FACULTAD
DE INGENIERIA
Universidad de Buenos Aires

75.73 - Arquitectura del Software
Trabajo Práctico 1

Integrantes:

Diego Alfonso Oderigo, 83969
Marcelo Calarota, 94839
Malena Maffei, 97426
Santiago Monsech, 92968

Objetivo

El presente trabajo tiene como objetivo el análisis de atributos de calidad de diversos tipos de *web services*, con especial foco en la *performance* y la *disponibilidad*. Se encuentra dividido en dos partes:

- En la primera, se analizan estos atributos para dos endpoints, cada uno a su vez implementado en dos lenguajes distintos (NodeJS y Python), que simulan dos casos de uso. Uno de ellos hace un uso de CPU intensivo con escaso o nulo I/O, mientras que el otro simula un contexto de operaciones de I/O de larga duración y que podría beneficiarse del uso del asincronismo. Se incluye además, en los casos en los que se consideró que tenía sentido, un tercer endpoint que emula un *health check* (es decir, que recibe un request e instantáneamente envía una respuesta).
- En la segunda, a partir de los *insights* logrados en la primera parte, se analizan dos endpoints desconocidos para tratar de determinar sus características.

Resumen ejecutivo

Se someterán a evaluaciones servicios distintos implementados en Node y Python, a fin de entender ventajas, desventajas, problemas de cada uno. Asimismo, aprender herramientas como cAdvisor, Graphite, Grafana, Artillery y Nginx.

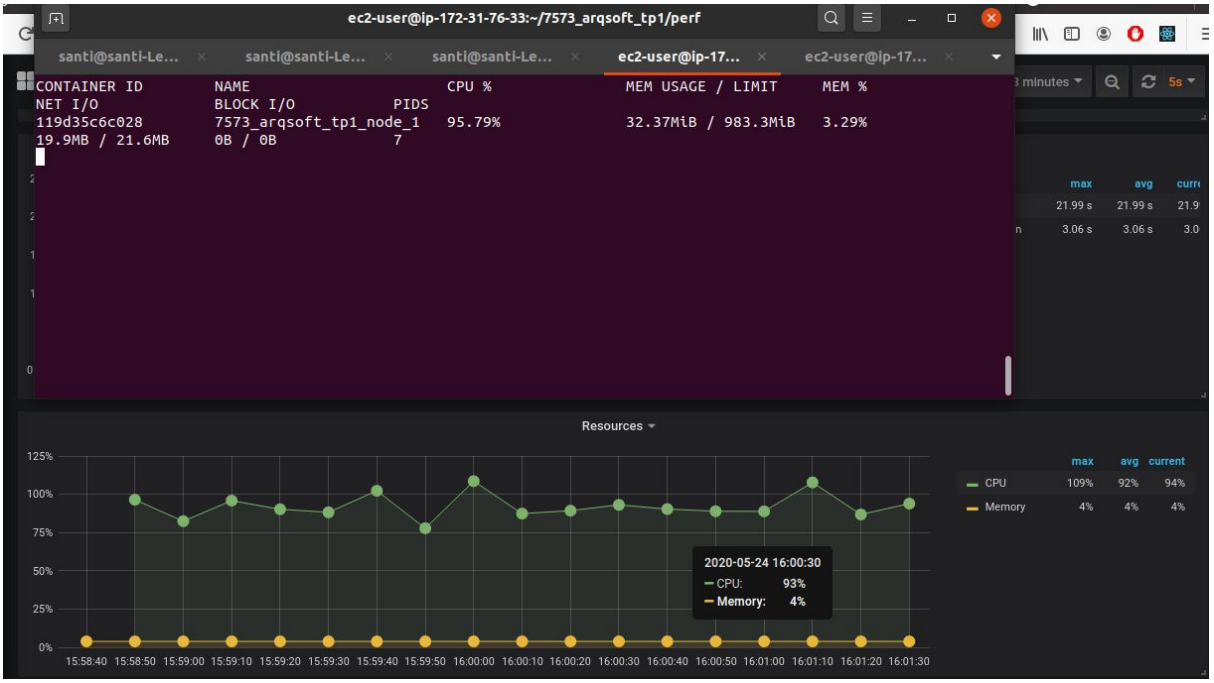
Setup de la infraestructura

Con el fin de tener un banco de pruebas estable y unificado, se desplegaron los servicios en instancias EC2 de AWS (de tipo t2.micro). De esta manera, se evita cualquier interferencia debida a circunstancias ajenas a la prueba (como podría ser por ejemplo un antivirus corriendo en paralelo a las pruebas de CPU intensivo) y se logran resultados repetibles en el tiempo.

