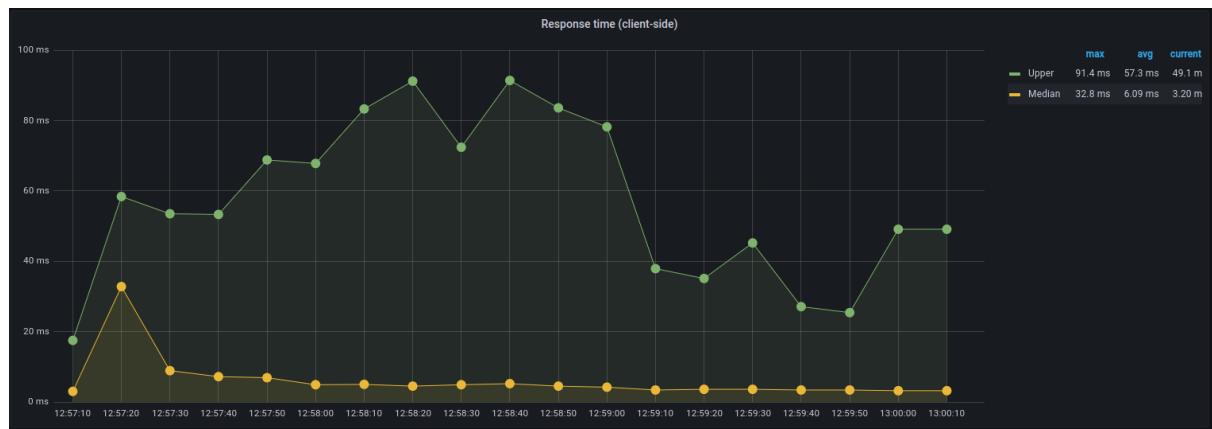
Scenarios launched:



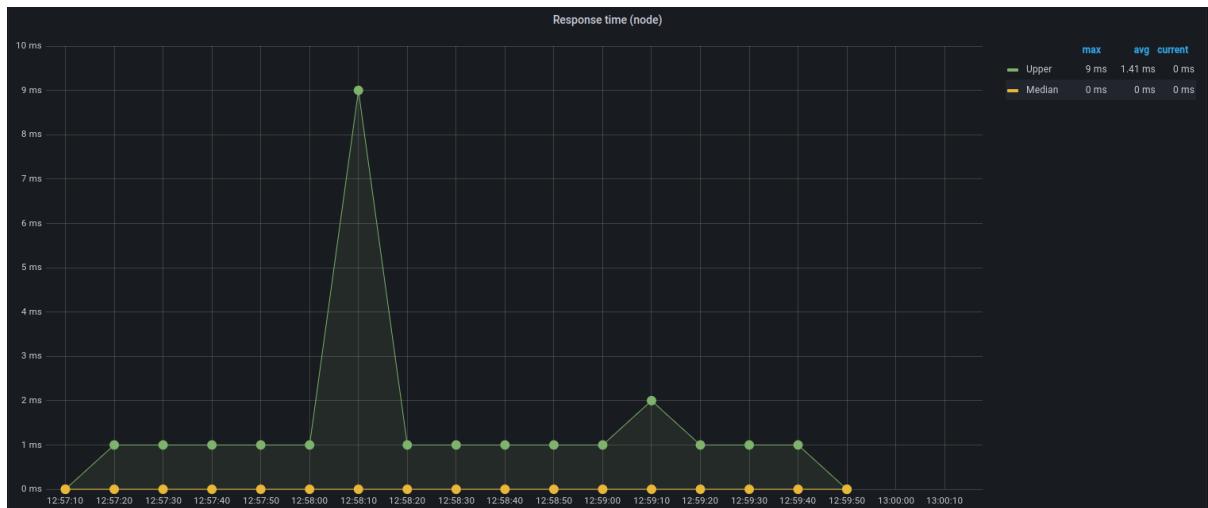
Requests status:



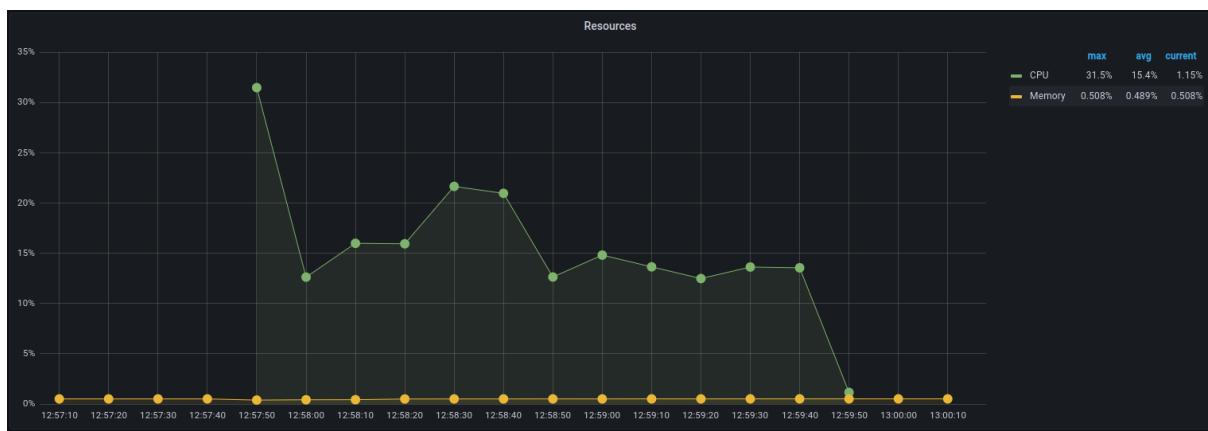
Response time client:



Response time node:



Recursos:

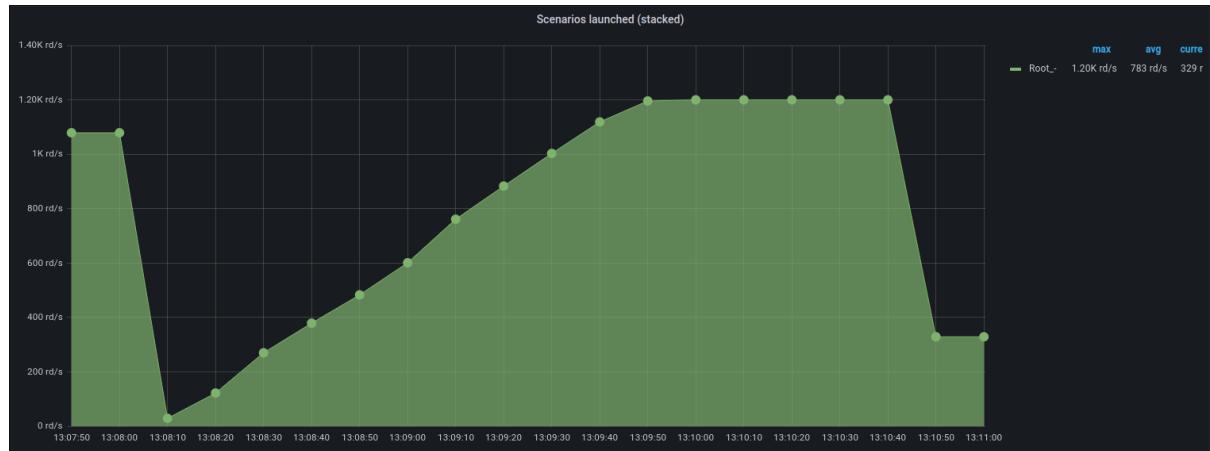


Summary:

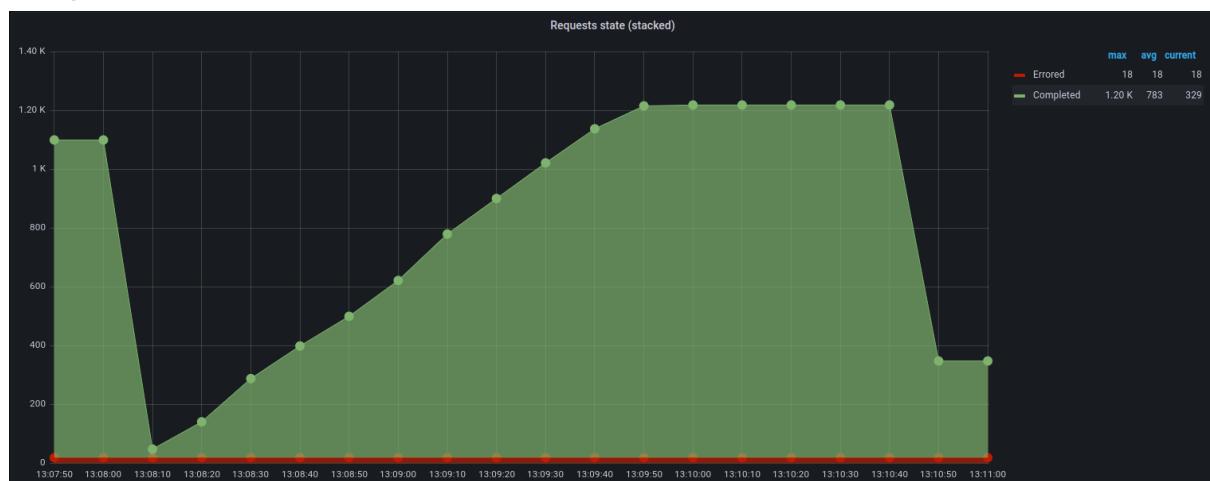
http.codes.200: ..	13341
http.request_rate: ..	80/se
http.requests: ..	13341
http.response_time:	
min: ..	0
max: ..	53
median: ..	2
p95: ..	7
p99: ..	13.9
http.responses: ..	13341
vusers.completed: ..	13341
vusers.created: ..	13341
vusers.created_by_name.Root (/): ..	13341
vusers.failed: ..	0
vusers.session_length:	
min: ..	1.7
max: ..	91.4
median: ..	3.9
p95: ..	21.1
p99: ..	40.9

Load testing para nodos replicados:

