Arquitectura de Software (75.73)

Trabajo Práctico Nº 1

Grupo: Quality Crunchers

Nombre	Padrón
Cozza, Fabrizio	97.402
Prieto, Pablo	91.561
Condori, Guillermo	98.688

Índice

Objetivos

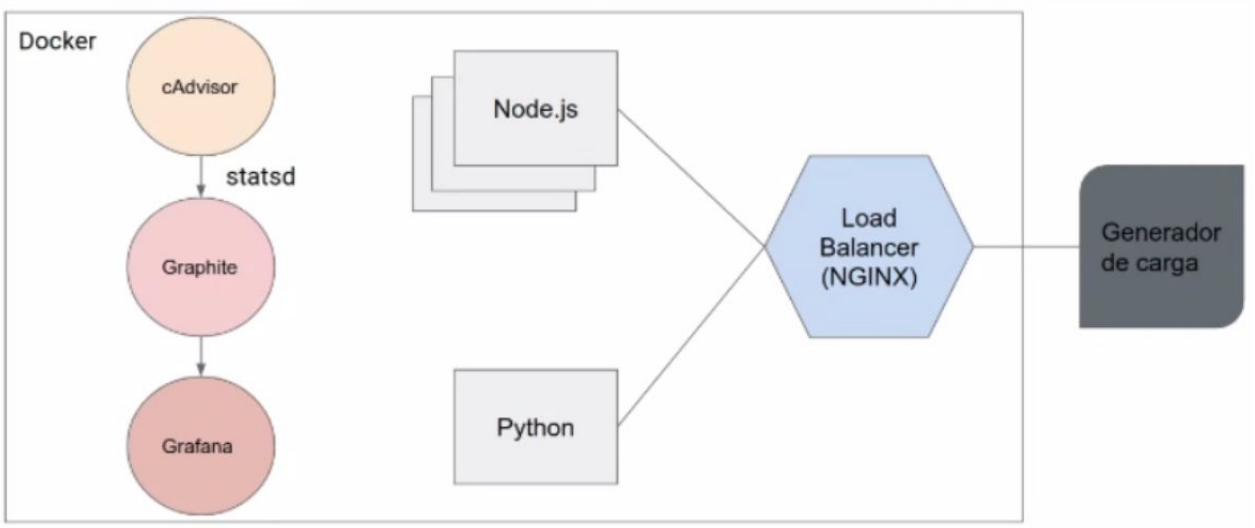
El objetivo principal es comparar algunas tecnologías, ver cómo diversos aspectos impactan en los atributos de calidad y sugerir qué cambios se podrían hacer para mejorarlos. El objetivo menor es que aprendamos a usar una variedad de tecnologías útiles y muy usadas hoy en día, incluyendo:

- Node.js (+ Express)
- Python (+ Flask y Gunicorn)
- Docker
- Docker-compose
- Nginx
- Algún generador de carga (se utilizó Artillery)
- Tomar mediciones varias y visualizarlas, preferentemente en tiempo real, con persistencia, y en un dashboard unificado (Artillery + cAdvisor + StatsD + Graphite + Grafana).

Distribución del informe

Al momento de ver los escenarios se realizaron comentarios básicos de los gráficos para asi tenerlos en cuenta y poder realizar un análisis aglomerativo al final del informe

Arquitectura



Tipos de endpoints para comparar los servidores

Caso	Implementado como	Representa
Ping	Respuesta de un valor constante (rápido y de procesamiento mínimo)	Healthcheck básico
Proxy/timeout	Duerme cierto tiempo y responde (lento y de procesamiento mínimo)	Llamada a otro servicio (request HTTP, llamada a DB, etc.) casi sin procesamiento de datos.
Intensivo	Loop de cierto tiempo (lento y de alto procesamiento)	Cálculos pesados sobre los datos (ej: algoritmos pesados, o simplemente muchos cálculos)

Escenarios

PC Specs
Procesador: Intel Core i7-7700k @ 4.20GHz
RAM: 8 GB

Sección 1

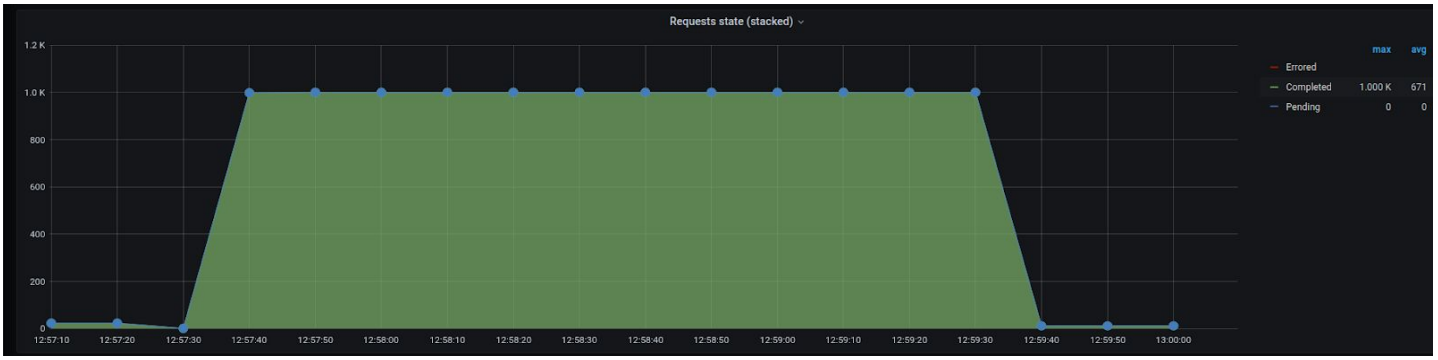
Nota: en algunos casos se pueden ver algunos picos al comienzo de las llamadas a endpoints pero se trata de un problema de inicialización que fue mencionado en clase y es tomado como no relevante para el análisis.

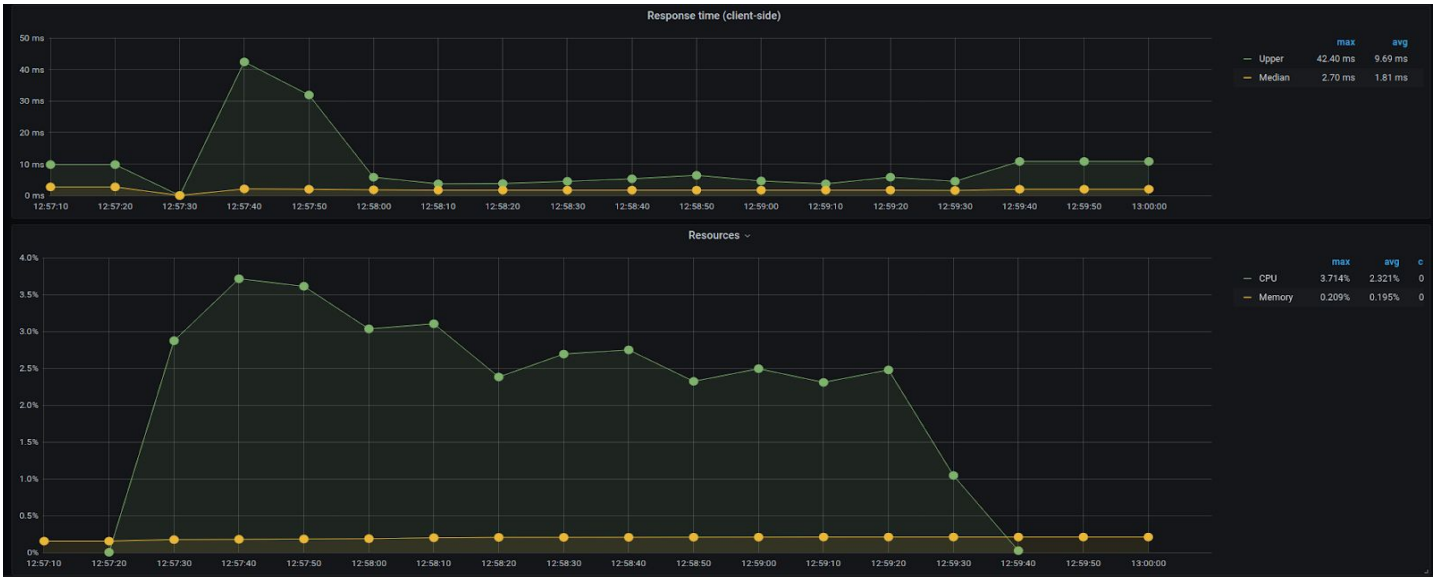
Caso 1: Ping Endpoint

Lo que primero se intentó ver en este caso fue el límite del servidor al enviar reiteradamente una gran cantidad de solicitudes para este endpoint que prácticamente no necesita procesamiento.

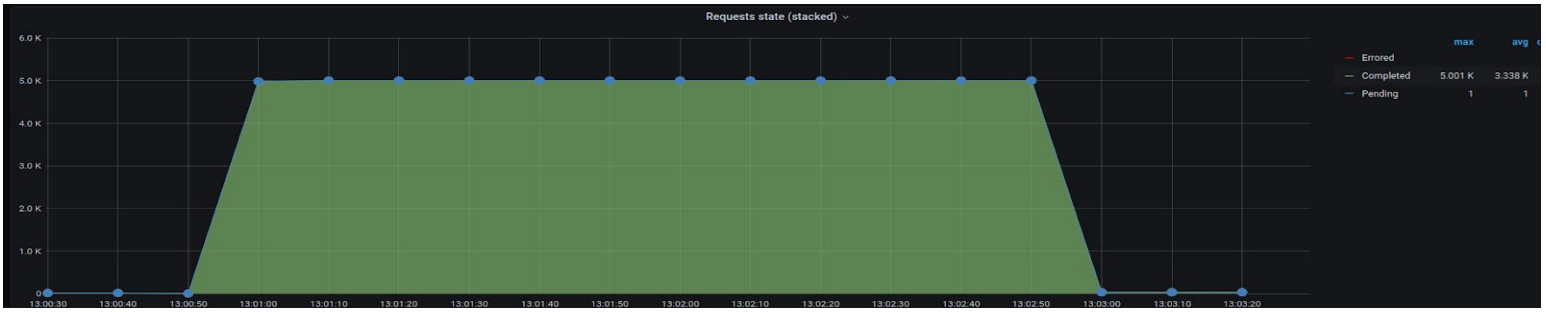
➢ **Server**: Servidor en node, con un solo proceso. Un solo container.

