.UBAfiuba
FACULTAD DE INGENIERÍA

# Trabajo Práctico N°1

[75.73] Arquitectura del Software
1° Cuatrimestre 2022

| **Grupo 4** | |
|---|---|
| **Integrantes** | **N° Padron** |
| Bianco, Santiago | 93607 |
| Pacheco, Federico Jose | 104541 |
| Veron, Lucas Maximiliano | 89341 |

# Objetivos

El objetivo principal del trabajo práctico es analizar los atributos de calidad tales como performance, disponibilidad y escalabilidad a través de las diferentes configuraciones y escenarios a probar.
En cada escenario vamos a predecir resultados en base a argumentos aprendidos en la materia, obtener métricas que luego servirán para comparar resultados con otros escenarios y en base a eso armar las conclusiones.

Se podría decir que otro de los objetivos es familiarizarnos con las herramientas utilizadas para construir la arquitectura a ejecutar en cada uno de los escenarios.

# Resumen

La idea sería, primero configurar toda una arquitectura a través de docker compose como orquestador, implementar una app muestra que en algunos casos servirá de pasamanos y en otro nosotros tendremos que implementar la solución.

Una vez que tenemos la configuración, correr escenarios exploratorios para cada endpoint a estudiar para saber cómo se comportan, además de verificar el correcto funcionamiento de la arquitectura de punta a punta.

Endpoints a medir y analizar:

- **PING**: health check implementado en node, no consume ningún servicio externo
- **PASAMANO_9090**: endpoint implementado en NODE, consume el servicio 9090 de bbox de forma sincrónica y no bloqueante.
- **PASAMANO_9091**:endpoint implementado en NODE, consume el servicio 9090 de bbox de forma sincrónica y no bloqueante.
- **HEAVY**: endpoint implementado en NODE, tiene varias operaciones matemáticas que consumen CPU, el response time es considerablemente mayor comparado con el PING. No consume ningún servicio externo.

De ahora en más cuando usemos los nombres de estos endpoints nos referiremos a los que acabamos de definir aquí.

# Configuraciones

## Un solo nodo

Para este caso se tiene a **Artillery** como cliente, quien genera carga en diferentes escenarios los cuales son analizados. Las peticiones http son tomadas por un nginx quien hace de **Load Balancer/Proxy** y luego se comunica con el servicio de **NODE.** Para las pruebas en donde se ve involucrado otro servicio externo, se tiene una instancia de **BBox** quien escucha en 2 puertos diferentes(9090 y 9091) y responden a las peticiones del servicio de **NODE**. Uno de los servicios expuestos por **BBox** es **sincrónico**, mientras que el otro es **asincrónico**.

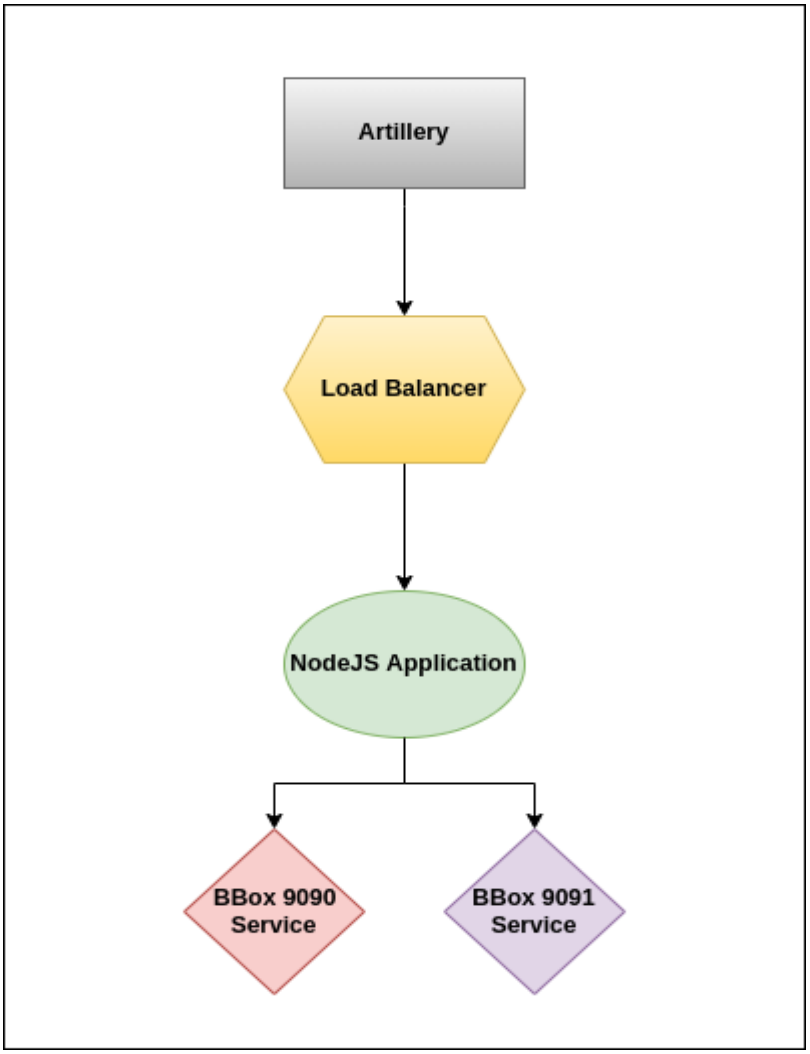
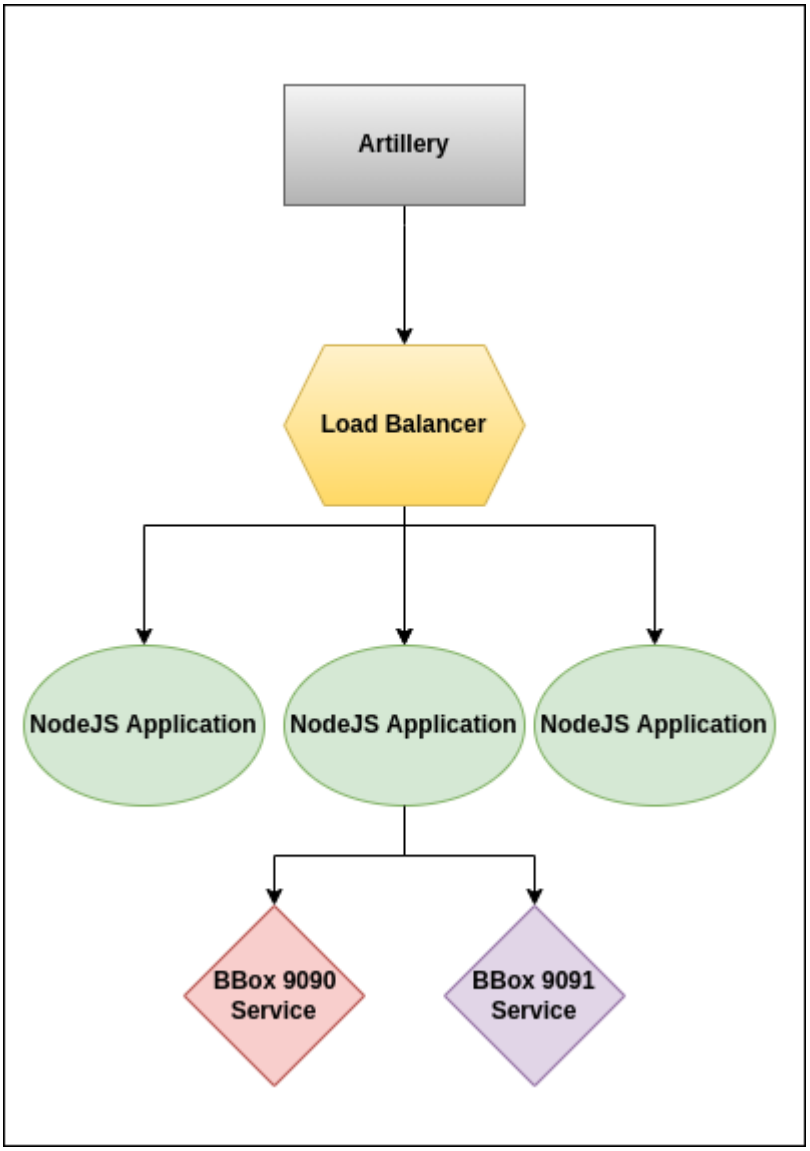



Fig.1. Vista de componentes y conectores sobre la configuración de 1 solo nodo.

## Réplicas

En este caso se tiene una configuración similar a la usada para el caso de 1 solo nodo, pero se escala el servicio de NODE a 3 instancias para poder llevar a cabo los mismos escenarios de prueba pero obtener métricas diferentes. Estas las analizaremos, compararemos y sacaremos nuestras conclusiones.




## Buscando el SETUP ideal

Uno de los factores de éxito que consideramos crucial para lograr los objetivos planteados es lograr que las mediciones sean lo más limpias posibles, es decir que los componentes intermedios entre nuestro generador de carga y el componente a medir, tienen que afectar lo menos posible, y para esto creemos que es clave la configuración ideal.

En esta búsqueda nos encontramos varios inconvenientes, más allá de la dificultad que tuvimos para ejecutar la configuración inicial provista por la cátedra, dificultad nuestra al no

poseer casi conocimiento previo de estas tecnologías, uno de los grandes problemas fue la implementación del pasamano  en NODEJS.

## Servicio NODE

Una de las grandes dificultades aquí fue también nuestra falta de experiencia en este lenguaje. No teníamos bien en claro los conceptos acerca de “EVENT LOOP”, ni sincronía/asincronía o cómo declarar una función para que sea bloqueante o no.

Al no saber, arrancamos inicialmente creando nuestras API REST que hacen de pasamano a bbox de forma bloqueante, con llamados al servicio de bbox de manera asincrónica. Justo al revés de lo deseado.

- “No bloqueante” para no bloquear en EVENT LOOP mientras que se esperan las respuestas de BBOX.
- Llamados sincrónicos desde el servicio NODE a BBOX para que los tiempos de espera repercutan en los resultados de las mediciones, y así poder analizar a posteriori algunas características de bbox.

## Artillery

Una vez que logramos implementar estos servicios pasamanos nos topamos con otro problema, creamos un escenario de carga mínima para ver cómo se comportaba el servicio PASAMANO_9090 de **bbox** usando la configuración “un solo nodo”.
Y con esta configuración de ARTILLERY

```
phases
  - name: Rampa
    duration: 10
    arrivalRate: 5
    rampTo: 10
  - name: Plano - 10 request por segundo
    duration: 30
    arrivalRate: 10
  - name: Rampa
    duration: 10
    arrivalRate: 10
    rampTo: 13
  - name: Plano - 13 request por segundo
    duration: 30
    arrivalRate: 13
  - name: Rampa
    duration: 10
    arrivalRate: 13
    rampTo: 16
  - name: Separador
    duration: 10
    arrivalRate: 1
```

Obtenemos estos resultados

```
--------------------------------
Summary report @ 20:18:30(-0300)
--------------------------------

errors.ETIMEDOUT: ..... 896
http.codes.200: ..... 147
http.request_rate: ..... 5/sec
http.requests: ..... 1043
http.response_time:
  min: ..... 904
  max: ..... 9975
  median: ..... 4492.8
  p95: ..... 9416.8
  p99: ..... 9801.2
http.responses: ..... 147
vusers.completed: ..... 147
vusers.created: ..... 1043
vusers.created_by_name.Root (/): ..... 1043
vusers.failed: ..... 896
vusers.session_length:
  min: ..... 906.2
  max: ..... 9977
  median: ..... 4492.8
  p95: ..... 9416.8
  p99: ..... 9801.2
```

Y lo que mas nos llamo la atencion fue este mensaje de nginx
**"GET /proxy_9090 HTTP/1.1" 499 0 "**

Buscamos a que se podía deber este error y al parecer se lanza  porqué el  cliente (artillery para nginx) cierra la conexión. La pregunta es, ¿por qué?

Para buscar una razón se nos ocurrió hacer esta prueba, correr el mismo escenario pero primero apuntando directamente a bbox y luego apuntando al servicio node, sin pasar por nginx. La razón de esta prueba es ver si hay algún cuello de botella en el servicio NODE que dispare este cierre de conexión.

Artillery apuntando a bbox

```
--------------------------------
Summary report @ 18:07:46(-0300)
--------------------------------

errors.ETIMEDOUT: ..... 911
http.codes.200: ..... 146
http.request_rate: ..... 5/sec
http.requests: ..... 1057
http.response_time:
  min: ..... 900
```

*max: ..... 9938*
*median: ..... 4231.1*
*p95: ..... 9230.4*
*p99: ..... 9801.2*
*http.responses: ..... 146*
*vusers.completed: ..... 146*
*vusers.created: ..... 1057*
*vusers.created_by_name.Root (/): ..... 1057*
*vusers.failed: ..... 911*
*vusers.session_length:*
*min: ..... 903.2*
*max: ..... 9940*
*median: ..... 4231.1*
*p95: ..... 9230.4*
p99: ..... 9801.2

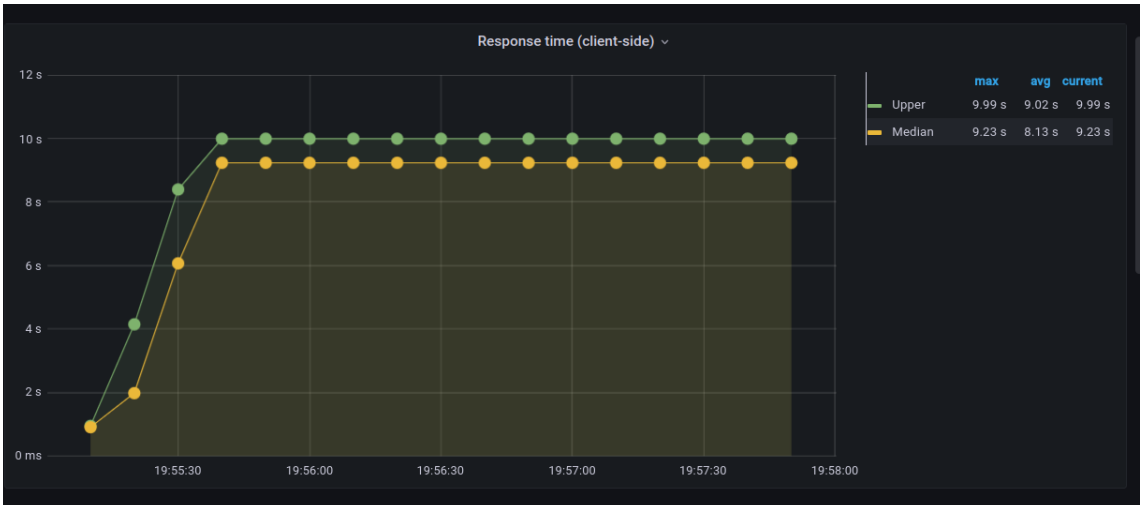
Artillery apuntando a servicio NODE

```
--------------------------------
Summary report @ 18:14:32(-0300)
--------------------------------

errors.ETIMEDOUT: ..... 896
http.codes.200: ..... 145
http.request_rate: ..... 5/sec
http.requests: ..... 1041
http.response_time:
  min: ..... 904
  max: ..... 9889
  median: ..... 4492.8
  p95: ..... 9230.4
  p99: ..... 9801.2
http.responses: ..... 145
vusers.completed: ..... 145
vusers.created: ..... 1041
vusers.created_by_name.Root (/): ..... 1041
vusers.failed: ..... 896
vusers.session_length:
  min: ..... 906.9
  max: ..... 9890.4
  median: ..... 4492.8
  p95: ..... 9230.4
  p99: ..... 9801.2
```

Como observamos los resultados dieron muy parecido, por eso concluimos que el servicio que implementamos en NODE no sería el cuello de botella.

Para analizar un poco mejor la situación usamos el gráfico que nos provee Grafana. Acá se puede ver el tiempo de respuesta de las request acumuladas (cada diez segundos) en función del tiempo avanzado en el escenario.

Mismo escenario que usamos antes pero con ARTILLERY pegandole al servicio PASAMANO_9090, y todo esto detrás de NGIX como balanceador de carga




Vemos que el servicio PASAMANO_9090 de bbox cada vez tarda más en responder a

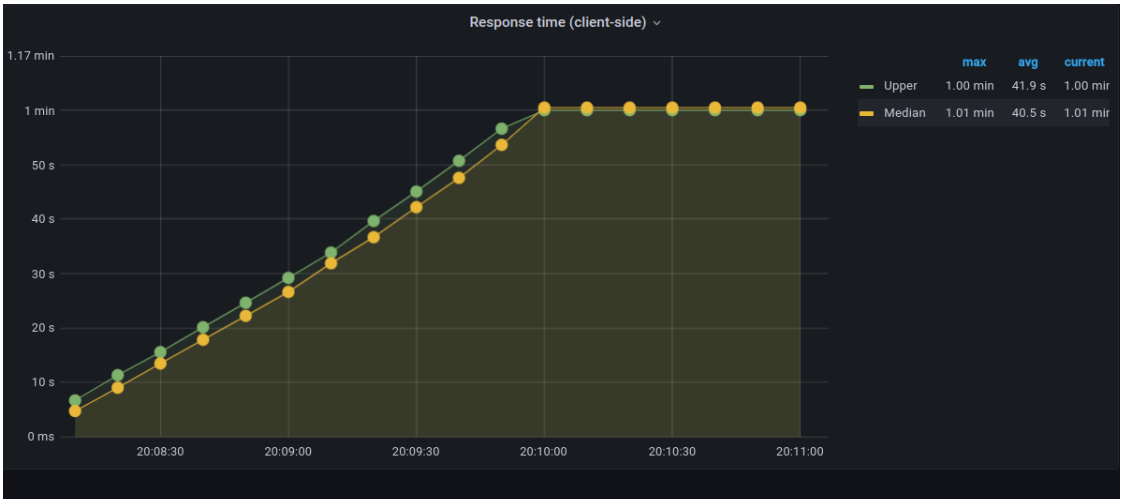
medida que va acumulando peticiones en tiempo de espera para ser procesadas (aumenta el response time).

Al observar este gráfico notamos algo raro, primero el rápido crecimiento hasta alcanzar el pico de 10 segundos de response time, y luego una meseta prolongada a partir de este pico.