El siguiente paso fue calibrar los parámetros de la prueba dadas las características de la infraestructura (1 vCPU, 1 GB de memoria RAM). Para determinar, por ejemplo, la cantidad de unidades de procesamiento que representan el 100% de uso de CPU, se levantó el conjunto de contenedores y se corrió una prueba de carga para el *endpoint* `/heavy`, que hace un uso intensivo del procesador, pero con el generador de carga (*Artillery*) ejecutándose en una PC fuera de la instancia, es decir, a través de la red. De esta manera, se eliminó el ruido que pudiera provocar sobre las mediciones el propio generador de carga. Para ver el consumo de recursos de los contenedores se utilizó el comando:

```
docker stats
```

La salida de este comando indica el uso de memoria y CPU de cada contenedor. Esta información se utilizó para calibrar las visualizaciones de uso de recursos en el *dashboard* de *Grafana*, como se muestra en la siguiente imagen:



Podemos ver que en la consola, el docker stats muestra un consumo del 95% del CPU del AWS y un consumo de memoria de más del 3%. Asimismo, en el chart de Grafana que muestra los recursos, vemos que el CPU ronda el 90% del consumo, y la memoria ronda un 4%. Comparando la consola y grafana, decimos que Grafana esta debidamente calibrado en la escala.

Configuraciones

El trabajo se desarrolla utilizando tres sets distintos de configuraciones para la implementación de los servicios:

- NodeJS + Express - 1 contenedor
- Python + Flask + GUnicorn - 1 contenedor con un único worker
- NodeJS + Express - 3 contenedores, balanceados por *nginx* con una estrategia de *Round Robin* (todos los contenedores con igual “peso”).

Todos los servicios implementan los mismos endpoints:

- Para el **/ping**, este recibe la llamada, la ejecuta instantáneamente y devuelve una respuesta en en acto, por lo que el nginx no va a encolar requests.
- Para el **/timeout**, este recibe la llamada, la ejecuta y queda procesando en background 5 segundos a través de un *sleep*, por lo que el nginx va a encolar las llamadas a medida que le vayan llegando, pero no de una manera muy intensa.
- Para el **/heavy**, se intenta simular un escenario sincrónico independiente de la tecnología, y como el server tiene un solo core, entonces solo puede procesar un request por vez. Esto va a ocasionar que el nginx encole todas las requests que le vayan llegando, y solo podrá ir desencolando a medida que el servicio resuelva la request anterior.

Sección 1:

Escenarios

Exploratory Test Ping

Objetivo

Comparar las características básicas de las tecnologías con las que vamos a trabajar en el resto de los escenarios con el fin de obtener insights y entender la naturaleza de cada una. También, ver cómo se muestran las diferencias entre sincrónico / asincrónico en un endpoint común del estilo *ping*, el cual simplemente devuelve un mensaje cuando le llega una request.

Resultado esperado

A priori antes de ejecutar el escenario, no esperamos ningún resultado en particular. El objetivo final de este test es ver las diferencias de un mismo proceso implementado en tecnologías diferentes. Esperamos que se comporten de manera distinta, y el propósito es ver cuáles son esas diferencias a nivel de recursos, tiempo de respuesta y manejo de requests.

Ejecución de la prueba

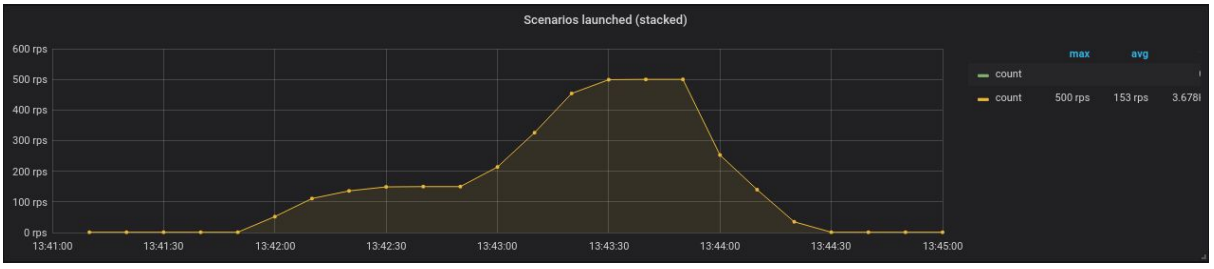
Escenario utilizado: `basic_scenario_ping.yaml`

Hacer un escenario que empiece con una cantidad fija de requests y hacer una rampa y después mantenerla por otro lapso de tiempo X (probar con el ping).