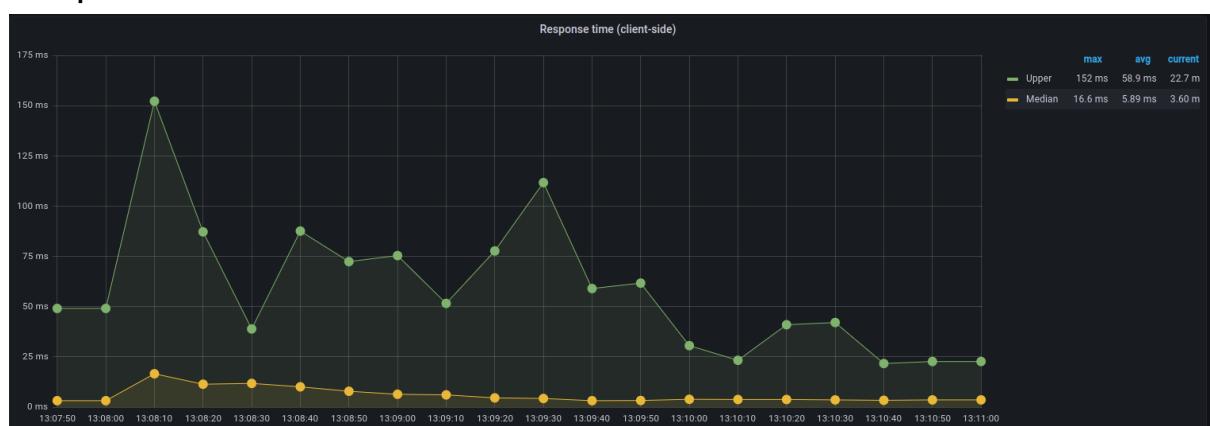
Scenarios launched:



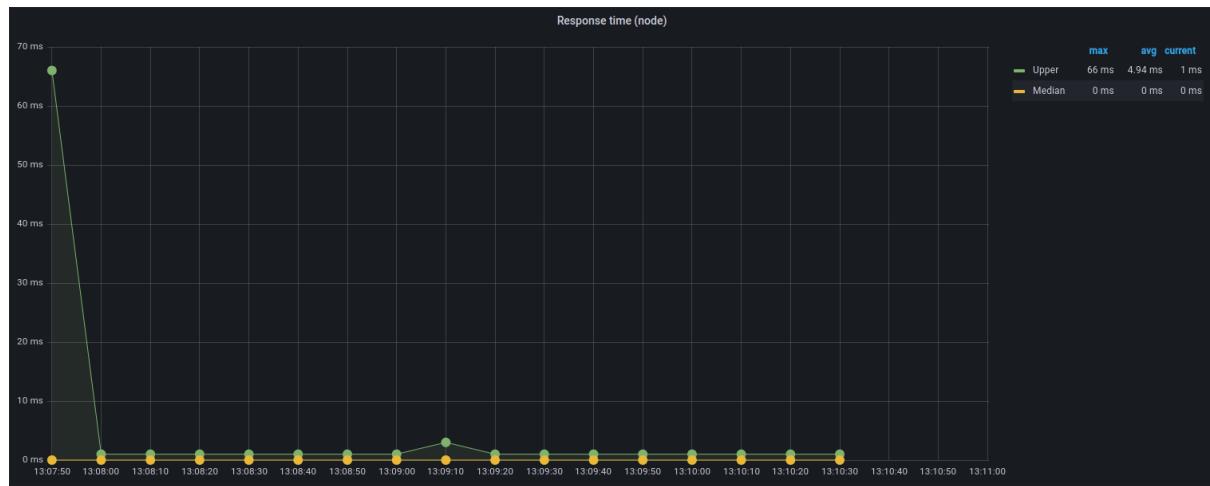
Requests status:



Response time client:



Response time node:



Nodo 1:

Resources:



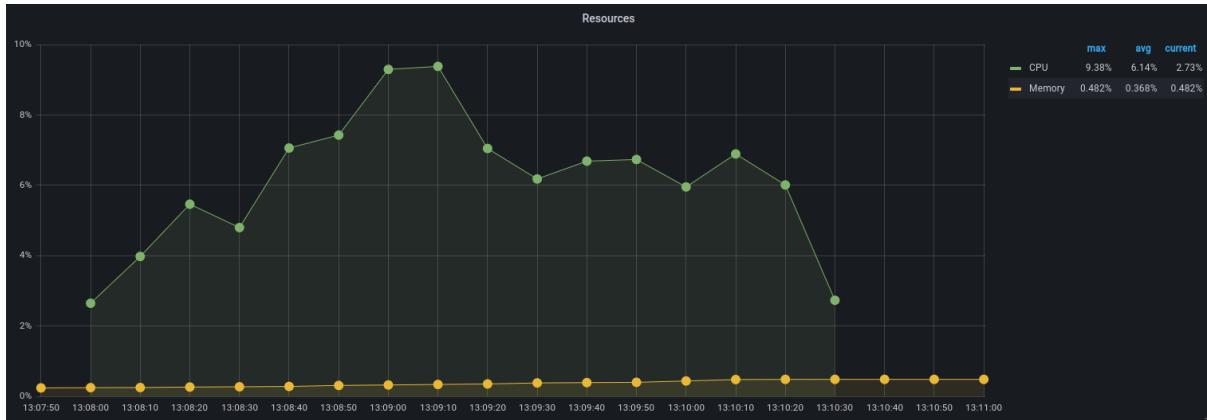
Nodo 2:

Resources



Nodo 3:

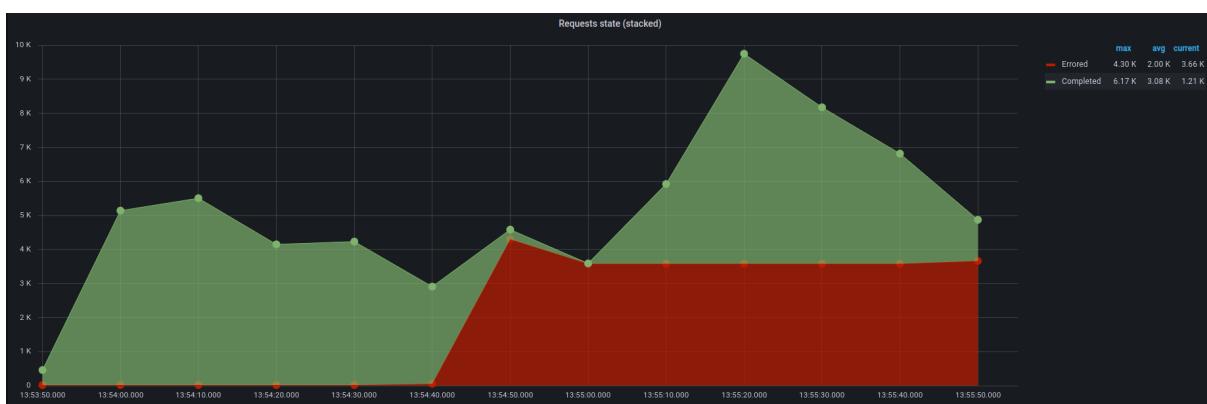
Resources:

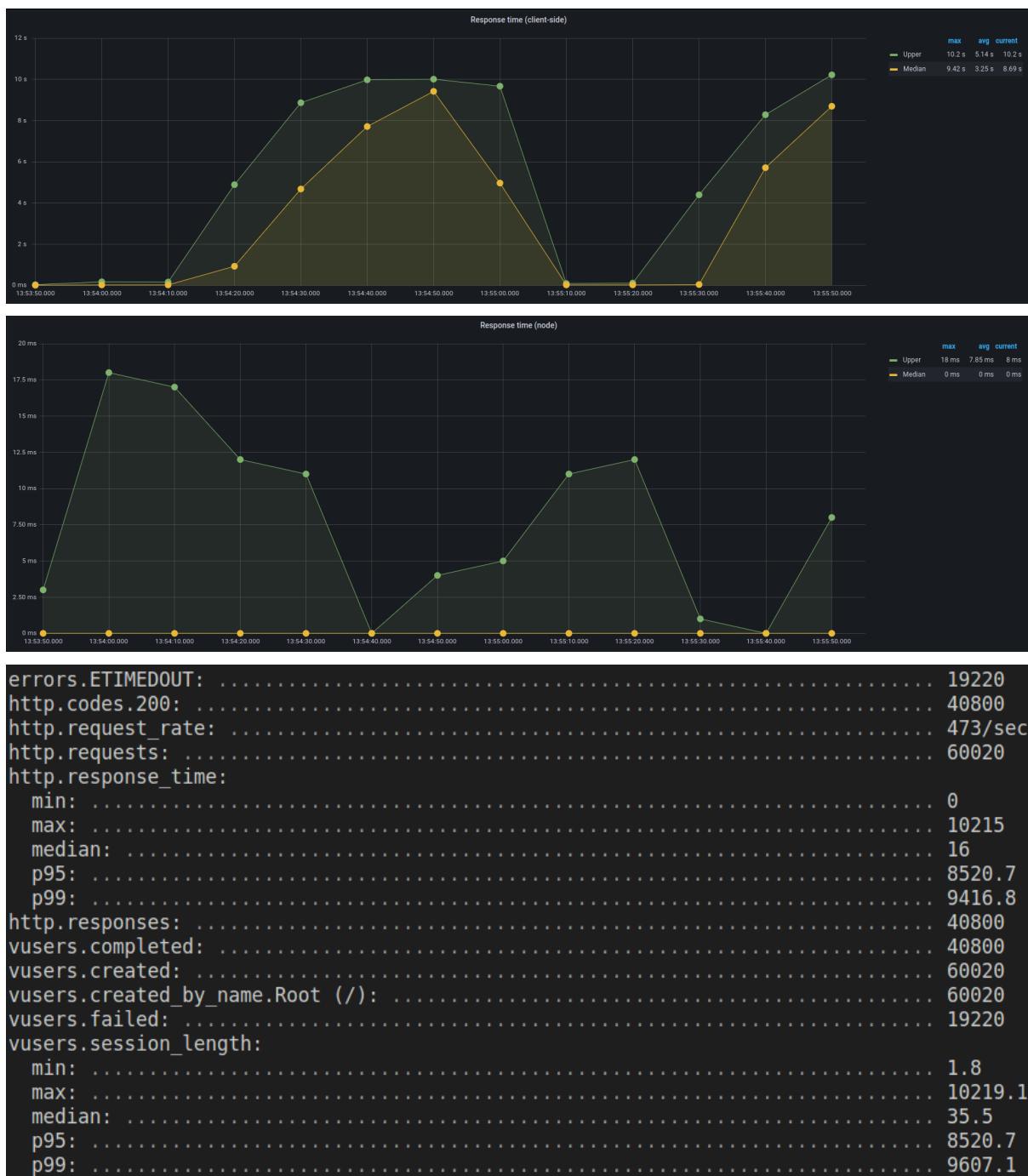


Summary:

http.codes.200:	13175
http.request_rate:	83/sec
http.requests:	13175
http.response_time:	
min:	0
max:	87
median:	2
p95:	7
p99:	13.9
http.responses:	13175
vusers.completed:	13175
vusers.created:	13175
vusers.created_by_name.Root (/):	13175
vusers.failed:	0
vusers.session_length:	
min:	2
max:	152.2
median:	3.9
p95:	18
p99:	34.1

Spike para nodos replicados:





Conclusión:

Lo más importante a observar es que el request ping no requiere grandes costos de ningún tipo, por lo cual en general funcionará bien con una gran cantidad de usuarios. Los tiempos de respuesta no se ven comprometidos con una gran cantidad de solicitudes por segundo, podríamos decir que su performance hasta el momento es buena.