Esto nos llevó a inferir que probablemente lo que está pasando es que ARTILLERY tenga poca tolerancia a la espera. Buscamos en su documentación y vimos que efectivamente el TIMEOUT por defecto es de 10 segundos, probamos cambiarlo a 100 segundos para comprobar que este fuere el problema y obtuvimos estos resultados corriendo el mismo escenario contra NGNIX.

*---*
*Summary report @ 20:10:20(-0300)*
*---*

*http.codes.200: ..... 692*
*http.codes.504: ..... 351*
*http.request_rate: ..... 3/sec*
*http.requests: ..... 1043*
*http.response_time:*
*min: ..... 903*
*max: ..... 60010*
*median: ..... 42205.5*
*p95: ..... 60495.1*
*p99: ..... 60495.1*
*http.responses: ..... 1043*
*vusers.completed: ..... 1043*
*vusers.created: ..... 1043*
*vusers.created_by_name.Root (/): ..... 1043*
*vusers.failed: ..... 0*
*vusers.session_length:*
*min: ..... 905.7*
*max: ..... 60019.7*
*median: ..... 42205.5*
*p95: ..... 60495.1*
*p99: ..... 60495.1*




Como vemos los responses exitosos son considerablemente mayores al subir el umbral de TIMEOUT (692 con timeout de 100 seg vs 146 con timeout de 10 segundos).

Podemos concluir que subir este umbral en ARTILLERY es de nuestro interés, ya que para las mediciones que queremos realizar sobre los servicios, deseamos que los componentes afecten lo menos posible.

**Observación**: como vemos en el resumen de ARTILLERY , dejamos de tener ***errors.ETIMEDOUT*** y comenzamos a ver ***http.codes.504*** ,esto se debe a que luego de aumentar el TIMEOUT de artillery el que los comenzó a tirar los timeouts fue NGNIX . Buscando en la documentación oficial de NGNIX y encontramos que por defecto son 60 segundos de tolerancia a una respuesta, por lo tanto al cambiar a 100 segundos de TIMEOUT en Artillery provocará que primero fallara NGINX.

El plugin de grafana muestra los errores que evidencia ARTILLERY , si bien podríamos tocar la configuración del dashboard de grafana para que también muestre los errores de NGNIX , por una cuestión del tiempo restante para terminar el trabajo práctico no vamos a invertir dedicación aquí, entonces vamos a setear los timeouts de tal forma que se puedan visualizar en grafana asi como esta la configuración del dashboard por default. Es decir, vamos a dejar un TIMEOUT más alto en NGNIX que en ARTILLERY para que el primero que arroje el timeout sea ARTILLERY .

- 110 para nginx.
- 100 segundos de timeout para artillery

La decisión de la cantidad de segundos en los TIMEOUTS es totalmente arbitraria, pero aumentamos lo que venía por default porque las peticiones fallaban demasiado pronto en los escenarios, dificultando asi el análisis de las mediciones.

# Sección 1

## Configuración un solo nodo

### Escenario exploratorio

#### ENDPOINT PING

##### Configuración escenario

El escenario a utilizar para la fase exploratoria fue el siguiente

```
phases:
  - name: Rampa I - Light
    duration: 30
    arrivalRate: 25
    rampTo: 50
  - name: Plain I
    duration: 30
    arrivalRate: 50
  - name: Rampa II - Pico
    duration: 30
    arrivalRate: 50
    rampTo: 100
  - name: Plain II
    duration: 30
    arrivalRate: 100
  - name: Plain III
    duration: 30
    arrivalRate: 50
  - name: Rampa III - Pico
    duration: 30
    arrivalRate: 50
    rampTo: 100
  - name: Plain III
    duration: 30
    arrivalRate: 100
  - name: Plain III
    duration: 30
    arrivalRate: 50
  - name: Separador
    duration: 30
    arrivalRate: 1
```

Vamos aumentando la carga de forma ligera, y buscando picos con una bajada y subida de carga.

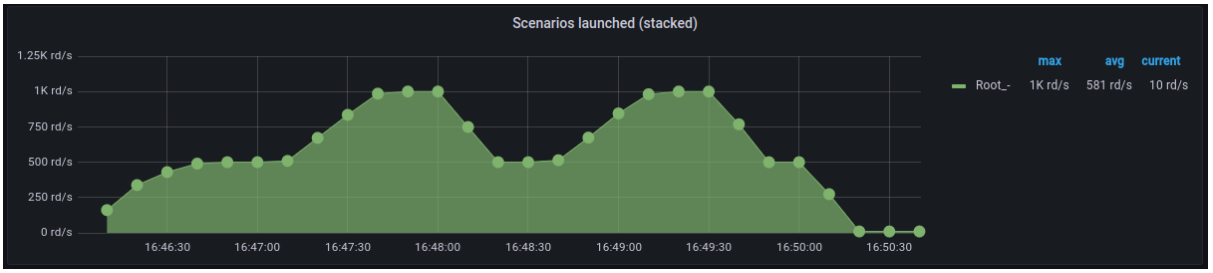
## Predicción de resultado

Para este endpoint y con este escenario, no se esperan grandes expectativas más que observar un leve uso de CPU debido al tipo de endpoint, como también una response time bastante corta a medida que se aumenta la carga.

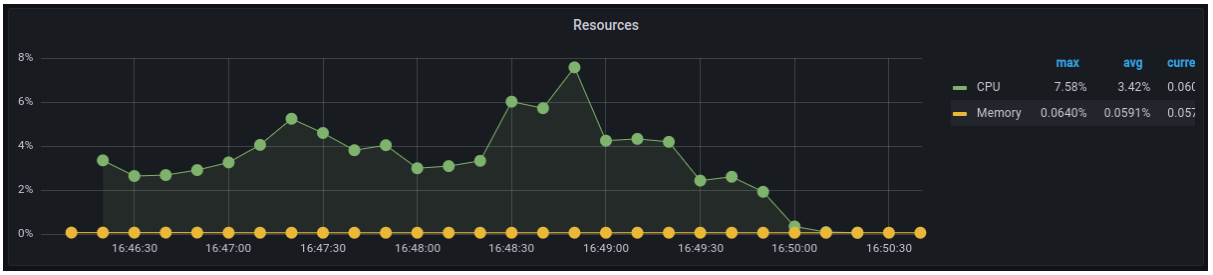
## Resultados

```
--------------------------------
Summary report @ 16:50:26(-0300)
--------------------------------

http.codes.200: ..................................................... 5576
http.request_rate: ..................................................... 20/sec
http.requests: ..................................................... 5576
http.response_time:
  min: ..................................................... 0
  max: ..................................................... 29
  median: ..................................................... 2
  p95: ..................................................... 5
  p99: ..................................................... 10.1
http.responses: ..................................................... 5576
vusers.completed: ..................................................... 5576
vusers.created: ..................................................... 5576
vusers.created_by_name.Root (/): ..................................................... 5576
vusers.failed: ..................................................... 0
vusers.session_length:
  min: ..................................................... 1.4
  max: ..................................................... 85.8
  median: ..................................................... 4.3
  p95: ..................................................... 13.6
  p99: ..................................................... 23.8
```

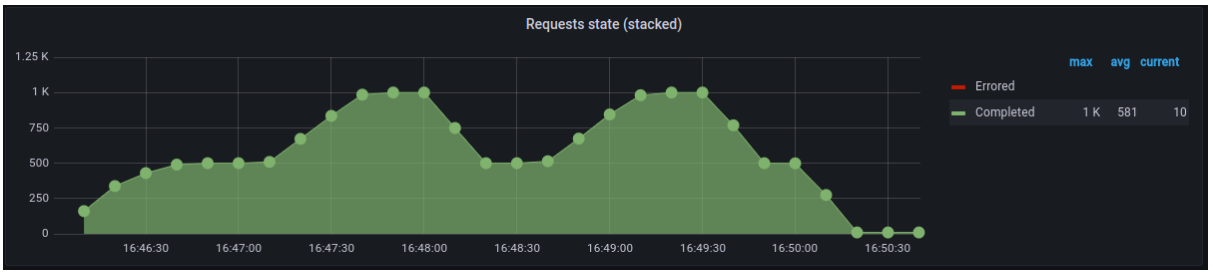
Grafana - request enviadas en el escenario



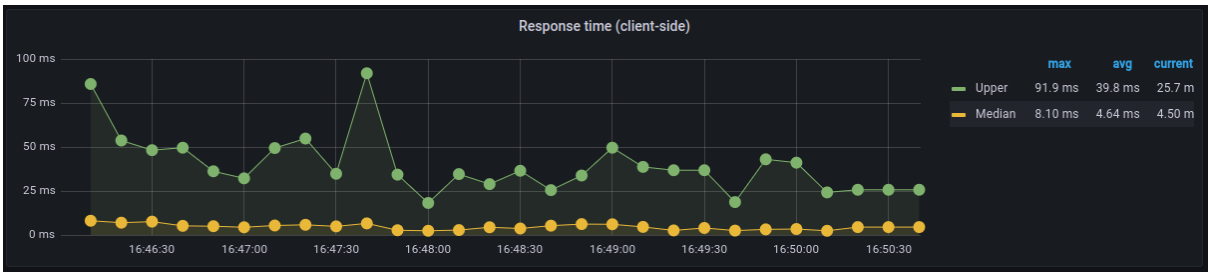

Grafana - consumos de recursos a lo largo del escenario



Grafana - request completadas vs request con respuesta de error




Grafana - response time a lo largo del escenario




### Conclusión o observaciones

Se puede observar como hay un mínimo uso de CPU y de memoria, logrando utilizar un promedio de 3.42% de uso de CPU. Solo una vez ocurrió un pico de 7.58% del consumo de CPU y lo interesante a observar es que durante la etapa en donde transcurre el periodo del crecimiento de una rampa es donde aparecen leves picos de consumo de CPU hasta que una vez se llegue al periodo donde hay requests enviados de forma constante.

## PASAMANO_9090

### Configuración escenario

El escenario a utilizar para la fase exploratoria fue el siguiente

```
phases:
  - name: Rampa I - Light
    duration: 30
    arrivalRate: 3
    rampTo: 5
  - name: Plain I
    duration: 30
    arrivalRate: 5
  - name: Rampa II - Pico
    duration: 30
    arrivalRate: 5
    rampTo: 10
  - name: Plain II
    duration: 30
    arrivalRate: 10
  - name: Plain III
    duration: 30
    arrivalRate: 3
```

```
  - name: Rampa III - Pico
    duration: 30
    arrivalRate: 5
    rampTo: 10
  - name: Plain IV
    duration: 30
    arrivalRate: 10
  - name: Plain V
    duration: 30
    arrivalRate: 3
  - name: Separador
    duration: 30
    arrivalRate: 1
```

A diferencia del endpoint PING, vamos a estar mandando carga muy ligera debido a que ya no es un “health check”, es decir, algo mínimo que responde rápido. Buscamos hacer el mismo efecto aumentando la carga, y buscando picos con una bajada y subida de carga sin llegar a romper el servicio.

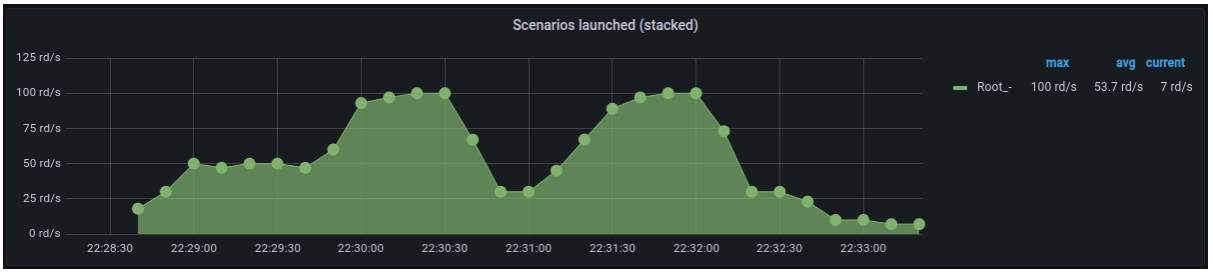
### Predicción de resultado

Con poca carga enviada no esperamos un gran consumo de CPU, ni mucho menos obtener requests con errores. Además esperamos que el gráfico de response time pueda ser de dos formas (aproximadamente): creciendo de manera lineal o como una constante (esto es debido a que, a priori, no sabemos si el tipo de endpoint es sincrónico o asincrónico).

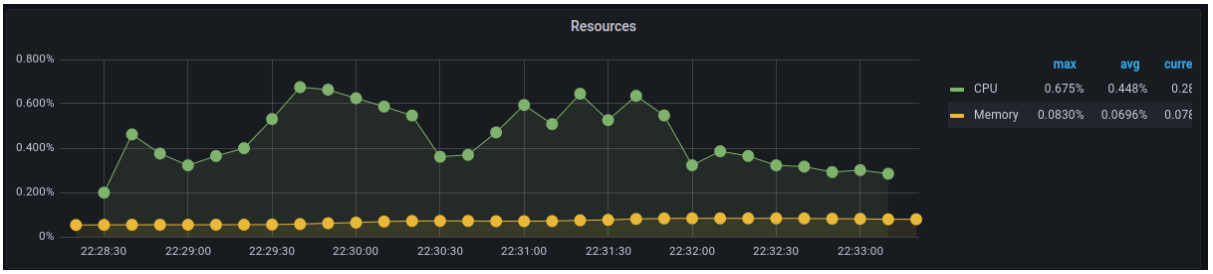
### Resultados

```
--------------------------------
Summary report @ 22:33:18(-0300)
--------------------------------

http.codes.200: ..................................................... 1550
http.request_rate: .................................................. 6/sec
http.requests: ...................................................... 1550
http.response_time:
  min: .............................................................. 901
  max: ............................................................. 59740
  median: ........................................................... 25598.5
  p95: .............................................................. 55843.8
  p99: .............................................................. 59297.1
http.responses: ..................................................... 1550
vusers.completed: ................................................... 1550
vusers.created: ..................................................... 1550
vusers.created_by_name.Root (/): .................................... 1550
vusers.failed: ...................................................... 0
vusers.session_length:
  min: .............................................................. 905.8
  max: ............................................................. 59742.8
  median: ........................................................... 25598.5
  p95: .............................................................. 55843.8
  p99: .............................................................. 59297.1
```

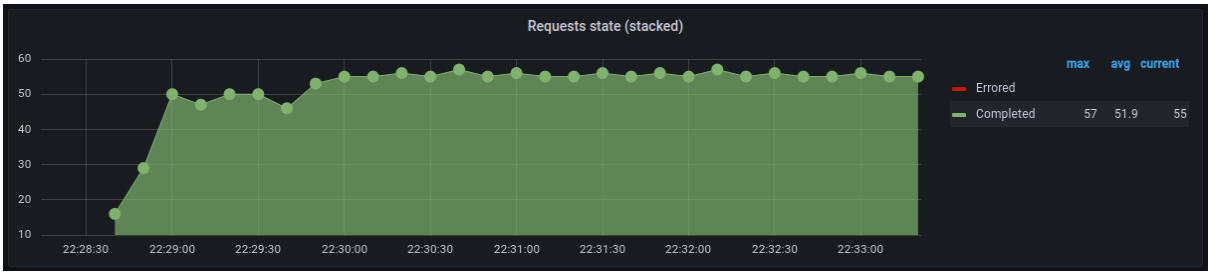
Grafana - request enviadas en el escenario



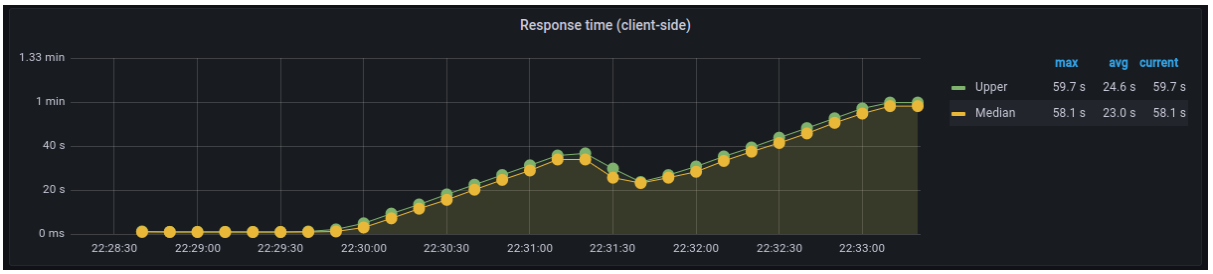

Grafana - consumos de recursos a lo largo del escenario




Grafana - request completadas vs request con respuesta de error




Grafana - response time a lo largo del escenario




## Conclusión o observaciones

Tal como lo esperado, y como en PING, hay un mínimo uso de CPU (no llega ni al 1%) y de memoria, no hay requests con errores y además el response time presenta una de las dos características esperadas: un crecimiento aproximado en forma lineal, aunque luego hay un decrecimiento del response time debido a que las requests enviadas en ese lapso de tiempo fueron mucho menores. En definitiva se van encolando las requests y, entonces **cuando hablamos del response time hablamos de un tiempo que tiene 2 componentes: el primero es el tiempo que tarda una request desde que se encola hasta que llega su turno de ser procesada, y la segunda es el tiempo de procesamiento, donde este último es aproximadamente constante, mientras que el primero se va incrementando con la cantidad de peticiones que recibe**.