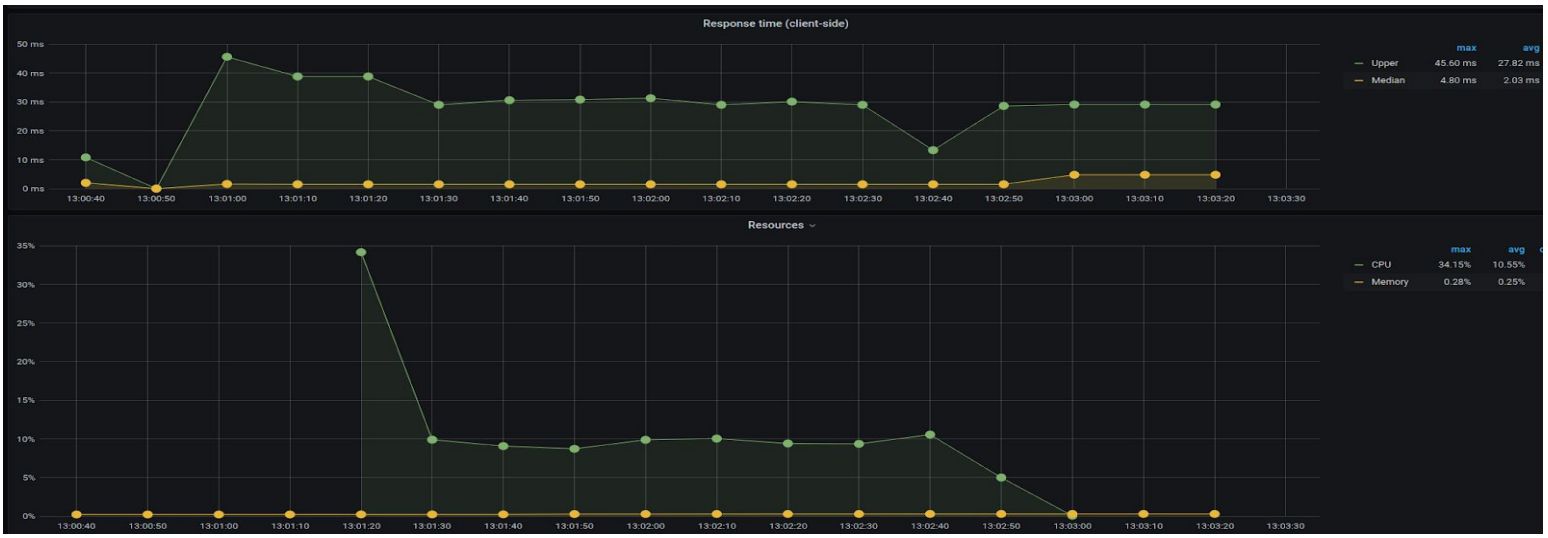
100 requests:



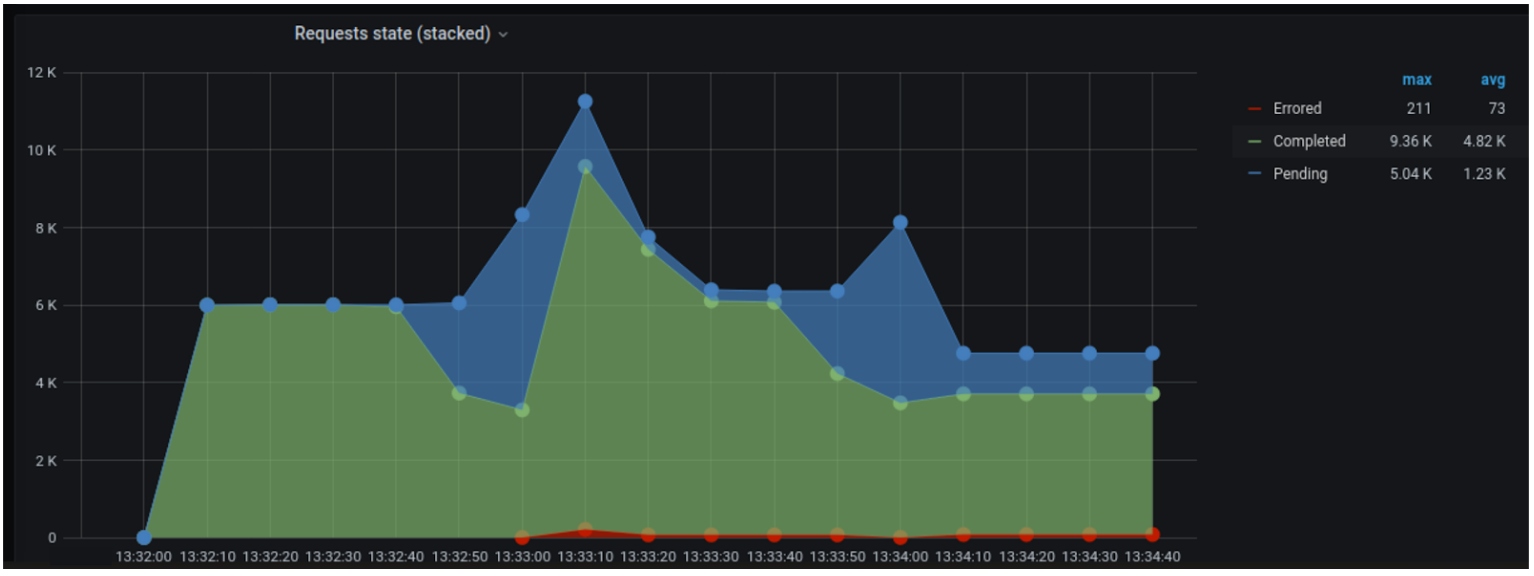


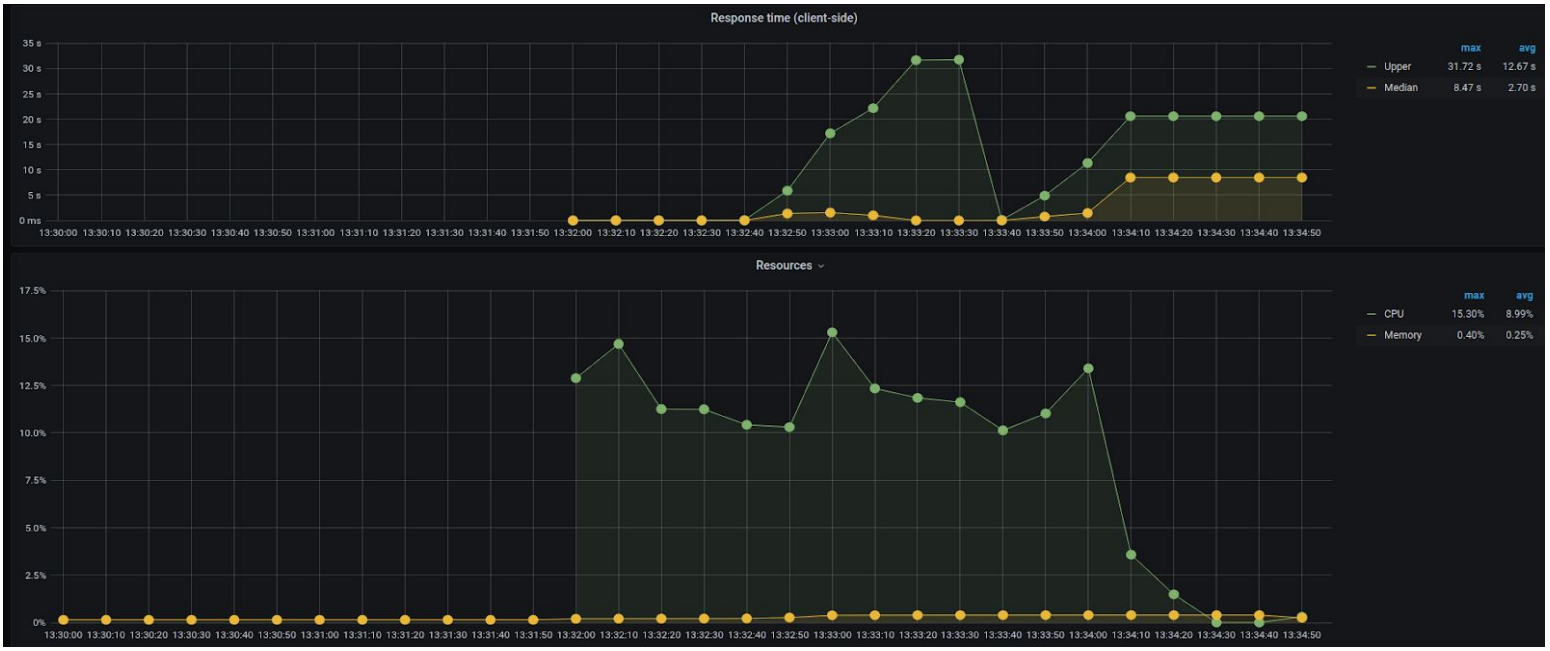
500 requests:





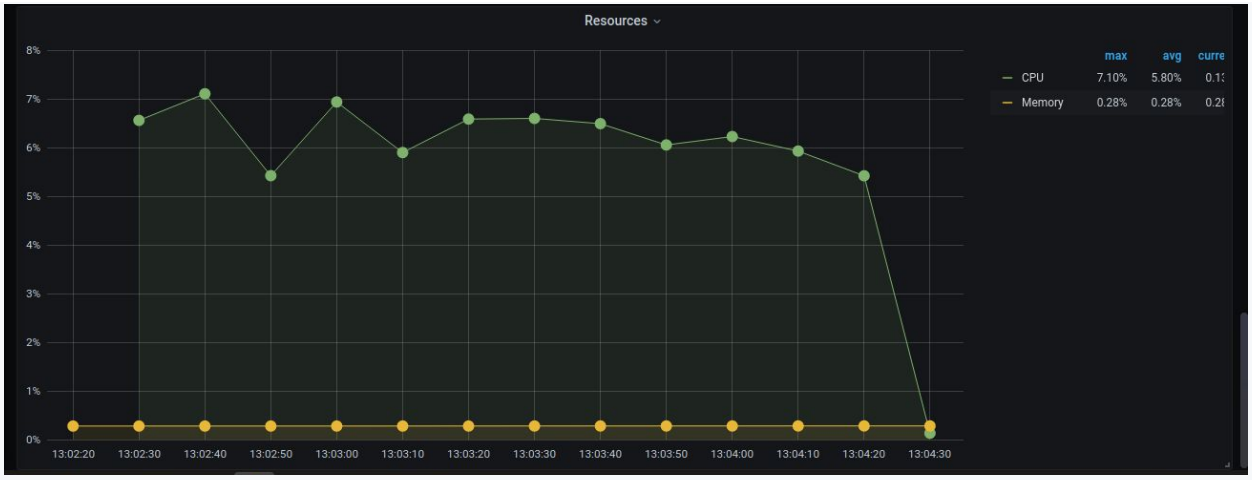
600 requests:



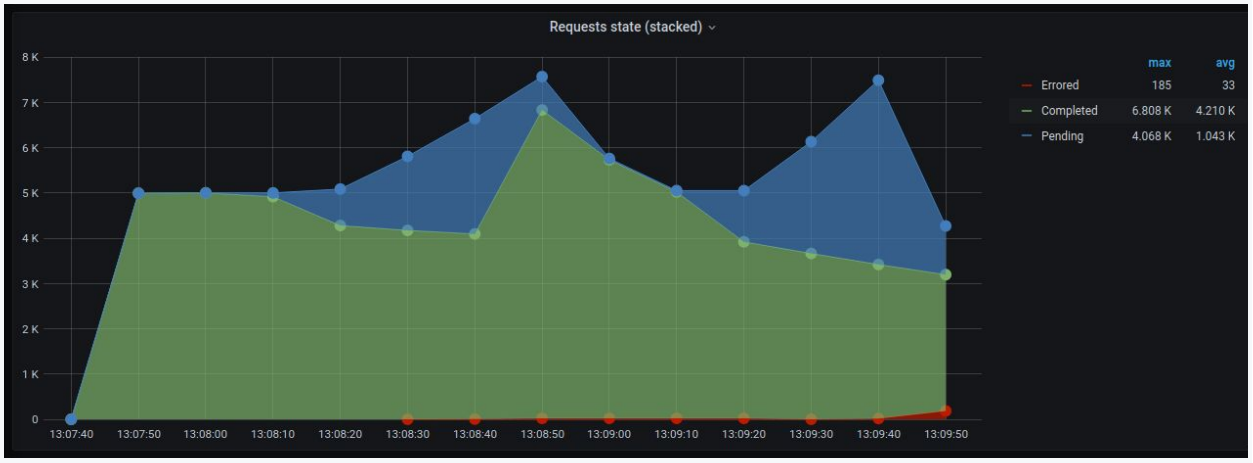


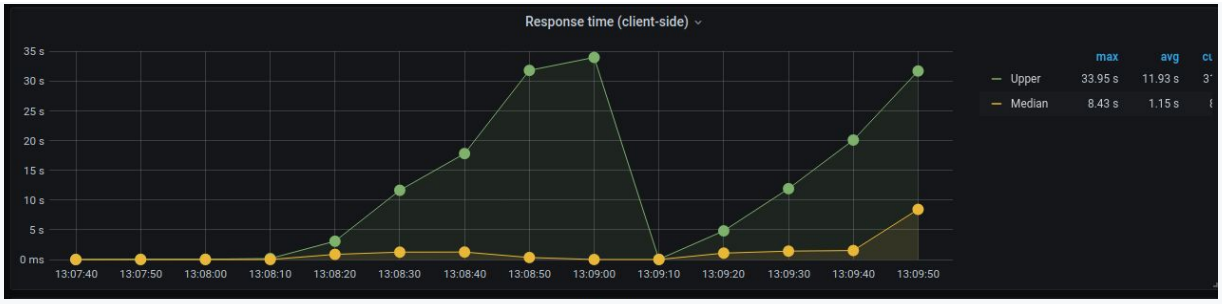
➢ **Server**: Servidor en python con gunicorn, usando un solo worker sincrónico (el del tipo default). Un solo container.

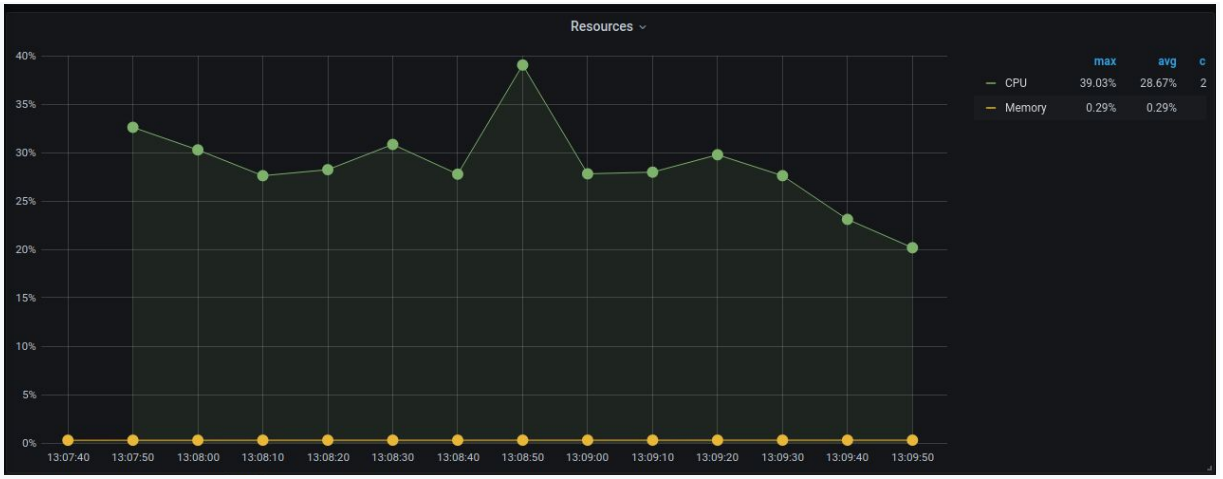
100 requests:



500 requests:







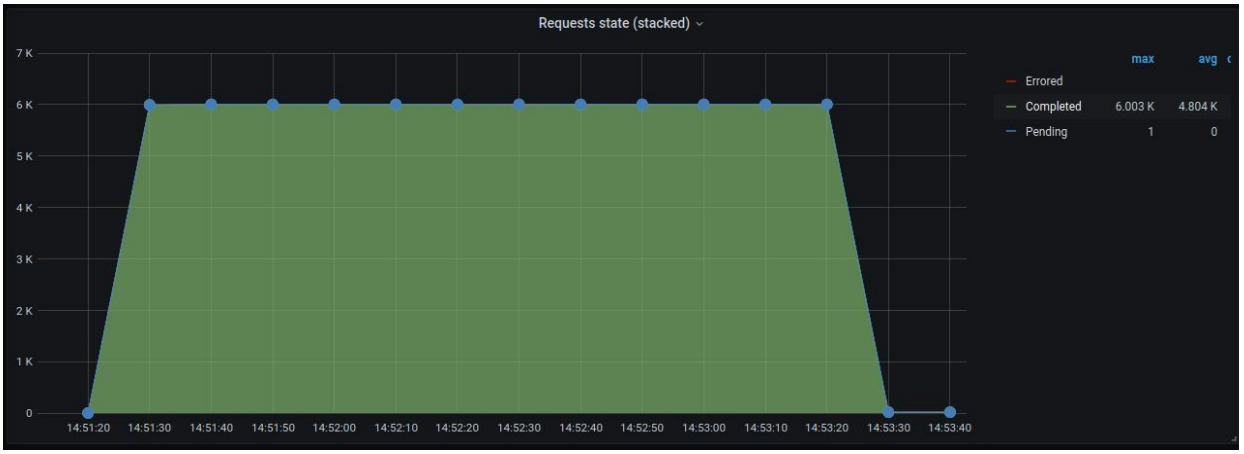
Comentarios servidor único

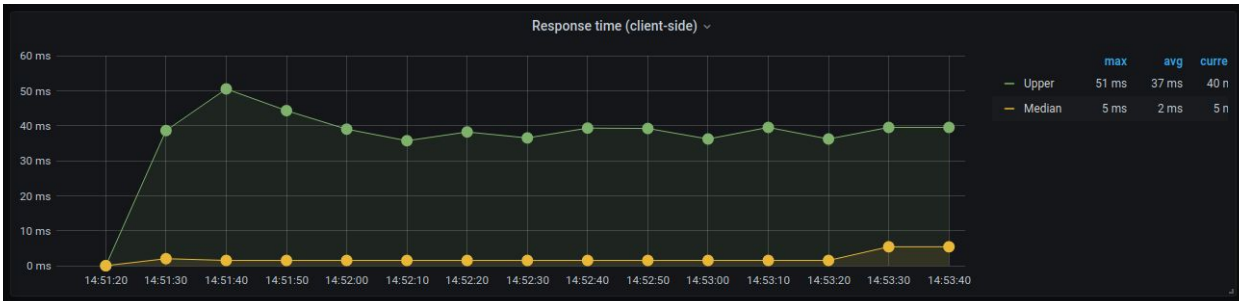
Si bien los gráficos hablan por sí solos y podemos ver que en el caso de Node el límite máximo se encuentra alrededor de los 500 requests por segundo, en gunicorn el límite es menor y su **comportamiento es inesperado**, ya que no estaría apilando las requests a medida que llegan, sino que estaría realizandolos en paralelo hasta cierto límite.
En Node podemos ver que recién por ejemplo en 600 requests comienza a fallar, pero incluso luego de fallar en ese límite podemos ver que hace un trabajo relativamente bueno si vemos el porcentaje de escenarios completados, por lo cual un primer análisis base nos indica que Node es más tolerante a escalar en relación a gunicorn.

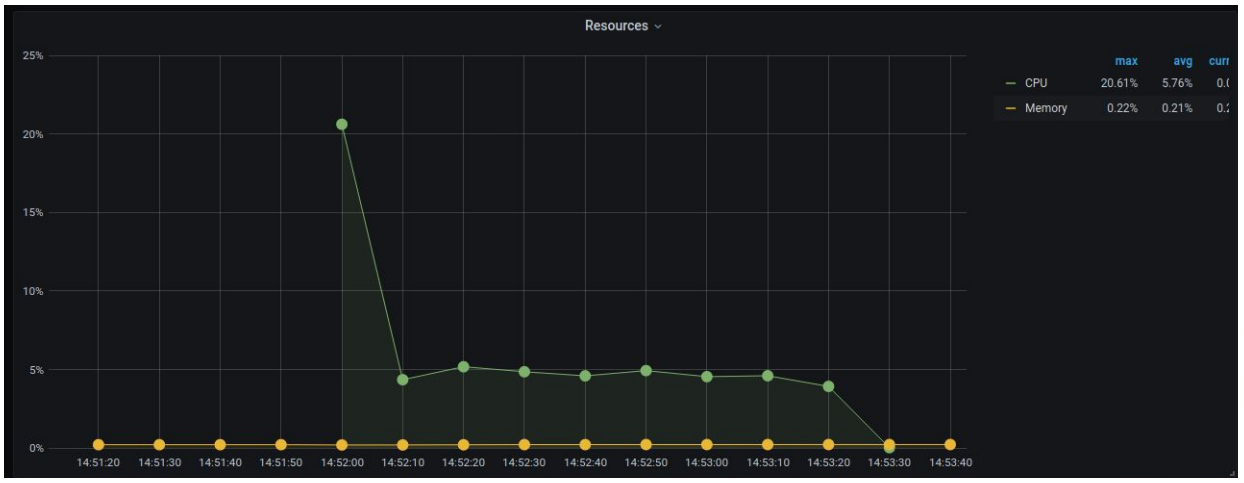
- **Server**: Servidor en node, replicado en 3 containers, con load balancing a nivel de nginx.

Basando en lo visto en los servidores unicos se decidio ver directamente que sucedía en el caso límite para este caso de escalado horizontal.

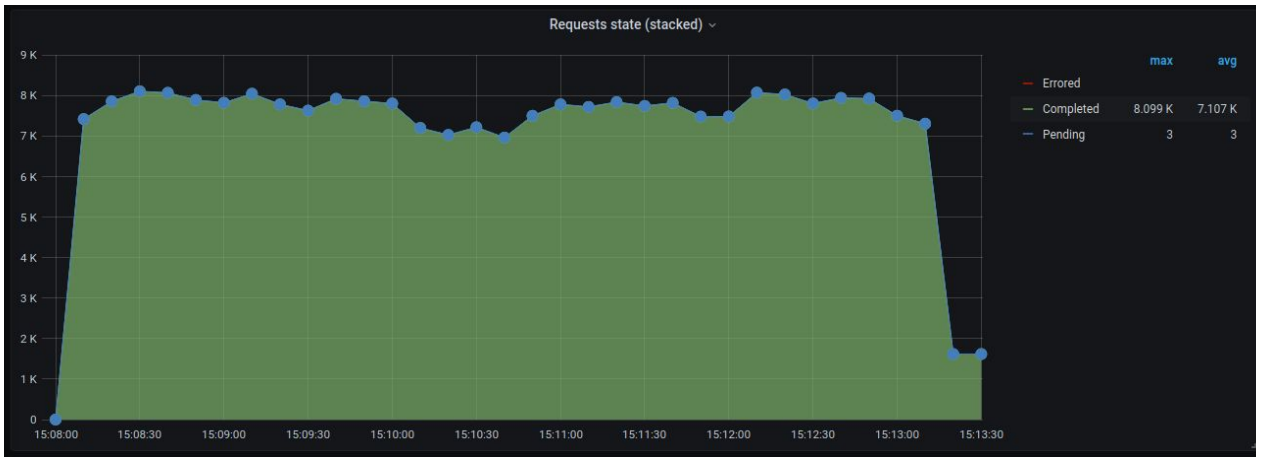
600 requests:

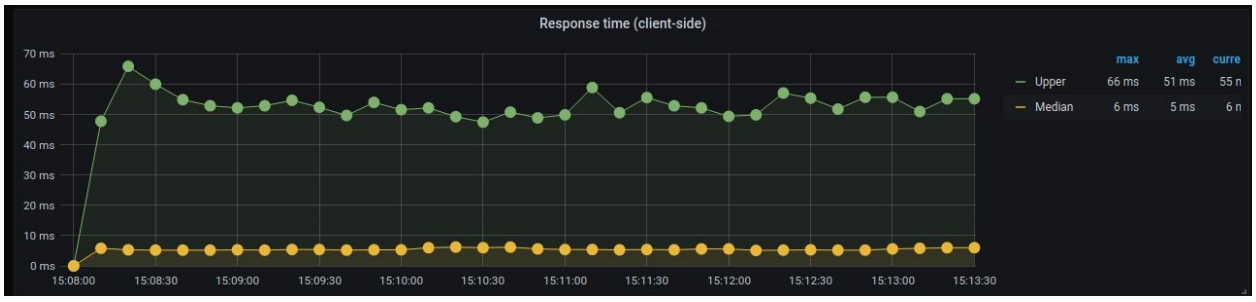


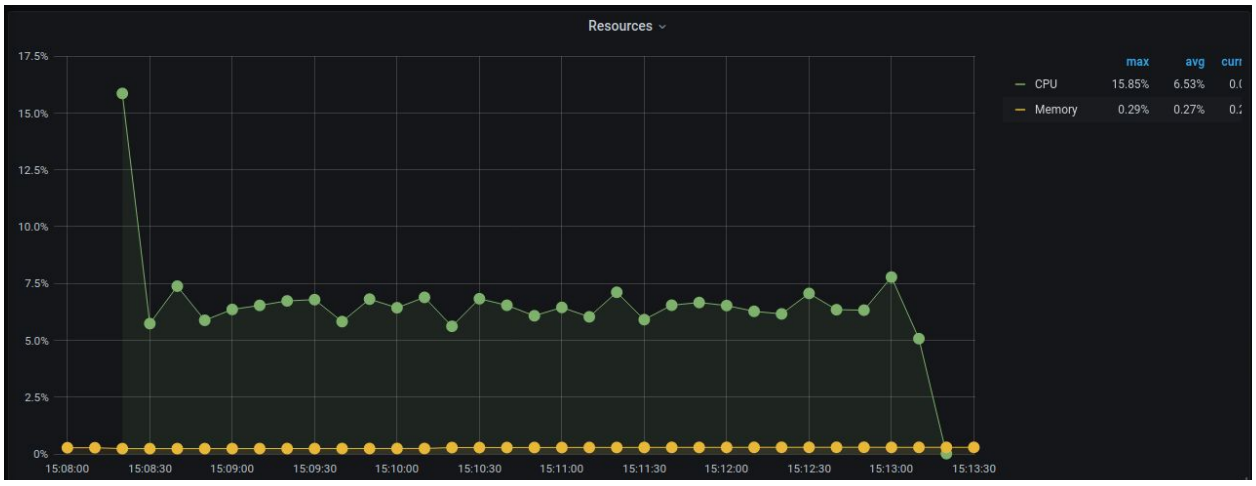




2000 requests:







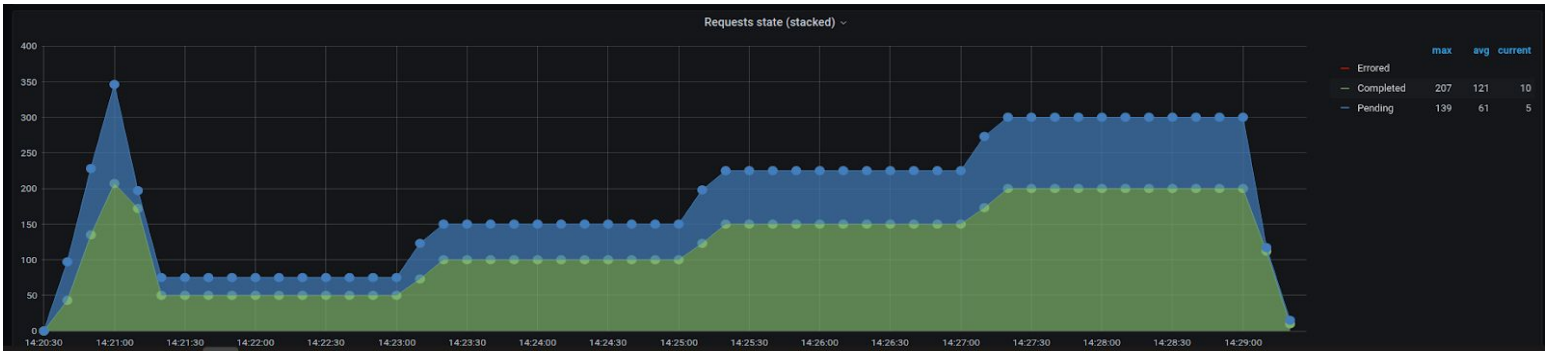
Comentarios servidor replicado

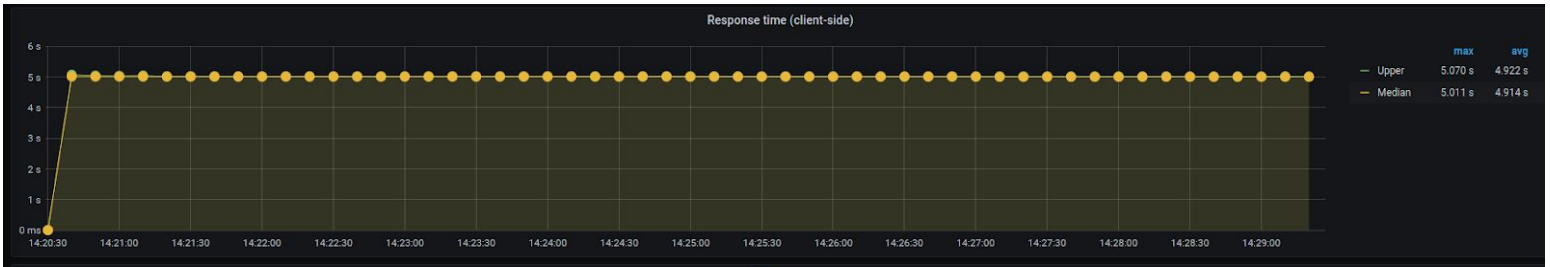
Como era de esperarse, al tener 3 servidores replicados y un load balancer, la carga es distribuida de forma “justa” y los servidores pueden aguantar mucho más carga. Si bien tardan más en tiempo (no en respuesta, sino en la completitud de todas las solicitudes), no hay errores al procesarlas, incluso para el caso en el que son un poco más del triple de request probadas inicialmente.

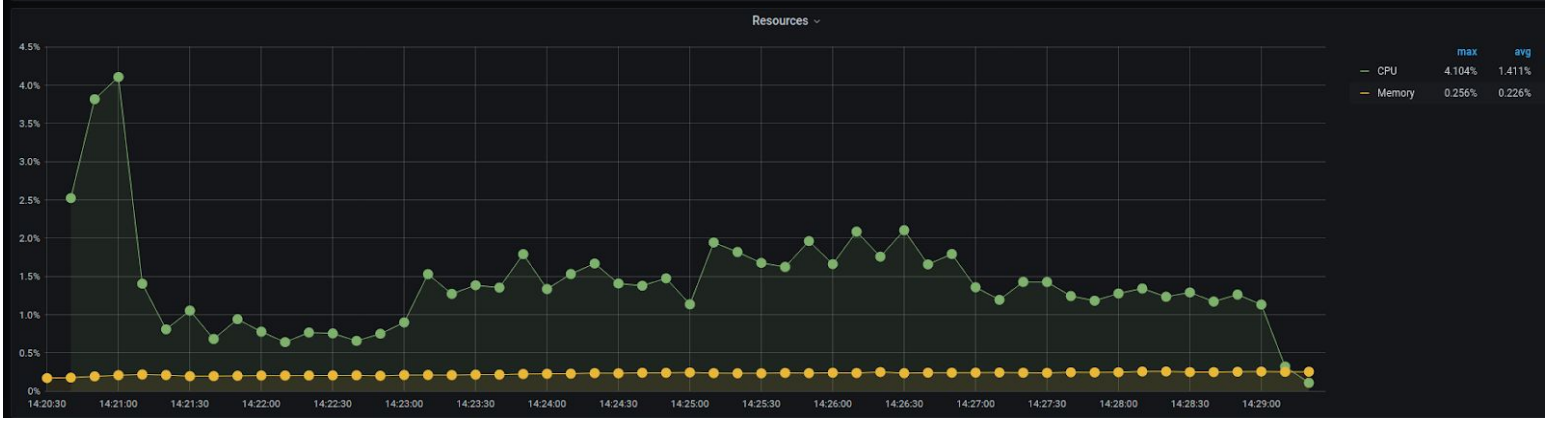
Caso 2: Proxy/timeout Endpoint

Aquí se intentó ver qué causaban solicitudes en los que su objetivo es dormir el proceso 5 segundos, se los fue incrementando progresivamente para notar su efecto.

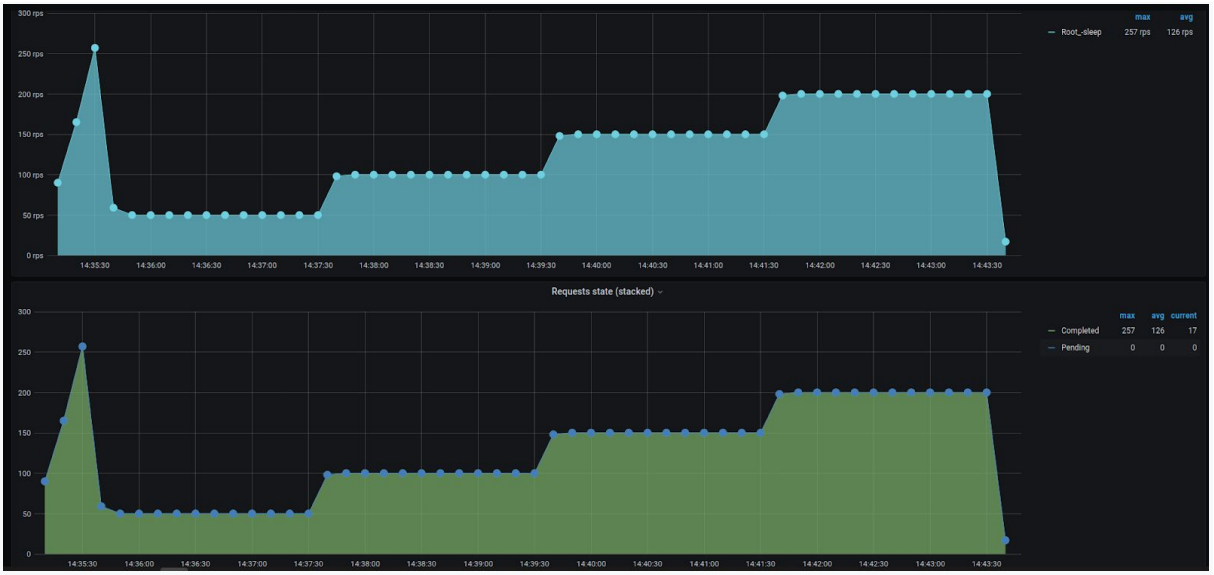
- **Server**: Servidor en node, con un solo proceso. Un solo container.

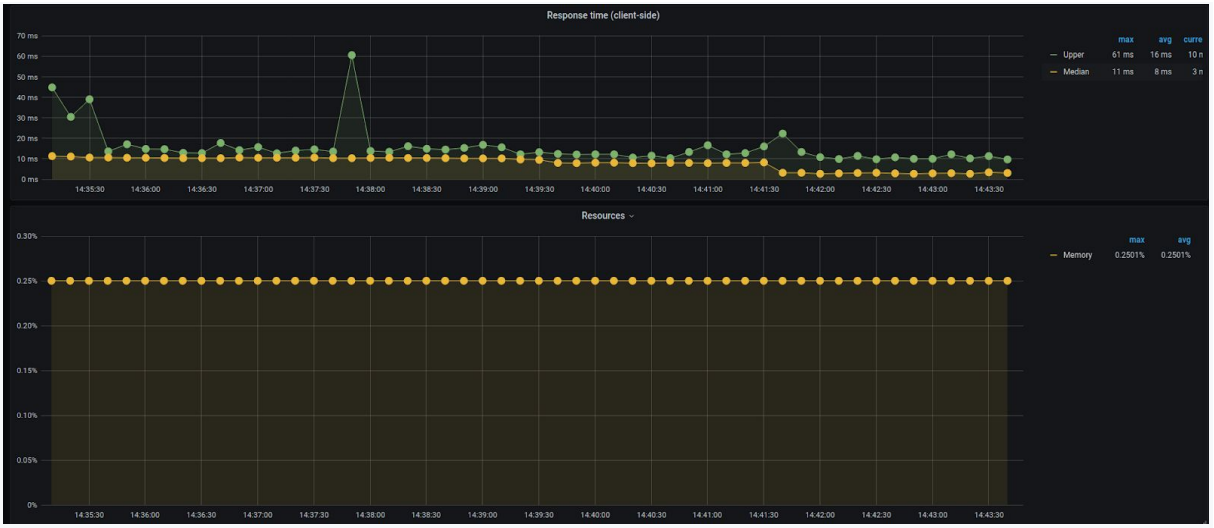






➢ **Server**: Servidor en python con gunicorn, usando un solo worker sincrónico (el del tipo default). Un solo container.





