Trabajo Práctico 1

Arquitectura de Software

Segundo Cuatrimestre 2020

Grupo COMB

Integrantes:

Olivia Fernandez	99732
Matias Fonseca	98591
Cecilia Hortas	100687
Brian Zambelli Tello	98541

Sección 1

Introducción

El objetivo de esta sección es analizar los resultados obtenidos de someter a distintas pruebas a los endpoints de dos servicios HTTP implementados en Node y Python y que corren en los web servers Express y Gunicorn respectivamente.

Se busca utilizar herramientas como Artillery, Graphite, Grafana, CAdvisor y Nginx para recopilar información de los endpoints y poder llevar adelante las pruebas con el uso de escenarios de carga y de un dashboard para visualizar diversas métricas.

Los endpoints que se utilizarán como punto de partida para la comparación entre ambos servidores son:

- **Ping**: respuesta de un valor constante sin procesamiento que representa el *healthcheck* básico
- **Timeout**: el servidor no responde el request inmediatamente sino que pausa unos segundos para simular la llamada a otro servicio (sin procesamiento de datos)
- **Intensivo**: loop de cierto tiempo para simular cálculos pesados sobre los datos

Los casos de análisis que se utilizarán para la comparación son:

- **Node**: un servidor en Node de un worker y un container
- **Gunicorn**: un servidor en Python con Gunicorn de un worker sincrónico y un container
- **Node replicado**: un servidor en Node de un worker replicado en 3 containers con el uso de Nginx como *load balancer*

Por último lugar, el análisis expuesto se basará en las pruebas de performance:

- **Load testing:** se busca testear el sistema al aumentar la carga en el sistema hasta que la carga llega a su valor umbral
- **Stress testing:** se busca testear la estabilidad del software cuando los recursos de hardware no son los suficientes por lo que se carga al sistema con una cantidad de procesos que no pueden ser manejados con los recursos del sistema

Pruebas de performance

En línea con lo mencionado previamente se detallan a continuación las pruebas realizadas para los distintos endpoints y servers.

Load testing

Las pruebas consisten de 4 fases distintas:

1. **Cleanup**: con una duración de 30 segundos donde no se envían requests
2. **Ramp**: con una duración de 30 segundos, se envían 5 request por segundo, aumentando hasta llegar a 30 request por segundo
3. **Plain:** con una duración de 120 segundos, se envían 30 request por segundo
4. **Cleanup**: con una duración de 30 segundos donde no se envían requests

Stress testing

Estas pruebas mantienen una estructura similar a las de *load testing* solo que se busca realizar un uso más intensivo del servidor. De esta manera se mantienen las mismas fases pero se aumenta la cantidad de request por segundo y la duración del período de ramp.

1. **Cleanup**: con una duración de 30 segundos donde no se envían requests
2. **Ramp**: con una duración de 60 segundos, se envían 5 request por segundo, aumentando hasta llegar a 600 request por segundo
3. **Plain:** con una duración de 120 segundos, se envían 600 request por segundo
4. **Cleanup**: con una duración de 30 segundos donde no se envían requests

La elección de la cantidad tope de request por segundo a la que se somete el servidor se basa en que es 20 veces más que la prueba de load, un número que debería sobrecargar a los recursos de hardware del servidor. De la misma manera se aumenta la cantidad de tiempo del ramp para que se encolen requests y pueda observarse en el gráfico cómo se comportan los servidores ante una cantidad muy grande de requests en un período más largo de tiempo.

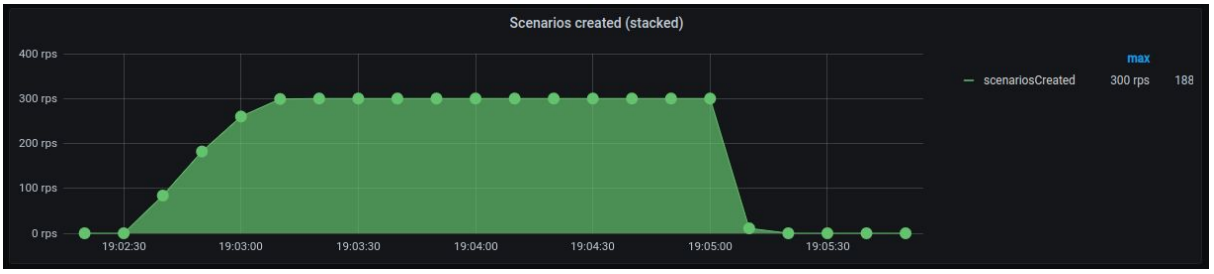
Ping

Es un endpoint que representa un *healthcheck* básico por lo que el procesamiento es mínimo. Por lo tanto, se espera un tiempo de respuesta bajo y también un consumo de recursos escaso.

Node

Load testing

Escenarios creados

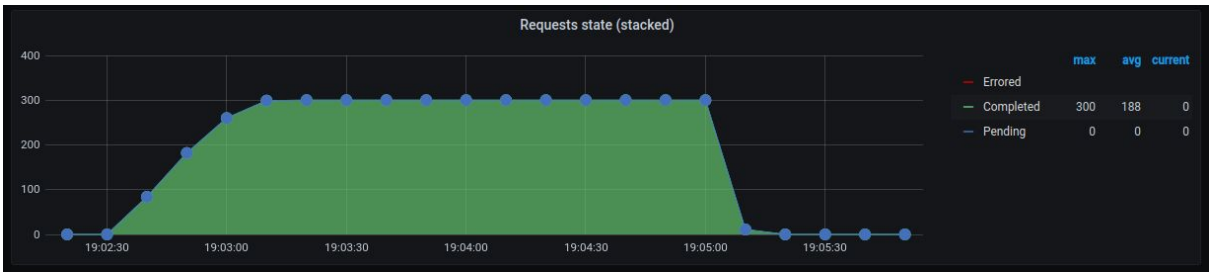


Se pueden visualizar la cantidad de escenarios creados en forma de request por segundo. Se muestran las fases que constituyen el escenario, en un primer lugar el cleanup con 0 rps como fue definido, luego el ramp con una duración de 30 segundos donde aumentan la cantidad de request hasta llegar a 30 request por segundo y luego la meseta donde se mantiene esta frecuencia de request para luego terminar con un cleanup final.

Este gráfico tendrá una forma equivalente en las siguientes pruebas de carga que se expondrán del endpoint ping por lo que no se incluirá.

Se puede notar que el gráfico muestra el valor de 30 request por segundo aumentado en un factor de 10 y esto es porque se muestran los request por segundo en los últimos 10 segundos.

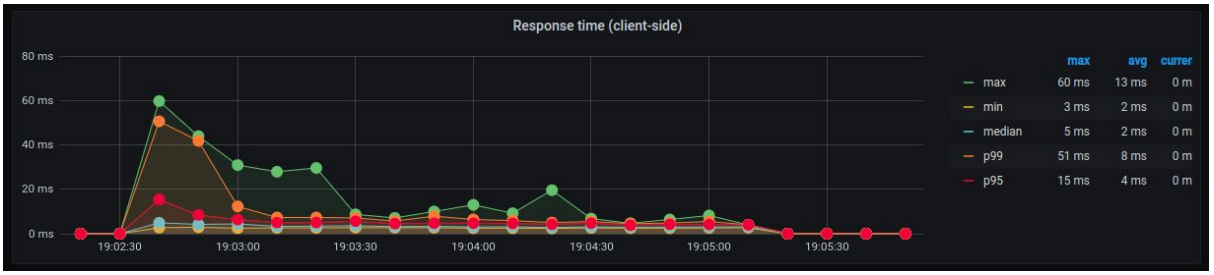
Requests completados, pendientes y fallidos



Se muestran la cantidad de request completados en el período de tiempo que dura el escenario. Acá se puede observar que para todo instante no queda ningún request pendiente por lo que se completaron en su totalidad. Esto es consecuencia de lo

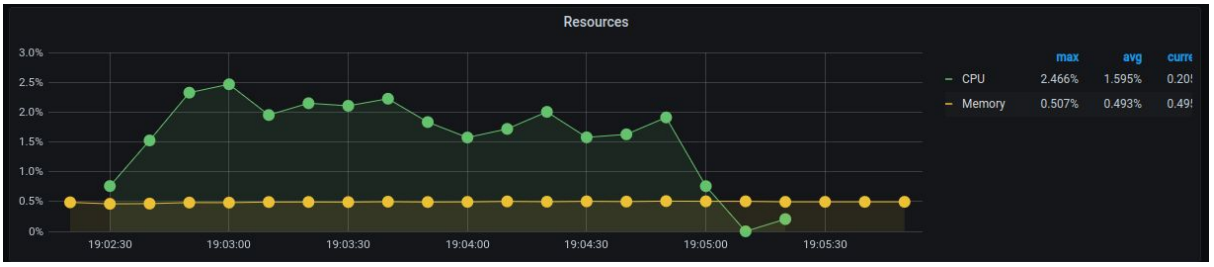
mencionado previamente sobre el bajo procesamiento del endpoint que implica una respuesta casi inmediata. Así mismo, la cantidad de request enviados por segundo se condice con la del gráfico anterior.

Tiempo de respuesta



Se pueden observar distintas medidas del tiempo de respuesta como el mínimo, el máximo, la mediana y distintos percentiles. La mediana a lo largo de toda la prueba tiene un valor bajo, lo cual se corresponde con el procesamiento mínimo propio del endpoint. Además, inicialmente se observa un pico que podría estar vinculado con el pico que se observa en el gráfico de recursos que se muestra a continuación en cuanto al uso de la CPU.

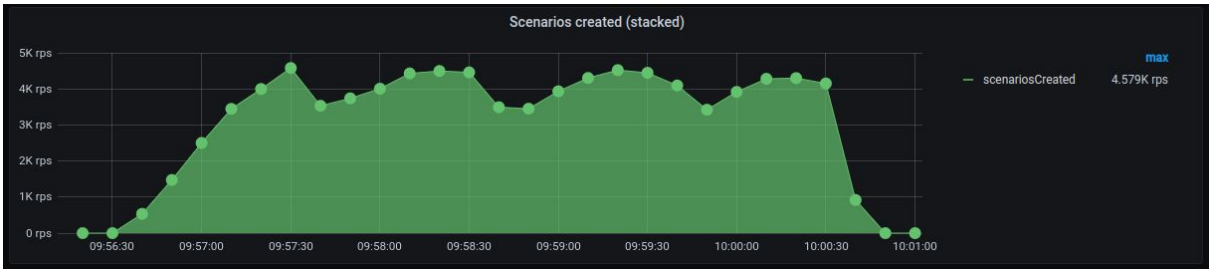
Recursos



Se puede afirmar que el uso de la CPU se mantiene bajo durante toda la prueba y así mismo el uso de memoria, en línea con el análisis expuesto previamente. Notar que el uso de memoria es constante dado que no hay instrucciones de escritura que se ejecuten.

Stress testing

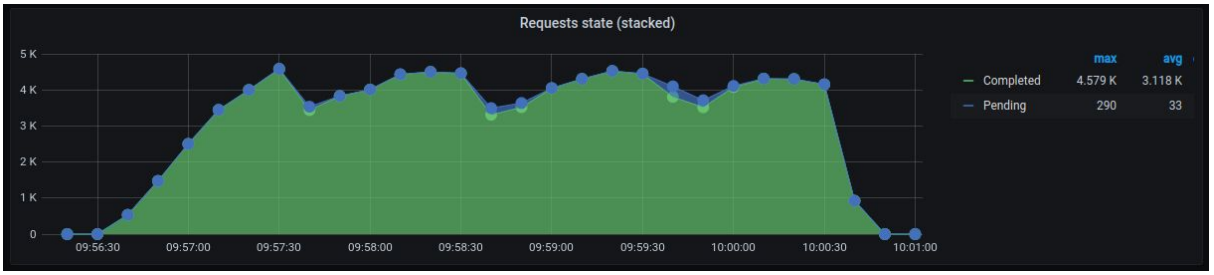
Escenarios creados



Se muestra el mismo gráfico que fue expuesto anteriormente solo que la cantidad de escenarios creados varía ya que la cantidad de request por segundo aumenta considerablemente al ser una prueba de estrés.

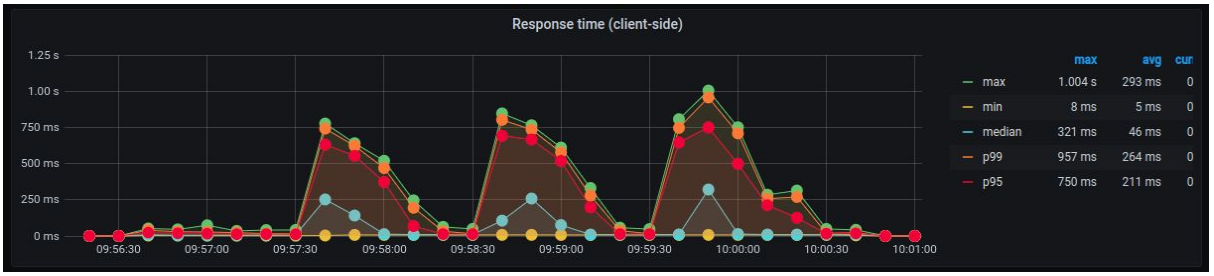
Este gráfico tendrá una forma equivalente en las siguientes pruebas de estrés que se expondrán del endpoint ping por lo que no se incluirá.

Requests completados, pendientes y fallidos



A diferencia del gráfico anterior, se observa que 290 requests quedaron como request pendientes ya que evidentemente el servidor Node no pudo ejecutar la cantidad de request solicitadas y tuvo que encolar el resto. Esto era esperado ya que la cantidad de request enviadas por segundo era 20 veces más que en la prueba de carga anterior. Igualmente, se considera que la cantidad de request encolados en relación a la cantidad de request completados es sustancialmente menor pero esto se debe a que el endpoint es un ping y el procesamiento es mínimo.