## PASAMANO_9091

### Configuración escenario

El escenario a utilizar para la fase exploratoria será el mismo que el escenario de 9090 debido a que ambos son servicios ofrecidos por bbox y su única variación era uno sincrónico y el otro asincrónico por ende sería interesante comparar dichos casos manteniendo el mismo escenario:

```
  phases:
  - name: Rampa I - Light
    duration: 30
    arrivalRate: 3
    rampTo: 5
  - name: Plain I
    duration: 30
    arrivalRate: 5
  - name: Rampa II - Pico
    duration: 30
    arrivalRate: 5
    rampTo: 10
  - name: Plain II
    duration: 30
    arrivalRate: 10
  - name: Plain III
    duration: 30
    arrivalRate: 3
  - name: Rampa III - Pico
    duration: 30
    arrivalRate: 5
    rampTo: 10
  - name: Plain IV
    duration: 30
    arrivalRate: 10
  - name: Plain V
    duration: 30
    arrivalRate: 3
  - name: Separador
    duration: 30
    arrivalRate: 1
```

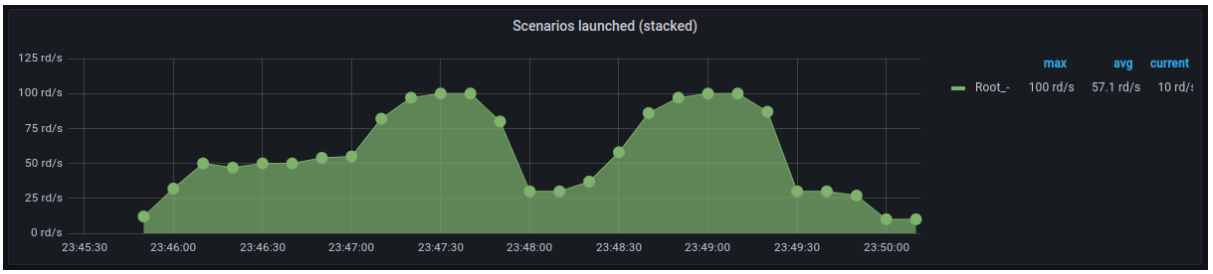
### Predicción de resultado

Mantendremos las mismas predicciones que el caso de PASAMANO_9090, pero con la salvedad que el gráfico de response time se mantendrá aproximadamente como una función constante debido a que el anterior caso de bbox terminó siendo lo contrario.

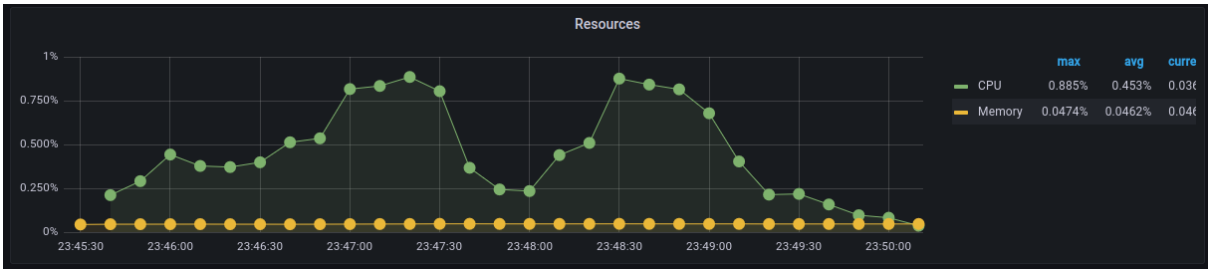
## Resultados

```
--------------------------------
Summary report @ 23:50:09(-0300)
--------------------------------

http.codes.200: ....................................................... 626
http.request_rate: ..................................................... 2/sec
http.requests: ........................................................ 626
http.response_time:
  min: ................................................................ 701
  max: ................................................................ 1023
  median: ............................................................. 713.5
  p95: ................................................................ 727.9
  p99: ................................................................ 727.9
http.responses: ....................................................... 626
vusers.completed: ..................................................... 626
vusers.created: ....................................................... 626
vusers.created_by_name.Root (/): ...................................... 626
vusers.failed: ........................................................ 0
vusers.session_length:
  min: ................................................................ 705.3
  max: ................................................................ 1066
  median: ............................................................. 713.5
  p95: ................................................................ 727.9
  p99: ................................................................ 742.6
```

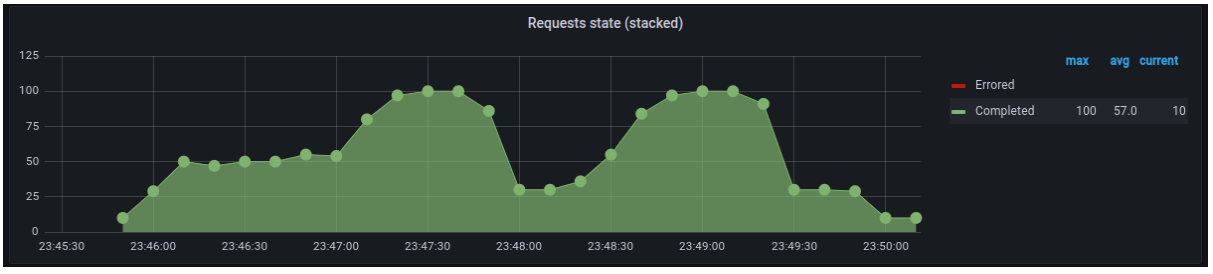
Grafana - request enviadas en el escenario



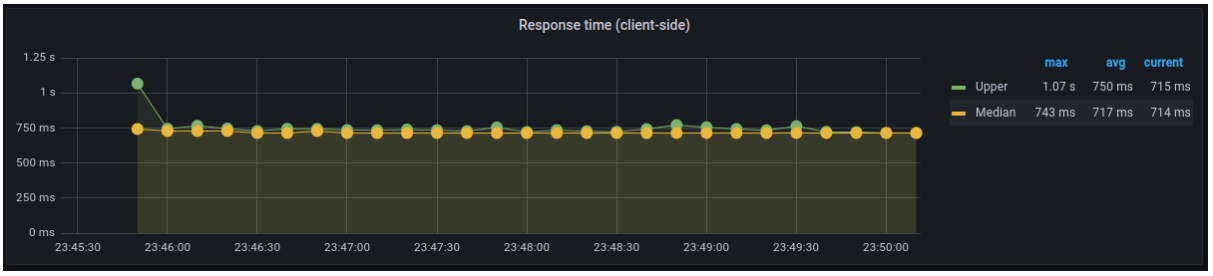

Grafana - consumos de recursos a lo largo del escenario




Grafana - request completadas vs request con respuesta de error



Grafana - response time a lo largo del escenario




### Conclusión o observaciones

Podemos observar que en el gráfico de response time es tal como lo anticipado, el servicio de PASAMANO_9091 procesa las requests a medida que los va llegando, manteniendo el promedio de 750 ms de tiempo de respuesta. Esto también se puede verificar viendo el gráfico de las request completadas vs request con respuesta de error, pues es similar al de los escenarios de request enviadas.
Además, tanto el servicio PASAMANO_9090 como PASAMANO_9091 sigue con bajo consumo de CPU, esto tiene sentido ya que nuestro endpoint solamente realiza un llamado a una api externa, que en este caso es bbox, no debería procesar nada más que una invocación y esperar para dar su respuesta.
Tampoco tendría sentido tener un alto elevado uso de memoria (esto aplica para los 4 endpoints) ya que el código escrito hace poco uso de la memoria.

## HEAVY

### Configuración escenario

El escenario a utilizar, similar al de PING pero con mucha menor carga (debido a que se realiza internamente un cálculo numérico), y buscando picos con una bajada y subida de carga:

```
  phases:
  - name: Rampa I - Light
    duration: 30
    arrivalRate: 3
    rampTo: 5
  - name: Plain I
    duration: 30
    arrivalRate: 5
  - name: Rampa II - Pico
    duration: 30
    arrivalRate: 5
    rampTo: 7
  - name: Plain II
    duration: 30
    arrivalRate: 7
  - name: Plain III
    duration: 30
    arrivalRate: 5
```

```
  - name: Rampa III - Pico
    duration: 30
    arrivalRate: 6
    rampTo: 7
  - name: Plain IV
    duration: 30
    arrivalRate: 7
  - name: Plain V
    duration: 30
    arrivalRate: 5
  - name: Separador
    duration: 30
    arrivalRate: 1
```

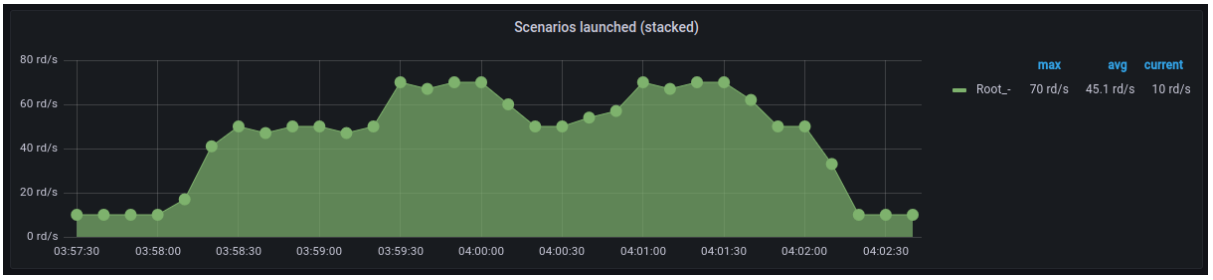
### Predicción de resultado

A diferencia de los anteriores endpoints se esperaría tener un mayor uso de CPU, como también un alto response time que debería ir aumentado debido al event loop de NodeJS.

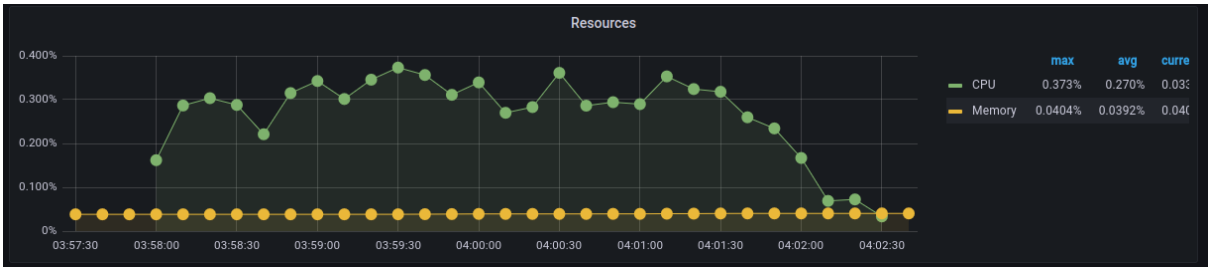
### Resultados

```
-------------------------------
Summary report @ 03:09:26(-0300)
-------------------------------

http.codes.200: ..... 583
http.request_rate: ..... 2/sec
http.requests: ..... 583
http.response_time:
  min: ..... 151
  max: ..... 4284
  median: ..... 788.5
  p95: ..... 3262.4
  p99: ..... 3534.1
http.responses: ..... 583
vusers.completed: ..... 583
vusers.created: ..... 583
vusers.created_by_name.Root (/): ..... 583
vusers.failed: ..... 0
vusers.session_length:
  min: ..... 153.7
  max: ..... 4286.9
  median: ..... 788.5
  p95: ..... 3262.4
  p99: ..... 3534.1
```

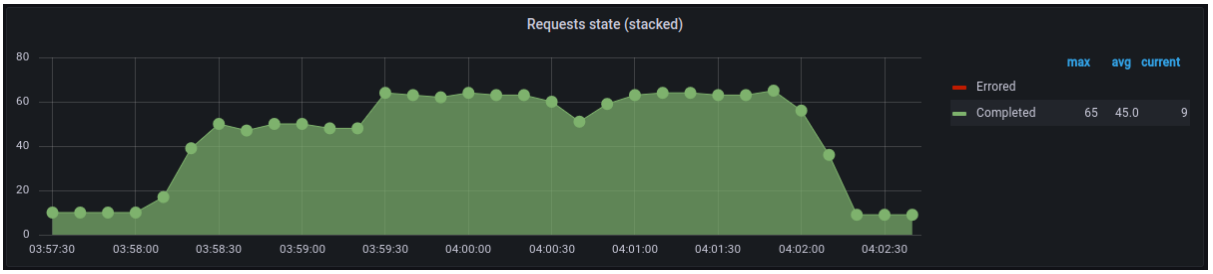
Grafana - request enviadas en el escenario



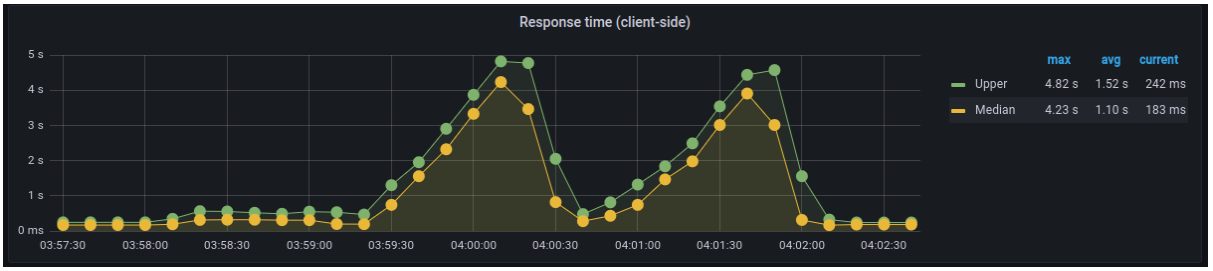

Grafana - consumos de recursos a lo largo del escenario




Grafana - request completadas vs request con respuesta de error




Grafana - response time a lo largo del escenario




## Conclusión o observaciones

Se esperaba un alto consumo de CPU pero terminó siendo relativamente bajo (no llega al 1% de consumo); esto podría explicarse debido a que no es una carga tan intensa a la que es sometido el endpoint.

Lo que resulta destacado observar es como el gráfico del response time, en el transcurso donde crece los picos de carga, se va comportando (aproximadamente) como una función lineal (y luego decrece al disminuir la carga).Esto es debido a que al ir aumentando la carga se está llenando la cola del event loop (api rest declarada como async, por lo tanto su invocación se resuelve en el event loop) y reparte los tiempos de cómputo para terminar de procesar el algoritmo de todos los requests mandados intensamente. Es decir, a mayor carga, con un único nodo, y estando en Node donde solo se utiliza un solo thread, mayor será el response time.

## Escenarios intensivos

Lo que vamos a buscar en estos escenarios es encontrar el cuello de botella de cada uno de los endpoints, detectar qué componentes el que falla y a partir de que carga. Cada uno de los endpoints que vamos a analizar son los implementados en nuestro servicio NODE.

Por cada escenario se plantea una estrategia para las diferentes fases de intensidad, y previo a la ejecución se intentará predecir resultados en base a los escenarios exploratorios previamente ejecutados.

### PING

Servicio implementado en NODE que evidencia que la aplicación está levantada y es capaz de responder a una petición simple.

#### Configuración escenario

```
phases:
  - name: Rampa
    duration: 10
    arrivalRate: 10
    rampTo: 20
  - name: Plano - 20 request por segundo
    duration: 30
    arrivalRate: 20
  - name: Rampa
    duration: 10
    arrivalRate: 20
    rampTo: 50
  - name: Plano - 50 request por segundo
    duration: 30
    arrivalRate: 50
  - name: Rampa
    duration: 10
    arrivalRate: 50
    rampTo: 100
  - name: Plano - 100 request por segundo
    duration: 30
    arrivalRate: 100
  - name: Rampa
    duration: 10
    arrivalRate: 100
    rampTo: 500
  - name: Plano - 500 request por segundo
```

```
    duration: 30
    arrivalRate: 500
  - name: Rampa
    duration: 10
    arrivalRate: 100
    rampTo: 500
  - name: Plano - 500 request por segundo
    duration: 30
    arrivalRate: 500
  - name: Rampa
    duration: 10
    arrivalRate: 500
    rampTo: 1000
  - name: Plano - 1000 request por segundo
    duration: 30
    arrivalRate: 1000
  - name: Separador
    duration: 10
    arrivalRate: 1
```

### Predicción de resultado

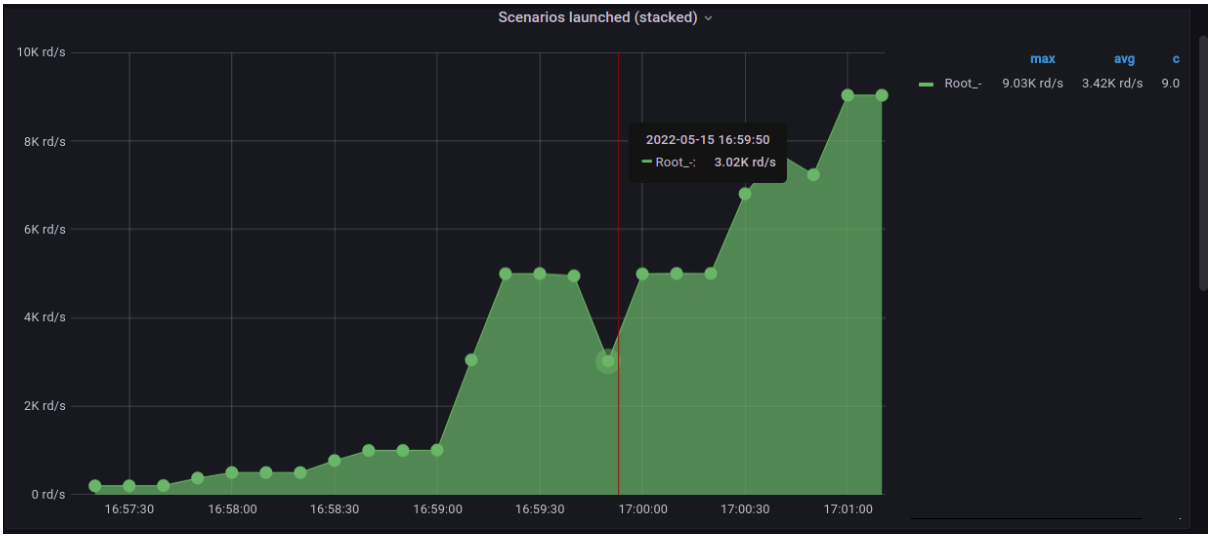
El servicio a probar aquí no hace ningún tipo de procesamiento ni llamada a un servicio externo, ni bien es invocado devuelve una respuesta OK. Lo que esperamos es un response time muy bajo en comparación a los otros servicios. Tampoco que consuma muchos recursos, aun con mucha intensidad de carga.