➢ **Server**: Servidor en node, replicado en 3 containers, con load balancing a nivel de nginx.

Comentarios

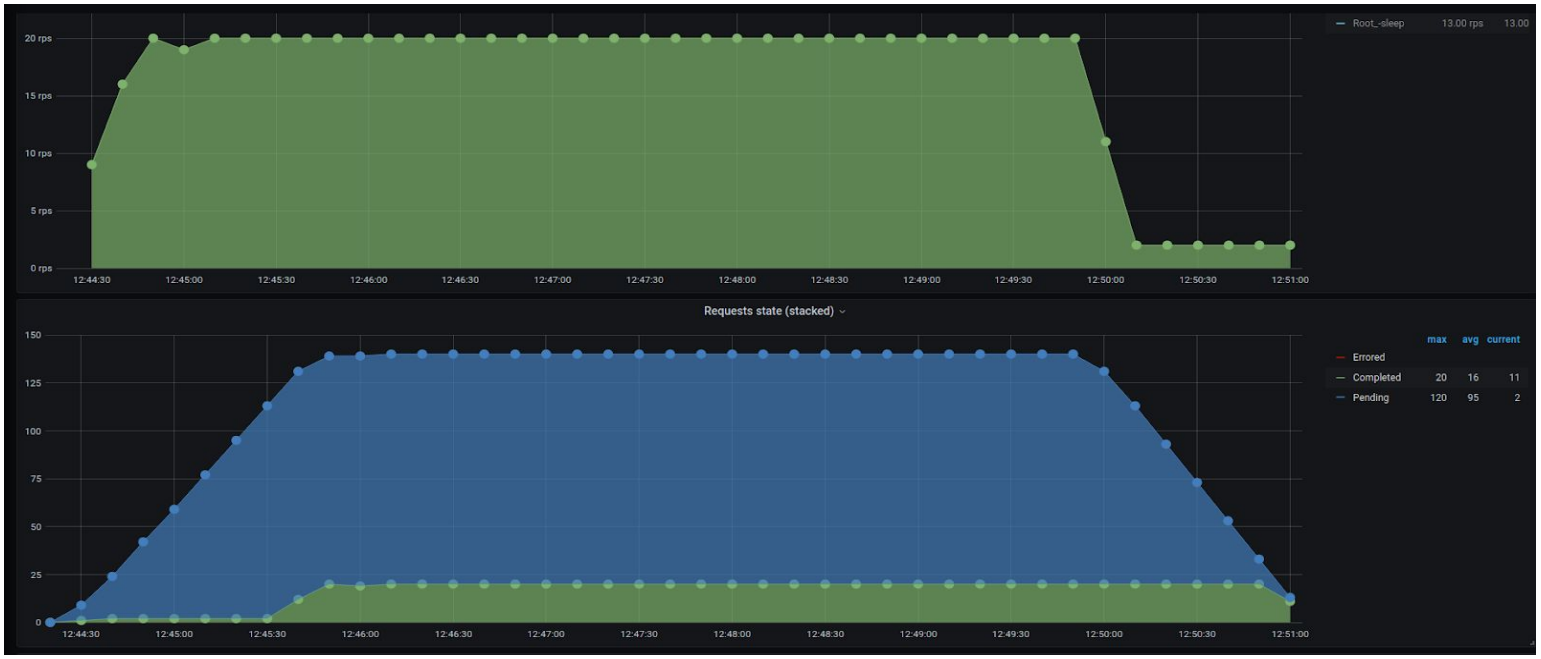
Aquí podemos ver la capacidad de Node y su Event Loop en postergar aquellas acciones que deben esperar para ser completadas, ya que continúa procesando los requests sin problema a medida que siguen llegando (los va acumulando mientras procesa otros).
La diferencia que se ve entre escenarios completados y pendientes son estos 5 segundos de timeout (como se puede ver en la mediana en toda la fase) que tarda el proceso en completarse, pero en ningún momento el servidor se bloquea ni genera problemas.
En el caso Gunicorn se comporta nuevamente de **forma inesperada** ya que debería poner en pendientes muchos de los requests ya que debe esperar 5 segundos para cada uno, pero en cambio los realiza paralelamente.

Caso 3: Heavy Endpoint

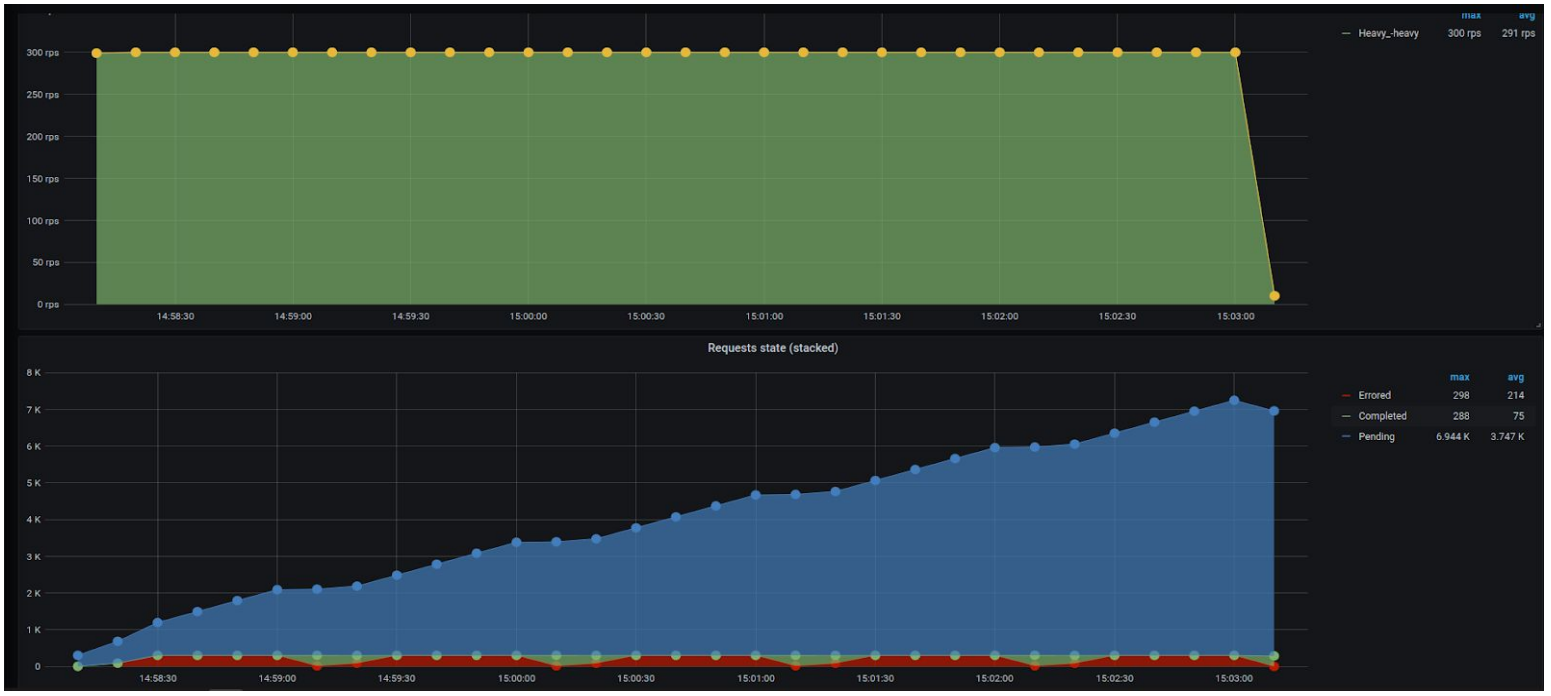
El objetivo aquí fue ver que sucedía al enviar solicitudes que causarán una acción bloqueante en el servidor

➢ **Server**: Servidor en node, con un solo proceso. Un solo container.

Heavy “tranquilo” (2 requests por segundo)



Heavy “full” (30 requests por segundo)