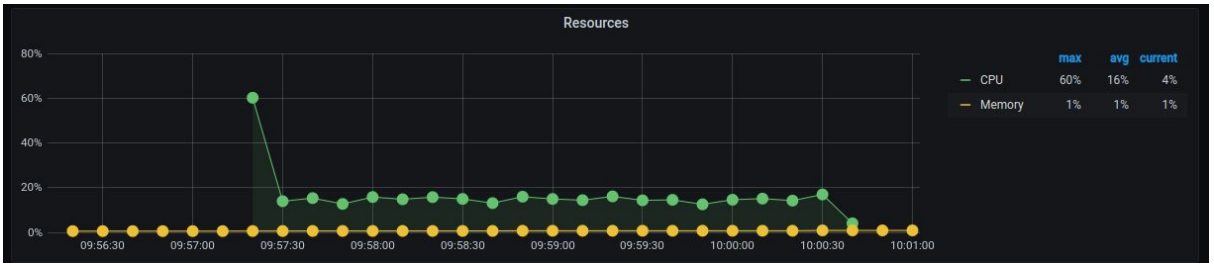
Tiempo de respuesta



En este gráfico el tiempo de respuesta aumenta de 5 ms como mediana a 321 ms, es decir, 60 veces más aproximadamente. Esto es lógico ya que el servidor fue sometido a una prueba de estrés y tuvo que organizar sus recursos para responder las request solicitadas.

Es pertinente aclarar que los picos pronunciados que se observan en el gráfico no implican que hay requests que se contestan en 0 segundos, sino que hay algunas que fallan y no registran tiempo de respuesta.

Recursos

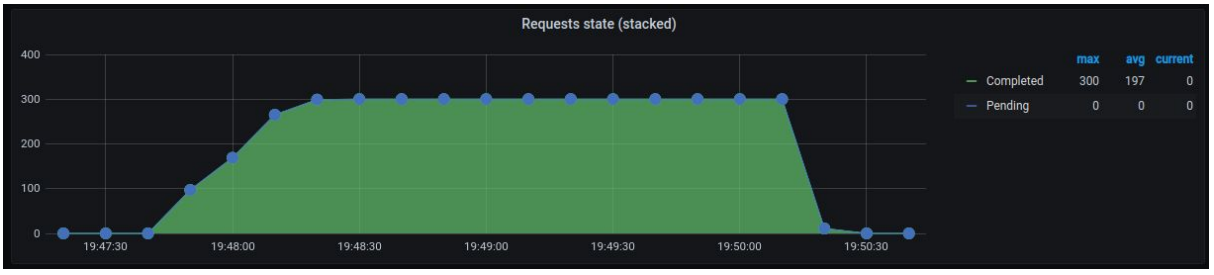


En línea con lo mencionado previamente el uso de la CPU aumenta a un 60%, lo cual era de esperar por el análisis expuesto a raíz de los anteriores gráficos.

Gunicorn

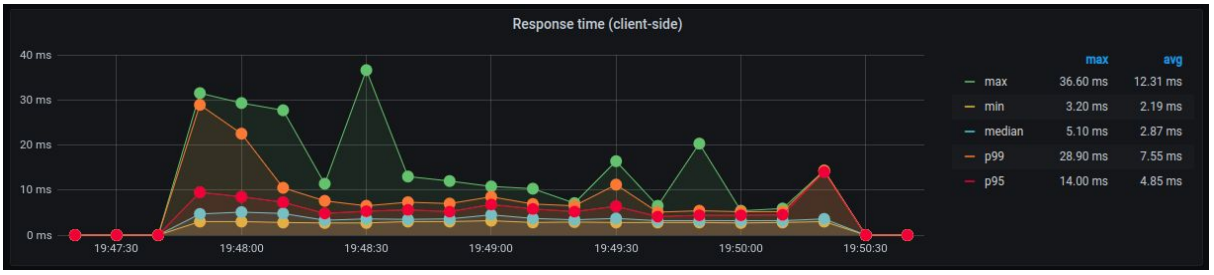
Load testing

Requests completados, pendientes y fallidos



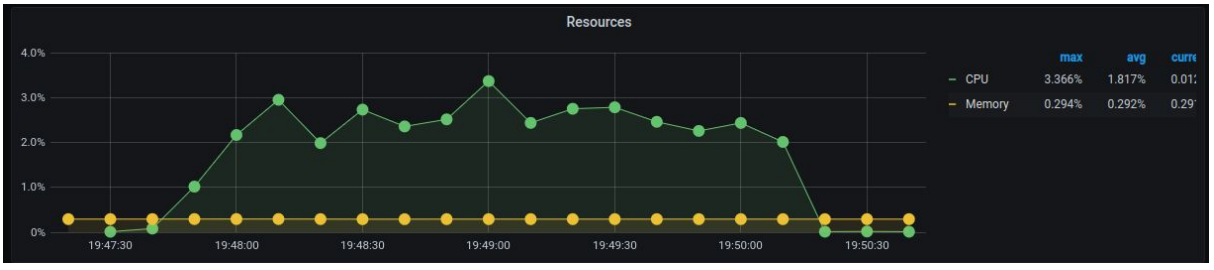
Acá se puede ver que el comportamiento es igual al observado con el server de Node, el bajo procesamiento del endpoint ping no permite ver las diferencias en el sincronismo/asincronismo entre los servers. Como era de esperarse no hay requests pendientes para ningún instante de la prueba.

Tiempo de respuesta



En los tiempos de respuesta se vuelve a ver una situación similar a la de Node, es decir, los tiempos se mantienen bajos para todo tiempo. Al comparar la mediana, el server Python tiene un tiempo de respuesta un poco mayor pero se considera que podrían asumirse muy similares. Así mismo, el máximo de tiempo de respuesta de Python (36.6 ms) es bastante menor que el Node (60 ms). Este gráfico tiene más picos que el de Node pero al tener valores similares en los distintos parámetros se considera que no es pertinente para el análisis de este gráfico.

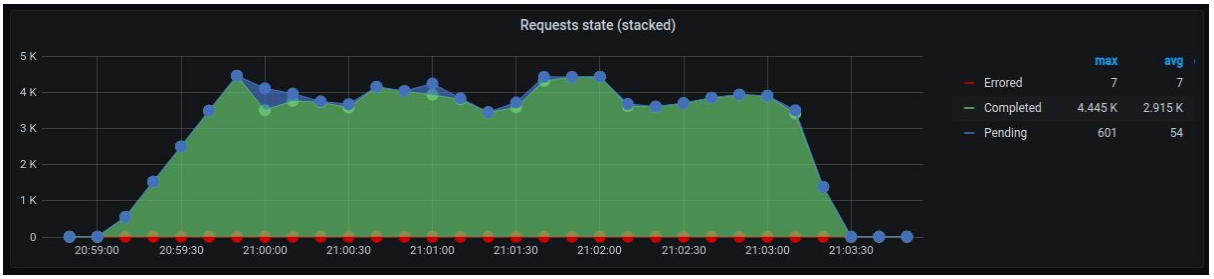
Recursos



Con respecto al consumo de recursos se ve que también se mantiene bajo con apenas una subida con respecto a Node: avg(Node) = 1,6 vs avg(Gunicorn) = 1,8.

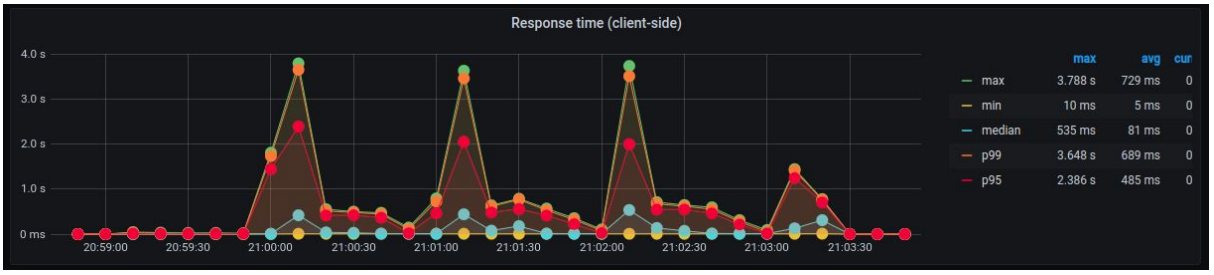
Stress testing

Requests completados, pendientes y fallidos



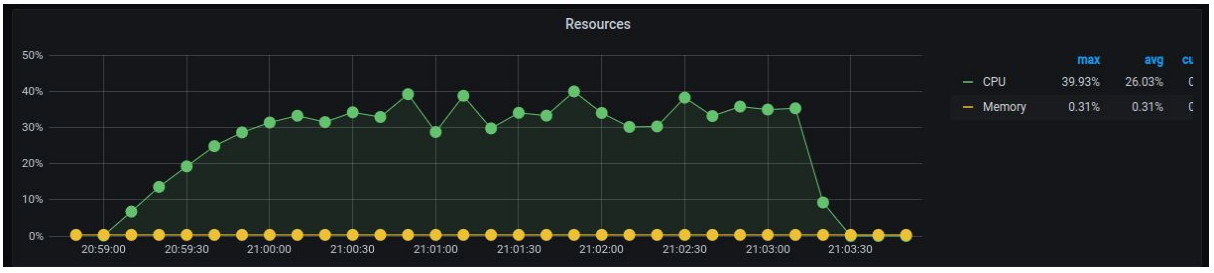
Se observa que algunos request fallaron y 601 quedaron como request pendientes ya que evidentemente el servidor Gunicorn no pudo ejecutar la cantidad de request solicitadas y tuvo que encolar el resto. Esto era esperado ya que la cantidad de request enviadas por segundo era 20 veces más que en la prueba de carga anterior. Igualmente, se considera que la cantidad de request encolados en relación a la cantidad de request completados es sustancialmente menor pero esto se debe a que el endpoint es un ping y el procesamiento es mínimo. Un detalle interesante a destacar es que la cantidad de request encolados es mayor que en el servidor Node.

Tiempo de respuesta



En este gráfico el tiempo de respuesta aumenta de 5 ms como mediana a 535 ms, es decir, 100 veces más aproximadamente. Esto es lógico ya que el servidor fue sometido a una prueba de estrés y tuvo que organizar sus recursos para responder las request solicitadas. En comparación con el servidor en Node, el tiempo de respuesta aumenta a casi el doble lo cual es un aspecto interesante en el análisis.

Recursos

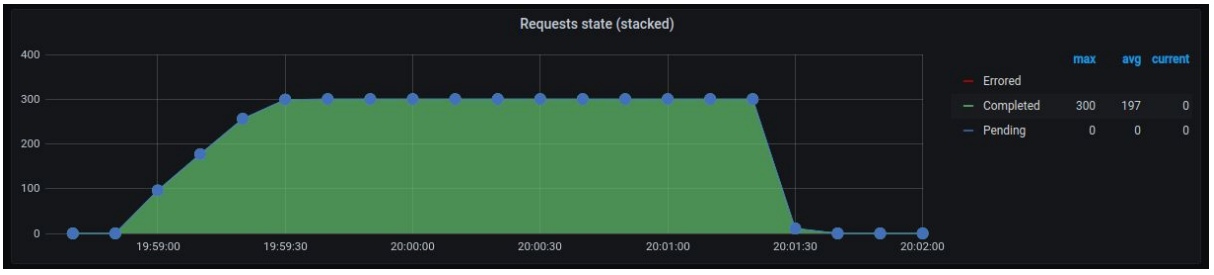


En línea con lo mencionado previamente el uso de la CPU aumenta a un 40%, lo cual era de esperar por lo expuesto en los anteriores gráficos.

Node replicado

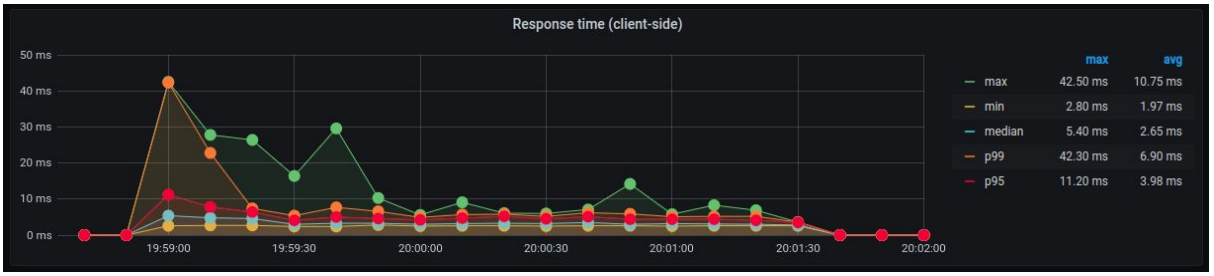
Load testing

Requests completados, pendientes y fallidos



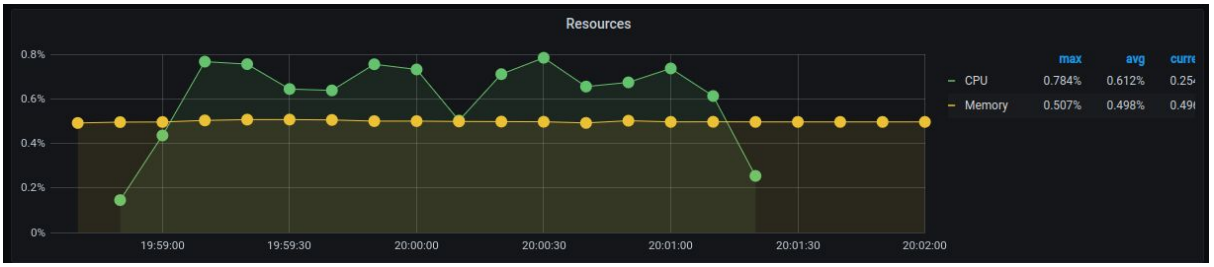
Al igual que en los casos anteriores se puede ver que no hay problemas en responder todas las requests que se van mandando instante a instante, por eso en ningún momento hay requests pendientes, comportamiento que era esperado dada la naturaleza simple del endpoint.