Algo interesante a ver es que en los momentos donde hay crecimiento en la cantidad de requests por segundo, los máximos en response time se ven mas grandes, esto podría ser que si bien el servicio node responde bien y en tiempo constante, nginx puede tener algún delay o también overhead en los tiempos de respuesta debido a la tarea de redirigir requests y responses en un tiempo ínfimo, dado que el servicio tarda prácticamente nada. En cuanto a

la comparación entre nodo único y nodos replicados podríamos inferir algún tipo de conclusión con los gráficos pero viendo el summary provisto por artillery podemos concluir que tanto con un nodo como con replicados en general funciona bien. Más precisamente el 99% de los requests han tardado lo mismo y tienen la misma media. En valores tan chicos, el gráfico y el detalle del mismo puede arrastrar algún error de captura de anteriores pruebas.

Donde sí se puede ver una mejora lineal es en el uso de los recursos, donde al tener 3 nodos replicados el uso de los mismos se reduce 3 veces en cada nodo.

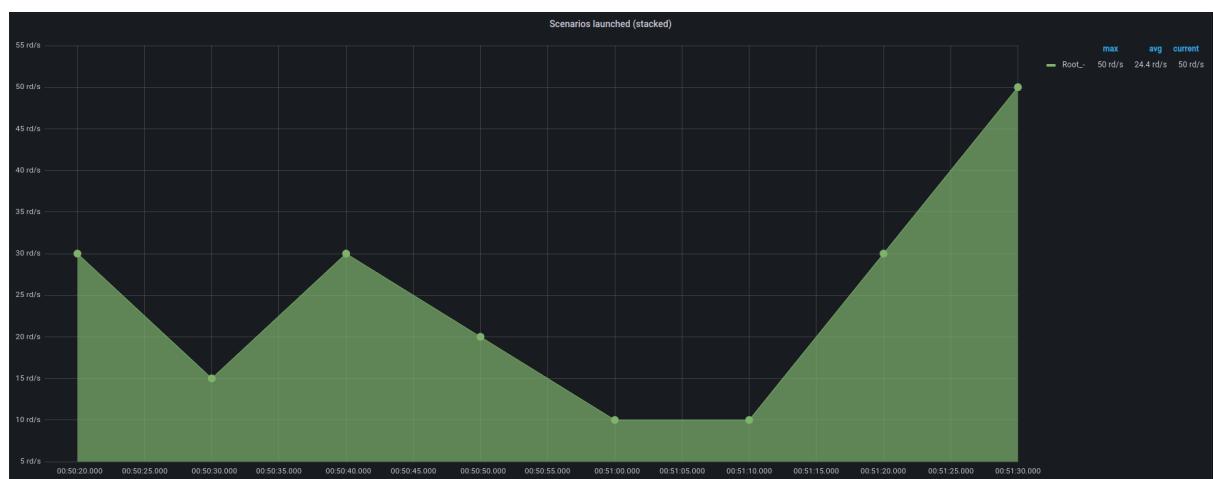
Además, hicimos spike testing para ver hasta donde aguantaba la carga el servidor. Si bien lo cierto es que resuelve todos los requests, a partir de un cierto punto con una cierta carga, empieza a resolver los requests con una cierta demora y como consecuencia de esta demora, se puede llegar a ver comprometida la performance ya que los tiempos de cara al cliente aumentan considerablemente. La mediana de los tiempos registrados por la aplicación node sigue siendo cercana a 0 ms, con picos de algunos ms, pero por los tiempos percibidos por artillery llegan hasta 10s que es justamente lo que genera esa sensación de timeout y donde todo parece que se cae. Luego cuando los requests empiezan a bajar, los tiempos de respuesta se acomodan y vuelve a funcionar normalmente.

Heavy

Para comprar los servidores a través del endpoint heavy utilizaremos el escenario spike, con los picos siendo de 100 r/s y los valles de 1 r/s

Para un solo nodo:

Scenarios launched:



Requests status:



Response time client:



Response time node:



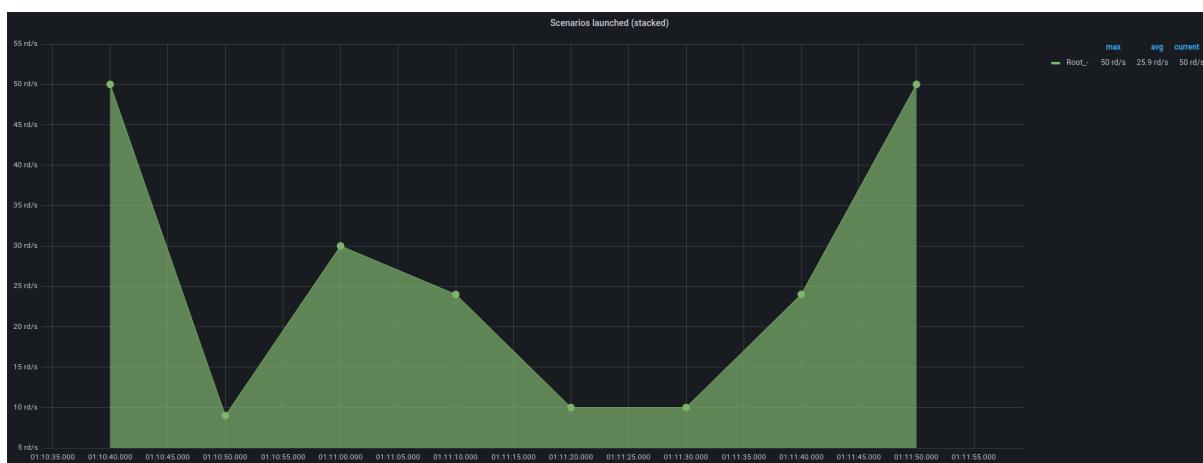
Recursos:



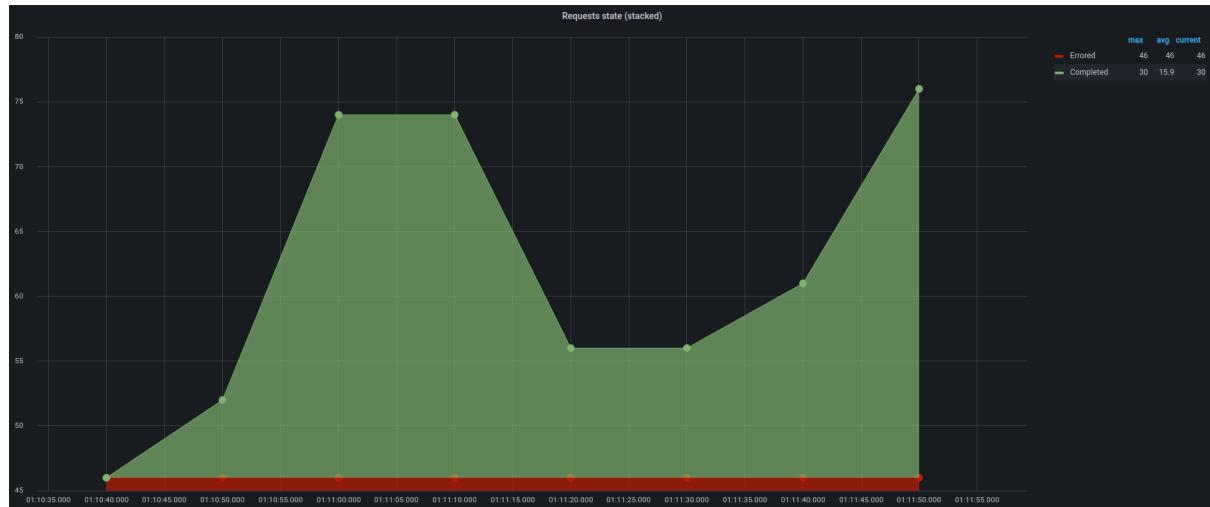
errors.ETIMEDOUT:	177
http.codes.200:	13
http.request_rate:	3/sec
http.requests:	190
http.response_time:	
min:	1005
max:	9061
median:	5065.6
p95:	9047.6
p99:	9047.6
http.responses:	13
vusers.completed:	13
vusers.created:	190
vusers.created_by_name.Root (/):	190
vusers.failed:	177
vusers.session_length:	
min:	1042.7
max:	9070
median:	5065.6
p95:	9047.6
p99:	9047.6

Para nodos replicados:

Scenarios launched



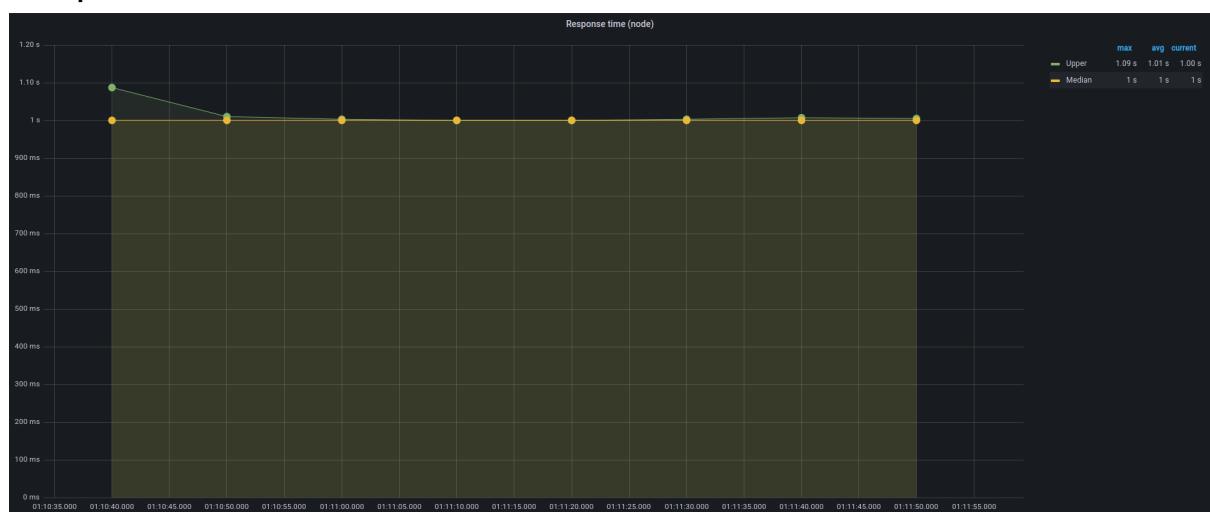
Requests status:



Response time client:



Response time node:



Nodo 1:

Resources:



Nodo 2:

Resources:



Nodo 3:

Resources:



errors.ETIMEDOUT:	35
http.codes.200:	155
http.request_rate:	3/sec
http.requests:	190
http.response_time:	
min:	1001
max:	9594
median:	2018.7
p95:	8520.7
p99:	9047.6
http.responses:	155
vusers.completed:	155
vusers.created:	190
vusers.created_by_name.Root (/):	190
vusers.failed:	35
vusers.session_length:	
min:	1002.5
max:	9601
median:	2018.7
p95:	8520.7
p99:	9047.6

Conclusión:

Para estos dos escenarios hicimos un for que tarde 1000ms en responder dando como resultado un endpoint con una demora generada por uso del hardware. El mínimo tiempo de respuesta brindado por cualquiera de los servicios independientemente de la cantidad de nodos, será el tiempo mencionado anteriormente más algún overhead generado por nginx o artillery. En el escenario con solo un nodo podemos ver como la mayoría de las requests no fueron satisfactorias o por lo menos, que arrojaron un timeout superior al de artillery, lo cual tiene sentido por ser un endpoint que demanda una cantidad de procesamiento considerable y encola sus requests entrantes por lo tanto no puede paralelizar requests dentro de una misma instancia. Esto se puede ver bajo los gráficos de recursos (todos los nodos llegan cerca del 100% de utilización). Una imagen que deja en evidencia lo recién comentado es la siguiente:

```
2022-10-13T14:45:09.633072419Z 172.22.0.1 - - [13/Oct/2022:14:45:09 +0000] "GET /app/heavy HTTP/1.1" 200 5 "-" "Artillery (https://artillery.io)" "-"
2022-10-13T14:45:10.640306705Z 172.22.0.1 - - [13/Oct/2022:14:45:10 +0000] "GET /app/heavy HTTP/1.1" 200 5 "-" "Artillery (https://artillery.io)" "-"
2022-10-13T14:45:11.642953107Z 172.22.0.1 - - [13/Oct/2022:14:45:11 +0000] "GET /app/heavy HTTP/1.1" 200 5 "-" "Artillery (https://artillery.io)" "-"
2022-10-13T14:45:12.643843985Z 172.22.0.1 - - [13/Oct/2022:14:45:12 +0000] "GET /app/heavy HTTP/1.1" 200 5 "-" "Artillery (https://artillery.io)" "-"
2022-10-13T14:45:13.645958478Z 172.22.0.1 - - [13/Oct/2022:14:45:13 +0000] "GET /app/heavy HTTP/1.1" 200 5 "-" "Artillery (https://artillery.io)" "-"
2022-10-13T14:45:14.647034737Z 172.22.0.1 - - [13/Oct/2022:14:45:14 +0000] "GET /app/heavy HTTP/1.1" 200 5 "-" "Artillery (https://artillery.io)" "-"
2022-10-13T14:45:15.655042989Z 172.22.0.1 - - [13/Oct/2022:14:45:15 +0000] "GET /app/heavy HTTP/1.1" 200 5 "-" "Artillery (https://artillery.io)" "-"
2022-10-13T14:45:16.655890685Z 172.22.0.1 - - [13/Oct/2022:14:45:16 +0000] "GET /app/heavy HTTP/1.1" 200 5 "-" "Artillery (https://artillery.io)" "-"
2022-10-13T14:45:17.657070962Z 172.22.0.1 - - [13/Oct/2022:14:45:17 +0000] "GET /app/heavy HTTP/1.1" 200 5 "-" "Artillery (https://artillery.io)" "-"
2022-10-13T14:45:18.657940940Z 172.22.0.1 - - [13/Oct/2022:14:45:18 +0000] "GET /app/heavy HTTP/1.1" 200 5 "-" "Artillery (https://artillery.io)" "-"
2022-10-13T14:45:19.659016491Z 172.22.0.1 - - [13/Oct/2022:14:45:19 +0000] "GET /app/heavy HTTP/1.1" 200 5 "-" "Artillery (https://artillery.io)" "-"
2022-10-13T14:45:20.659952879Z 172.22.0.1 - - [13/Oct/2022:14:45:20 +0000] "GET /app/heavy HTTP/1.1" 200 5 "-" "Artillery (https://artillery.io)" "-"
2022-10-13T14:45:21.660903541Z 172.22.0.1 - - [13/Oct/2022:14:45:21 +0000] "GET /app/heavy HTTP/1.1" 200 5 "-" "Artillery (https://artillery.io)" "-"
2022-10-13T14:45:22.512073178Z 172.22.0.1 - - [13/Oct/2022:14:45:22 +0000] "GET /app/heavy HTTP/1.1" 499 0 "-" "Artillery (https://artillery.io)" "-"
2022-10-13T14:45:23.513522679Z 172.22.0.1 - - [13/Oct/2022:14:45:23 +0000] "GET /app/heavy HTTP/1.1" 499 0 "-" "Artillery (https://artillery.io)" "-"
2022-10-13T14:45:23.497859562Z 172.22.0.1 - - [13/Oct/2022:14:45:23 +0000] "GET /app/heavy HTTP/1.1" 499 0 "-" "Artillery (https://artillery.io)" "-"
2022-10-13T14:45:23.499879932Z 172.22.0.1 - - [13/Oct/2022:14:45:23 +0000] "GET /app/heavy HTTP/1.1" 499 0 "-" "Artillery (https://artillery.io)" "-"
2022-10-13T14:45:23.500524878Z 172.22.0.1 - - [13/Oct/2022:14:45:23 +0000] "GET /app/heavy HTTP/1.1" 499 0 "-" "Artillery (https://artillery.io)" "-"
2022-10-13T14:45:24.499660741Z 172.22.0.1 - - [13/Oct/2022:14:45:24 +0000] "GET /app/heavy HTTP/1.1" 499 0 "-" "Artillery (https://artillery.io)" "-"
2022-10-13T14:45:24.503051318Z 172.22.0.1 - - [13/Oct/2022:14:45:24 +0000] "GET /app/heavy HTTP/1.1" 499 0 "-" "Artillery (https://artillery.io)" "-"
2022-10-13T14:45:24.504158151Z 172.22.0.1 - - [13/Oct/2022:14:45:24 +0000] "GET /app/heavy HTTP/1.1" 499 0 "-" "Artillery (https://artillery.io)" "-"
2022-10-13T14:45:25.497081554Z 172.22.0.1 - - [13/Oct/2022:14:45:25 +0000] "GET /app/heavy HTTP/1.1" 499 0 "-" "Artillery (https://artillery.io)" "-"
2022-10-13T14:45:25.498453823Z 172.22.0.1 - - [13/Oct/2022:14:45:25 +0000] "GET /app/heavy HTTP/1.1" 499 0 "-" "Artillery (https://artillery.io)" "-"
2022-10-13T14:45:25.504608834Z 172.22.0.1 - - [13/Oct/2022:14:45:25 +0000] "GET /app/heavy HTTP/1.1" 499 0 "-" "Artillery (https://artillery.io)" "-"
2022-10-13T14:45:26.498039156Z 172.22.0.1 - - [13/Oct/2022:14:45:26 +0000] "GET /app/heavy HTTP/1.1" 499 0 "-" "Artillery (https://artillery.io)" "-"
```

Donde podemos ver que a partir de una cierta cantidad de status code 200, empieza a arrojar errores 499 debido a que el cliente artillery empieza a cerrar las conexiones. Nunca tiene el suficiente tiempo de cerrar todas las viejas y procesar las nuevas, por lo cuál solo procederá exitosamente las primeras 13.

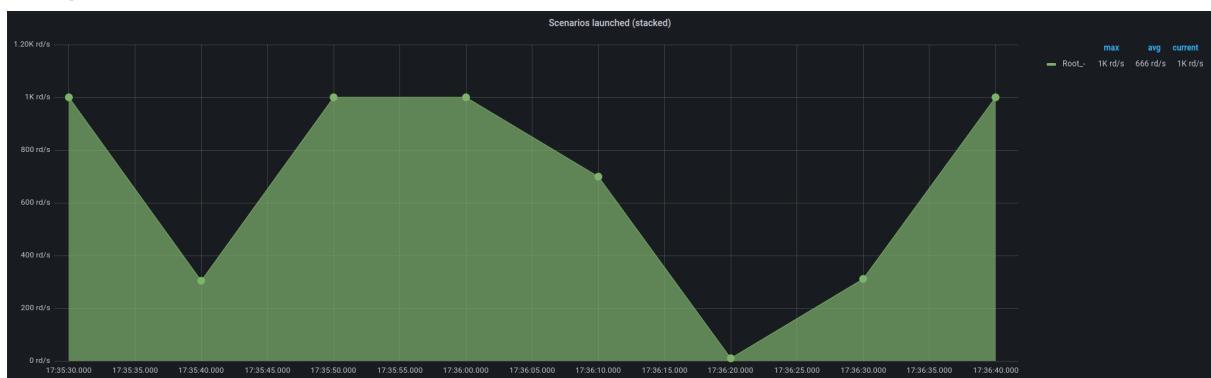
Lo que observamos es que para el caso de nodos replicados obtuvimos muchas más respuestas satisfactorias, los nodos llegaron a una capacidad bastante menor. Esto se debe a que las request son balanceadas entre varios nodos y al aumentar la performance no se sobrepasa el tiempo dado por el cliente artillery y percibe un tiempo de respuesta muchísimo más bajo con una tasa de recepción exitosa muchísimo más alta.

Bbox A

En este caso, para comparar este endpoint también utilizaremos el escenario spike. Hicimos pruebas con load testing y spike.