### Resultados

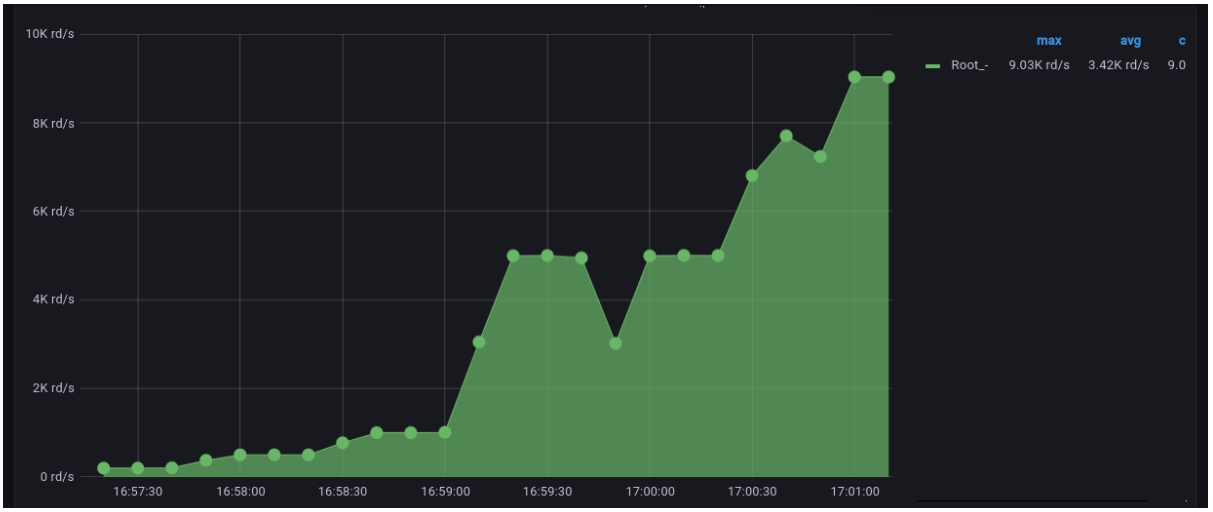
```
-------------------------------
Summary report @ 17:01:08(-0300)
-------------------------------

http.codes.200: ..................................................... 80268
http.request_rate: ..................................................... 156/sec
http.requests: ..................................................... 80268
http.response_time:
  min: ..................................................... 0
  max: ..................................................... 212
  median: ..................................................... 4
  p95: ..................................................... 49.9
  p99: ..................................................... 102.5
http.responses: ..................................................... 80268
vusers.completed: ..................................................... 80268
vusers.created: ..................................................... 80268
vusers.created_by_name.Root (/): ..................................................... 80268
vusers.failed: ..................................................... 0
```

*vusers.session_length:*
*min: ..... 1.4*
*max: ..... 217.2*
*median: ..... 9.7*
*p95: ..... 64.7*
*p99: ..... 113.3*

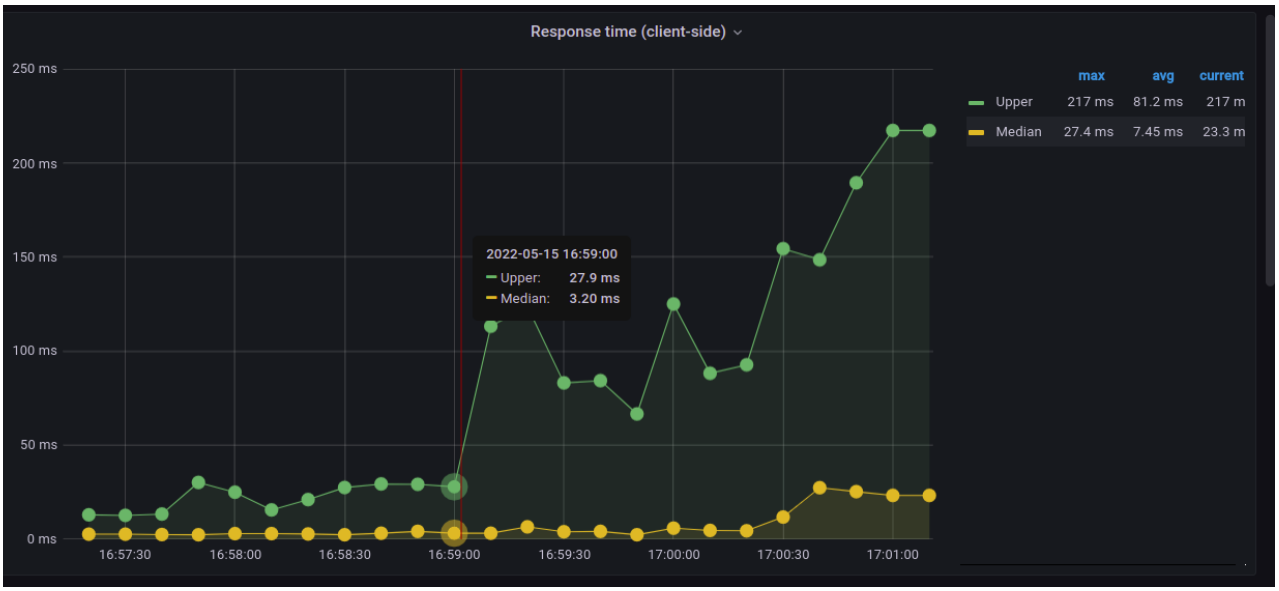
Grafana - request enviadas en el escenario



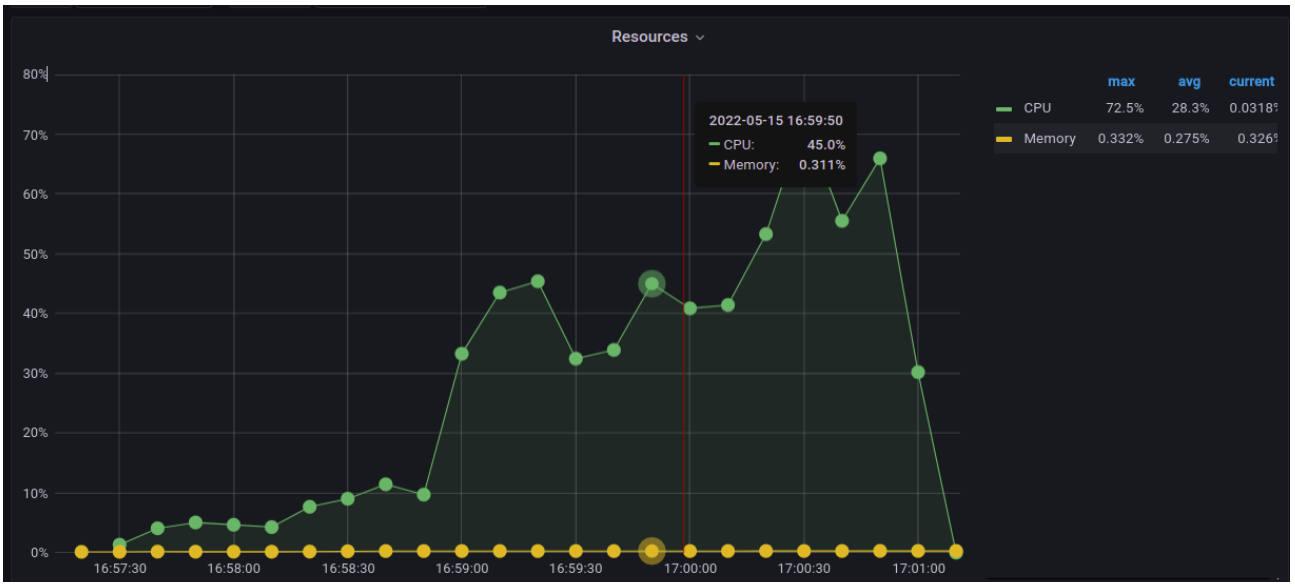

Grafana - request completadas vs request con respuesta de error



Grafana - response time a lo largo del escenario




Grafana - consumos de recursos a lo largo del escenario




## Conclusión o observaciones

Como dijimos previamente este servicio consume pocos recursos, para llegar a que sea notorio el consumo del CPU tuvimos que superar las 600 request por segundo.
La intensidad de carga no afectó la disponibilidad y performance como en los otros endpoints.

De todas formas la prueba tiene su límite, y el punto de falla es el nginx, superados los 500 request por segundos arroja este error.

**nginx_1 | 2022/05/15 20:00:54 [warn] 25#25: 1024 worker_connections are not enough, reusing connections**

Probablemente lo que está sucediendo aquí es que al haber declarado esto de forma asincrónica, por más mínimo de procesamiento que tenga el endpoint, se empiezan a encolar las request en el EVENT-LOOP de NODE. Las conexiones entre NGNIX y el servicio NODE duran más por este response time, mientras ARTILLERY sigue mandando carga y por lo tanto se siguen abriendo conexiones, entonces la cantidad de conexiones comienza a ser insuficiente y ahí aparece este error.
Luego de esto las pruebas comienzan a trabajar de forma errática, ya que se empiezan a cerrar conexiones.

## PASAMANO_9090

### Configuración escenario

```
phases:
   - name: Rampa
     duration: 10
     arrivalRate: 5
     rampTo: 10
   - name: Plano - 10 request por segundo
     duration: 30
     arrivalRate: 10
   - name: Rampa
     duration: 10
     arrivalRate: 10
     rampTo: 20
   - name: Plano  - 20 request por segundo
     duration: 80
     arrivalRate: 20
   - name: Separador
     duration: 10
     arrivalRate: 1
```

### Predicción de resultado

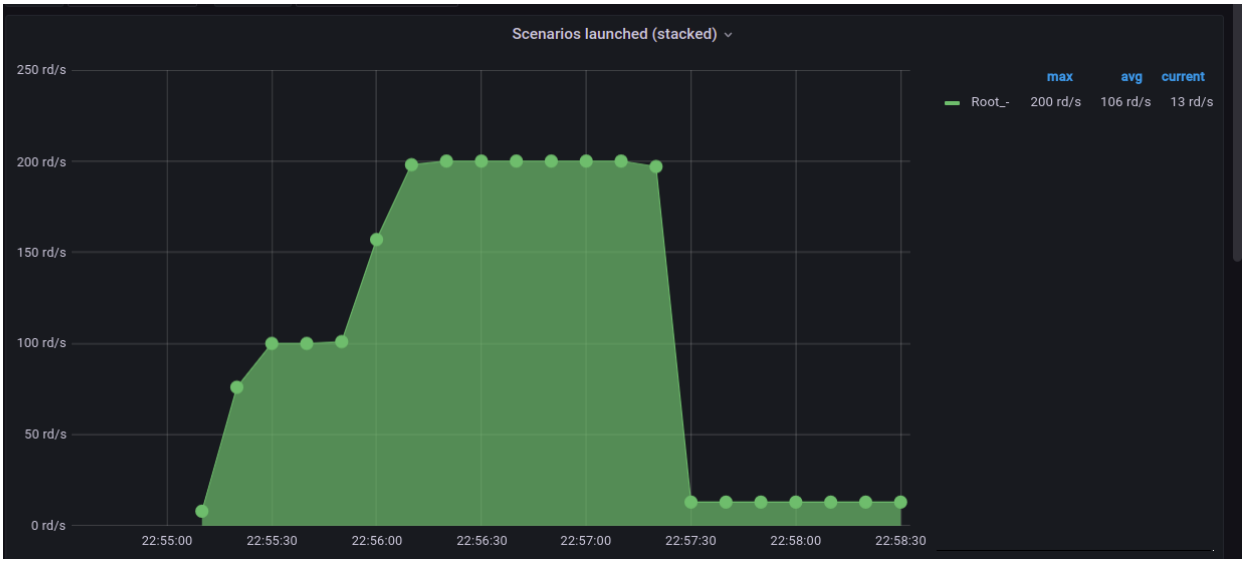
Como vimos en el escenario exploratorio para este endpoint, el response time crece linealmente a medida que se incrementa la carga y se comienzan a encolar las request, visto que este nuevo escenario tiene mayor intensidad de carga y mayor duración creemos que el cuello de botella será en el mismo artillery al alcanzar su límite de timeout de 100 segundos.

## Resultados

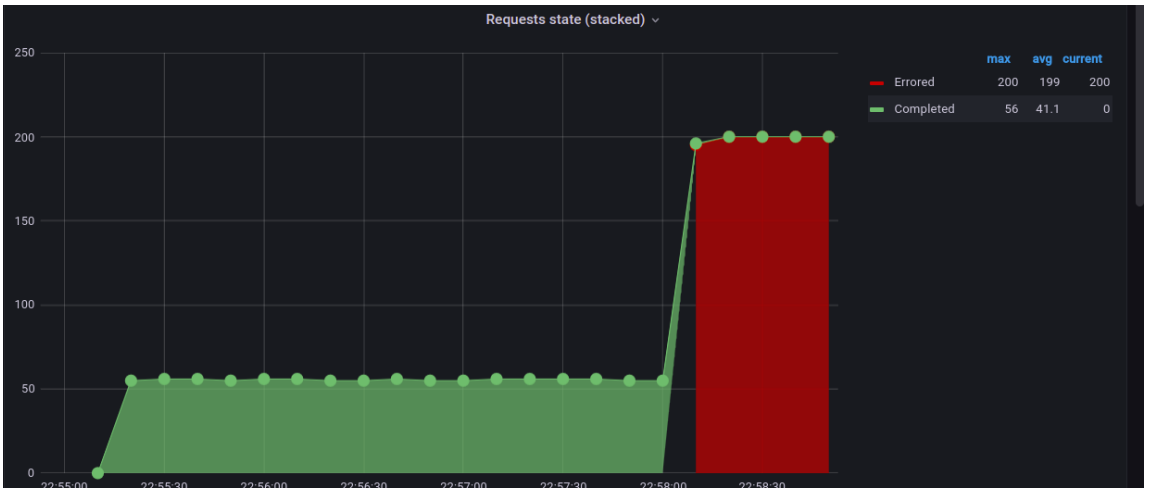
*---------------------------------*
*Summary report @ 22:58:59(-0300)*
*---------------------------------*

*errors.ETIMEDOUT: ..... 1205*
*http.codes.200: ..... 945*
*http.request_rate: ..... 9/sec*
*http.requests: ..... 2150*
*http.response_time:*
*  min: ..... 906*
*  max: ..... 99924*
*  median: ..... 38960.5*
*  p95: ..... 93932.5*
*  p99: ..... 97766.1*
*http.responses: ..... 945*
*vusers.completed: ..... 945*
*vusers.created: ..... 2150*
*vusers.created_by_name.Root (/): ..... 2150*
*vusers.failed: ..... 1205*
*vusers.session_length:*
*  min: ..... 907.4*
*  max: ..... 99925.7*
*  median: ..... 38960.5*
*  p95: ..... 93932.5*
*  p99: ..... 97766.1*

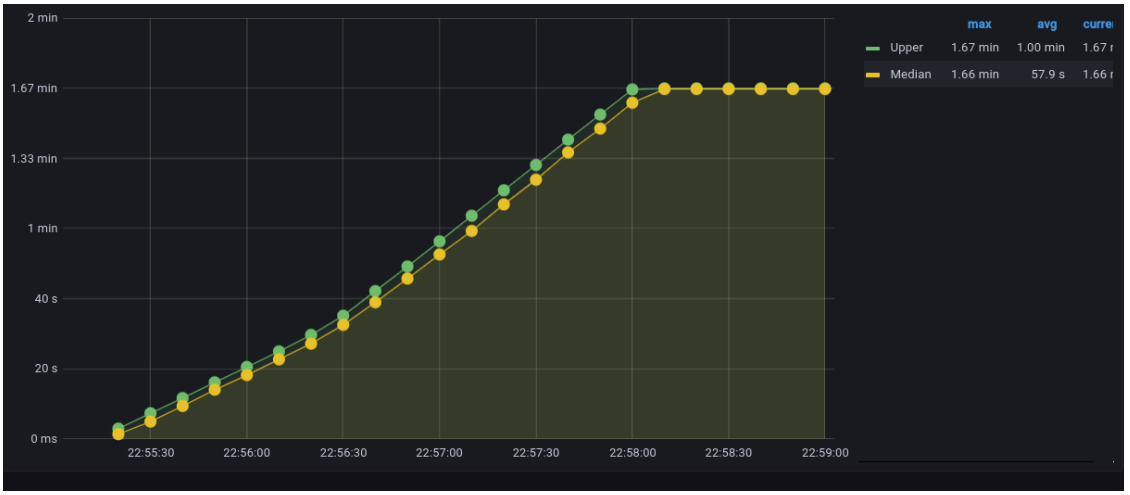
Grafana - request enviadas en el escenario



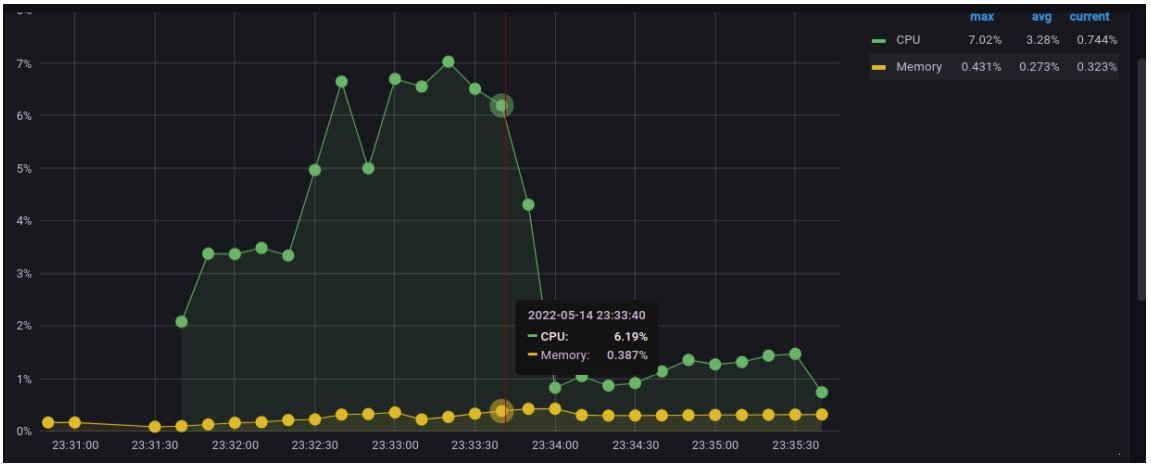
Grafana - request completadas vs request con respuesta de error




Grafana - response time a lo largo del escenario




Grafana - consumos de recursos a lo largo del escenario



### Conclusión o observaciones

El response time crece porque el servicio de bbox no llega a atender todas las request  por arriba de cierta intensidad de carga, entonces las empieza  a encolar, y esta cola cada vez es más grande mientras se mantenga o aumente la intensidad de carga de cierto umbral (este umbral se vio en el escarnio exploratorio de este endpoint).

Como dijimos en la predicción de resultados, una vez que los tiempos de respuestas superan el umbral de 100 segundos, ARTILLERY comienza a cortar la conexión de esa petición y los considera timeout. Todo esto es lo que vemos en el segundo y tercer gráfico.

Con esta configuración de un solo nodo, al aumentar la carga no encontramos perjudicar la **disponibilidad,**  pero si la **performance** cuando vemos el aumento de response time .