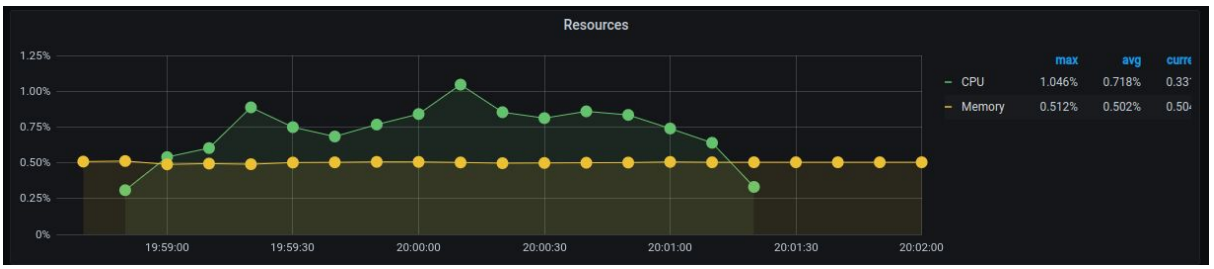
Tiempo de respuesta

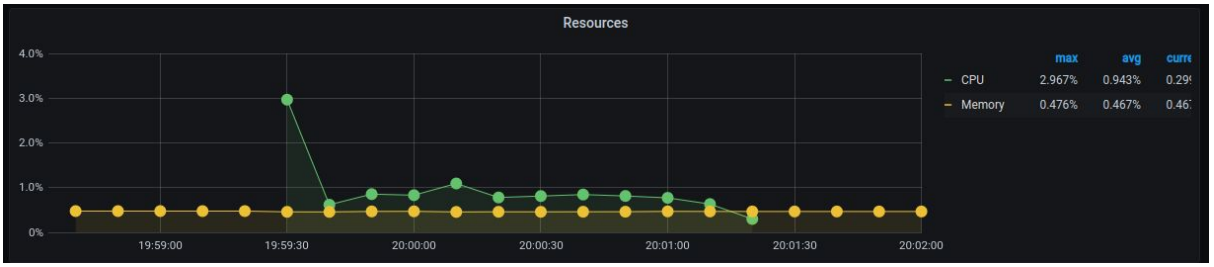


Para el tiempo de respuesta se ve que sucede lo mismo que previamente con una única instancia de Node, un pico inicial y después se mantiene bajo para toda la prueba.

Recursos



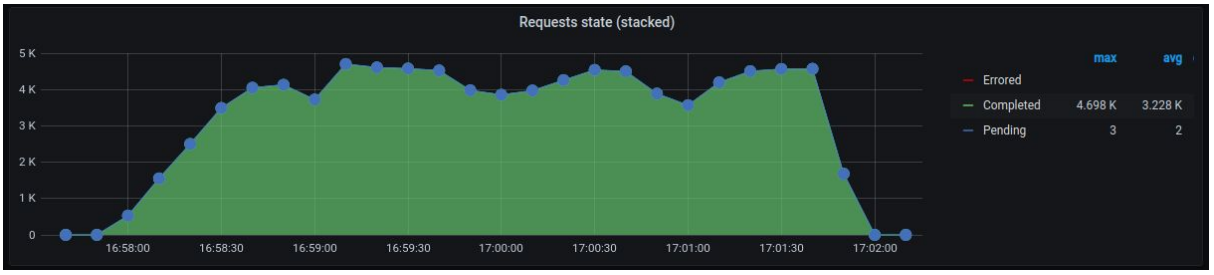




Para analizar el uso de recursos en este caso se requerían 3 gráficos, que muestran el uso de recursos en cada una de las instancias del servidor de Node (cada uno de los containers replicados). Se puede ver que en ninguno de los contenedores se supera el uso de CPU que se vio para el caso de una única instancia de un server en Node. Esto sucede porque al tener 3 instancias el load balancer (*nginx*) lo que hace es distribuir la carga entre los 3 servidores, provocando que cada una de las instancias reciba menos requests de lo que recibe cuando hay una sola.

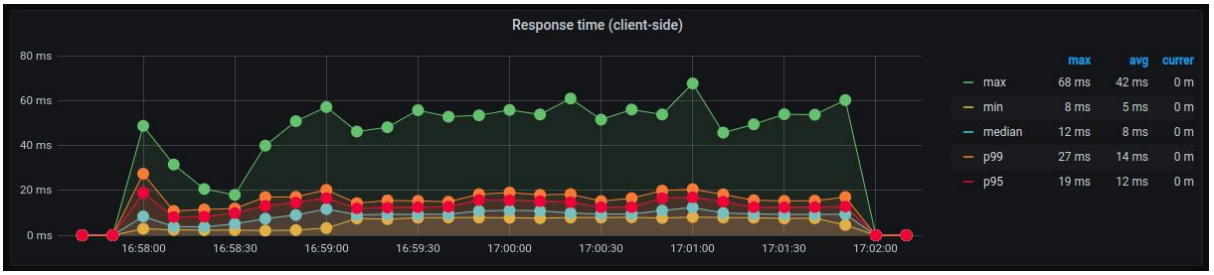
Stress testing

Requests completados, pendientes y fallidos



Es interesante comparar este gráfico con el del endpoint Ping del servidor Node. En ese caso quedaron 290 request pendientes y aquí solo 3. Esto era esperado por el uso del load balancer para distribuir los request para evitar que se encolen.

Tiempo de respuesta



Nuevamente, la mediana de tiempo de respuesta es significativamente menor que en el caso de la prueba de estrés de Node (12 vs 321 ms). Esto es lógico porque al distribuir la carga entre los 3 servidores, son capaces de responder en un tiempo menor. Es interesante remarcar que igualmente el tiempo de respuesta aumenta a casi el doble con respecto a la prueba de carga.

Recursos

Este gráfico resulta particularmente interesante porque el uso de la CPU para la prueba de estrés es de un 20% para el caso de mayor valor, lo cual es significativamente menor al 60% para el caso de la prueba de estrés de Node. Esto demuestra que el load balancer está haciendo un buen trabajo y el uso de la CPU está distribuido en las 3 instancias. Nuevamente el uso de la CPU es mayor al caso de la prueba de carga.

Timeout

El objetivo del endpoint de timeout es simular una llamada a otro servicio como un request HTTP o una llamada a DB, lo cual implica que es un endpoint que va a tardar un cierto tiempo pero sin tener mucho procesamiento.

En este caso se podrán ver las diferencias entre el uso de Gunicorn (Python) y Node, a pesar de que la implementación del endpoint sea igual en ambos casos (un timeout de cierto tiempo y luego se responde). Ambos fueron probados con distintas cargas y se obtuvieron resultados (tiempos de respuesta) diferentes.

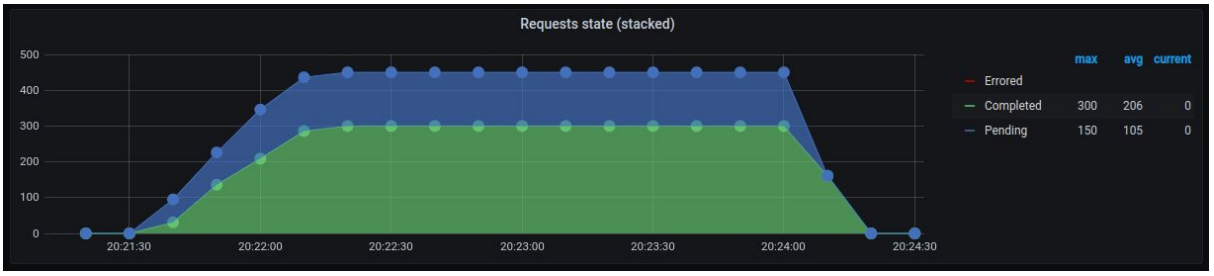
El tiempo de timeout que fue usado para simular el endpoint fue de 5000 ms (5 segundos).

Las diferencias en las respuesta se deben principalmente al procesamiento sincrónico o asincrónico que tienen las distintas tecnologías. Node usa procesamiento asincrónico, es decir, que a pesar de tener un solo hilo de ejecución no es bloqueante, entonces cuando llega a una operación de espera no frena sino que sigue con otras cosas, como puede ser responder otros requests. En cambio Python es sincrónico, por lo tanto si se bloquea en la espera a pesar de que no esté procesando nada en el momento. Lo que permite Gunicorn es lanzar más de un thread con distintos workers del mismo server de Python para poder distribuir la carga, en este caso como fue corrido siempre con un solo worker (un solo hilo de ejecución) se podrán ver las diferencias entre los distintos tipos de procesamiento.

Node

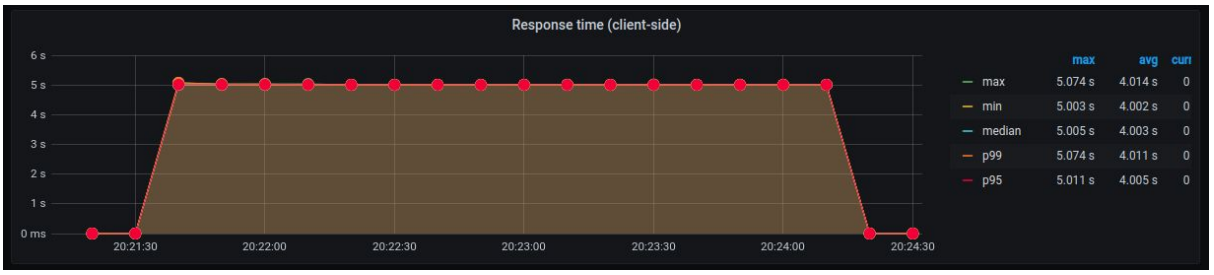
Load testing

Requests completados, pendientes y fallidos



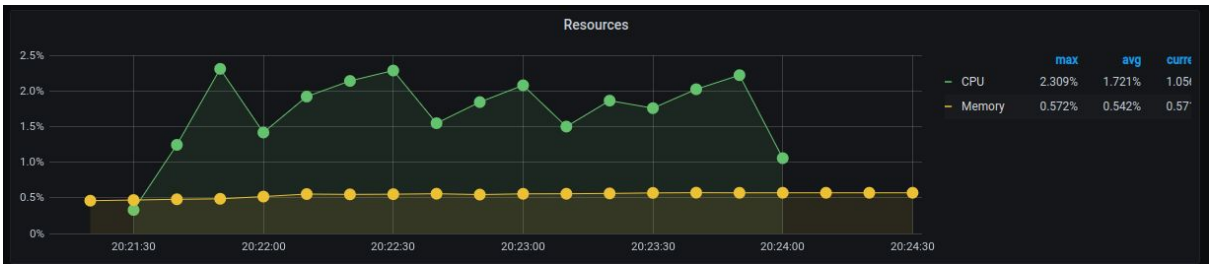
En este gráfico se puede observar cómo a pesar de que se van acumulando requests pendientes finalmente se logran contestar todas y ninguna llega a dar error, esto es porque el tiempo de timeout obliga a que tenga algunas pendientes por 5 segundos y en ese tiempo puede seguir con otros requests.

Tiempo de respuesta



Como era de esperarse el tiempo de respuesta es siempre de 5 segundos, a pesar de tener requests pendientes no se refleja en el tiempo de respuesta que recibe el usuario porque el tiempo de espera de timeout no imposibilita seguir recibiendo o contestando requests.

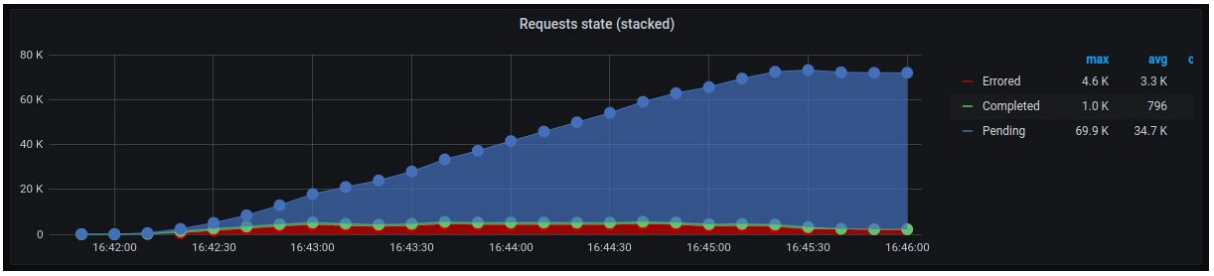
Recursos



Se puede ver que el uso de recursos es casi igual al observado en el caso de endpoint ping, esto se debe a que no hay procesamiento en este endpoint tampoco.