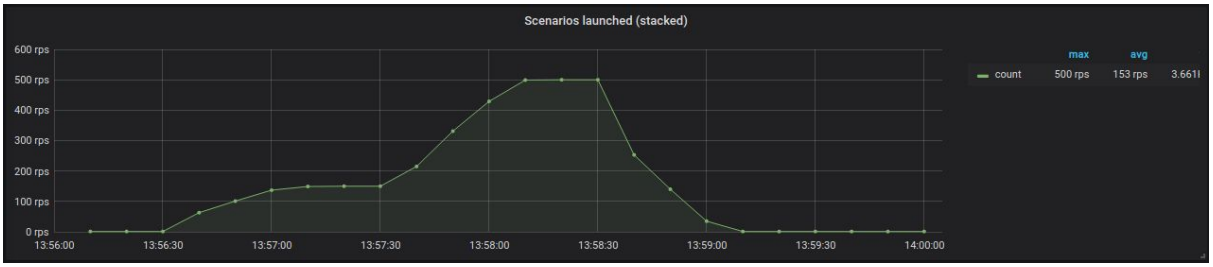
Este escenario es una prueba bastante simple, en la cual tiene:

1. una fase de warm-up mediante un ramp-up liviano,
2. una meseta liviana
3. otra ramp-up un poco más exigente
4. una meseta con la misma exigencia que el ramp-up anterior
5. finaliza con una ramp-down

RPS node ping:

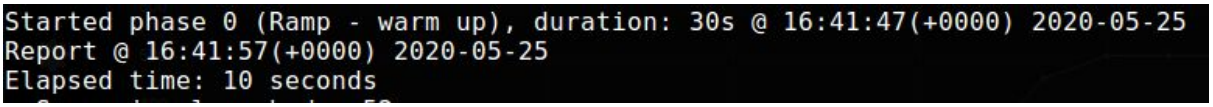


RPS gunicorn ping:



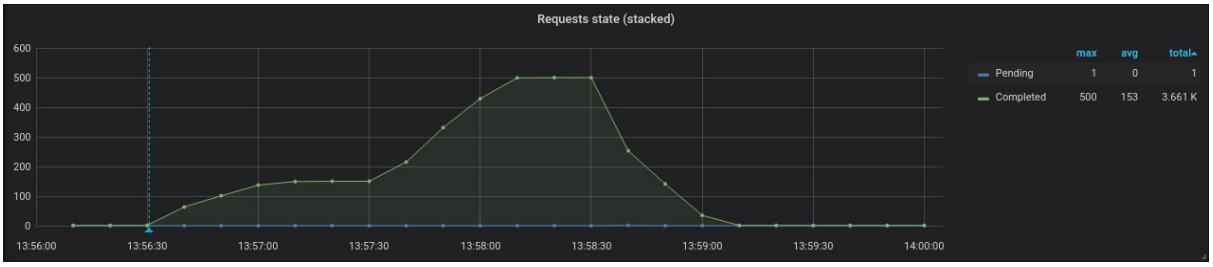
- Ver que pasa en los primeros **requests** (servidores lazy)

node ping:



La respuesta es en tiempo real a medida que llegan más rps.

gunicorn ping:



La respuesta es en tiempo real a medida que llegan más rps.

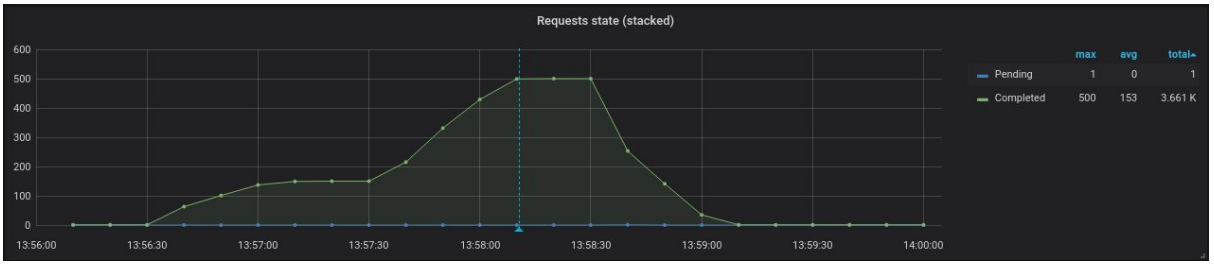
- Ver qué pasa con las **requests en general**: hay picos? (completed, pending, errors)?