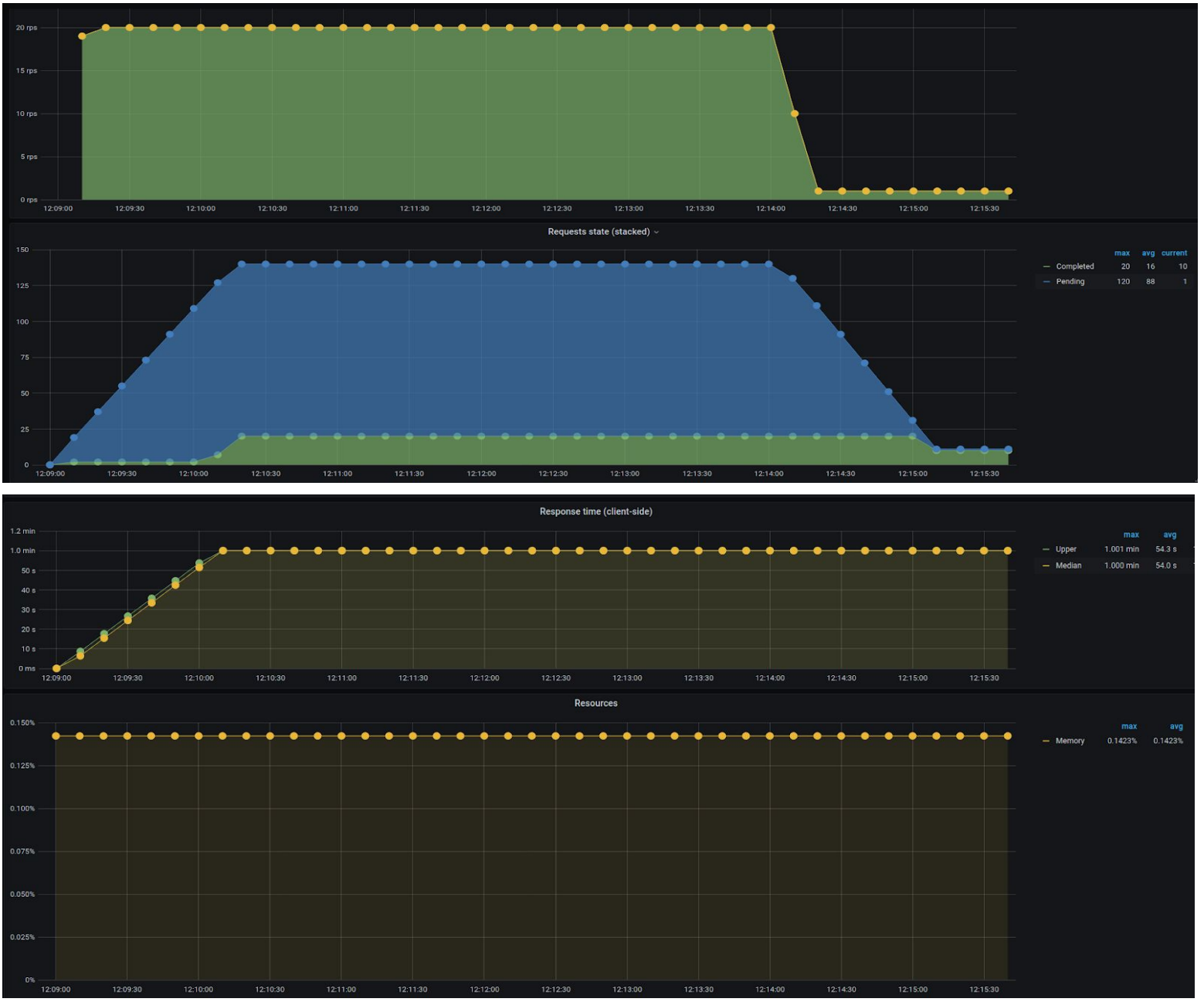

max avg
Heavy_-heavy 300 rps 291 rps
300 rps
250 rps
200 rps
150 rps
100 rps
50 rps
0 rps
14:58:30
14:59:00
14:59:30
15:00:00
15:00:30
15:01:00
15:01:30
15:02:00
15:02:30
15:03:00

Requests state (stacked)
max avg
Errored 298 214
Completed 288 75
Pending 6.944 K 3.747 K
8 K
7 K
6 K
5 K
4 K
3 K
2 K
1 K
0
14:58:30
14:59:00
14:59:30
15:00:00
15:00:30
15:01:00
15:01:30
15:02:00
15:02:30
15:03:00

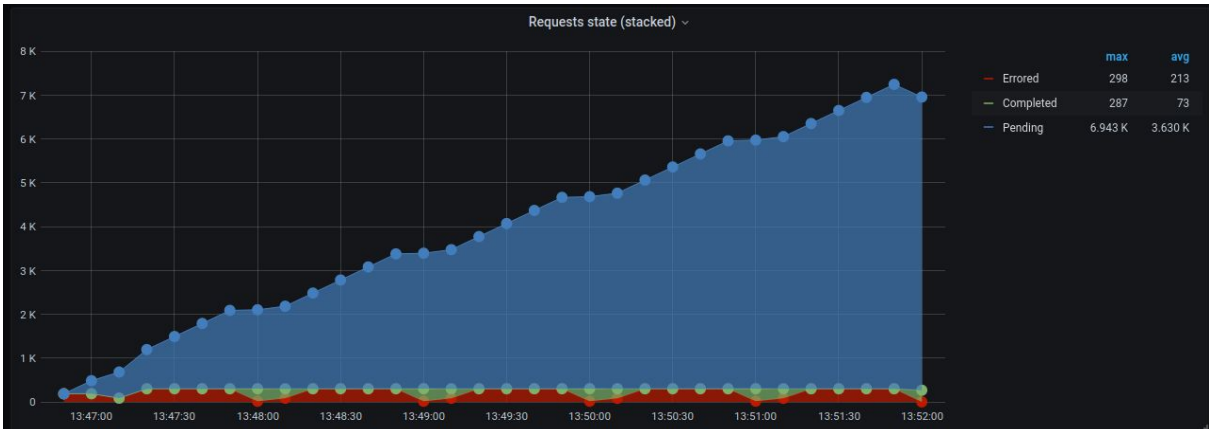

Response time (client-side)
max avg
Upper 1.001 min 54.2 s
Median 1.000 min 53.8 s
1.2 min
1.0 min
50 s
40 s
30 s
20 s
10 s
0 ms
14:58:30
14:59:00
14:59:30
15:00:00
15:00:30
15:01:00
15:01:30
15:02:00
15:02:30
15:03:00

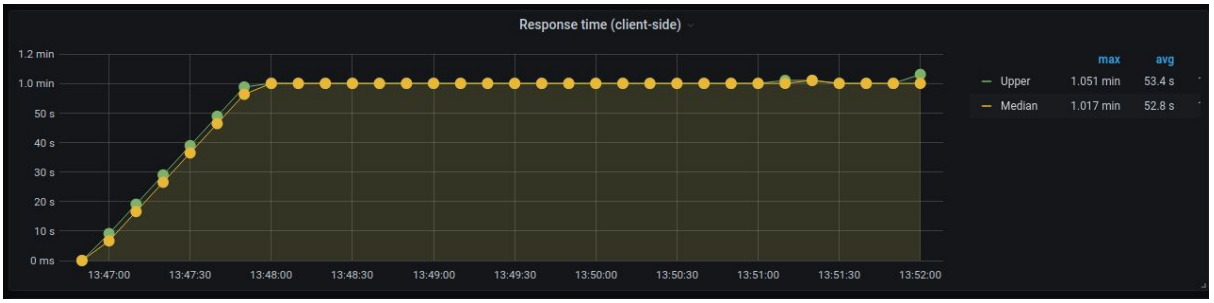
Resources
max avg current
CPU 251% 106% 94%
Memory 0% 0% 0%
300%
250%
200%
150%
100%
50%
0%
14:58:30
14:59:00
14:59:30
15:00:00
15:00:30
15:01:00
15:01:30
15:02:00
15:02:30
15:03:00

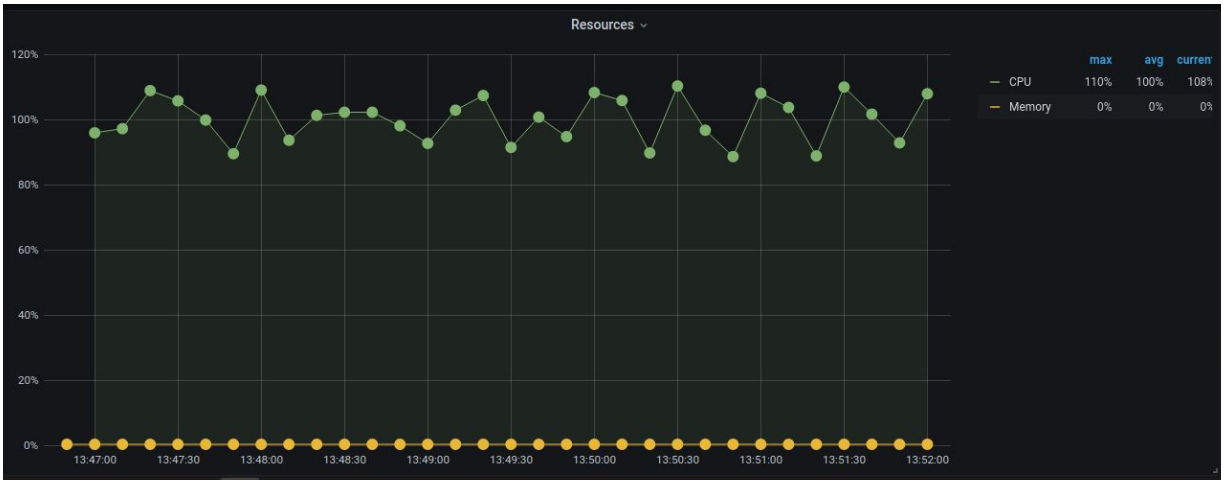
➢ **Server**: Servidor en python con gunicorn, usando un solo worker sincrónico (el del tipo default). Un solo container.