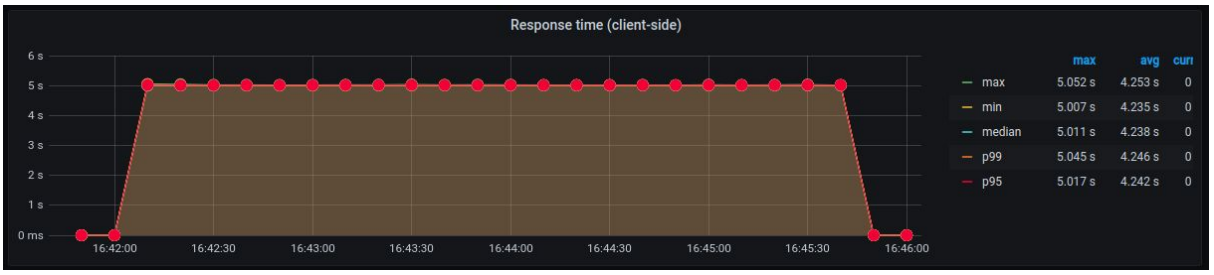
Stress testing

Requests completados, pendientes y fallidos



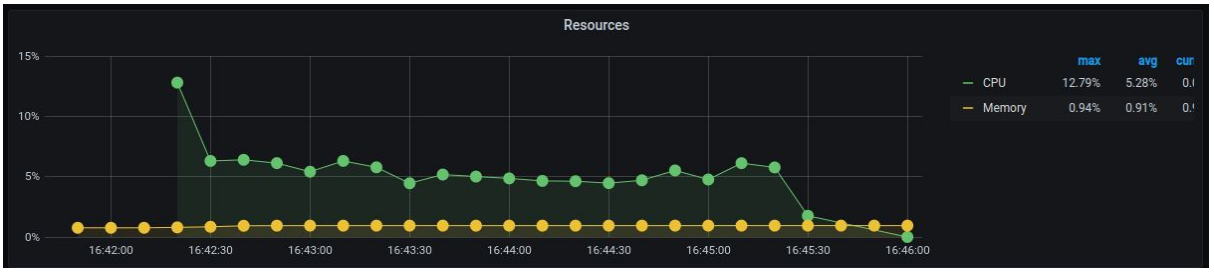
Un aspecto a destacar de este gráfico es que quedan encoladas muchas más request que en el caso de la prueba de carga, con un máximo de 70k en este gráfico y de 100 en el previo. Además, algunas request llegan a fallar que es algo que antes no sucedía. Este aumento sustancial puede deberse a que el timeout es un endpoint que de por sí plantea un tiempo donde la request debe ser encolada, entonces pasa automáticamente al estar como pendiente y eso se potencia con la gran cantidad de requests que está recibiendo el servidor.

Tiempo de respuesta



Este gráfico parece exponer algo contradictorio a lo que se muestra en los anteriores ya que se esperaría que en una prueba de estrés el tiempo de respuesta aumenta. Sin embargo, una explicación para esto podría ser que las request que completa, lo hace en 5 segundos (tiempo del timeout) y las que no, directamente no las responde (ya sea porque quedan pendientes o porque fallaron). Esto está íntimamente relacionado con el asincronismo de Node ya que apenas llega una request, la encola y cuando pasan los 5 segundos contesta. Las que quedan fuera de este esquema son las que fallaron o quedan pendientes.

Recursos

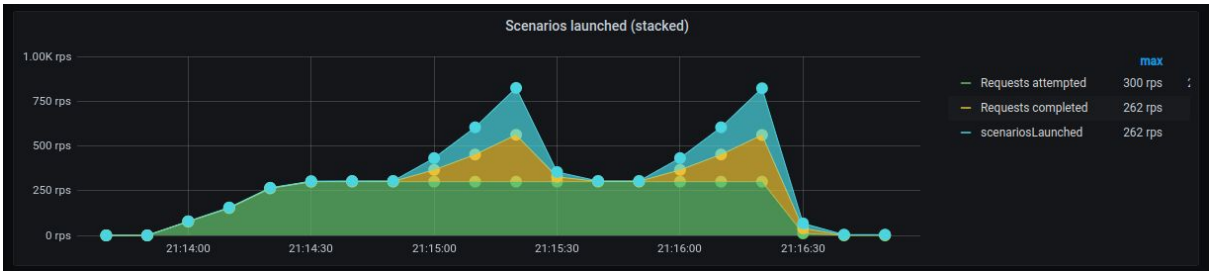


Se observa que el consumo de la CPU aumenta con respecto a la prueba de carga, lo cual es completamente lógico con el análisis planteado.

Gunicorn

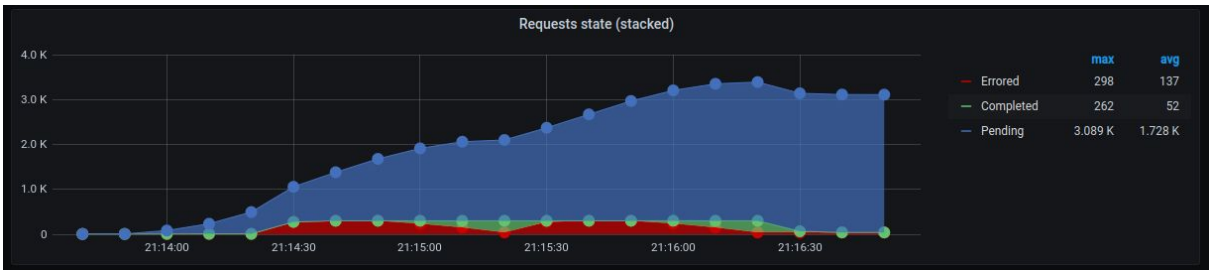
Load testing

Escenarios creados, request completados y request lanzados



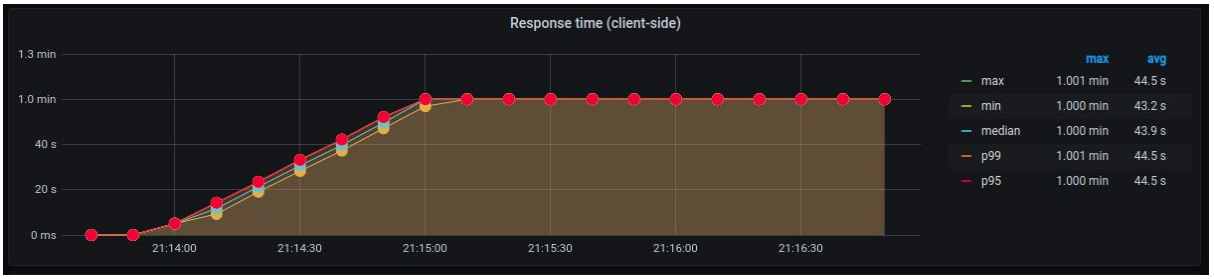
De este gráfico se puede afirmar que de todos los escenarios creados (es decir, request lanzadas) no lograron completarse todas y esto se debe a la naturaleza del endpoint. Este análisis será detallado con mayor profundidad en los gráficos que siguen.

Requests completados, pendientes y fallidos



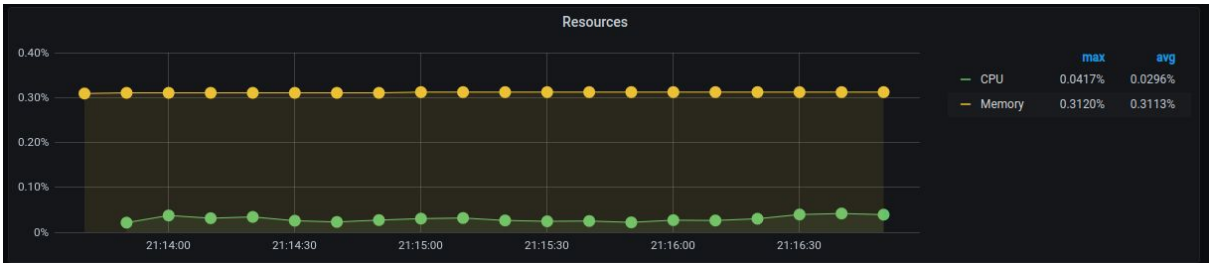
Acá se puede ver claramente la diferencia con el servidor en Node. A medida que van llegando más request el servidor en Python se queda bloqueado esperando los timeouts y no logra contestar todas las requests. Así es cómo se acumulan muchas request pendientes y por ende algunas terminan en error.

Tiempo de respuesta



En el tiempo de respuesta se ve un incremento notable con respecto a lo observado anteriormente en Node. Antes se mantenía estable en los 5 segundos de timeout (estipulado por la implementación propia del endpoint) en cambio con Python se ve que va subiendo hasta mantenerse en 1 minuto estable. Esto se debe al tiempo que se agrega para poder procesar todas las requests de manera sincrónica.

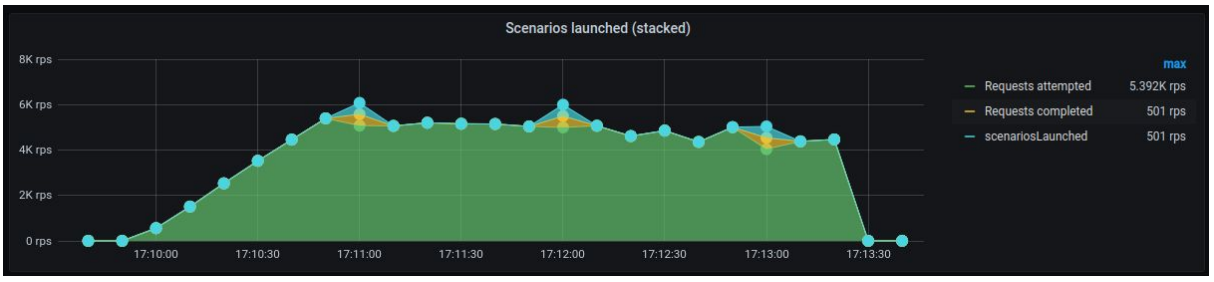
Recursos



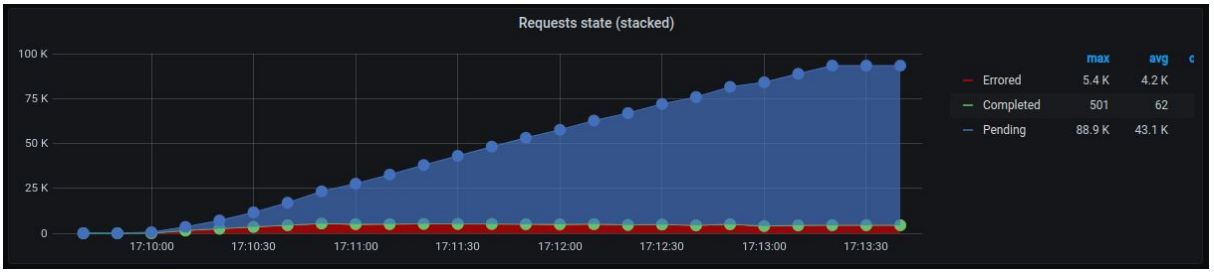
Por último, en lo que respecta al uso de recursos se puede observar que es casi nulo, lo cual era esperable por la naturaleza del endpoint. El hecho de que se bloquee el hilo de ejecución no implica que consuma más recursos, sino que se queda esperando hasta que termine la operación de timeout.

Stress testing

Escenarios creados, request completados y request lanzados

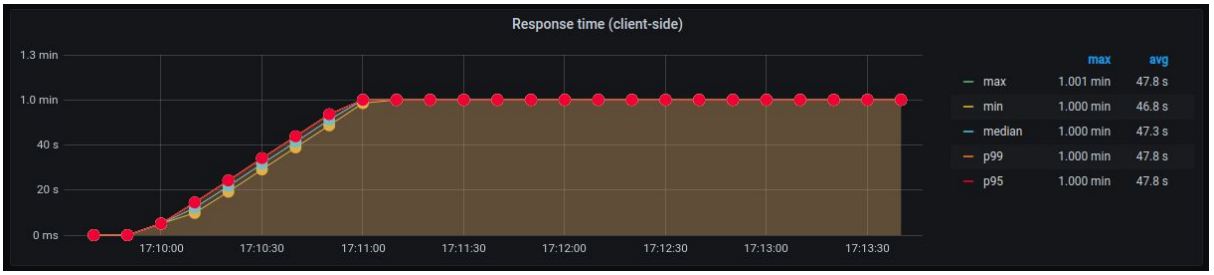


Requests completados, pendientes y fallidos



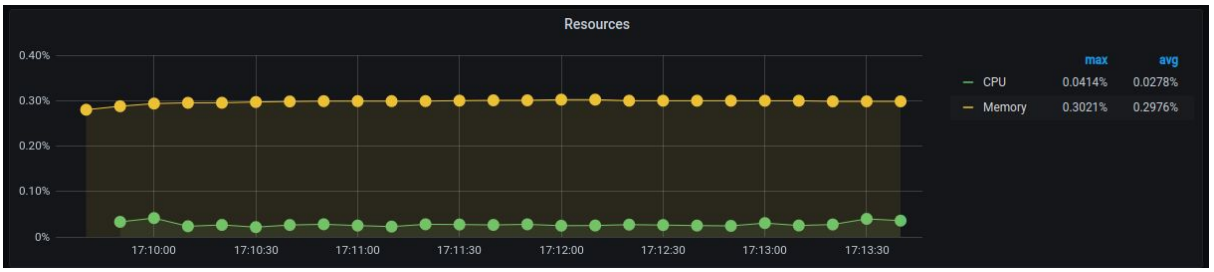
Se observa que la cantidad de request pendientes y con error es mayor al caso de la prueba de carga de Gunicorn, lo cual era de esperar. Así mismo, es interesante remarcar que si bien tiene más requests pendientes y con error que el servidor Node para la misma prueba de estrés, la diferencia no es tan grande. Esto llevaría a pensar que ante una prueba de estrés la diferencia de sincronismo entre ambos servidores no se hace notar de gran manera.

Tiempo de respuesta



Igual que en el caso del servidor Node el tiempo de respuesta se mantiene muy similar al caso de la prueba de carga. Esto puede justificarse con un enfoque similar al de Node, las request que contesta las hace en ese umbral de 1 minuto y el resto no las contesta.

Recursos

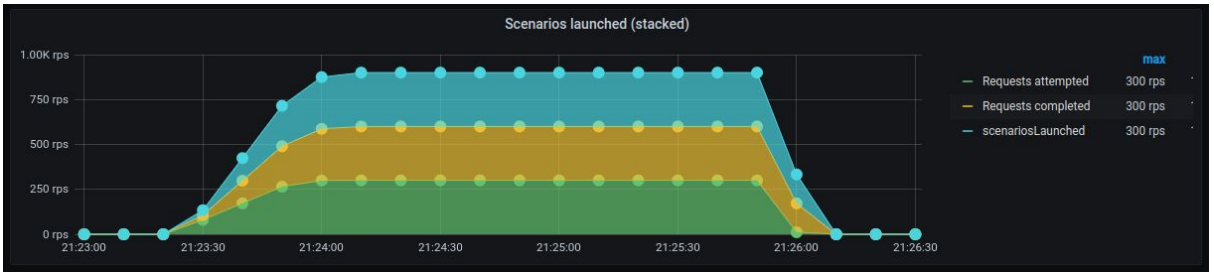


El consumo de CPU se mantiene muy similar al de la prueba de carga. Esto es porque a pesar de que sean muchas requests, las mismas son bloqueantes así que no generan un consumo extra de CPU.