## PASAMANO_9091

### Configuración escenario

```
phases:
  - name: Rampa
    duration: 10
    arrivalRate: 10
    rampTo: 20
  - name: Plano - 20 request por segundo
    duration: 30
    arrivalRate: 20
  - name: Rampa
    duration: 10
    arrivalRate: 20
    rampTo: 50
  - name: Plano - 50 request por segundo
    duration: 30
```

```
    arrivalRate: 50
  - name: Rampa
    duration: 10
    arrivalRate: 50
    rampTo: 100
  - name: Plano - 100 request por segundo
    duration: 30
    arrivalRate: 100
  - name: Rampa
    duration: 10
    arrivalRate: 100
    rampTo: 500
  - name: Plano - 500 request por segundo
    duration: 50
    arrivalRate: 500
  - name: Separador
    duration: 10
    arrivalRate: 1
```

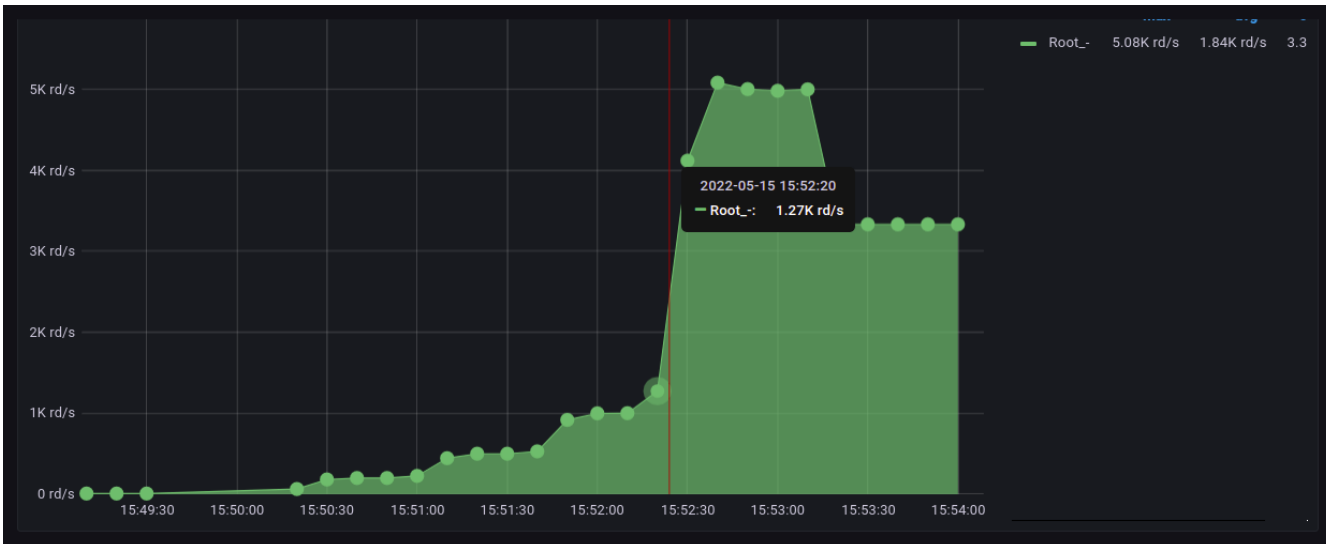
Predicción de resultado

Situación similar al análisis del servicio PASAMANO_9090, analizado en la sección anterior. Una vez que se llega al umbral de un response time mayor a 100 segundos, ARTILLERY comenzará a considerar como TIMEOUTS estas requests. Pero viendo como se comporta este servicio en base al análisis del escenario exploratorio, el response time aumenta mucho más lentamente que el PASAMANO_9090. Por eso escribimos un escenario con mayor intensidad de carga.

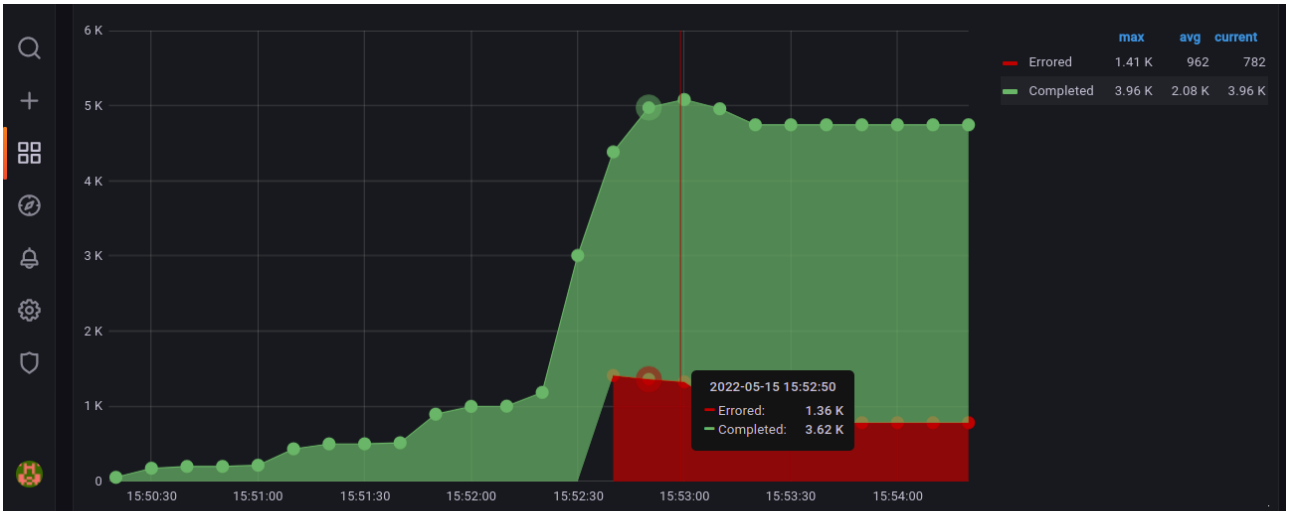
## Resultados

```
-------------------------------
Summary report @ 00:39:36(-0300)
-------------------------------

errors.ECONNRESET: ..... 8671
http.codes.200: ..... 25067
http.codes.500: ..... 556
http.codes.504: ..... 229
http.request_rate: ..... 161/sec
http.requests: ..... 34523
http.response_time:
  min: ..... 0
  max: ..... 60005
  median: ..... 3072.4
  p95: ..... 10617.5
  p99: ..... 34554.7
http.responses: ..... 25852
vusers.completed: ..... 25852
vusers.created: ..... 34523
vusers.created_by_name.Root (/): ..... 34523
vusers.failed: ..... 8671
vusers.session_length:
  min: ..... 1.2
  max: ..... 60162.1
  median: ..... 3072.4
  p95: ..... 10617.5
  p99: ..... 34554.7
```

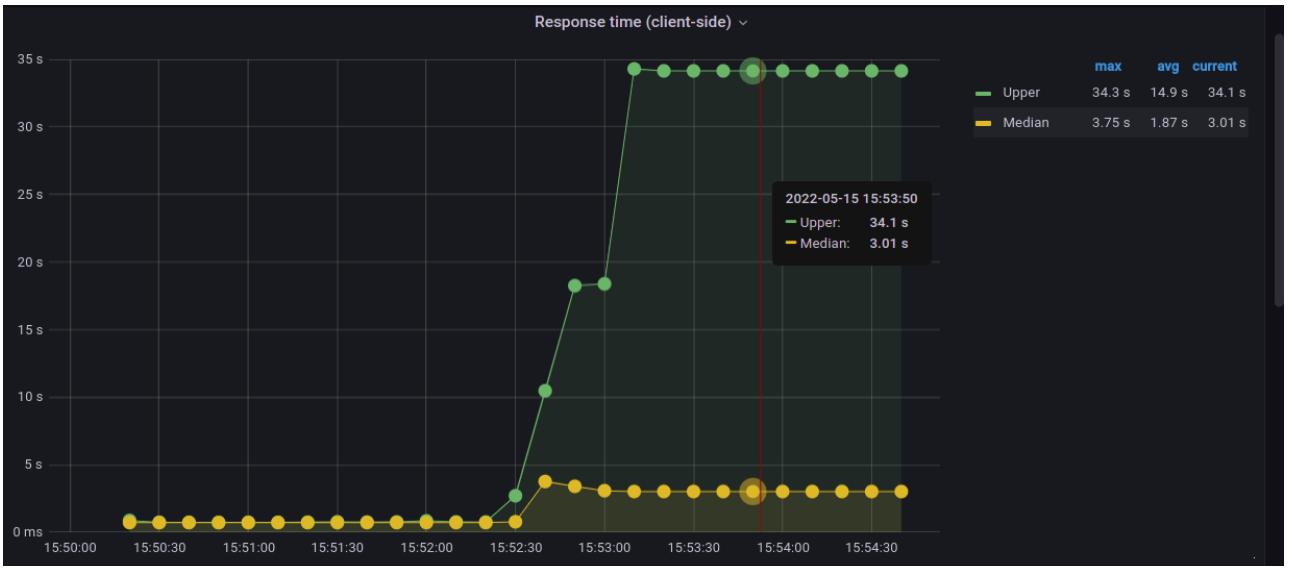
Grafana - request enviadas en el escenario



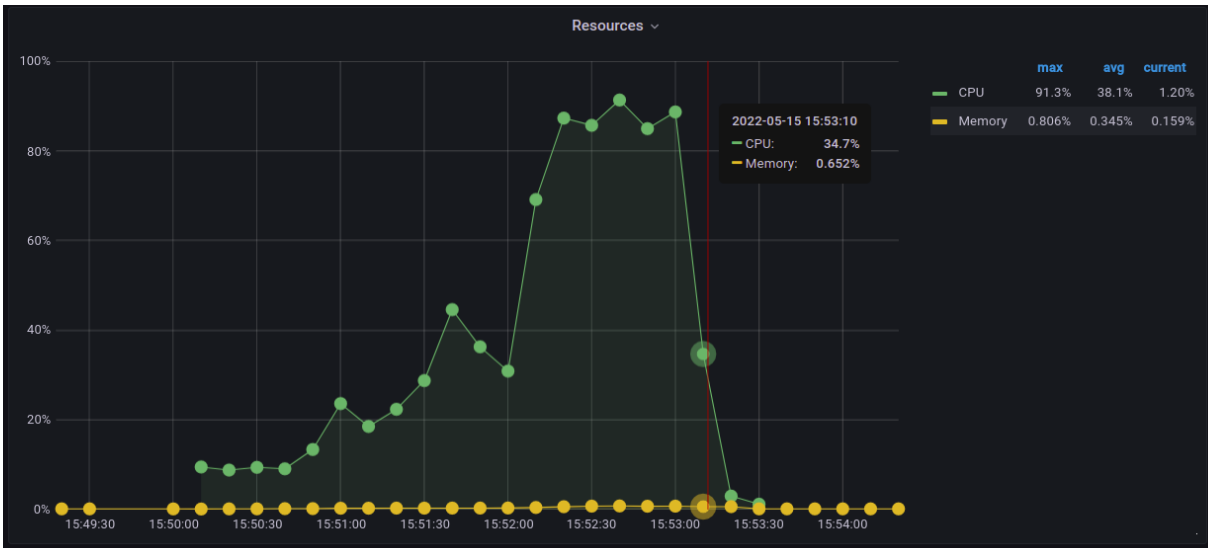
Grafana - request completadas vs request con respuesta de error




Grafana - response time a lo largo del escenario



Grafana - consumos de recursos a lo largo del escenario




## Conclusión o observaciones

El servicio comenzó a fallar, pero por otros motivos y antes de lo esperado, con este error:

**nginx_1 | 2022/05/16 01:19:48 [warn] 26#26: 1024 worker_connections are not enough, reusing connections**

La ejecución del escenario superó el límite configurado de conexiones de nginx, debido a la intensidad de carga una vez que las 500 request por segundo por segundo. El cuello de botella fue NGNIX y no la configuración de TIMEOUT de ARTILLERY.

## ENDPOINT  HEAVY

### Configuración escenario

```
phases:
  - name: Rampa
    duration: 10
    arrivalRate: 5
    rampTo: 10
  - name: Plano - 10 request por segundo
    duration: 30
    arrivalRate: 10
  - name: Rampa
    duration: 10
    arrivalRate: 10
    rampTo: 20
  - name: Plano  - 20 request por segundo
    duration: 80
    arrivalRate: 20
  - name: Separador
    duration: 10
    arrivalRate: 1
```

### Predicción de resultado

Este escenario tiene similitudes con el que prueba el 9090 de NODE, el response time crece muy rápido, y una vez que alcanza la cota de 100 segundos comienzan se comienzan considerar timeout

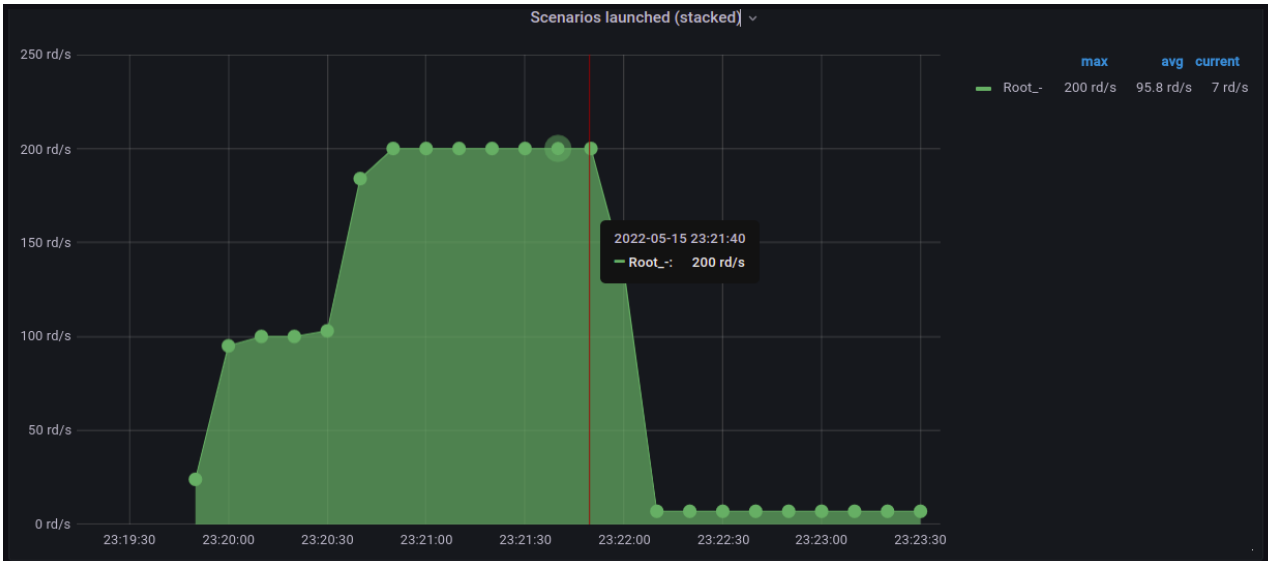
### Resultados

*---*
*Summary report @ 23:23:34(-0300)*
*---*

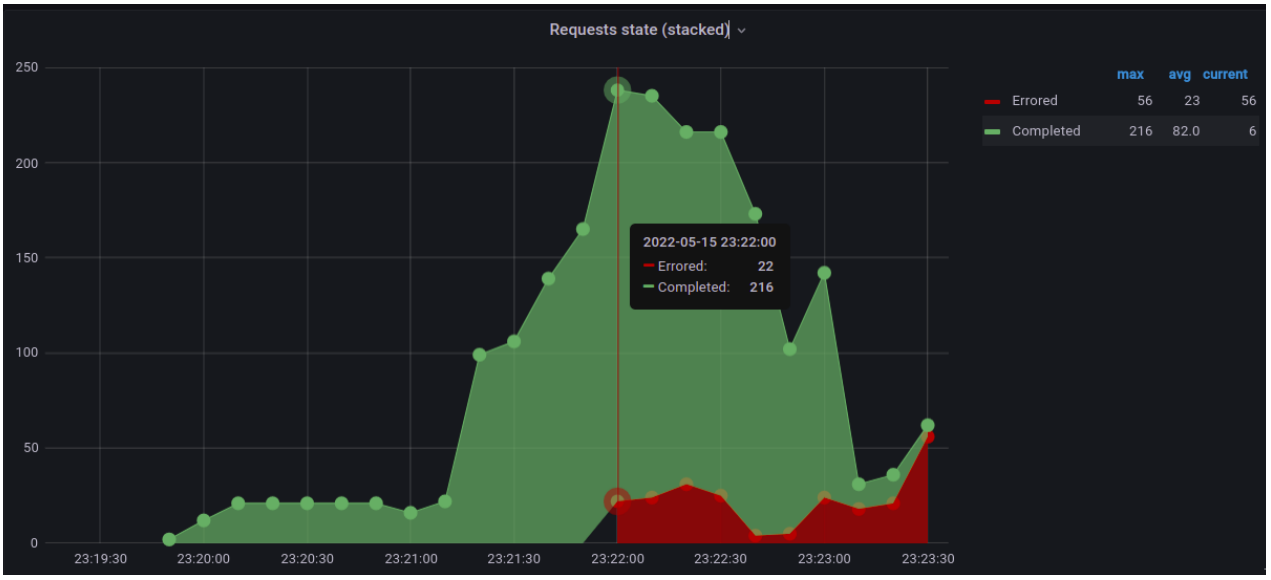
*errors.ETIMEDOUT: ..... 261*
*http.codes.200: ..... 339*
*http.codes.502: ..... 123*
*http.codes.504: ..... 1425*
*http.request_rate: ..... 5/sec*
*http.requests: ..... 2148*
*http.response_time:*
*  min: ..... 2147*
*  max: ..... 99925*
*  median: ..... 60495.1*

*p95: ..... 84993.4*
*p99: ..... 95830.1*
*http.responses: ..... 1887*
*vusers.completed: ..... 1887*
*vusers.created: ..... 2148*
*vusers.created_by_name.Root (/): ..... 2148*
*vusers.failed: ..... 261*
*vusers.session_length:*
*min: ..... 2171.5*
*max: ..... 99927.2*
*median: ..... 60495.1*
*p95: ..... 84993.4*
*p99: ..... 95830.1*