Heavy full:





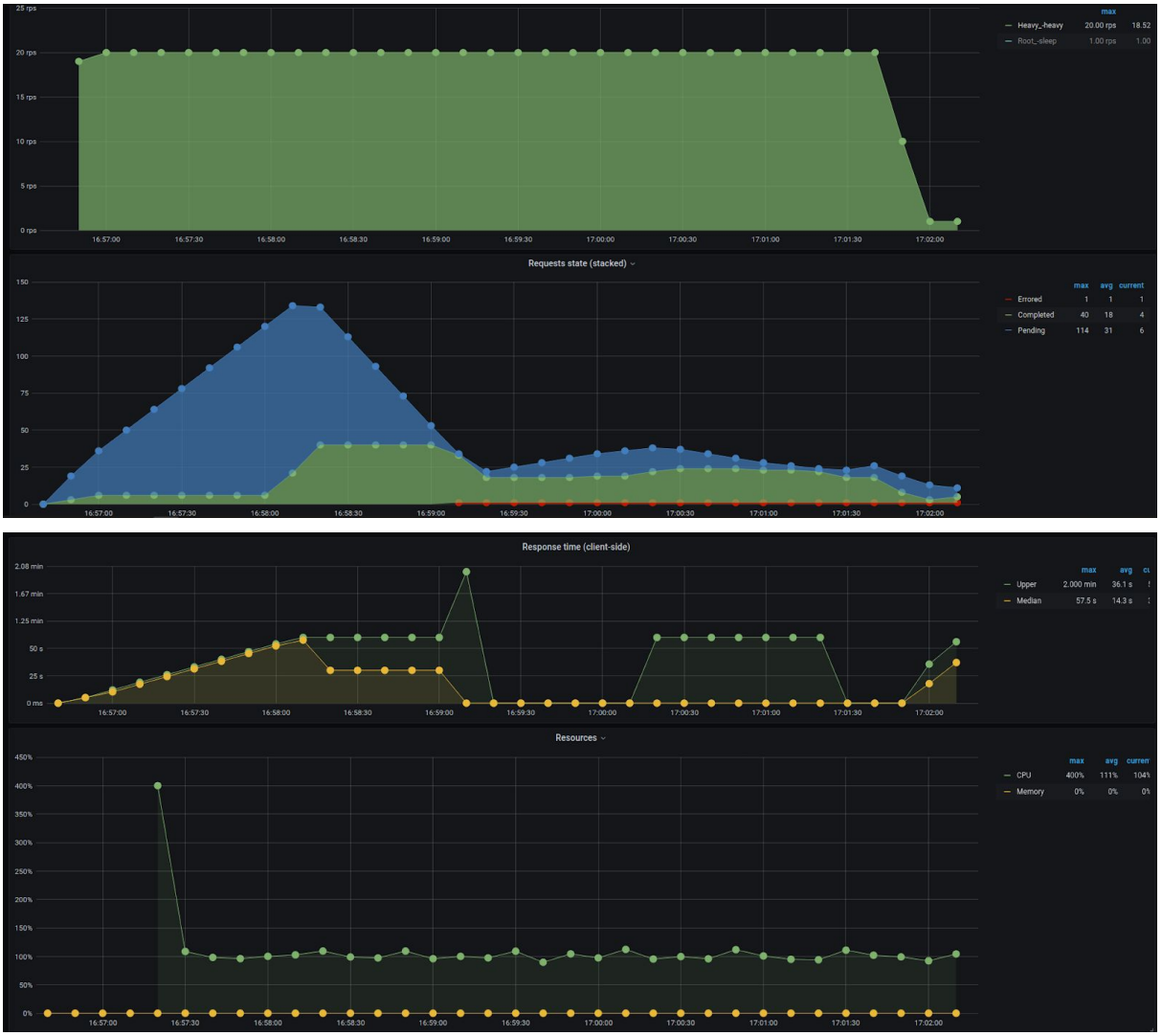


Comentarios servidor único

Como era de esperarse podemos ver como este caso no es viable para ninguno de los dos servidores. Si bien cuando la cantidad de requests son pocos las solicitudes logran completarse, el tiempo de espera que tenemos para la completitud de las mismas es elevado como también lo es el procesamiento del servidor, por lo cual esto podría generar un congelamiento para todo aquel que intente acceder en este momento.
Como el caso es bloqueante, ambos servidores se comportan de manera similar, independientemente de su funcionamiento y paralelismo

- **Server**: Servidor en node, replicado en 3 containers, con load balancing a nivel de nginx.

40 requests:



Comentarios servidor replicado

Nótese que las acciones bloqueantes siguen costando al servidor sus recursos, pero debido a la distribución de carga dada, como sucedió anteriormente con el ping, los servidores replicados son capaces de mucho mayor procesamiento, notar que ahora con una gran cantidad mayor (40 solicitudes) se provee un solo error, cosa que en los casos singulares causaba muchos problemas y ni siquiera se acercaran a ser completados.

Sección 2

Caso 1

➢ **Endpoint**: GET al primer servicio, puerto: 9090

Summary report:

```
  Scenarios launched:  6644
  Scenarios completed: 6644
  Requests completed:  6644
  Mean response/sec: 10.37
  Response time (msec):
    min: 1300.6
    max: 1338.2
    median: 1302.3
    p95: 1305.1
    p99: 1309.4
  Scenario counts:
    Root (/): 6644 (100%)
  Codes:
    200: 6644
```

Propiedades:

Caso asincrónico, basándonos en los gráficos de node sleep de la primera sección se responden varios request al mismo tiempo (y se completan mientras algunos quedan pendientes) con aproximadamente el mismo response time para todos. Según la mediana el response time es de alrededor 1.3 segundos.

Caso 2

➢ **Endpoint**: GET al segundo servicio, puerto: 9091

Summary report:

Scenarios launched: 6633
Scenarios completed: 109
Requests completed: 109
Mean response/sec: 8.84
Response time (msec):
min: 1210.5
max: 119481
median: 59001.7
p95: 113904.7

```
  p99: 118871.5
Scenario counts:
  Root (/): 6633 (100%)
Codes:
  200: 109
Errors:
  ESOCKETTIMEDOUT: 6524
```

Propiedades:

Caso sincrónico, las solicitudes se van acumulando a medida que llegan y aquellas que se realizan no producen procesamiento alto pero si toman el worker produciendo una cola larga de requests acumulados.
Según la mediana el response time es de casi 2 minutos.
Mirando los gráficos podemos decir que el indicio de la cantidad de workers es 2, ya que el procesamiento se realiza dentro del tiempo de 10 minutos y 9 requests son completados. Teniendo en cuenta que cada una de las solicitudes dura casi 2 minutos, para completar esta cantidad de 9 solicitudes se tendría de manera más “matemática” que:
Cant_req_completadas = 9
T_procesamiento = 10 min
T_response = casi 2 min

Se debe cumplir lo siguiente:
T_response * Cant_req_completadas = T_procesamiento

Para que esto se cumpla se necesita separar a la cant_req_completadas en 2 threads tal que:
T_response * Cant_req_completadas_W1 = T_procesamiento
T_response * Cant_req_completadas_W2 = T_procesamiento
Cant_req_completadas_W1 + Cant_req_completadas_W2 = Cant_req_completadas

Ya que es la única manera de que los números tengan sentido y por eso se piensa que la cantidad de workers es de 2.

Análisis final

Luego de mirar en detalle ambas secciones lo primero a resaltar es el **funcionamiento imprevisto de gunicorn en las pruebas de ping y sleep**, no supimos deducir si se debe a un error de configuración pero este tipo de pruebas deberían comportarse como en el caso sincrónico de la sección 2.

Atributos de calidad

A continuación vamos a ir analizando qué sucedió con los atributos de calidad principales en relación a lo visto a lo largo del informe.

Disponibilidad

Comenzando con este atributo podemos decir que en el caso de Node, siempre y cuando no sea una solicitud “heavy”, el servidor se mantuvo utilizable gracias a su carácter asincrónico, mientras que por otro lado, debido al sincronismo que caracteriza gunicorn, la respuesta fue de 1 solicitud a la vez (en el caso de 1 worker) y si era de mediano o intenso procesamiento se acumulan solicitudes a medida que el tiempo transcurre generando fallas y tiempos de respuesta altos.

En ambos casos, si la solicitud fuera intensiva la disponibilidad se ve afectada por la naturaleza del proceso ya que es bloqueante, generando baja disponibilidad para ambos casos.

Performance

Comparando el rendimiento de ambos servidores, hay una clara ventaja a favor de Node ya que procesa una mayor cantidad de solicitudes en paralelo sin problemas. Esto se debe al esquema de trabajo del Event Loop de Node contra el sincronico de gunicorn con 1 worker.

Escalabilidad

Se pudo ver claramente como la escalabilidad horizontal aumentaba en el caso de 3 servidores replicados de Node, ya que podíamos aumentar la carga del sistema en gran tamaño sin que el mismo tuviese problemas.

Visibilidad

Esto está dado por el componente NGINX (load balancer) ya que modera la interacción entre el generador de carga y los servidores, particularmente visible en el caso de los 3 servidores de Node replicados, ya que distribuye las solicitudes del cliente con planeamiento Round-Robin.

Usabilidad

Si bien no hay una interfaz gráfica para el sistema, uno podría imaginarse que en el caso de la solicitud intensiva, el sistema podría congelarse ofreciendo una experiencia de uso insatisfactoria al usuario reduciendo su usabilidad.

Portabilidad

Gracias a Docker, el sistema es portable, y las aplicaciones pueden ejecutarse en cualquier entorno con Docker instalado, independientemente del sistema operativo, facilitando los despliegues.

Posibles alteraciones

Si quisiéramos que gunicorn tenga efecto en los atributos de calidad mencionados, podrían agregarse más workers escalando de esta manera en forma vertical y permitiendo múltiples request al mismo tiempo (hasta que se llegue al límite de los recursos del sistema).

En el caso de Node y el caso en el que se lo escala horizontalmente, pudimos ver que tenemos hasta por lo menos el triple de posibilidades de realizar solicitudes en paralelo y pudimos realizar hasta 2000 requests en el caso simple, lo cual no es dato menor, incrementando principalmente la disponibilidad y performance.

Estos posibles cambios mencionados generan que exista también una relación con el atributo de modifiability.

Un comentario a tener en cuenta es que en algunos casos gunicorn es más fácil de implementar que node por lo cual según lo que se necesite podría ser conveniente a pesar de todo ya que en el balance general pudimos ver que el caso asincrónico supera al sincronico en casi todos los casos probados.