Spike para un solo nodo:

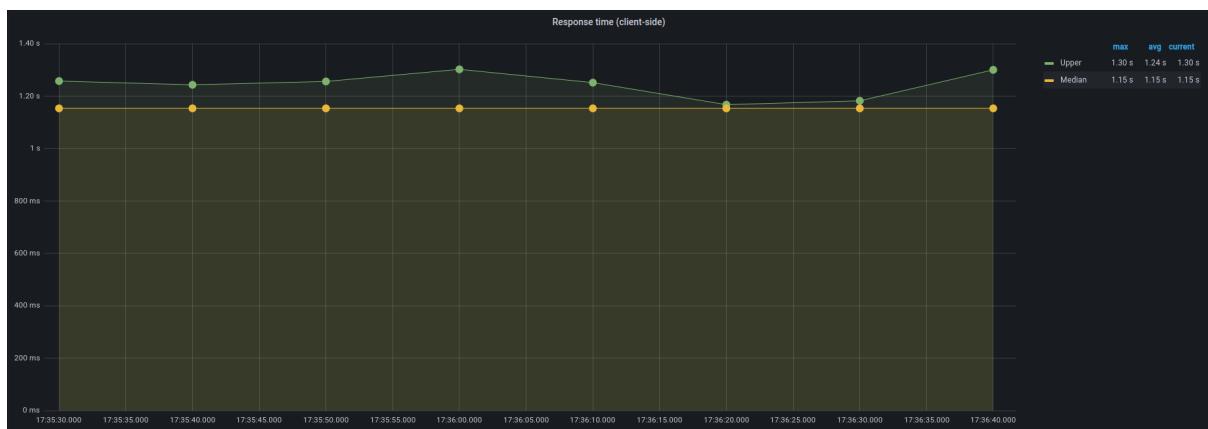
Requests launched:



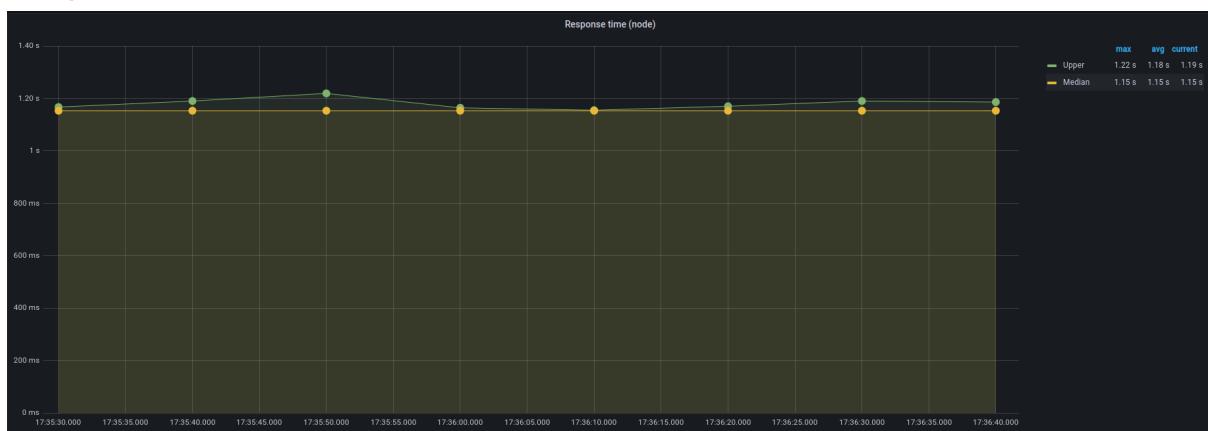
Requests status:



Request time client:



Request time server:



Resources:



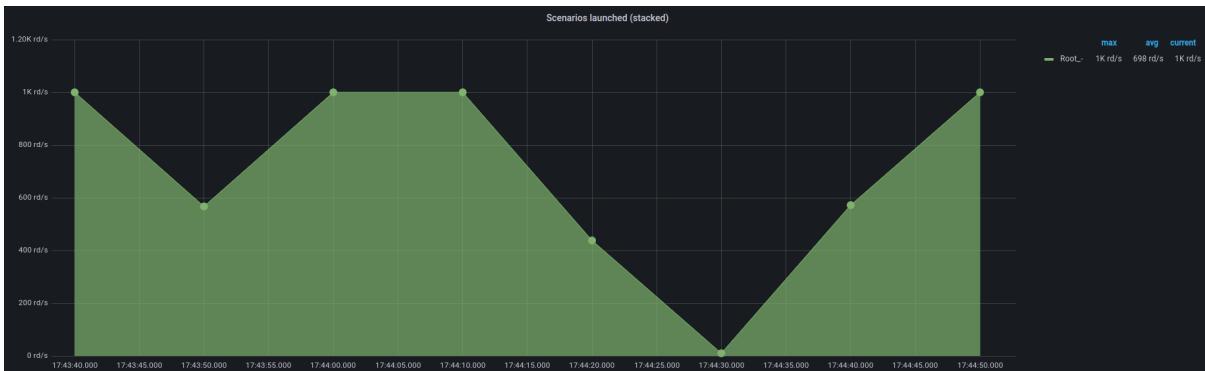
```

Summary report @ 17:36:49(-0300)
-----
http.codes.200: ..... 6020
http.request_rate: 93/sec
http.requests: ..... 6020
http.response_time:
  min: ..... 1132
  max: ..... 1298
  median: ..... 1153.1
  p95: ..... 1176.4
  p99: ..... 1200.1
http.responses: ..... 6020
vusers.completed: ..... 6020
vusers.created: ..... 6020
vusers.created_by_name.Root (/): ..... 6020
vusers.failed: ..... 0
vusers.session_length:
  min: ..... 1153.4
  max: ..... 1302
  median: ..... 1153.1
  p95: ..... 1200.1
  p99: ..... 1224.4

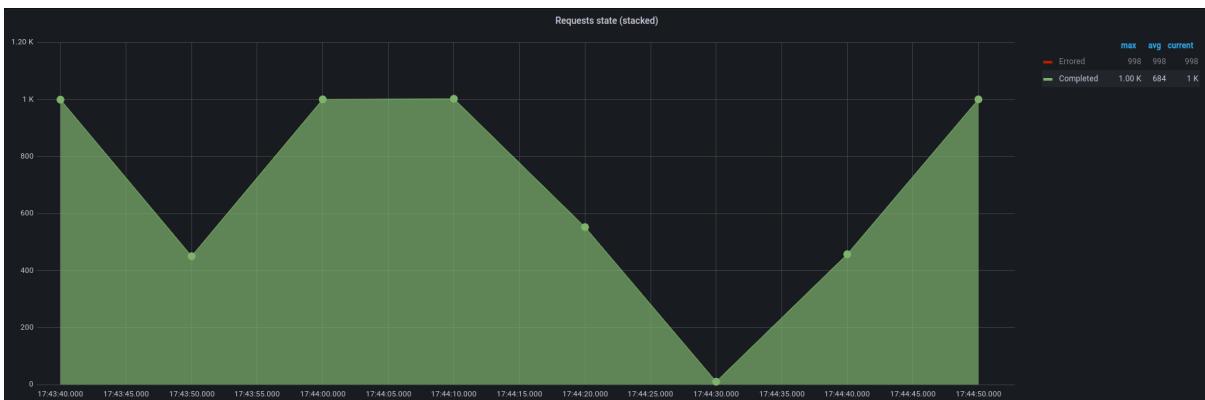
```

Spike para nodos replicados:

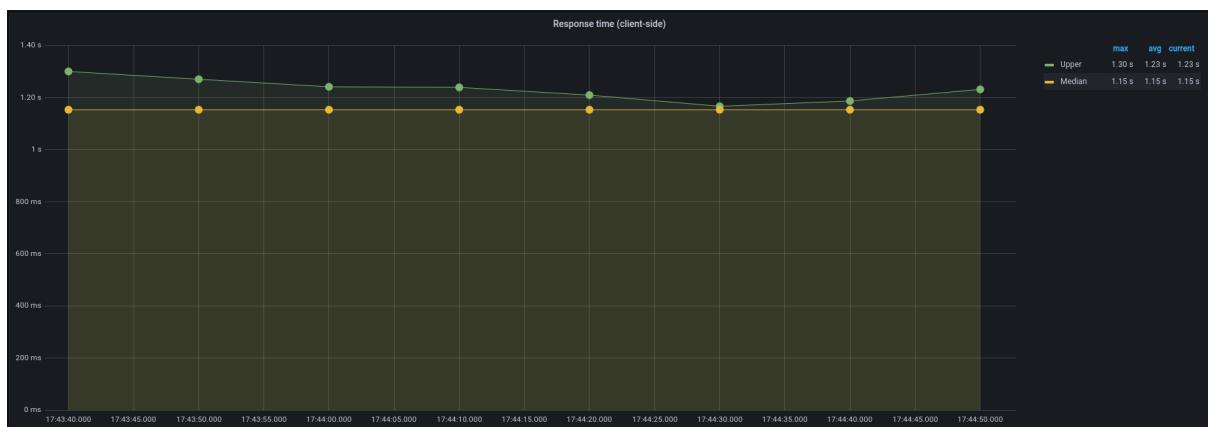
Scenarios launched:



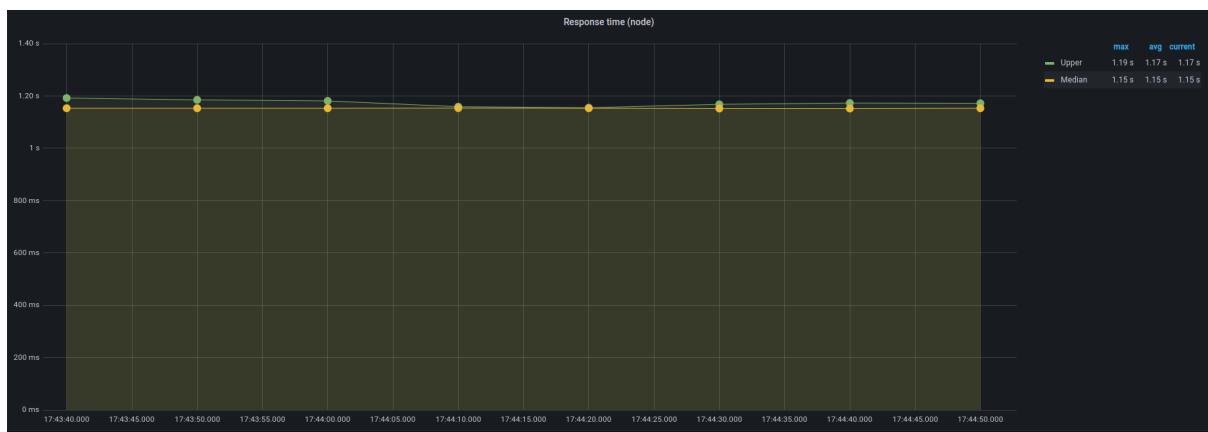
Requests state:



Response time client:



Response time server:



Nodo 1:

Resources:



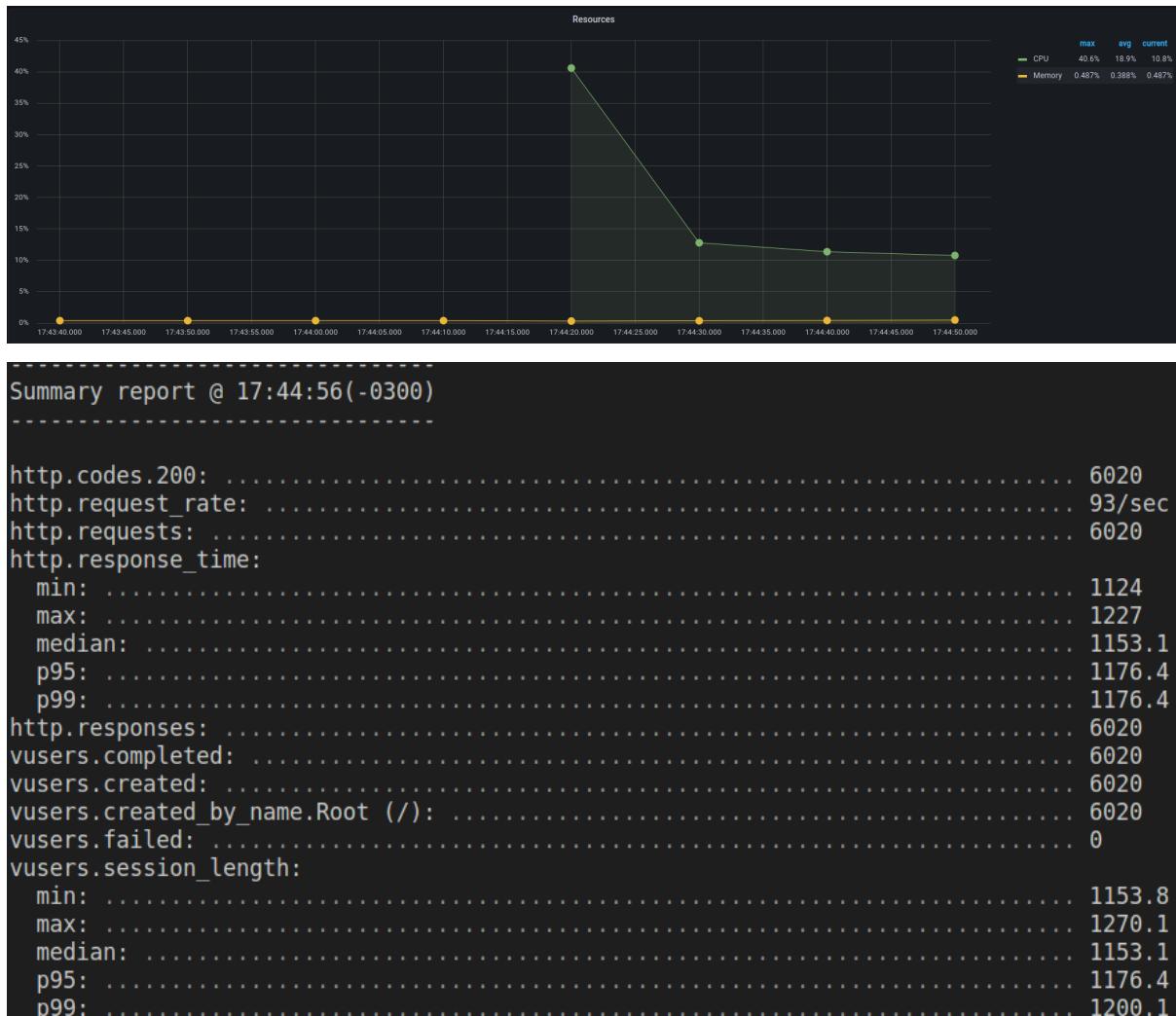
Nodo 2:

Resources:



Nodo 3:

Resources:



Load testing para un solo nodo:

Request state:



Conclusión:

El servicio bbox/a provisto por la cátedra tiene tiempos de respuesta constantes a incrementos repentinos en la cantidad de requests. Esta

capacidad de escalabilidad, que se ve con ambas infraestructuras, nos hace pensar que el servicio es asíncrono.

Un detalle importante a destacar es la escalabilidad horizontal provista por el servidor web cuando tenemos una cantidad determinada de nodos replicados, en este caso de tres nodos. Si bien tenemos la misma cantidad de requests bien resueltos porque el servidor no colapsa, podemos ver en las segundas imágenes como el caso de nodos replicados tiene una pendiente más pronunciada cuando termina la primera etapa del spike. Esto se debe a que la escalabilidad horizontal está bien aplicada y aumenta la performance.

En cuanto al response time vemos que tiene una correlación con el tiempo registrado mediante la métrica propia, tal como era esperado también.

Podemos ver como la media se mantiene casi igual, mientras que los máximos se ven afectados por el tiempo que tarda nginx en redireccionar las requests y responses.

Con respecto al uso de recursos, vemos un pico inicial que se lo atribuimos al estado inicial en el que tiene que hacer un calentamiento. Luego, en los nodos replicados, vemos que se mantienen utilizando una cantidad de cpu que en todos los casos individuales es inferior al cpu utilizado por el nodo simple, tal como era esperado.

También como podemos ver en el caso de load testing tiene un comportamiento muy similar al ping debido a su asíncronía y aunque tarda mas en contestar, que sea asíncrono permite que se puedan procesar muchos al mismo tiempo.

Bbox B

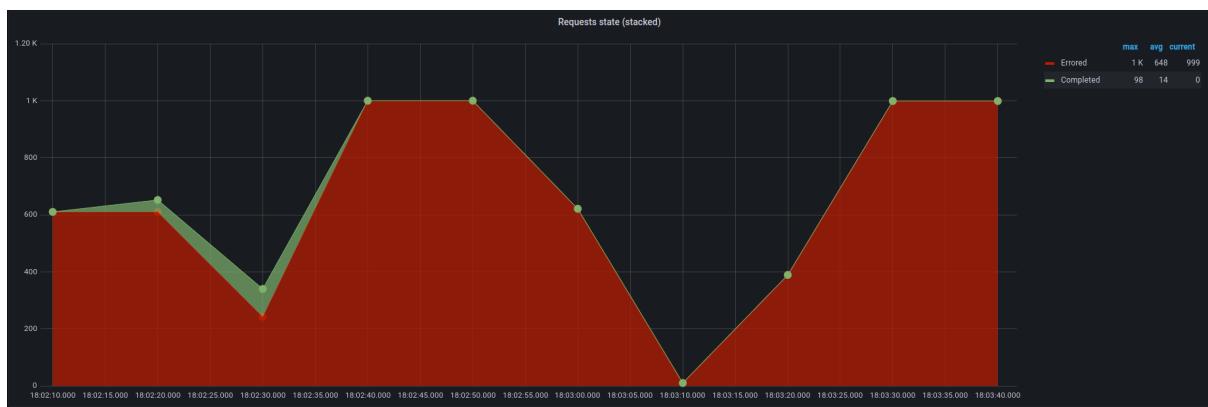
Para comparar este último endpoint también utilizaremos el escenario spike.

Para un solo nodo:

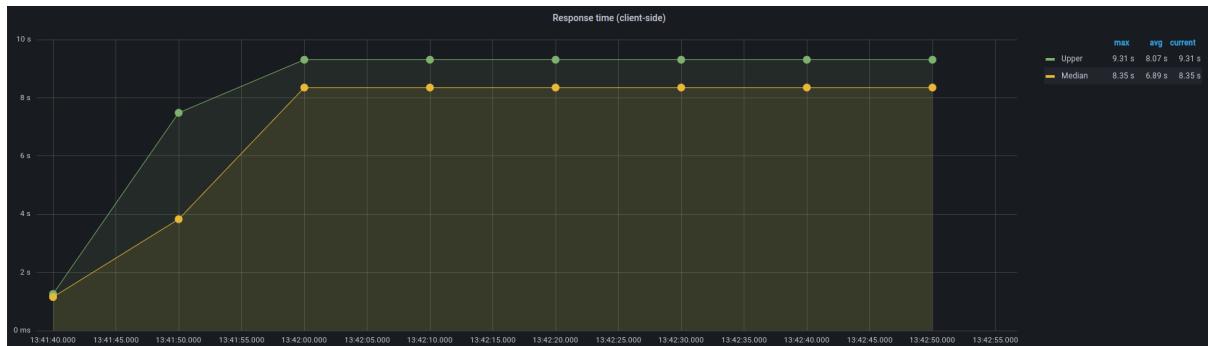
Scenarios launched:



Requests status:



Response time client:



Response time server:



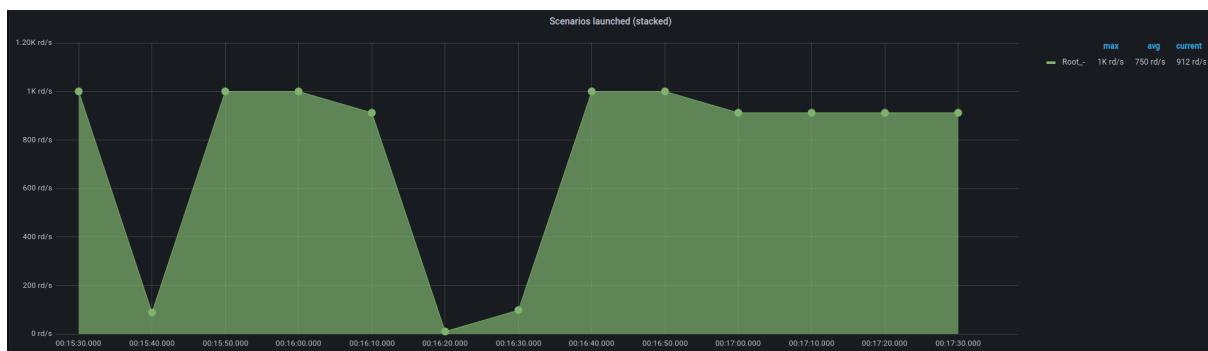
Resources:



Summary report @ 13:42:52 (-0300)		
<hr/>		
errors.ETIMEDOUT:	5880	
http.codes.200:	140	
http.request_rate:	78/sec	
http.requests:	6020	
http.response_time:		
min:	1058	
max:	9299	
median:	4770.6	
p95:	9230.4	
p99:	9230.4	
http.responses:	140	
vusers.completed:	140	
vusers.created:	6020	
vusers.created_by_name.Root (/):	6020	
vusers.failed:	5880	
vusers.session_length:		
min:	1067	
max:	9307.7	
median:	4770.6	
p95:	9230.4	
p99:	9230.4	

Para nodos replicados:

Scenarios launched:



Requests state:



Response time (client side):



Response time (node):



Node_2



Node_3



Node_4



Conclusión:

Lo que pudimos observar en este endpoint es que, al igual que para el bbox A, el comportamiento bajo ambas infraestructuras fue prácticamente el mismo. Esto se debe a que el bbox sigue siendo el mismo, independientemente de cual infraestructura se escoja.