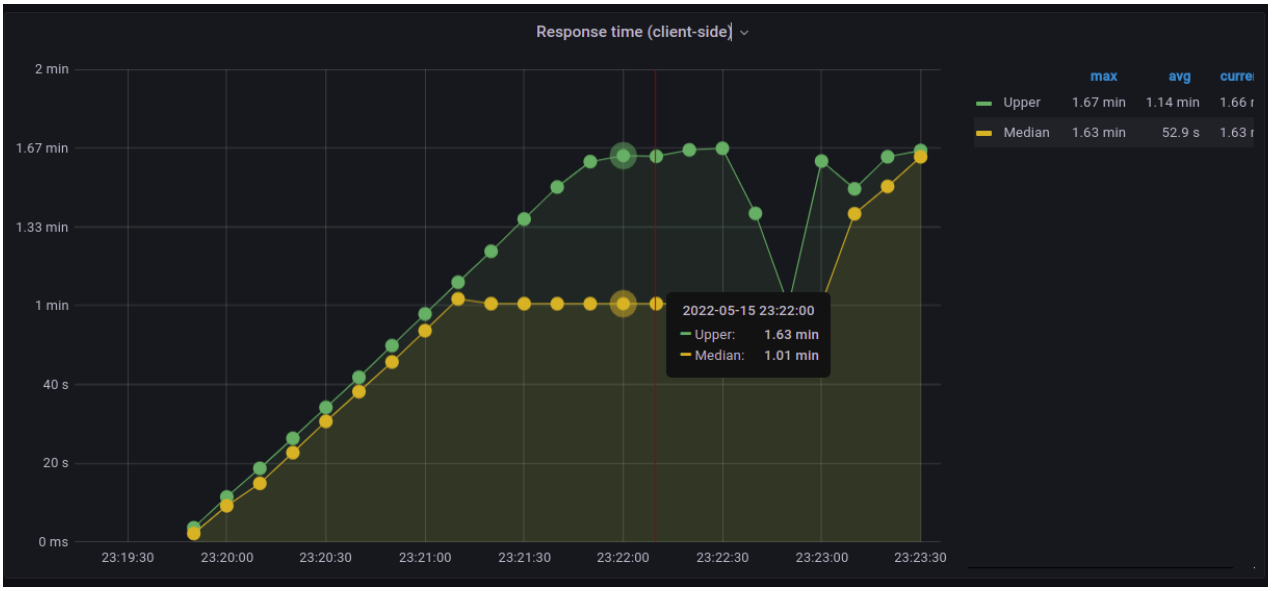
Grafana - request enviadas en el escenario




Grafana - request completadas vs request con respuesta de error



Grafana - response time a lo largo del escenario




Grafana - consumos de recursos a lo largo del escenario


## Conclusión o observaciones

Los resultados son parecidos al PASAMANO_9090, en el sentido que el response time crece de una forma bastante rápida y una vez alcanzado el TIMEOUT de ARTILLERY comienzan a aparecer los errores. Y con respecto también al 9090 también se consumen más CPU.

**Aclaración**: el gráfico del CPU no debería superar el 100 % debido a que NODE no usa más de un hilo, por ende no se usará más de un CORE, y tampoco este servicio está escalado. Lo que suponemos que sucede aquí cuando vemos un pico de 111 % es la técnica de “migration”, que usa el sistema operativo. Esta técnica aparece cuando un procesador está al 100 % y los otros no están con carga, el sistema operativo para que no

se degrade este procesador decide cambiar a otro procesador y le envía toda la carga a este.

# Configuración réplicas

Se levantaron los servicios con una configuración similar a la usada para levantar un solo nodo. Las modificaciones aplicadas son para que el servicio de nginx pueda balancear las request hacia los diferentes servicios de Node. El algoritmo de balanceo usado es el que viene por defecto: Round-Robin. Se usaron los mismos escenarios que en la sección de configuración de “un solo nodo” con carga intensiva.

Modificaciones:

**nginx_reverse_proxy.conf**

```
# List of application servers
   # el numero de puerto es el interno dentro del docker
   upstream api_servers {
      server 1c22-tp-1_node_1:3010;
      server 1c22-tp-1_node_2:3010;
      server 1c22-tp-1_node_3:3010;
   }
```

Para los escenarios se usaron los mismos.

## Escenarios intensivos

### ENDPOINT PING

Resumen de la corrida usando 3 nodos

```
--------------------------------
Summary report @ 22:21:35(-0300)
--------------------------------

http.codes.200: ..................................................... 80399
http.request_rate: .................................................. 154/sec
http.requests: ...................................................... 80399
http.response_time:
  min: ............................................................... 0
  max: ............................................................... 183
  median: ............................................................ 3
  p95: ............................................................... 19.1
  p99: ............................................................... 41.7
http.responses: ..................................................... 80399
vusers.completed: ................................................... 80399
vusers.created: ..................................................... 80399
vusers.created_by_name.Root (/): .................................... 80399
vusers.failed: ...................................................... 0
vusers.session_length:
  min: ............................................................... 1.7
  max: ............................................................... 202.4
  median: ............................................................ 11.8
  p95: ............................................................... 44.3
  p99: ............................................................... 71.5
```

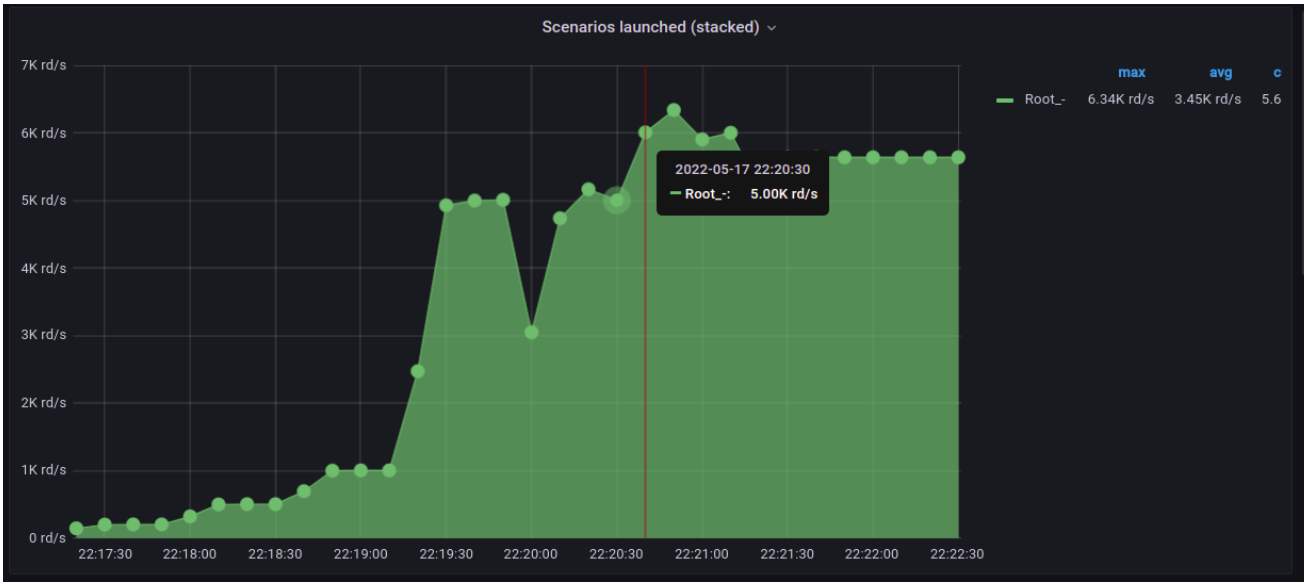



Fig 1. Gráfico launched. Ping endpoint, 3 nodos(n3)

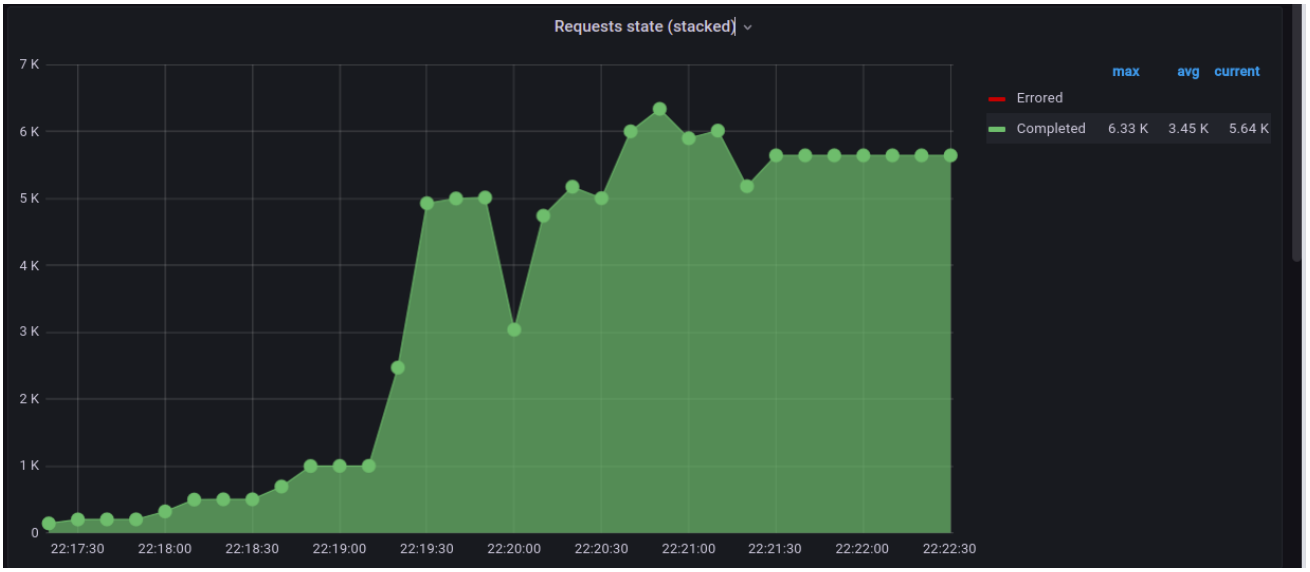



Fig 2. Gráfico request state. Ping endpoint, 3 nodos(n3)

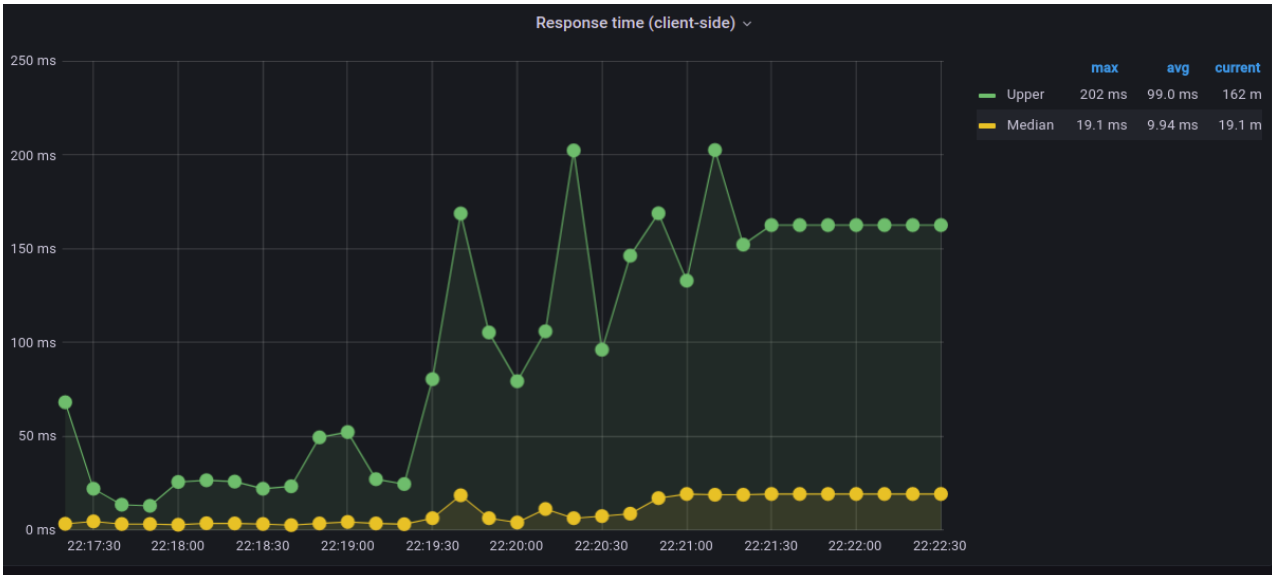



Fig 3. Gráfico response time. Ping endpoint, 3 nodos(n3)

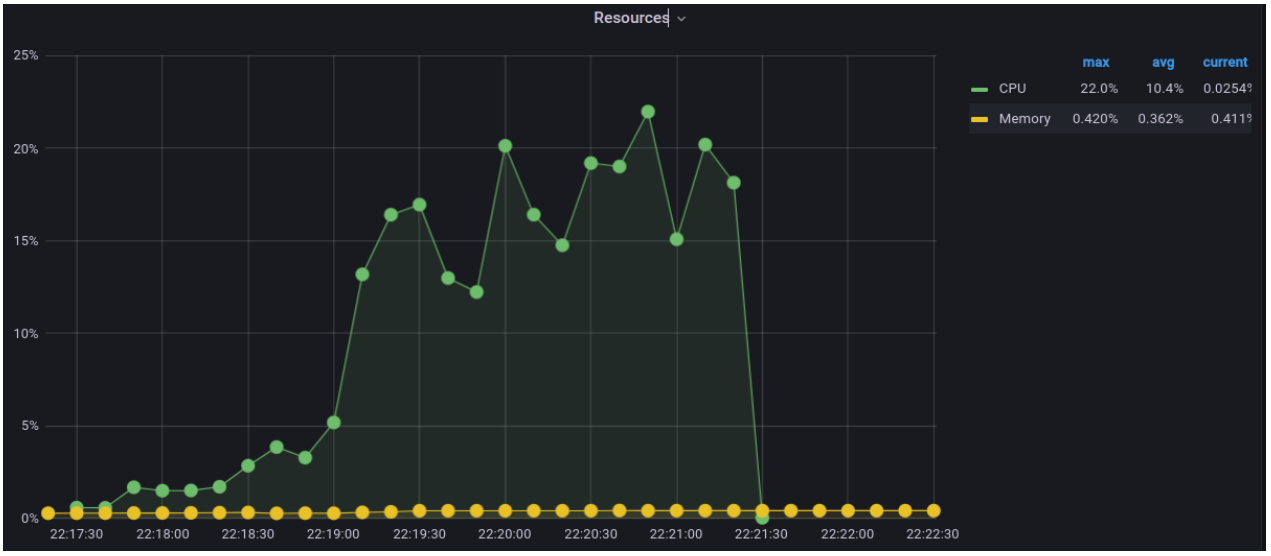



Fig 4. Resources. Ping endpoint, 3 nodos(n3)

Para esta configuración algunas diferencias son:

- El **response_time** disminuye pero muy poco respecto de la configuración simple(n1).
- El uso del CPU comparado a la configuración con solo nodo mejora, en el primer caso llegó a picos del 70% y un promedio de uso del 28 %, luego de escalarlo llegamos a un pico del 22% con un promedio de uso del CPU del 10,4%

El aumento de la cantidad de servicios Node no produjo una  gran mejora en el tiempo de respuesta medio, pero si mejora en el uso de CPU .

## PASAMANO_9090

### Configuración escenario

Mismo escenario que en la configuración de un solo nodo

```
phases:
  - name: Rampa
    duration: 10
    arrivalRate: 5
    rampTo: 10
  - name: Plano - 10 request por segundo
    duration: 30
    arrivalRate: 10
  - name: Rampa
    duration: 10
    arrivalRate: 10
    rampTo: 20
  - name: Plano  - 20 request por segundo
    duration: 80
    arrivalRate: 20
  - name: Separador
    duration: 10
    arrivalRate: 1
```

### Predicción de resultado

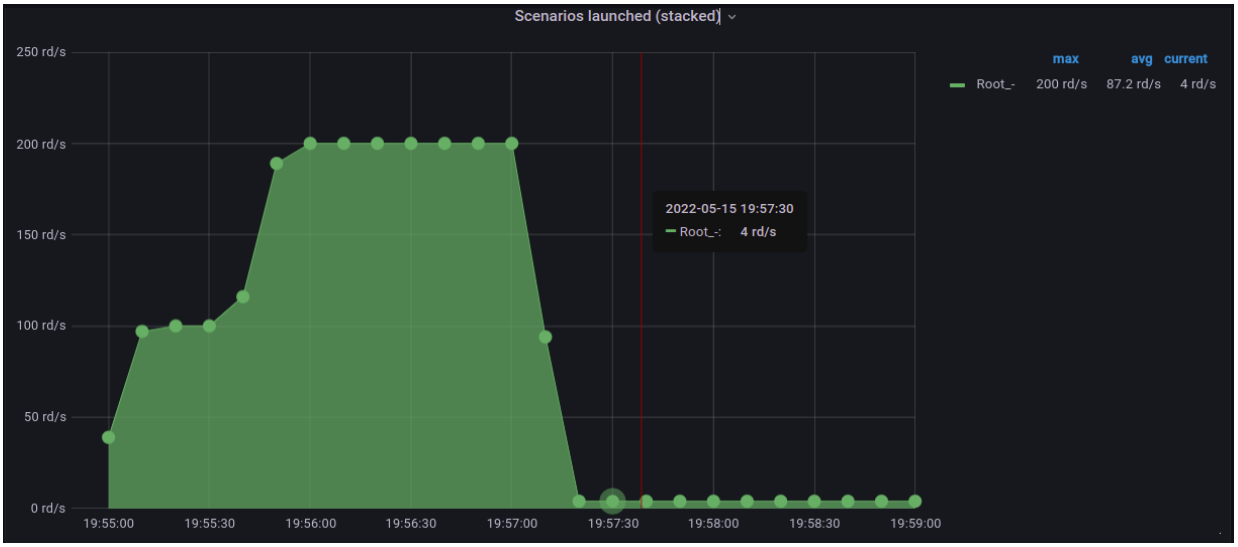
Los resultados no van a variar mucho con respecto a la configuración de un solo nodo, y es porque el cuello de botella para este caso no se encuentra en la aplicación NODE. Sino el crecimiento  del response time hasta que llega al límite de lo que artillery va a esperar por la respuesta a la request. Esto veremos reflejado en los gráficos y resultados de la ejecución.