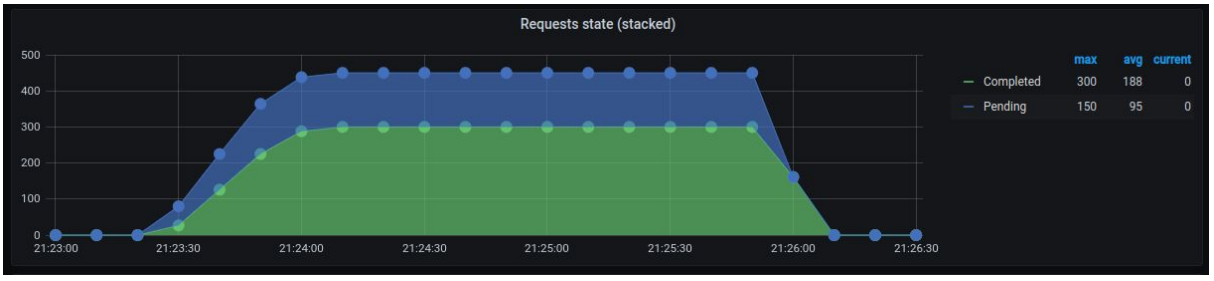
Node replicado

Load testing

Escenarios creados, request completados y request lanzados

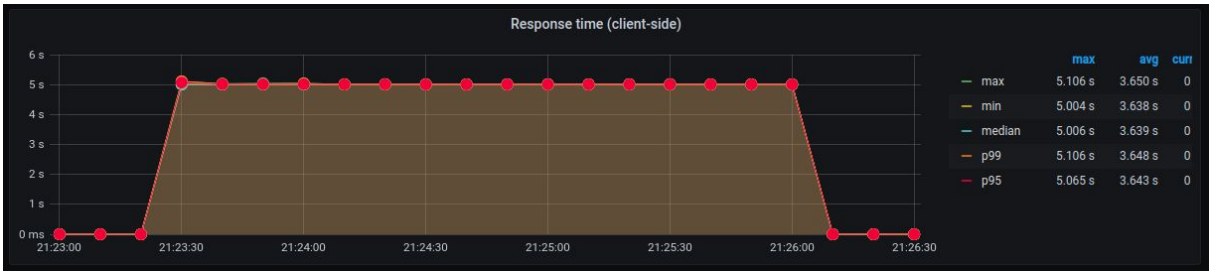


Requests completados, pendientes y fallidos



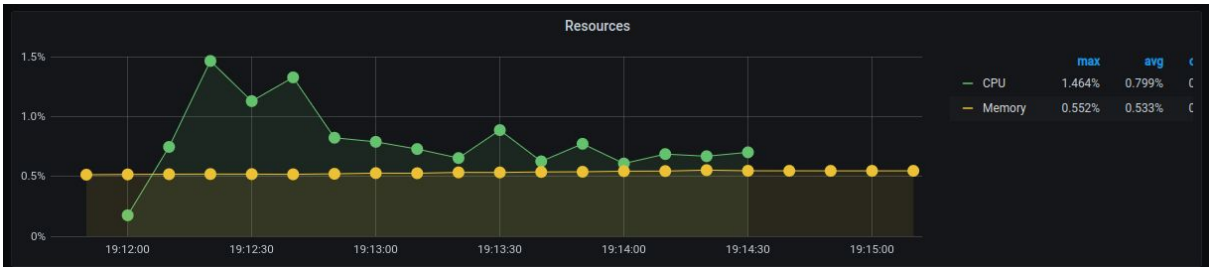
En este gráfico se puede ver que el comportamiento es igual al caso anterior con una solo instancia de Node.

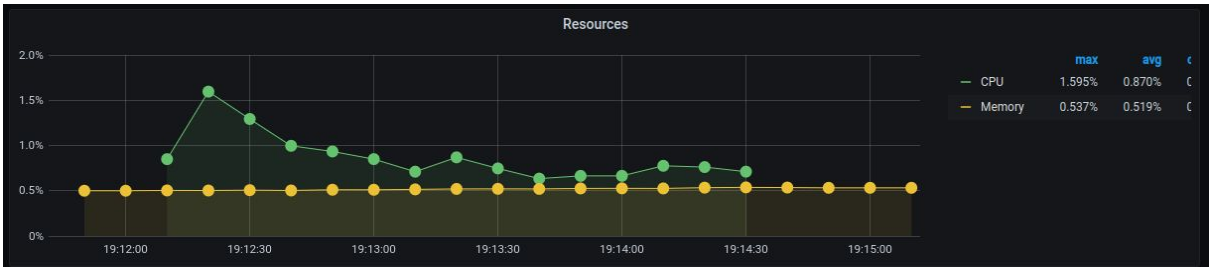
Tiempo de respuesta

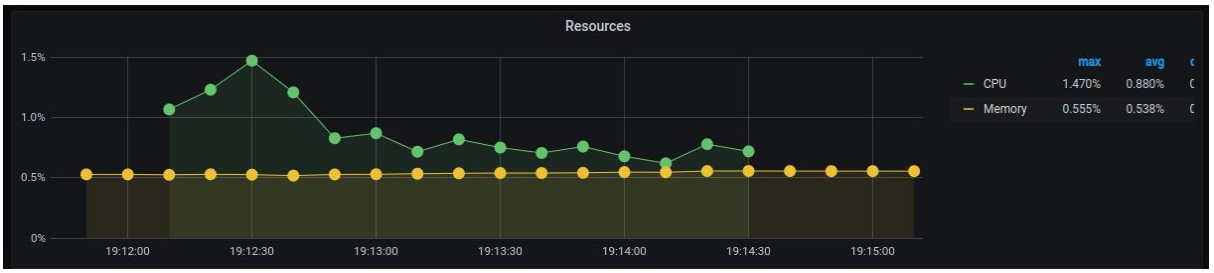


Al igual que cuando había una sola instancia de Node acá también se puede observar que el tiempo de respuesta se mantiene constante en el tiempo de timeout estipulado en la implementación del endpoint.

Recursos



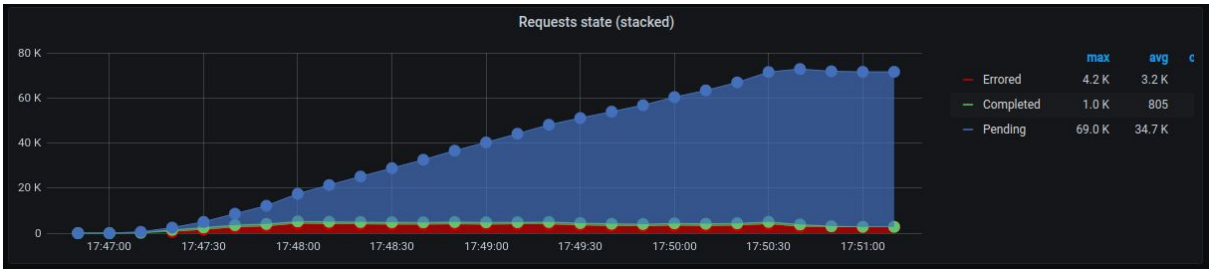




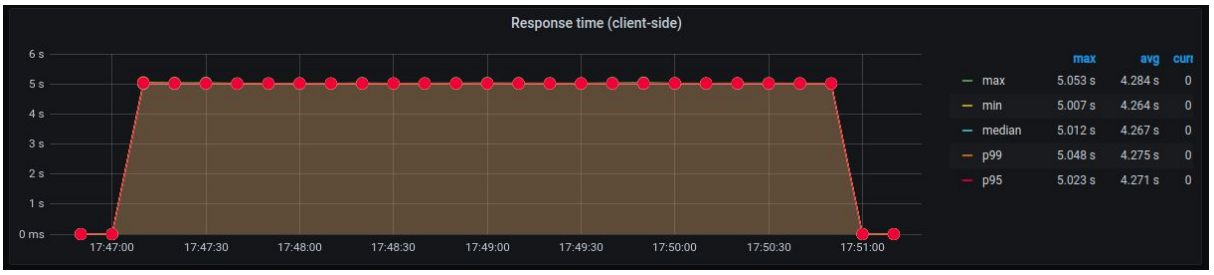
Acá igual que se vio en el Ping ninguna de las réplicas supera el uso de CPU a si se hubiese ejecutado con una sola instancia de Node, lo cual significa que la carga se está repartiendo entre todas las instancias. El hecho de que sea bajo en todos es porque el endpoint de timeout no tiene casi procesamiento.

Stress testing

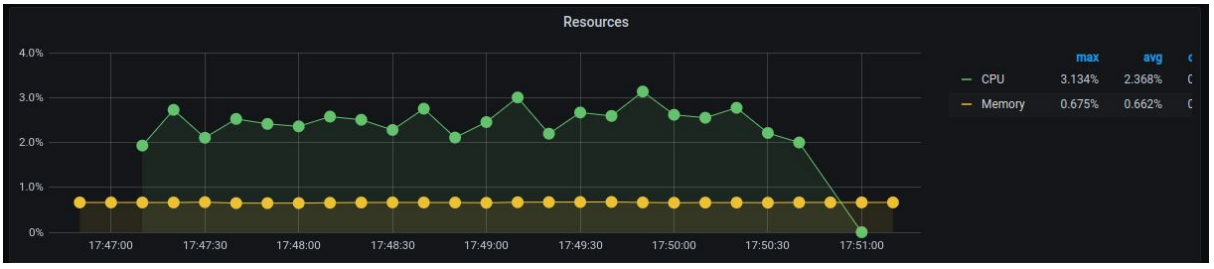
Requests completados, pendientes y fallidos

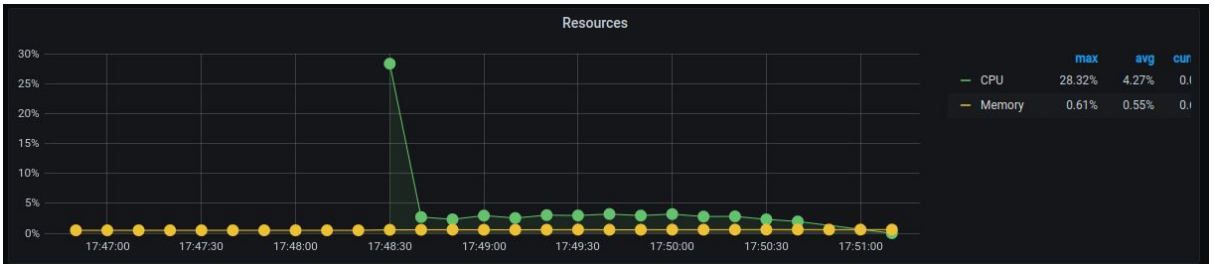


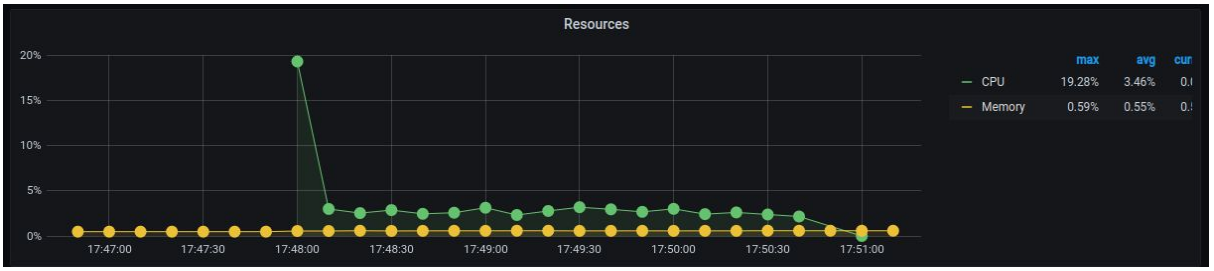
Tiempo de respuesta



Recursos







De estos gráficos se puede desprender que el comportamiento del endpoint timeout no varía significativamente en la prueba de estrés para un único servidor Node y un servidor Node replicado. El tiempo de respuesta es muy similar así como la cantidad de requests pendientes, completados y con error. Esto puede deberse a que cuando se somete al servidor a una cantidad de request como las simuladas, no aporta una ventaja significativa tener el servidor replicado.

Intense

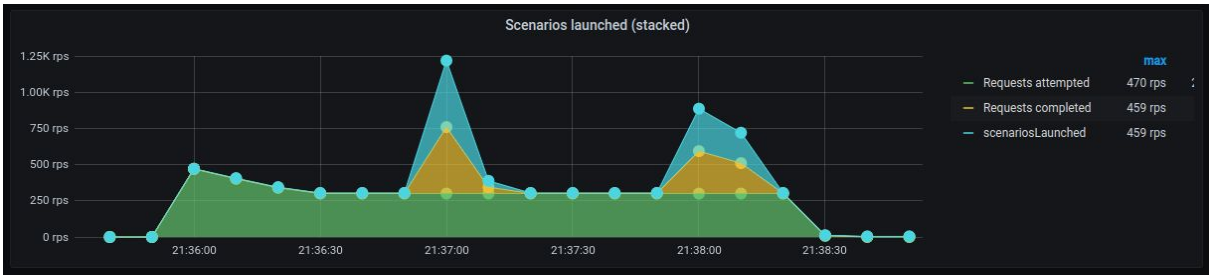
El objetivo de este endpoint es implementar un loop de cierto tiempo para simular cálculos pesados sobre los datos. Se espera que aumente el tiempo de respuesta junto con el uso de la CPU y en consecuencia que la cantidad de requests completados sea menor.