node ping:


```
Started phase 3 (Plain - más violenta), duration: 30s @ 16:43:17(+0000) 2020-05-25
Report @ 16:43:27(+0000) 2020-05-25
Elapsed time: 1 minute, 40 seconds
```

No hay picos por encima de la cantidad de rps, ni tampoco se empiezan a encolar requests en pending.

gunicorn ping:

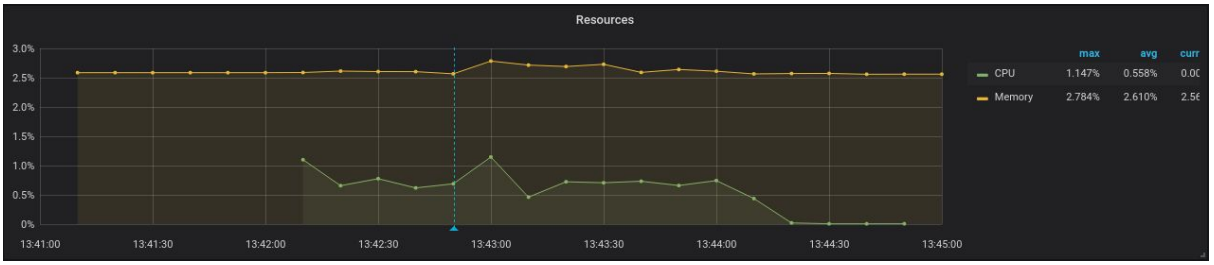



```
Started phase 3 (Plain - más violenta), duration: 30s @ 05:25:05(+0000) 2020-05-24
Report @ 05:25:15(+0000) 2020-05-24
Elapsed time: 1 minute, 40 seconds
```

No hay picos por encima de la cantidad de rps, pero aparece una sola request en pending.

- Ver que pasa en el **procesador** cuando empiezan a subir los requests por segundo.

node ping:




```
Started phase 2 (Ramp - más violenta), duration: 30s @ 16:42:47(+0000) 2020-05-25
Report @ 16:42:57(+0000) 2020-05-25
Elapsed time: 1 minute, 10 seconds
```

El procesador mantiene un consumo promedio del 0.5%, mostrando un leve pico llegando al 1% 10 segundos después de la rampa intensa.

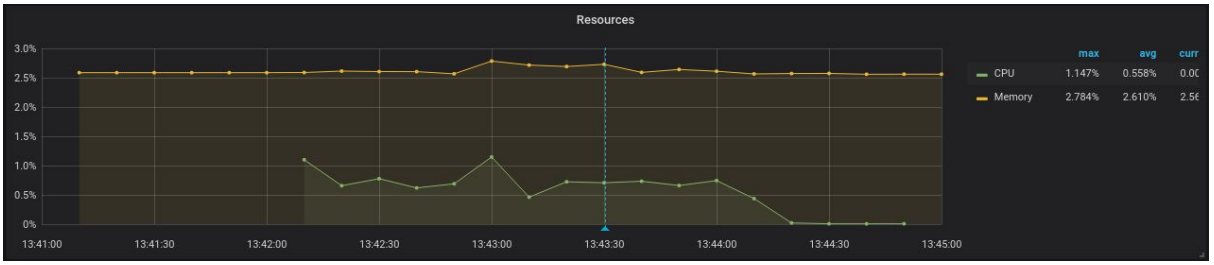
gunicorn ping:


```
Started phase 2 (Ramp - más violenta), duration: 30s @ 16:57:26(+0000) 2020-05-25
Report @ 16:57:36(+0000) 2020-05-25
Elapsed time: 1 minute, 10 seconds
```

El procesador mantiene un consumo promedio del 1%, mostrando un leve pico llegando al 2%, 10 segundos después de la rampa intensa.

- Ver que pasa en el **procesador** cuando se mantiene la corrida durante el tiempo

node ping:




```
Started phase 3 (Plain - más violenta), duration: 30s @ 16:43:17(+0000) 2020-05-25
Report @ 16:43:27(+0000) 2020-05-25
Elapsed time: 1 minute, 40 seconds
```

Luego en la meseta más intensa, el procesador sigue con el mismo ritmo, no mostrando alteraciones, ni replicando el pico que se vio posterior al inicio del Ramp-up.