### Resultados

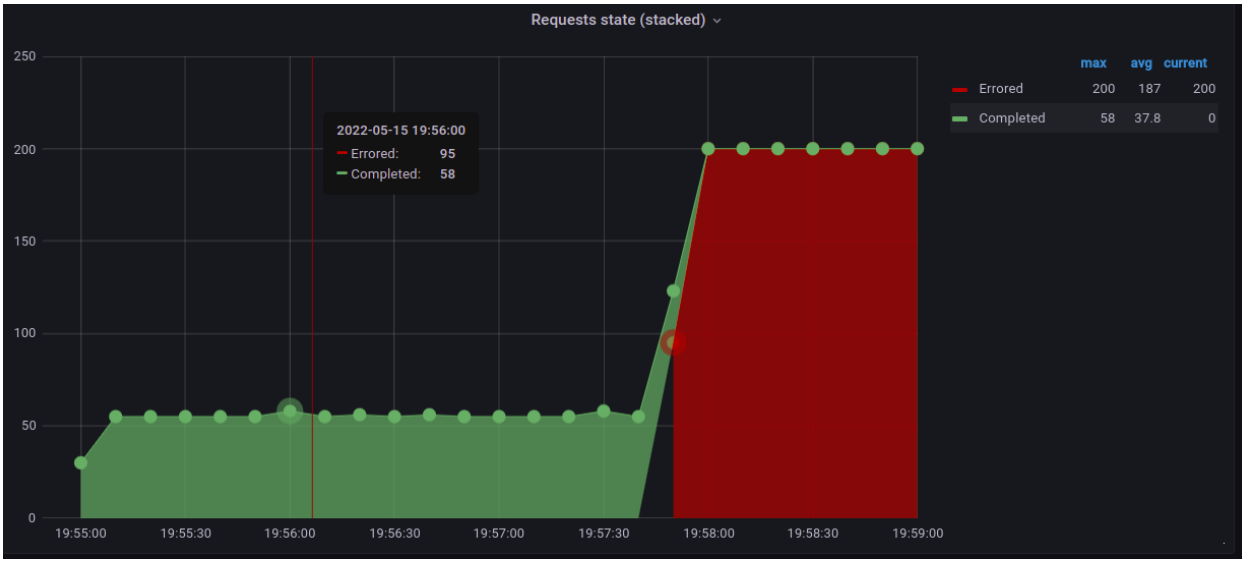
```
--------------------------------
Summary report @ 19:58:43(-0300)
--------------------------------

errors.ETIMEDOUT: ..................................................... 1193
http.codes.200: ....................................................... 946
http.request_rate: .................................................... 5/sec
http.requests: ........................................................ 2139
http.response_time:
  min: ................................................................ 973
```

*max: ..... 99978*
*median: ..... 38189*
*p95: ..... 93932.5*
*p99: ..... 97766.1*
*http.responses: ..... 946*
*vusers.completed: ..... 946*
*vusers.created: ..... 2139*
*vusers.created_by_name.Root (/): ..... 2139*
*vusers.failed: ..... 1193*
*vusers.session_length:*
*min: ..... 981.3*
*max: ..... 99980.8*
*median: ..... 38189*
*p95: ..... 93932.5*
*p99: ..... 97766.1*

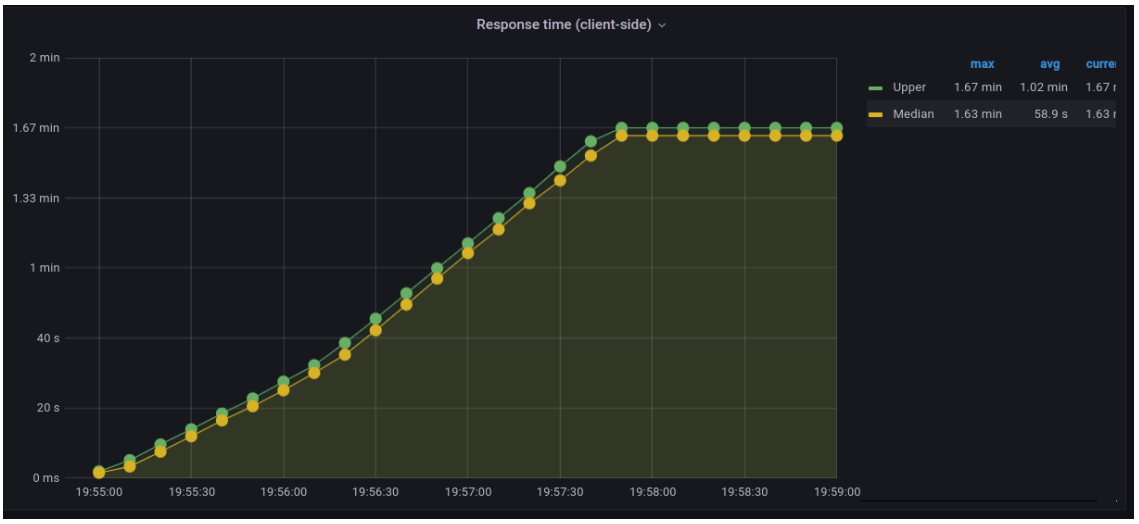
Grafana - request enviadas en el escenario



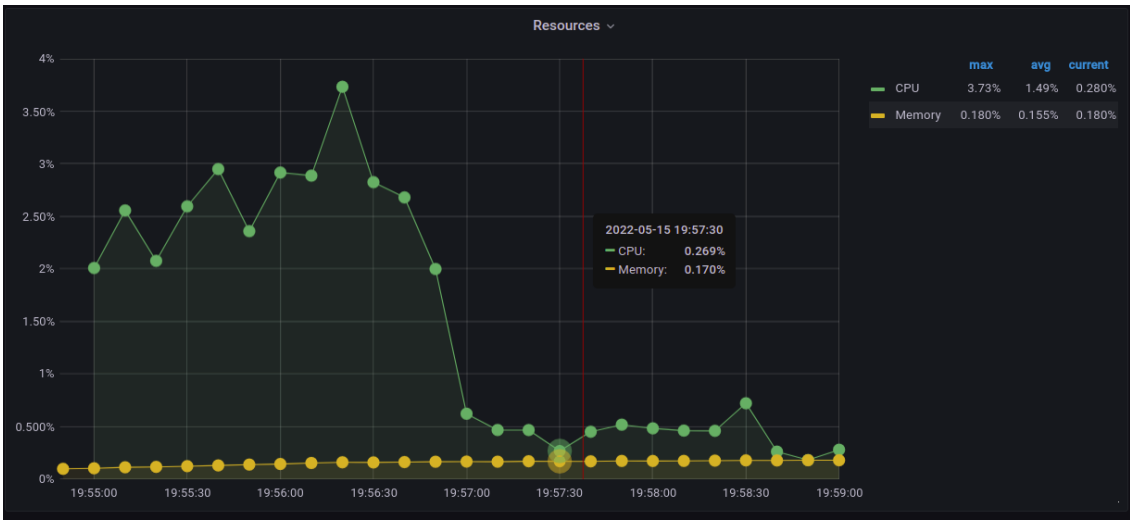

Grafana - request completadas vs request con respuesta de error



Grafana - response time a lo largo del escenario




Grafana - consumos de recursos a lo largo del escenario




## Conclusión o observaciones

Paso lo mencionado en las predicciones de resultado. La cantidad de TIMEOUTS y requests exitosos es muy parecida a la configuración de un solo nodo. Y el uso del CPU es mejor pero en ambos casos era muy poco.
El response time se comporta de forma  muy parecida y es porque el encolamiento no está en ninguna de las instancias de NODE sino en bbox, el cual no fue escalado.

PASAMANO_9091

Configuración escenario

```yaml
phases:
  - name: Rampa
    duration: 10
    arrivalRate: 10
    rampTo: 20
  - name: Plano - 20 request por segundo
    duration: 30
    arrivalRate: 20
  - name: Rampa
    duration: 10
    arrivalRate: 20
    rampTo: 50
  - name: Plano - 50 request por segundo
    duration: 30
    arrivalRate: 50
  - name: Rampa
    duration: 10
    arrivalRate: 50
    rampTo: 100
  - name: Plano - 100 request por segundo
    duration: 30
    arrivalRate: 100
  - name: Rampa
    duration: 10
    arrivalRate: 100
    rampTo: 500
  - name: Plano - 500 request por segundo
    duration: 50
    arrivalRate: 500
  - name: Separador
    duration: 10
    arrivalRate: 1
```

Resultados

---

*Summary report @ 21:15:07(-0300)*

---

*errors.ECONNRESET: ..... 13*
*http.codes.200: ..... 5041*
*http.request_rate: ..... 25/sec*
*http.requests: ..... 5054*
*http.response_time:*
*min: ..... 702*
*max: ..... 5187*
*median: ..... 757.6*
*p95: ..... 2780*
*p99: ..... 3678.4*
*http.responses: ..... 5041*
*vusers.completed: ..... 5041*
*vusers.created: ..... 5054*
*vusers.created_by_name.Root (/): ..... 5054*
*vusers.failed: ..... 13*
*vusers.session_length:*
*min: ..... 703.3*
*max: ..... 5197.8*
*median: ..... 757.6*
*p95: ..... 2780*
*p99: ..... 3752.7*

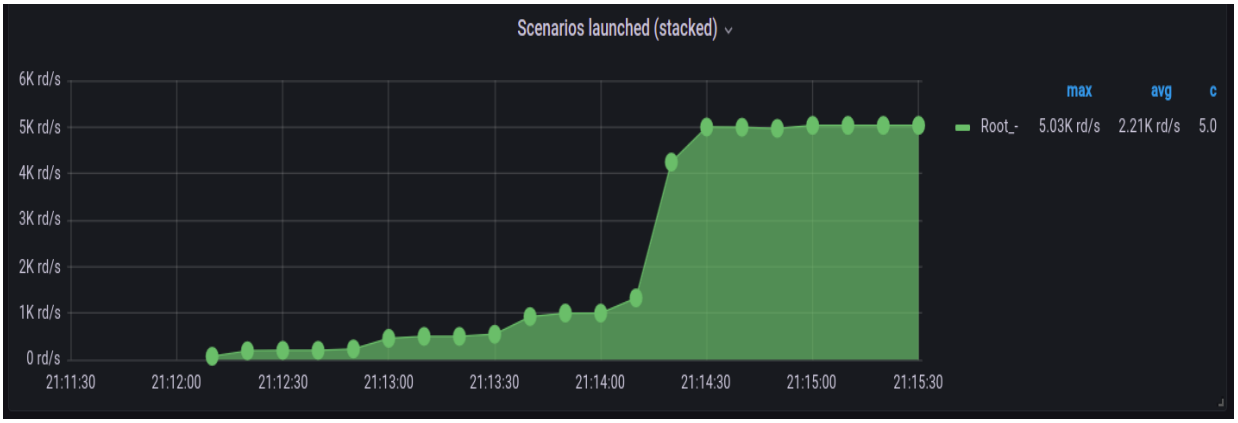



Fig 4. Gráfico launched. Bbox 9091 endpoint, 3 nodos(n3)

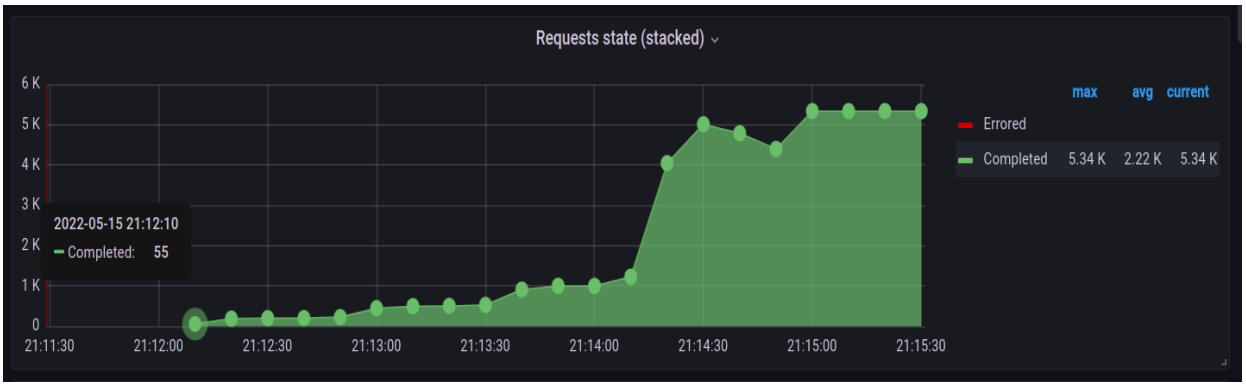



Fig 5. Gráfico request state. Bbox 9091 endpoint, 3 nodos(n3)

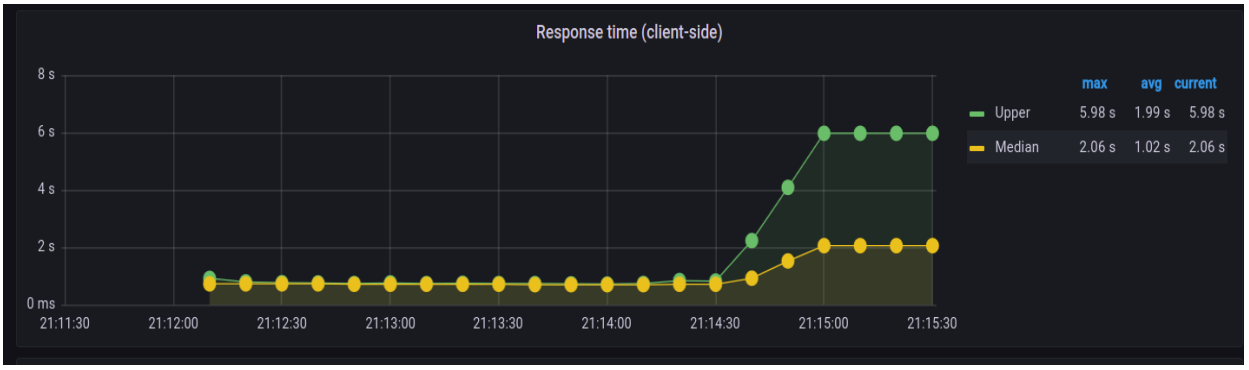



Fig 6. Gráfico response time. Bbox 9091 endpoint, 3 nodos(n3)

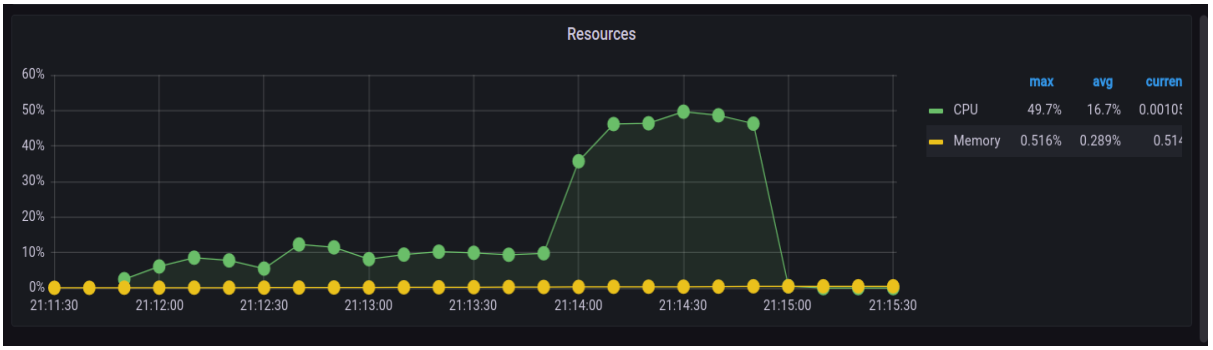



Fig 7. Gráfico resources. Bbox 9091 endpoint, 3 nodos(n3)

Durante la gran mayoría del tiempo de la prueba el uso del cpu fue importante. Se ve una disminución en el uso total respecto del caso en donde se tenía 1 solo nodo.
El uso de memoria es despreciable.

El **response_time** se ve disminuido notablemente. En el orden de 10 veces menos respecto de la prueba con 1 nodo solo.
No se ven errores en las respuestas durante toda la prueba. Todos los **response** terminan con status code 200. Lo cual nos da el indicio de que en la prueba con 1 nodo los errores se corresponden con la limitante de este servicio.