Este endpoint tiene la particularidad de que al agregar procesamiento en los cálculos no puede notarse una ventaja de Node por sobre Python en los tiempos de respuesta ya que ambos procesos son bloqueantes.

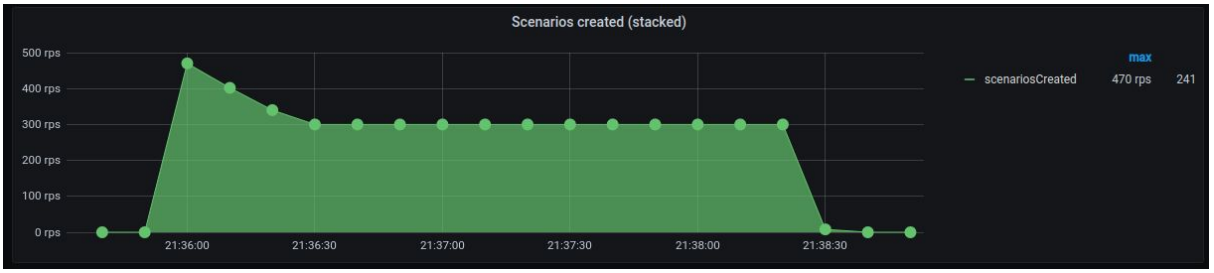
Node

Load testing

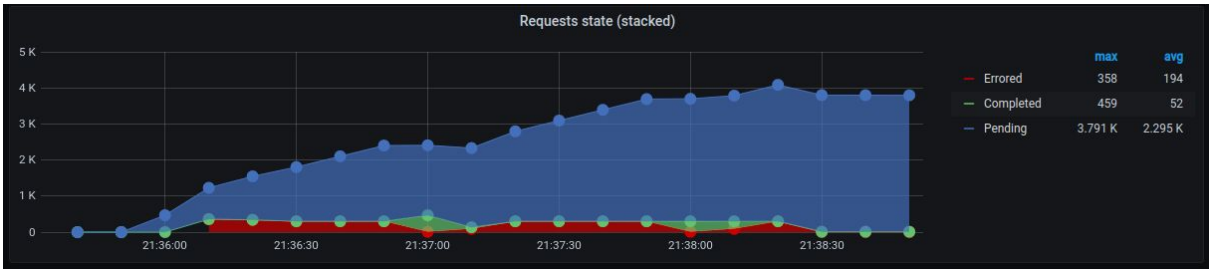
Escenarios creados, request completados y request lanzados



Escenarios creados



Requests completados, pendientes y fallidos

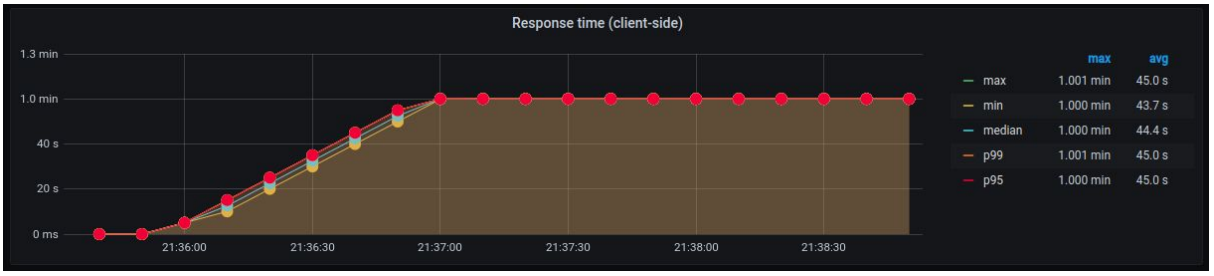


En este gráfico se puede ver que el asincronismo de Node no le da ninguna ventaja cuando no hay operaciones que pueda encolar y seguir con otras. Cabe recordar que Node tiene un único hilo de ejecución, entonces si la operación requiere de procesamiento no puede

seguir con otra hasta que termine; en este caso, el endpoint intense requiere de la "atención" del procesador en todo momento.

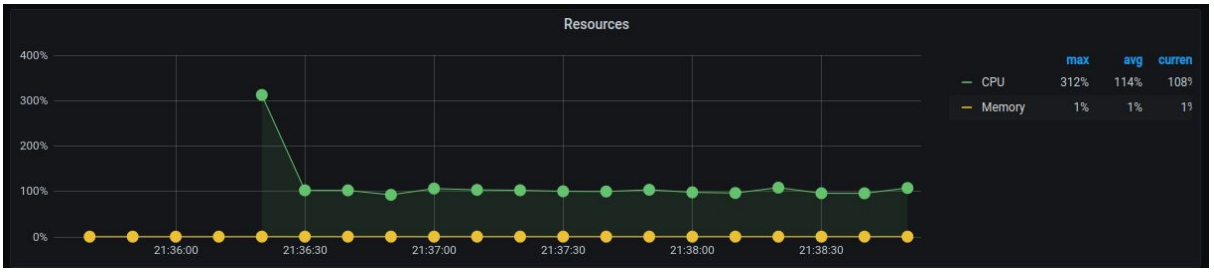
Se puede ver un comportamiento similar a lo observado en el endpoint de timeout con Gunicorn, donde la cantidad de requests pendientes se va aculumando en cada rafaga de requests porque se queda bloqueado procesando las primeras que llegaron, eso provoca que muchas terminan en error por el tiempo de espera.

Tiempo de respuesta



Aca tambien se puede ver una similitud con el server de Python en el endpoint de timeout, donde el tiempo de respuesta es mucho mayor al tiempo real de procesado de cada request y se agrega el tiempo de espera a que se terminen de procesar las request encoladas anteriormente. Se estanca en 1 minuto porque luego de ese tiempo se cree que da por error la request y no sigue esperando.

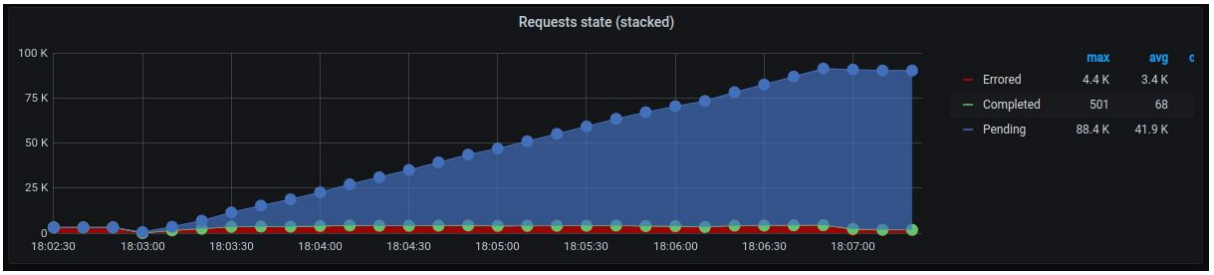
Recursos



A diferencia del endpoint timeout que no consume recursos, el endpoint intense sí consume, y se puede ver claramente en el gráfico que el uso de CPU se mantiene arriba durante toda la prueba ya que siempre está procesando alguna request.

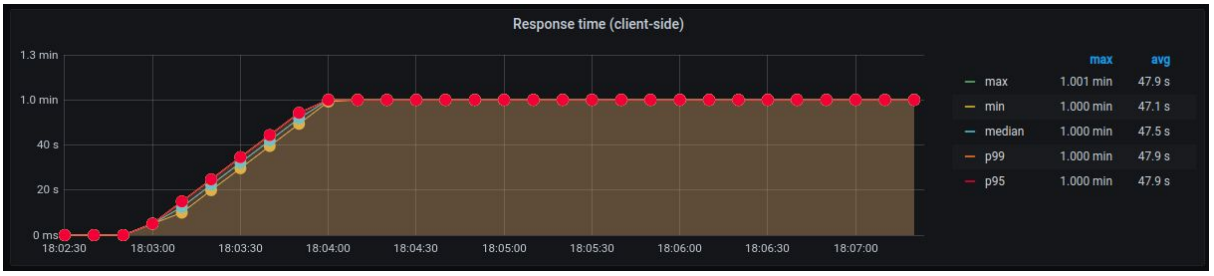
Stress testing

Requests completados, pendientes y fallidos



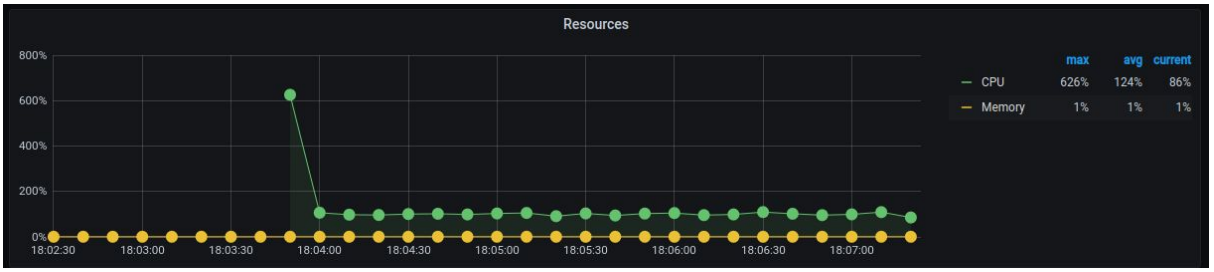
En este gráfico se puede observar que la cantidad de request pendientes y con error aumentó sustancialmente con respecto a la prueba de carga para el mismo servidor. Esto se debe a que la cantidad de request enviados es mucho mayor al tratarse de una prueba de estrés. Sin ir más lejos, se podría afirmar de igual manera que como el endpoint requiere de un procesamiento extra por request y como Node no puede aprovechar su asincronismo, la cantidad de request encoladas aumenta en mayor proporción.

Tiempo de respuesta



El tiempo de respuesta se mantiene similar al de la prueba de carga. Esto puede deberse a lo que fue mencionado anteriormente en relación al tiempo de espera del cliente, que es como máximo de 1 minuto.

Recursos

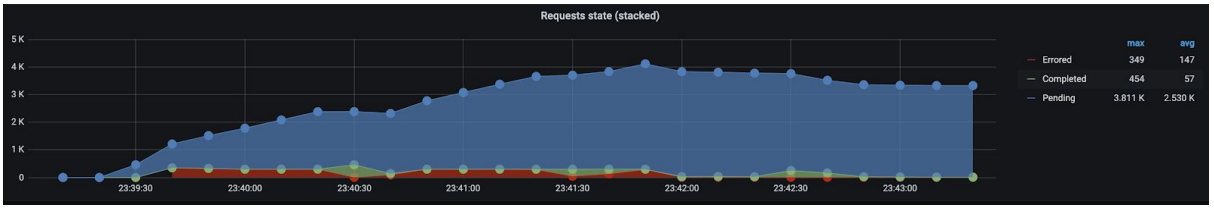


El uso de la CPU aumenta con respecto a la prueba de carga, lo cual es lógico con la línea de pensamiento expuesta en este análisis.

Gunicorn

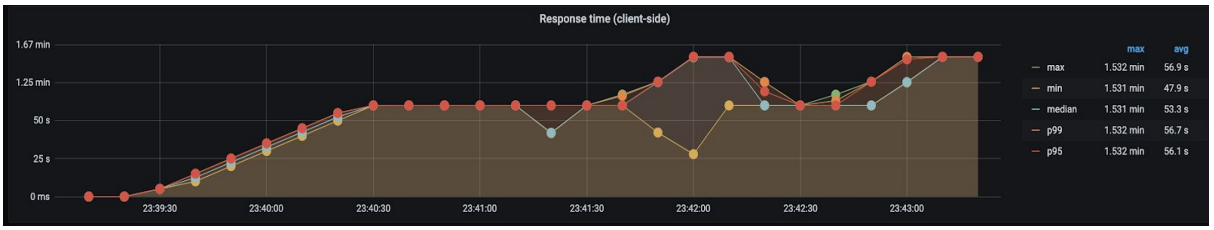
Load testing

Requests completados, pendientes y fallidos



En este caso se puede ver que para Python no es diferencia si hay un timeout o si la operación requiere procesamiento porque en ambos casos se va a quedar bloqueado el hilo de ejecución. Por eso es que el gráfico es igual al que se vio en el endpoint de timeout, donde las requests pendientes se van acumulando en cada rafaga que llega.

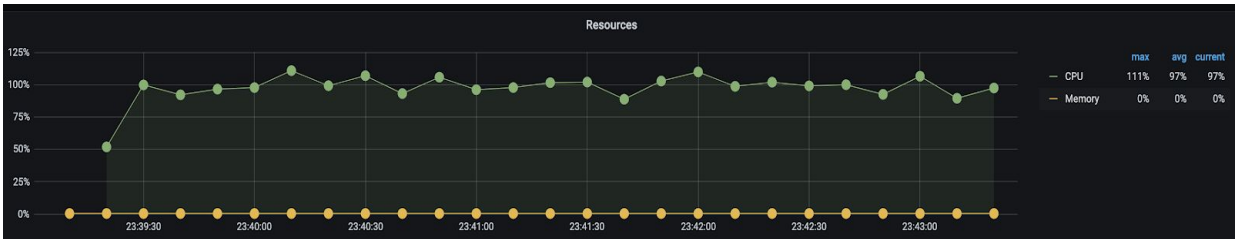
Tiempo de respuesta



Se observa que el tiempo de respuesta tiene como valor 1.5 minutos como máximo en todas las unidades de medición, probablemente porque sea el tiempo máximo de espera de la request para la CPU donde corre el server.

El gráfico tiene valores similares para el caso del endpoint timeout en Python aunque curiosamente con un tiempo de respuesta mayor. Esto puede deberse a dos motivos: se está corriendo con otra CPU o el procesamiento incrementa el tiempo de respuesta.