Algo interesante que pudimos observar es que los tests realizados con artillery dan a entender que el servidor se cae por un cierto timeout y a partir de ese momento todos los requests empiezan a fallar, algo similar a que su disponibilidad se vea vulnerada. Lo cierto es que Artillery tiene un timeout de 10 segundos donde luego de ese tiempo, asume que todo lo que sigue es un timeout, pero en realidad la aplicación sigue procesando tanto en un nodo como multi nodo. El servicio BBox B pareciera ser bastante resistente a altos timeouts, podemos ver en la imagen de **response time (node)** cómo pese a que artillery finalizó, sigue procesando por un buen rato.

Inferimos por este comportamiento que BBox B es el servicio sincrónico, más adelante en el punto 2 será analizado más en profundidad.

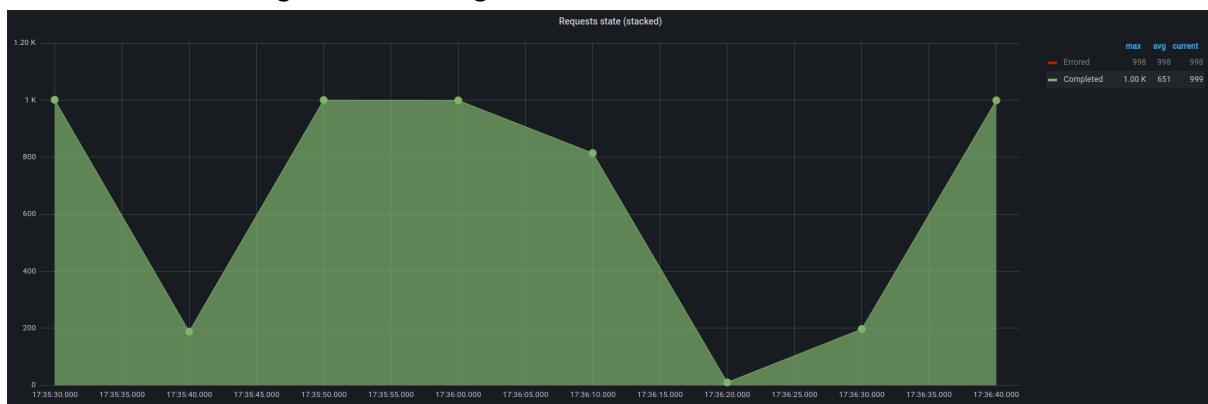
Salvo los picos iniciales de uso de CPU, el uso de recursos se mantuvo en valores razonables, sosteniendo que la culpa de las request no exitosas es de bbox y no del servicio node.

2. Sección 2

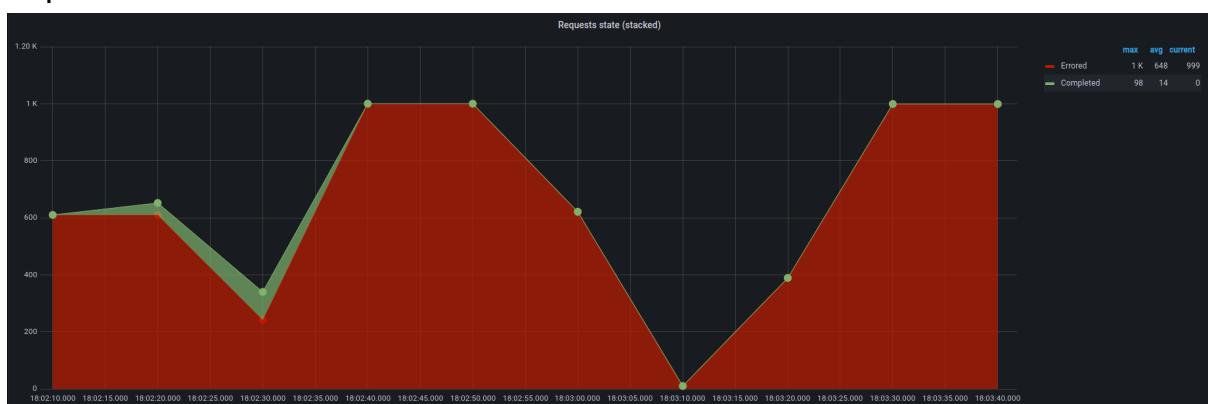
Analisis y caracterizacion:

Sincrónico/Asincronico:

Habiendo analizado las imágenes obtenidas en la sección 1 podemos concluir que el servicio asincrónico es el bbox 1 ya que este servicio pudo responder a una cantidad de requests alta sin alterar su disponibilidad, sus tiempos de respuesta fueron empeorando pero no hubo errores en el servicio. Por otro lado, el bbox 2 si se mostró limitado a la hora de responder, dando errores en las respuestas como timeouts, esto se debe a que el servicio no está preparado para crecer y solo dispone de una cantidad finita de workers que a su vez soportan una cantidad determinada de clientes. Esto lo podemos observar en las siguientes imágenes.



El bbox 1, respondiendo satisfactoriamente a una demanda significativa de requests.



El bbox 2, no pudiendo soportar una demanda significativa de requests.

Cantidad de workers:

Para comprobar la cantidad de workers en el bbox sincrónico lo que haremos es enviar solicitudes HTTP concurrentes. Esperamos que con n workers el bbox soportará N solicitudes concurrentes que realizarán la tarea en una cantidad muy similar de tiempo. En el momento en que al incrementar en 1 la cantidad de solicitudes concurrentes y que esta tarde

más que el resto, habremos llegado al límite de ejecuciones concurrentes y por lo tanto, a la cantidad de workers.

Para el caso de 1 solicitud única, el tiempo de ejecución fue de: 1073 ms. Esperamos entonces que un worker responda en un tiempo aproximado de 1000 milisegundos.

Luego de eso (todas las medidas están expresadas en milisegundos):

2da - 1074 ms 1078 ms

3ra - 1091 ms 1090 ms 1093 ms

4ta - 1079 1089 1086 1084

5ta - 1060 1057 1061 1061 1057

6ta - 1062 1066 1065 1065 1067 1067

...

14ta - 1063 1061 1060 1063 1060 1065 1068 1067 1073 1070 1069 1062
1073 1064

15ta - 1064 1065 1066 1066 1067 1068 1064 1074 1066 1067 1075 1074
1064 1073 2092

En esta décimo quinta iteración con 15 llamadas concurrentes obtuvimos un tiempo de respuesta de 2 milisegundos, lo que es prácticamente el doble de tiempo que el resto de las requests, a partir de aquí, al realizar requests concurrentes de más de 15, a partir de la quinceava obtuvimos un tiempo superior a los dos milisegundos. Es por esto que concluimos que tiene 14 workers con los cuales maneja hasta 14 requests a la vez, el resto de requests deberán esperar a que estas primeras finalicen para empezar.

Demora en responder:

Para comparar el servicio en cuanto a tiempo de respuesta, lo hicimos con pocas requests, sin utilizar herramientas de carga, para ver cómo se comportan. Nos guiamos por lo que reporta la aplicación de nodeJS, ya que esta registra el tiempo de respuesta en todos sus endpoints para generar la métrica propia de la sección 1. En el caso del bbox sincrónico pudimos observar que sus respuestas fueron en el rango de los 1069 ms - 1074 ms, mientras que en el bbox asincrónico fueron en el rango de los 1150 ms - 1350 ms. De estos datos concluimos que nuestra primera hipótesis es verdadera, ya que el bbox asincrónico es más lento que el bbox sincrónico porque el segundo devuelve una respuesta al cliente inmediatamente después de procesar la petición.

Característica	Bbox 1	Bbox 2
Demora en responder (visto desde el nodo)	Primera vez: 1350 ms Luego: 1159 ms 1158 ms 1157 ms	Primera vez: 1074 ms Luego: 1073 ms 1069 ms 1070 ms

Cantidad de workers	X	14 workers
Sincronicidad	Es asíncrono	Es sincrónico

3. Sección 3

En esta sección simularemos un sistema de inscripción similar al usado en la facultad y su flujo para estudiar su comportamiento bajo carga.

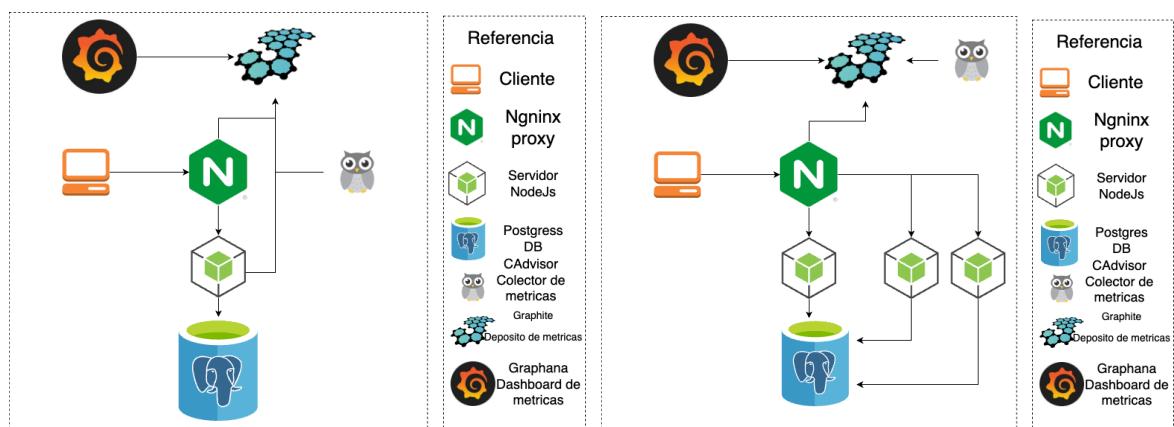
Hipótesis:

Con este objetivo planteamos ciertas hipótesis que describiremos a continuación:

- Pasos del Usuario:
 - Iniciar Sesión (POST /auth/login {username, password} => {token})
 - Inscribirse a N materias”
 - Ver lista de materias inscripto (GET /courses/enroll {token} => {courses})
 - Ver lista de materias disponibles (GET /courses {token} => {courses})
 - Inscribirse a una materia (POST /courses/enroll {token} => ok)
 - Cerrar Sesión (POST /auth/logout {token} => ok)
- Cantidad de alumnos: 8000 Usuarios
- Duración de prioridades: 4 Prioridades
- Duración de prioridades: 15 Min
- Distribución de usuarios: 90% de cada prioridad se conectara en los primeros 5 minutos de su Franja y el 10% restante se conectara en el tiempo siguiente.