## ENDPOINT HEAVY

Este servicio no invoca a ningún otro externo, es no bloqueante pero con una operatoria matemática cíclica que consume considerables recursos. También tarda un tiempo considerable en devolver la respuesta. Hay 3 instancias de este servicio detrás de un balanceador de carga

### Configuración escenario

Mismo escenario que en la configuración de un solo nodo de carga intensiva

```
phases:
  - name: Rampa
    duration: 10
    arrivalRate: 5
```

```
    rampTo: 10
  - name: Plano - 10 request por segundo
    duration: 30
    arrivalRate: 10
  - name: Rampa
    duration: 10
    arrivalRate: 10
    rampTo: 20
  - name: Plano  - 20 request por segundo
    duration: 80
    arrivalRate: 20
  - name: Separador
    duration: 10
    arrivalRate: 1
```

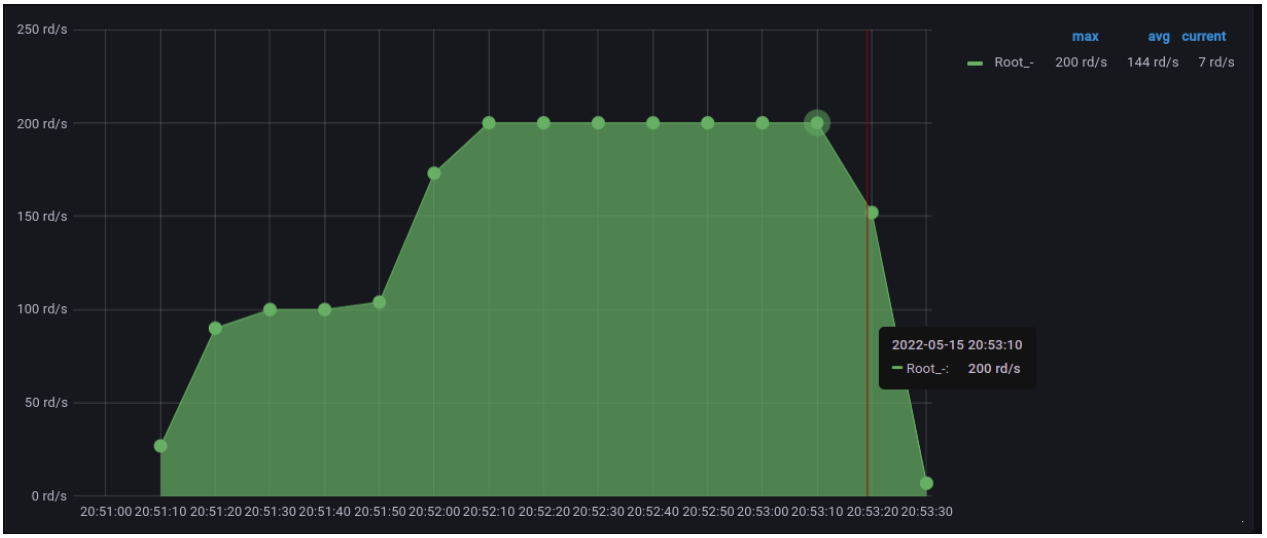
### Predicción de resultado

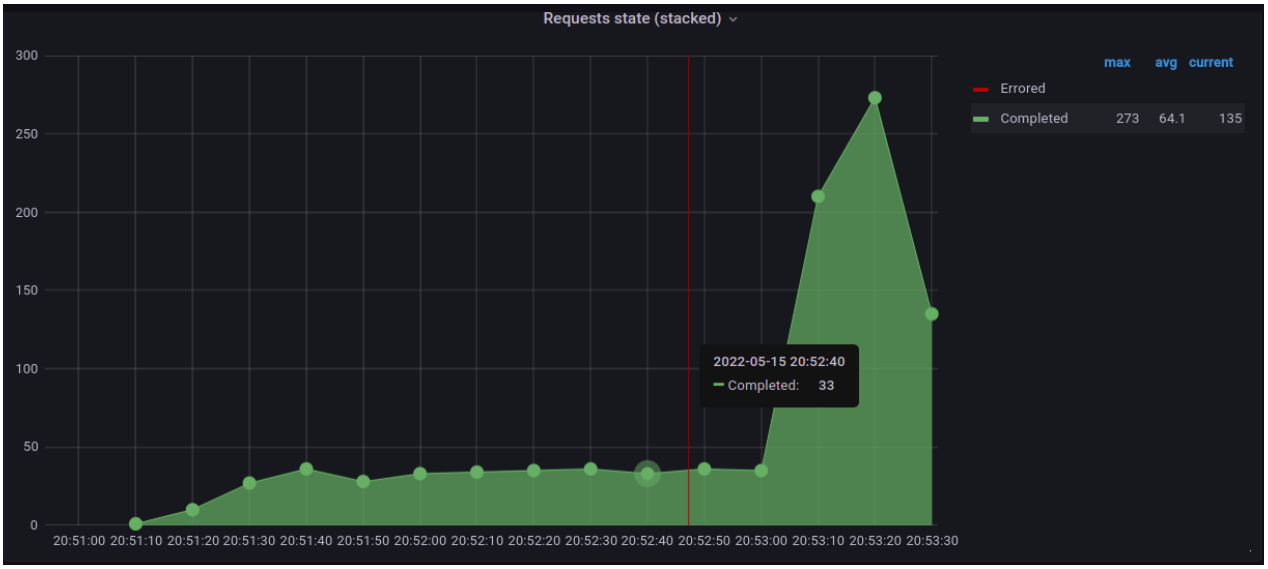
Para este caso si esperamos ver cambios con respecto al escenario intensivo de un solo nodo para el endpoint heavy. Esperamos ver que aumenta la disponibilidad y menor uso de CPU al distribuir la carga sobre los demás procesadores.

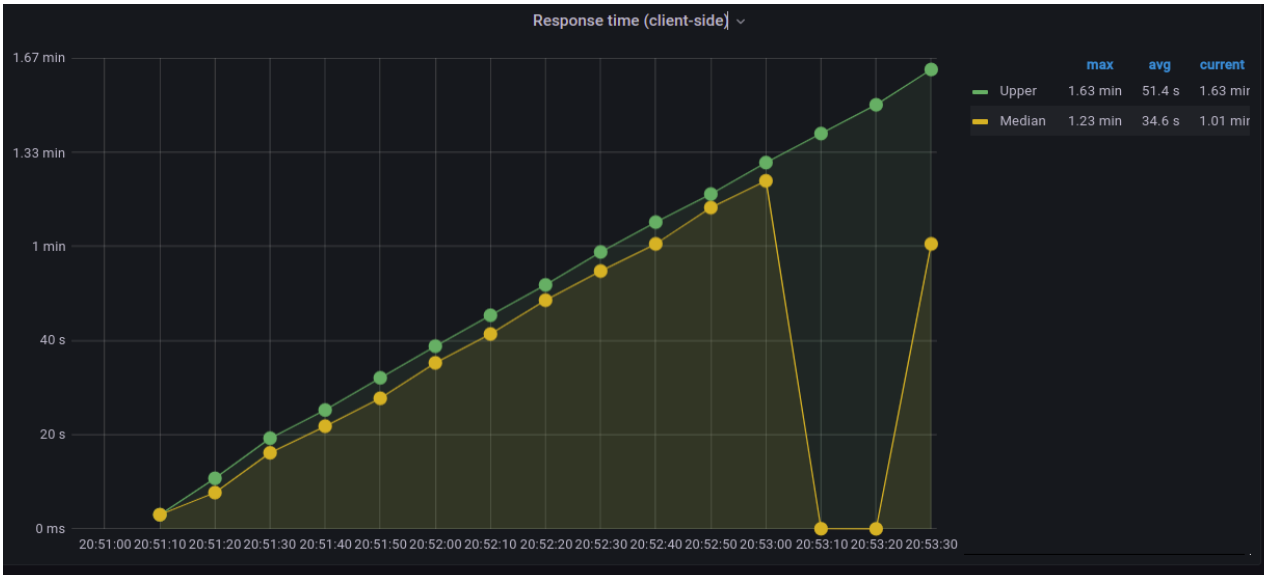
### Resultados

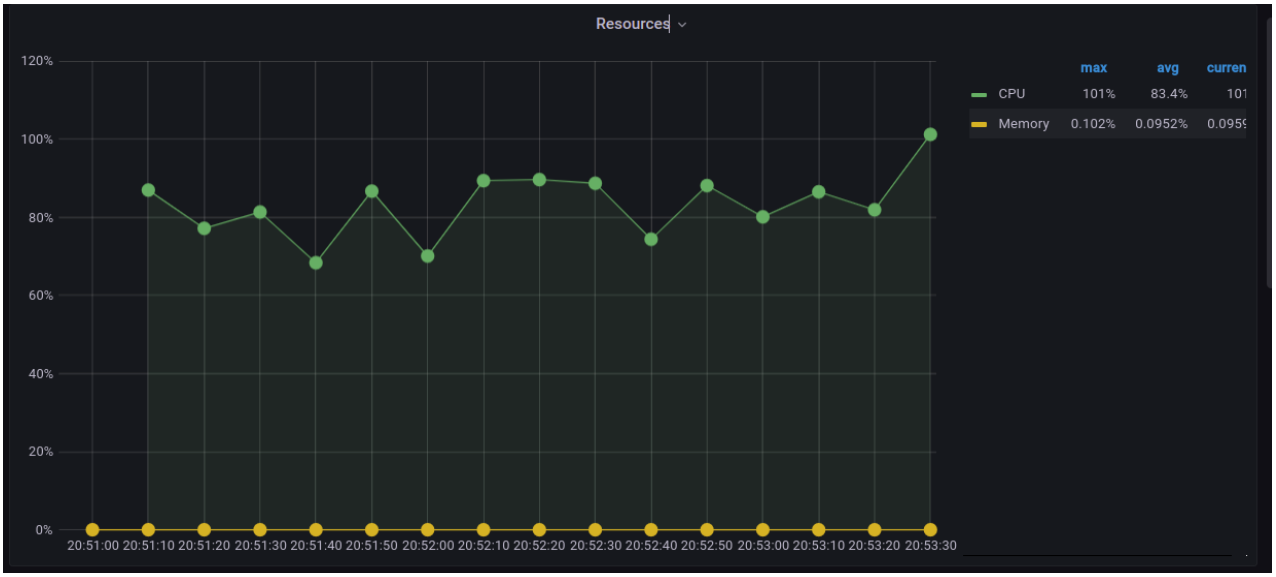
*---*
*Summary report @ 20:54:42(-0300)*
*---*

*errors.ETIMEDOUT: ..... 770*
*http.codes.200: ..... 463*
*http.codes.502: ..... 920*
*http.request_rate: ..... 5/sec*
*http.requests: ..... 2153*
*http.response_time:*
*  min: ..... 0*
*  max: ..... 99784*
*  median: ..... 60495.1*
*  p95: ..... 86710.4*
*  p99: ..... 97766.1*
*http.responses: ..... 1383*
*vusers.completed: ..... 1383*
*vusers.created: ..... 2153*
*vusers.created_by_name.Root (/): ..... 2153*
*vusers.failed: ..... 770*
*vusers.session_length:*
*  min: ..... 2*
*  max: ..... 99792.9*
*  median: ..... 60495.1*
*  p95: ..... 86710.4*
*  p99: ..... 97766.1*


250 rd/s
200 rd/s
150 rd/s
100 rd/s
50 rd/s
0 rd/s
max avg current
Root_- 200 rd/s 144 rd/s 7 rd/s
2022-05-15 20:53:10
Root_-: 200 rd/s
20:51:00 20:51:10 20:51:20 20:51:30 20:51:40 20:51:50 20:52:00 20:52:10 20:52:20 20:52:30 20:52:40 20:52:50 20:53:00 20:53:10 20:53:20 20:53:30


Requests state (stacked)
300
250
200
150
100
50
0
max avg current
Errored
Completed 273 64.1 135
2022-05-15 20:52:40
Completed: 33
20:51:00 20:51:10 20:51:20 20:51:30 20:51:40 20:51:50 20:52:00 20:52:10 20:52:20 20:52:30 20:52:40 20:52:50 20:53:00 20:53:10 20:53:20 20:53:30


Response time (client-side)
1.67 min
1.33 min
1 min
40 s
20 s
0 ms
max avg current
Upper 1.63 min 51.4 s 1.63 mir
Median 1.23 min 34.6 s 1.01 mir
20:51:00 20:51:10 20:51:20 20:51:30 20:51:40 20:51:50 20:52:00 20:52:10 20:52:20 20:52:30 20:52:40 20:52:50 20:53:00 20:53:10 20:53:20 20:53:30




Conclusión o observaciones

De acuerdo a lo que sabemos y venimos viendo, una vez escalado el servicio tenemos 3 instancias cada una con su event loop, y cada una con su cola donde se van acumulando las peticiones. Esperaríamos que al tener 3 colas, tardarían más tiempo en acumular suficiente cantidad de request  hasta que el response time llegue al umbral del 100 segundos y luego comience a fallar ARTILLERY debido a su seteo de TIMEOUT, esto no paso. Pensamos que configuramos algo mal o nos estamos perdiendo algún detalle. Pero en conclusión tendría que mejorar la disponibilidad, al menos por un tiempo hasta que todas las colas otra vez alcancen un tamaño suficiente para que el response time llegue otra vez a 100 y vuelvan a aparecer timeouts.

Otra de las principales ventajas en cuanto al uso de recursos es que si bien NODE  no es multithreading, con el escalamiento podemos obtener uso de los diferentes núcleos del procesador al correr cada una en un core distinto.

# Tabla comparativa

Los escenarios de la siguiente tabla son los intensivos (no exploratorios)

| | **Configuración 1 nodo** | | | | **Configuración 3 nodos** | | | |
|---|---|---|---|---|---|---|---|---|
| Escenario/End point | ping | Bbox 9090 | Bbox 9091 | heavy | ping | Bbox 9090 | Bbox 9091 | heavy |
| request_rate | *156/seg* | *9/seg* | *161/seg* | *5/sec* | *154/sec* | *5/sec* | *25/sec* | *5/sec* |
| response_time median[ms] | *4* | *38960.5* | *10617.5* | *60495.1* | *3000* | *38189* | *757.6* | *60495.1* |
| CPU avg | *28,3%* | *3,28%* | 38,1% | 100% | 10,4% | 1,49% | 16,7% | 83,4% |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Memory | *~0* | *0,27%* | 0,35% | 0,10% | 0,41% | 0,16% | 0,29% | 0,10% |
| errors/completed avg | *0/80268* | *199/41.1* | 962/2080 | 23/82 | 0/3.45 k | 187/37,8 | 0/2220 | 0/64,1 |

# Sección 2

## 1- Asincrónico/Sincronico

En vista de los resultados analizados en la sección 1. Podemos ver claramente que el response time crece de forma mucho más rápida en el servicio 9090 en comparación con el 9091.

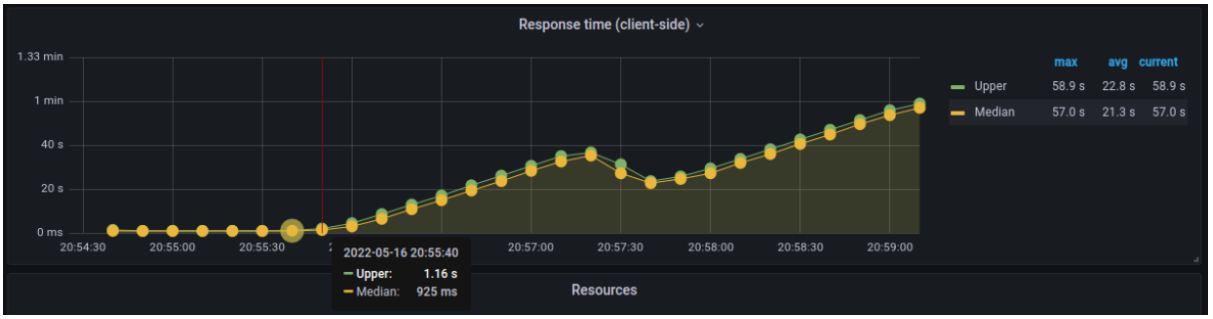
Como sabemos en los procesamiento asincrónicos son más rápidos que los sincrónicos debido a que no se esperan las respuestas de las funciones promesas, entonces como vemos que el servicio 9090 tiene un mayor response time, inferimos que es el servicio sincrónico y el 9091 es el asincrónico.

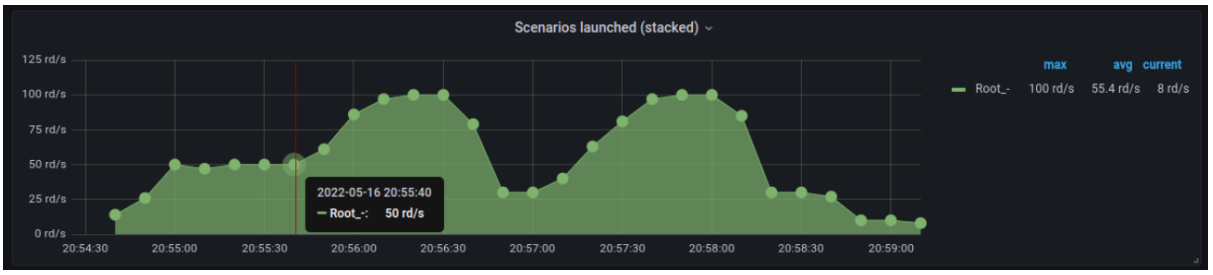
## 2- Cantidad de workers:

Para calcular la cantidad de workers del servicio sincrónico vamos a tener en cuenta los resultados obtenidos con un nodo en el caso exploratorio; hay un momento en la curva de response time que empieza a crecer aproximadamente de forma lineal debido al aumento de carga enviada, dando a entender que se empiezan a encolar los requests.

Debido a esto, tal como lo mencionado en las conclusiones, ese tiempo de “encolado” comienza a ser una componente del response (la otra componente es el tiempo de procesamiento propio del endpoint de **BBOX 9090**).

Entonces deberíamos calcular la cantidad de workers (teniendo en cuenta la cantidad de requests enviados por segundo (rps) y el tiempo de respuesta en ese momento) en el instante anterior  a que se que se empiezan a encolar las request (punto que corresponde a cuando se comienza a aumentar la carga), es decir especialmente en el siguiente punto:




*Estos últimos gráficos corresponden al escenario ejecutado con un solo nodo de la fase exploratoria para el endpoint pasamano_9090.*

Para obtener la cantidad de workers deberíamos tener en cuenta que, en el gráfico de la carga enviada, se visualiza acumulandose de a 10 segundos los requests y en ese punto para obtener la cantidad de requests enviados por segundos (rps) se debe dividir por 10, por ende el servicio de **bbox 9090** procesa 5 rps previo a empezar la congestión al mandar mayor carga. En ese instante, se tiene un response time de 0.925seg para procesar los requests, entonces **5 rps/0.925seg = 5.4054**, redondeando a 5, podemos decir de forma aproximada que el servicio de bbox dispone 5 workers para procesar.

## 3 - Demora en responder

Para hacer estas mediciones creamos 2 endpoints para medir cada uno de los servicios. La estrategia es guardar el la fecha/hora antes de invocarlo y luego guardar la fecha/hora después de la invocación de forma.

Con esto calculamos:

- 932 ms para 9090
- 709 ms para 9091

# Conclusiones

Como vimos a lo largo de todas las mediciones pudimos predecir muchos de los resultados y algunos nos sorprendieron. En muchas situaciones el cuello de botella lo encontramos en lugares donde no pensábamos, como por ejemplo la propia herramienta artillery.

Podemos concluir que cambiar la arquitectura para mejorar un atributo de calidad puede impactar negativamente en otro. Como nos pasó con al aumentar los TIMEOUTS de

ARTILLERY y NGNIX, esto nos dio mayor disponibilidad  en general pero en muchos otros casos repercutio de forma negativa en la **performance** ya que incrementó considerablemente el RESPONSE TIME.

También notamos que mejorar la **escalabilidad**  de forma horizontal no tuvo un impacto positivo en todos los endpoints, por ejemplo en el PING no mejoró el response time. Tampoco tuvo mejoras notables en el endpoint 9090 ya que cuello de botella donde se encolan las request era bbox, y no era este el servicio que estábamos escalando, sino el servicio NODE.

En definitiva, analizar la arquitectura, hacer mediciones bajo situaciones críticas y no criticas, conocer el límite de cada componente de nuestra arquitectura y otros análisis son factores de éxitos cruciales en la toma de decisiones, como por ejemplo donde invertir recursos.