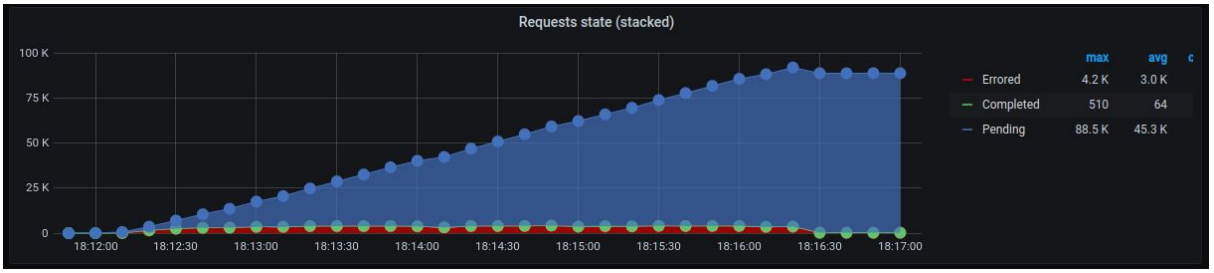
Recursos



En lo único que difiere el comportamiento del endpoint de intense contra el de timeout, usando un server en Python es el consumo de recursos. Se puede ver que en este caso el consumo de CPU se mantiene arriba a lo largo de toda la prueba, porque siempre está procesando alguna request.

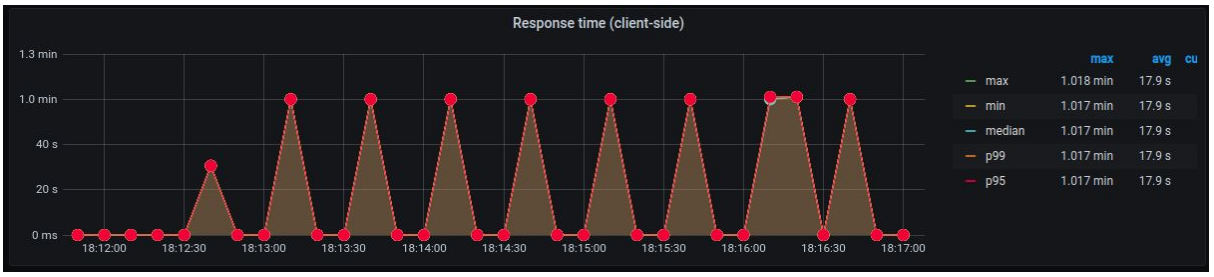
Stress testing

Requests completados, pendientes y fallidos



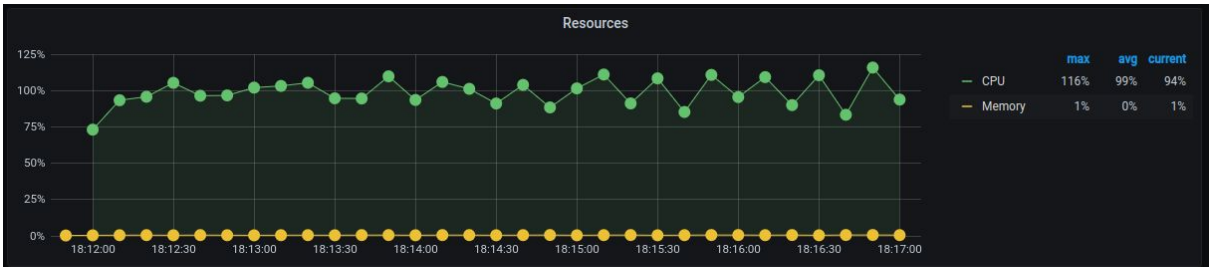
Se observa que la cantidad de request pendientes y con error aumenta con respecto a la prueba de carga. Esto es lógico considerando las razones expuestas anteriormente.

Tiempo de respuesta



Si bien pareciera que el tiempo de respuesta bajó en la prueba de estrés para el caso de Gunicorn, esto no es así ya que fue corrido en otra computadora. Es pertinente destacar en este gráfico que la serie de picos se manifiesta porque algunas request no se pueden contestar y no devuelven ningún tiempo de respuesta, lo que se traduce a distintos ceros en el gráfico.

Recursos

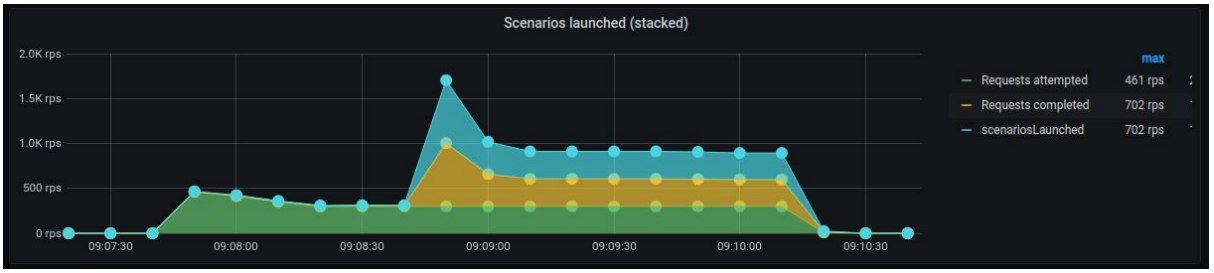


El uso de la CPU se mantiene parecido al de la prueba de carga pero nuevamente no se puede extraer esta conclusión ya que fue corrido con distintas computadoras.

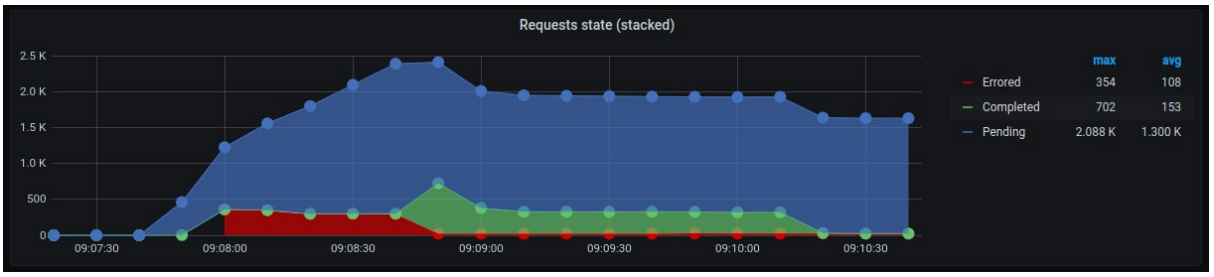
Node replicado

Load testing

Escenarios creados, request completados y request lanzados

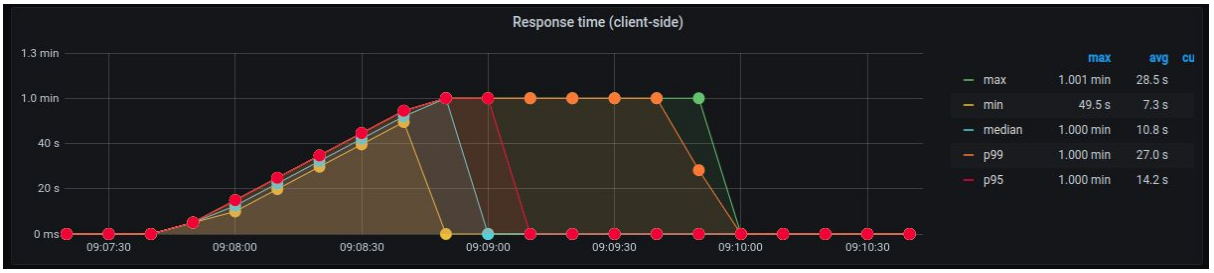


Requests completados, pendientes y fallidos



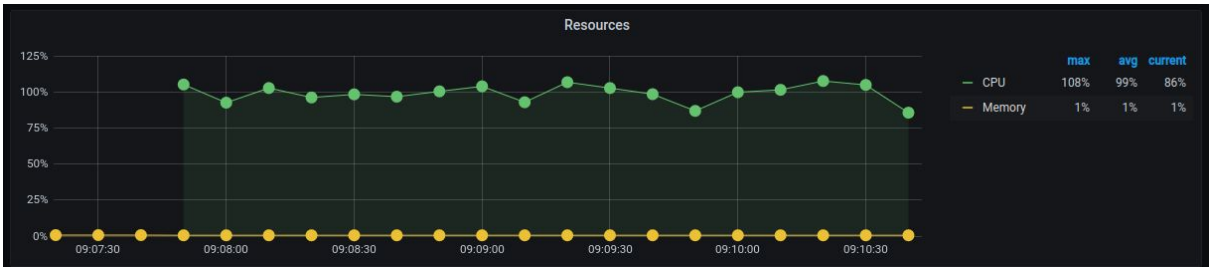
En este caso se puede observar que a pesar de no poder hacer uso del asincronismo de Node, el hecho de tener 3 instancias del servidor logra disminuir la carga en cada uno por ende poder lograr más cantidad de requests exitosas, menos pendientes y menos errores.

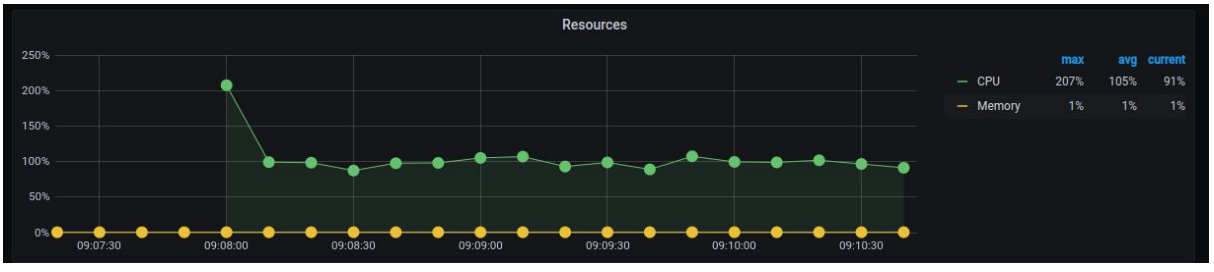
Tiempo de respuesta

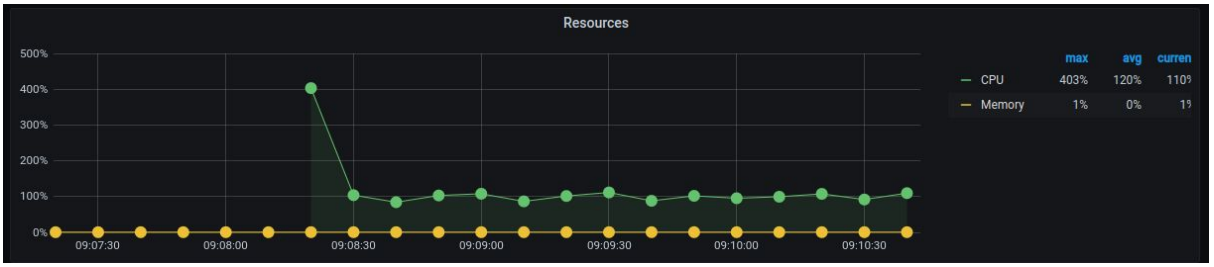


De manera similar al caso de Gunicorn, se observa que la mediana del tiempo de respuesta se mantiene con un máximo de 1 minuto, lo cual podría ser un límite de la CPU al responder el request. Se podría afirmar que Gunicorn fue más performante que Node para este endpoint pero se considera que se debe principalmente a que fueron corridos con diferentes CPUs.

Recursos



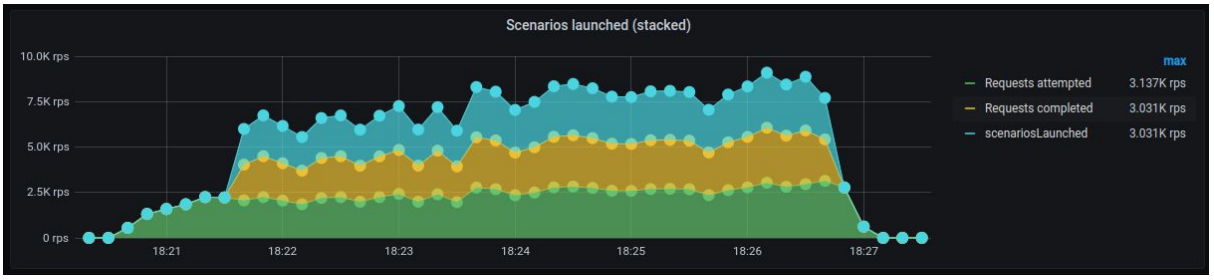




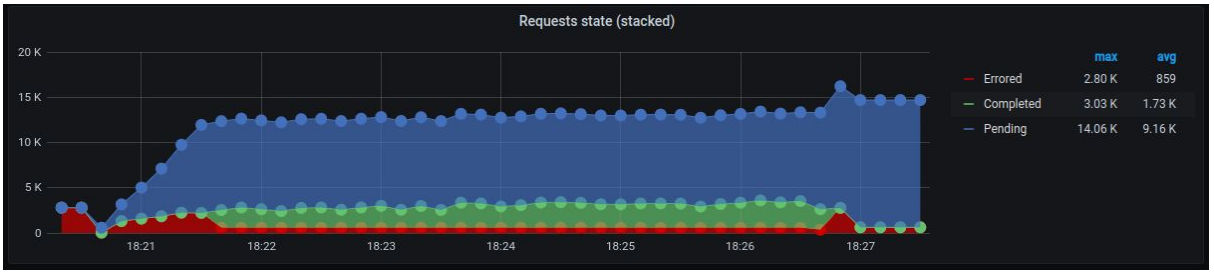
En los 3 gráficos se puede ver que las réplicas de node consumen la CPU que tienen al máximo.