Implementación:

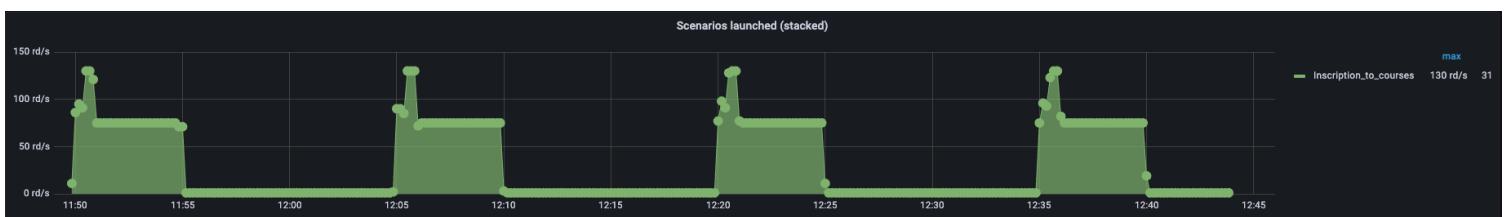
Para realizar la simulación del sistema de inscripción usamos la misma arquitectura que para los puntos anteriores y le agregamos una base de datos para almacenar los usuarios, las materias y las inscripciones.



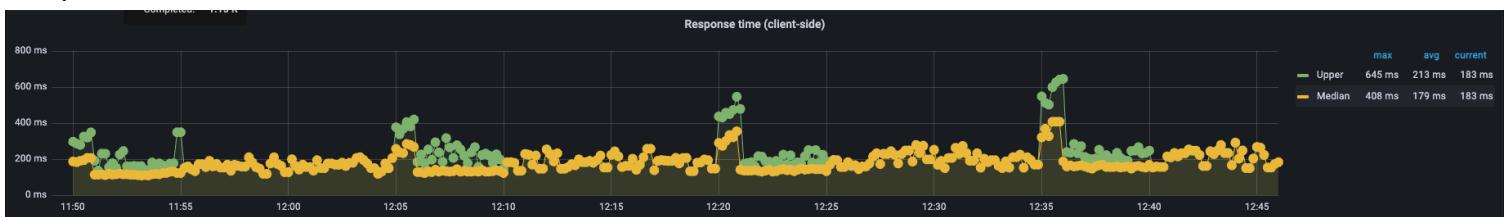
Graficos:

Single Node:

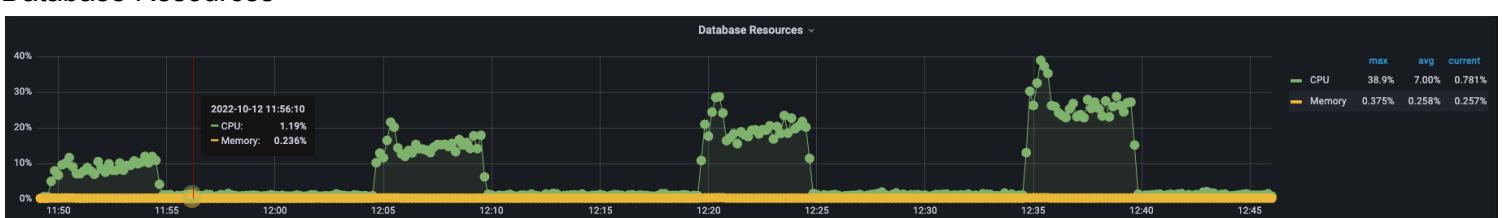
Scenarios Launched



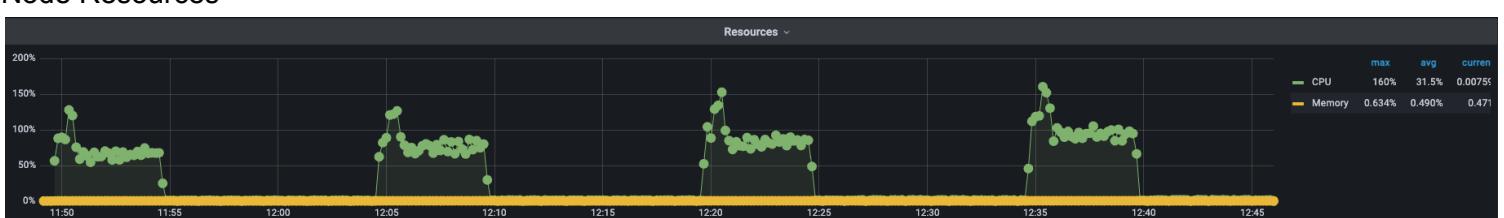
Response Time



Database Resources

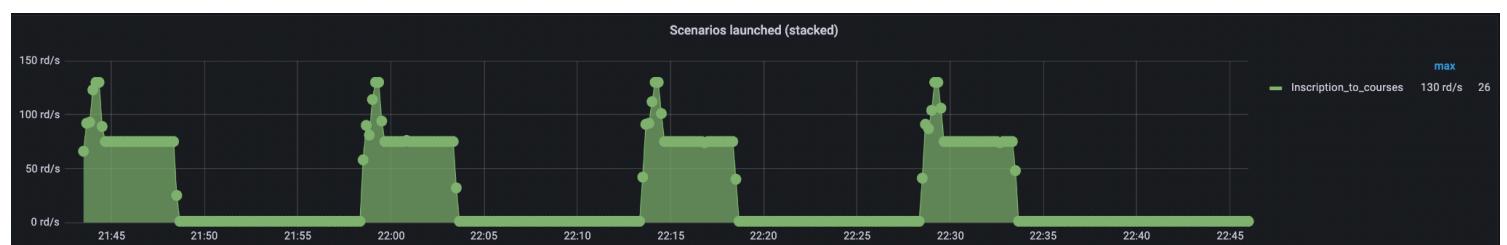


Node Resources

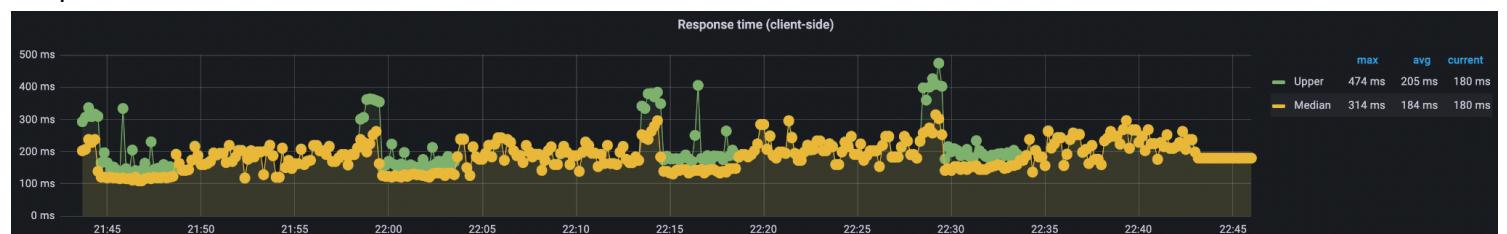


Replicated Node

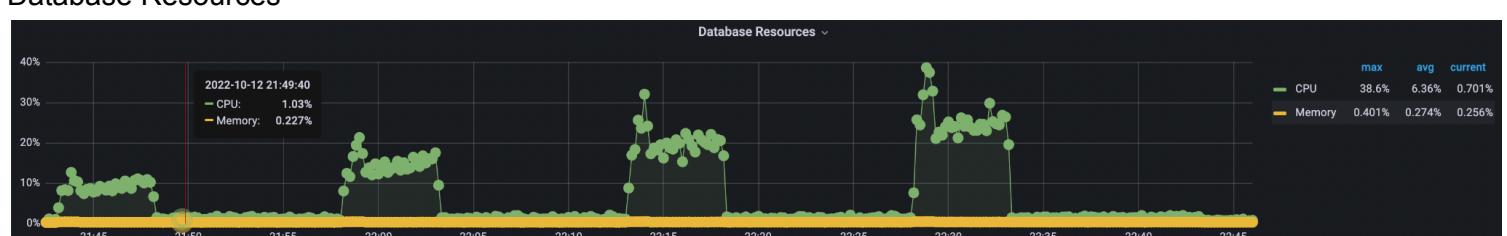
Scenarios Launched



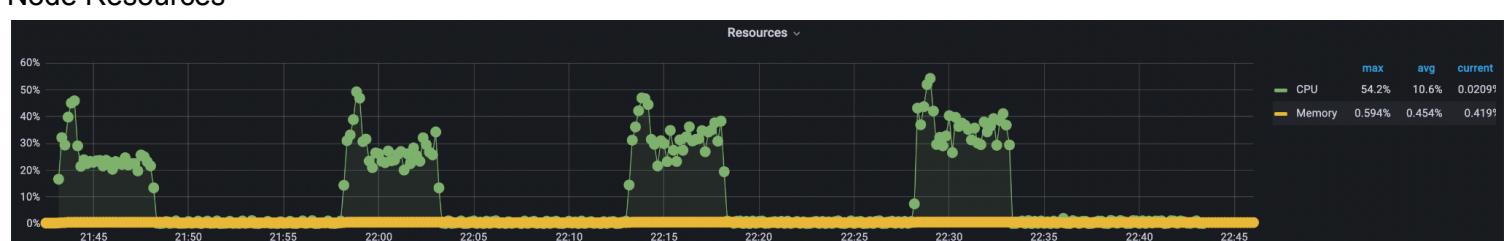
Response Time



Database Resources



Node Resources



Conclusion:

Como primera conclusión podemos decir que el sistema implementado resuelve satisfactoriamente el escenario planteado sin mayores problemas. En ningún momento el sistema falla y responde a todas las request realizadas

de forma satisfactoria. Tanto en el escenario single node como en el caso de nodos replicados el comportamiento del tiempo de respuesta es similar y este aumenta a medida que la cantidad de request sube esto puede deberse a que la base de datos es única y no fue escalada de forma horizontal a diferencia de los nodos, sin embargo si se puede ver que los máximos tiempos de respuesta sufren una significativa mejora al replicar los nodos, por lo cual concluimos que el escalamiento horizontal fue beneficioso. Por otro lado, podemos ver que la carga del CPU de la base de datos sufre de la misma forma en ambos escenarios lo cual es evidente ya que siempre es un solo nodo. Podemos ver que a medida que avanza la simulación la carga es cada vez mayor, esto se puede deber a que las búsquedas hechas son en un dataset mayor a medida que los usuario se inscriben y no se tuvo en cuenta en la implementación ninguna forma para mejorar esto como podrían ser los índices. En contraposición, la carga de los CPU del servidor es mucho más baja en el escenario donde el servidor está replicado.

Por otro lado, si hubiesen problemas de rendimiento se darían al inicio de un nuevo turno de inscripción. Por esto, una solución simple sería tener menos gente por turno, se teniendo más turnos del mismo tiempo o más cortos. En este caso, la duración de un turno no tiene un rol significativo en los resultados al concentrarse los ingresos en los primeros minutos pero sí podría afectar al usuario. Una buena idea podría ser que a medida que se avanza en los turnos, cada vez sean menos alumnos para mitigar el efecto de la carga donde más hay y afectar al usuario lo menos posible.