Stress testing

Escenarios creados, request completados y request lanzados

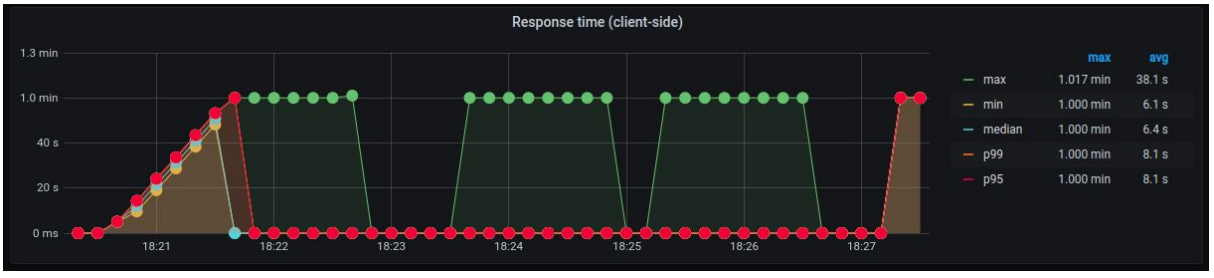


Requests completados, pendientes y fallidos



En este gráfico se puede observar que como era de esperarse la proporción entre cantidad de requests pendientes y completadas aumentó sustancialmente en comparación con la prueba de carga anteriormente analizada.

Tiempo de respuesta



Como fue mencionado previamente, es importante aclarar que en este gráfico se ve que por momento bajan a 0 los parámetros. Esto no significa que el tiempo de respuesta fue 0

segundos sino que el servidor no pudo contestar y dio error (502 / ECONNRESET / ESOCKETTIMEDOUT).

Como era de esperarse la performance para este caso fue baja, es decir, el hecho de tener 3 réplicas de Node no pudieron dar abasto para contestar la gran cantidad de requests que llegaban por segundo.

Recursos

Con respecto al consumo de recursos no hay grandes diferencias con la prueba de carga, es ambas situaciones los 3 containers usaron el máximo de CPU durante toda la prueba.

Sección 2

Objetivo general

Se trabajará en esta sección con las estrategias aprendidas y explicadas en la sección 1. Se busca analizar y caracterizar dos servicios web provistos por la cátedra.

Servicios Web

A continuación, se describe una serie de características de ambos servicios.

- Se interactúa con ambos de la misma manera. Ambos fueron agregados al archivo de Docker Compose para poder ser utilizados.
- Ambos devuelven un “Hello World”
- Uno de los servicios es sincrónico y el otro es asincrónico

Detección de sincronía

En esta sección se explica cuál fue el método de detección. El mismo parte de la utilización del timeout como herramienta para originar una disparidad entre ambos servicios. Teniendo en cuenta el funcionamiento del sincronismo, esto debería ocasionar que el tiempo de respuesta entre request sea mayor en el servicio sincrónico. Esto es por el carácter bloqueante del sincronismo, que hace que una vez que la cantidad de request supere la cantidad de workers, las request que llegan deban esperar a que se complete la request que se está procesando. Por otro lado, en el servicio asincrónico el procesamiento de ciertas partes de la request se irán encolando y contestando continuamente debido a su carácter no bloqueante.

Test Ejecutado

Se realiza un ramp desde 0 a 15 request durante 60 segundos.

Hipótesis y supuestos

- El escenario utilizado para ambos es idéntico.
- El tiempo de respuesta del sincrónico debería aumentar a medida que se aumentan los request por segundo una vez superado el umbral de rps que puede procesar a la vez.
- El tiempo de respuesta del asincrónico debería mantenerse estable, con poca variación.
- Cualquier otro aspecto externo capaz de modificar las mediciones es desestimado

Gráfico del servicio 9091

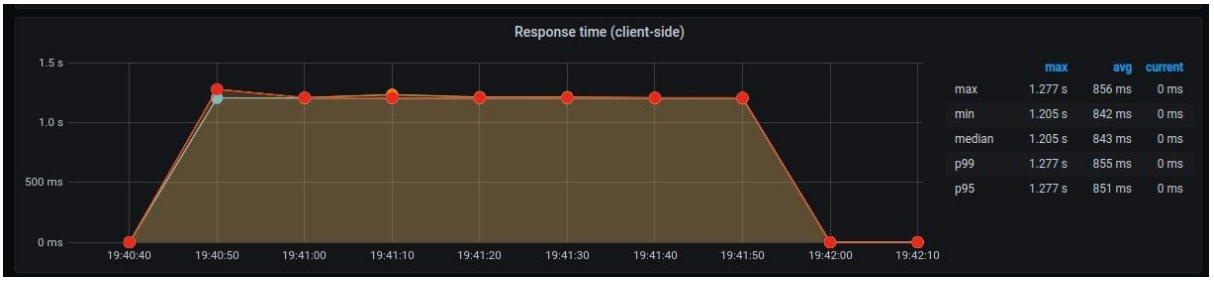
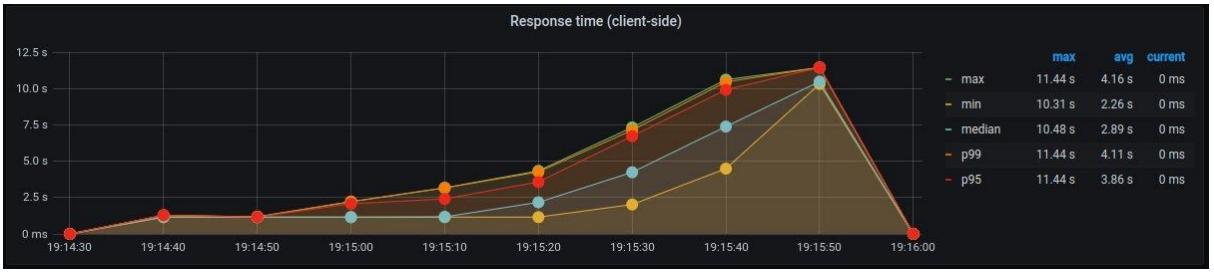


Gráfico del servicio 9090



Resultados

A modo de conclusión se detecta que el servicio sincrónico es el 9090 y el asincrónico es el 9091. En esta ocasión se llegó a la conclusión buscando cuál de los dos servicios mantenía tiempos relativamente constantes y este iba a ser el asincrónico. El análisis más exhaustivo del sincrónico se verá en las siguientes secciones.

Cantidad de workers

Objetivo

Se buscará estimar la cantidad de workers que posee el servicio web que corresponde al modelo sincrónico.

Test Ejecutado

Se realiza un ramp desde 0 a 15 request durante 60 segundos.

Hipótesis y supuestos

- Todos los workers funcionan de forma idéntica.
- Los request llegan de forma uniforme en el tiempo.
- No hay injerencia externa capaz de modificar los resultados obtenidos.
- Una vez llegado el umbral correspondiente, se limitará la cantidad de request respondidos de forma notable.

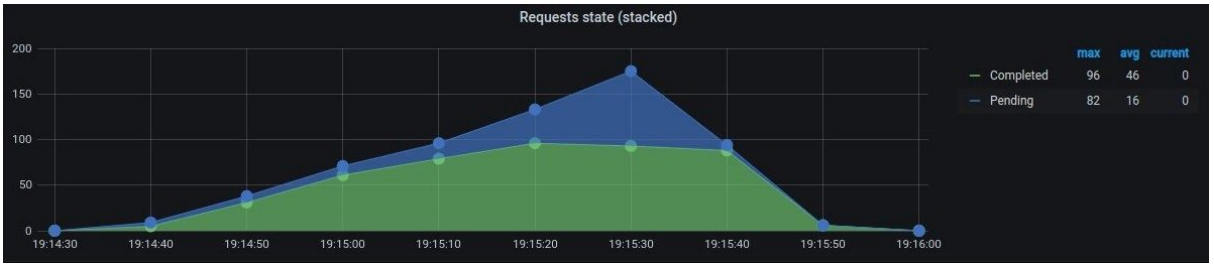
Metodología

Partiendo de las hipótesis previas, se diseñó un escenario tal que se pueda visualizar a simple vista el punto de quiebre de la métrica de la tasa de request respondidos sobre el total de las request. De esta manera se podrá calcular muy simplemente la cantidad de workers que posee el servicio. Por cada request se contará con un worker aproximadamente.

Nota: Es importante destacar que, como toda medición, existe un margen de error. Se considerará que todos los request han sido respondidos cuando se cumpla que tanto el tiempo mínimo como el máximo de respuesta sea similar.

Resultados

Escenario utilizado



El siguiente gráfico muestra el tiempo de respuesta. Se muestran los dos puntos más destacables para este análisis. En el primero donde se contestan 3 request, se ve el momento en el cual todavía se cumple la hipótesis establecida.

Mientras que en este segundo donde se contestan 6 request, ya se sobrepasó el punto de quiebre mencionado.

Análisis de resultados y conclusiones

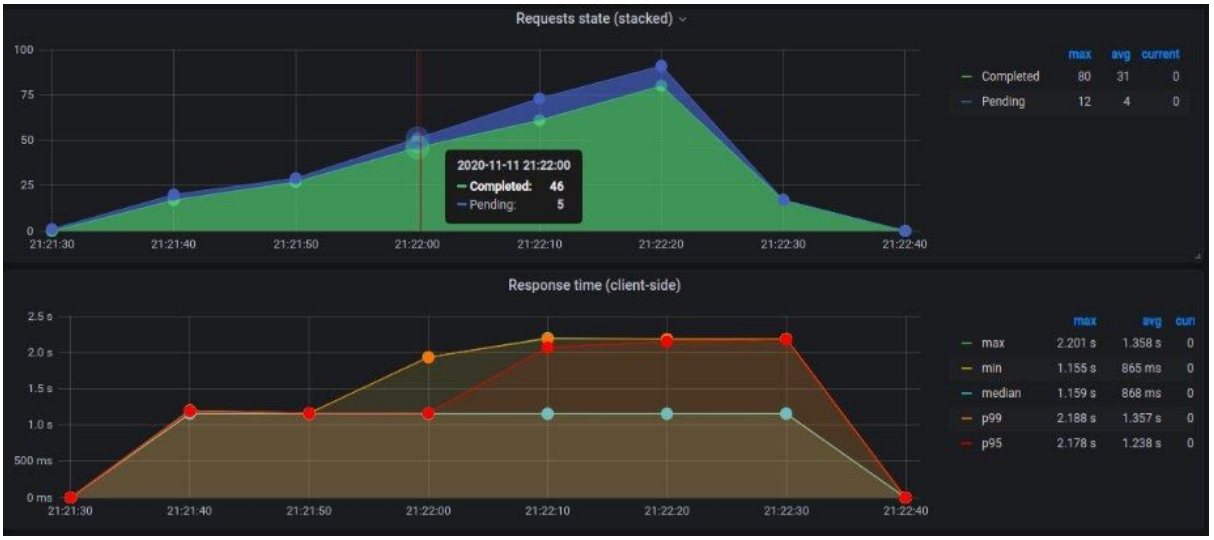
Por medio de lo expuesto se podría llegar a un espectro de posibilidades. Debido a que el gráfico está tomado en intervalos de 10 segundos se pierde la precisión necesaria para establecer un valor exacto. Es por ello que la cantidad de workers queda comprendido entre el siguiente intervalo inclusivo: [3-5].

Análisis posterior

Test Ejecutado

Se realiza un ramp de 0 a 8 request durante 60 segundos

Habiendo realizado el análisis previo, se decidió diseñar un nuevo escenario capaz de esclarecer el valor exacto de workers, o por lo menos acotarlo más. Siguiendo con las mismas hipótesis, se arriba al siguiente resultado:



Con este nuevo análisis, se concluye que la cantidad exacta más probable de workers es 4.

Tiempo de respuesta

Objetivo

En la última subsección se detallarán los tiempos de respuestas de cada uno de los servicios web suministrados.

Test ejecutado

Se envía una request durante 60 segundos.

Hipótesis y supuestos

- No hay ningún agente externo involucrado no tomado en cuenta.

- Las medidas obtenidas con Grafana carecen de error importante para la medición en cuestión.
- El escenario que se utilizará para medir será el mismo para ambos.
- El escenario utilizado no genera en ningún momento mayor cantidad de request por segundo (rps) que las que superan el umbral previamente mostrado y explicado anteriormente.

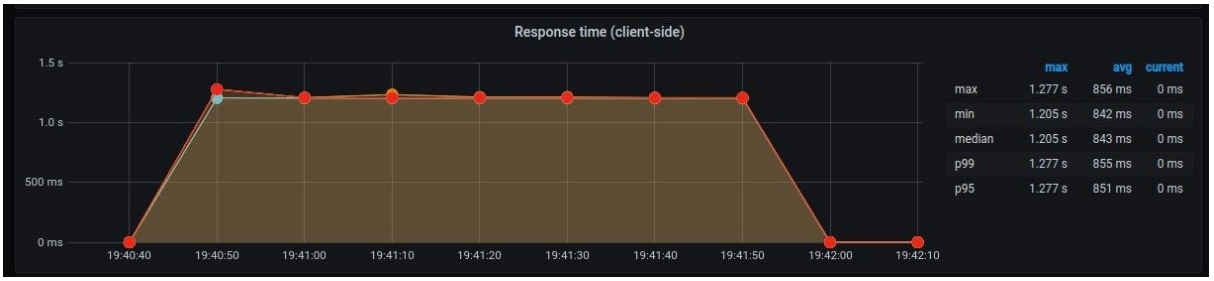
Resultados esperados

Debido a la ausencia de componentes sincrónicos, el tiempo del servicio web debería no solo ser constante, o tender a ello, sino que debería ser menor.

Resultados obtenidos

Escenario utilizado y tiempo de respuesta del servicio 9091

Tiempo de respuesta del servicio 9090



Análisis de resultados

Se puede visualizar fácilmente la diferencia en promedio de respuesta entre un sistema y el otro. Por más de que ambos servicios alcancen valores máximos similares, sus promedios y

medianas difieren en aproximadamente 200 ms. Esto hace que el servicio 9090 tenga menor tiempo de respuesta.

Resumen de sección

Servicio	Comportamiento	Tiempo de respuesta (mediana)	Workers
9090	Sincrónico	1,157s	4
9091	Asincrónico	938ms	-