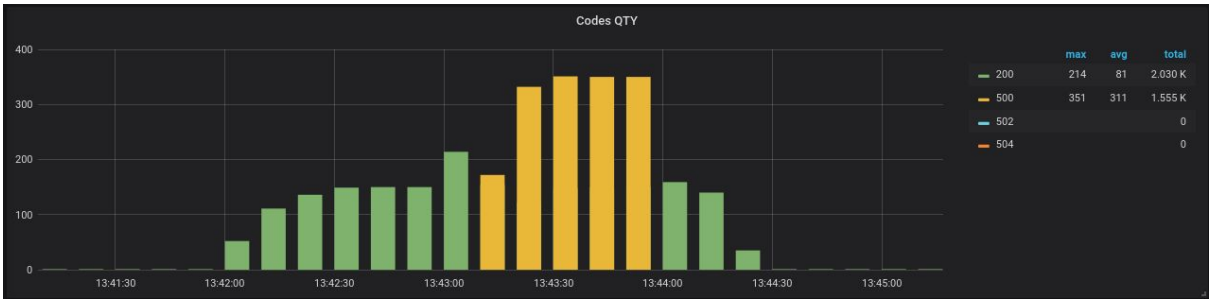
gunicorn ping:


```
Started phase 3 (Plain - más violenta), duration: 30s @ 16:57:56(+0000) 2020-05-25
Report @ 16:58:06(+0000) 2020-05-25
Elapsed time: 1 minute, 40 seconds
```

Luego en la meseta más intensa, el procesador sigue con el mismo ritmo, no mostrando alteraciones, ni replicando el pico que se vio posterior al inicio del Ramp-up.

- Ver qué responde el servidor en cada caso

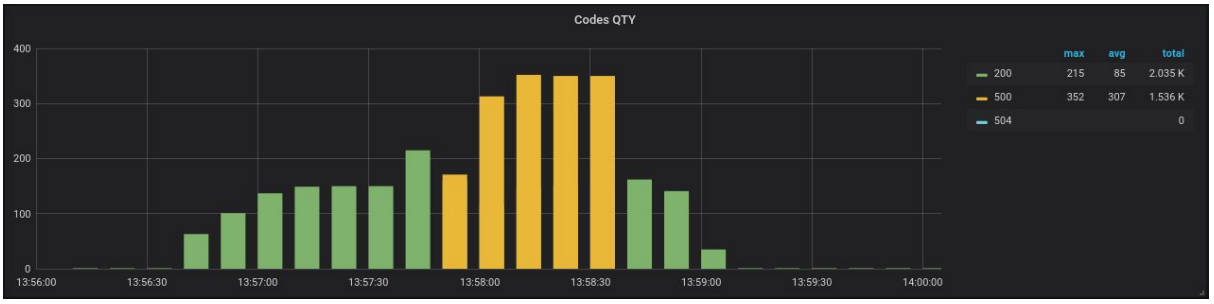
node ping:




```
Started phase 3 (Plain - más violenta), duration: 30s @ 16:43:17(+0000) 2020-05-25
Report @ 16:43:27(+0000) 2020-05-25
Elapsed time: 1 minute, 40 seconds
```

En la rampa más intensa, empiezan a devolver 500. El servidor llega a responder, pero no de la mejor manera.

gunicorn ping:

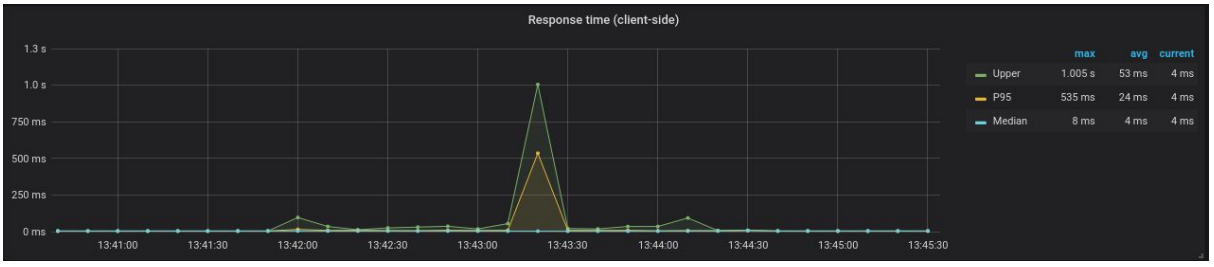



```
Started phase 3 (Plain - más violenta), duration: 30s @ 16:57:56(+0000) 2020-05-25
Report @ 16:58:06(+0000) 2020-05-25
Elapsed time: 1 minute, 40 seconds
```

En la rampa más intensa, empiezan a devolver 500. El servidor llega a responder, pero no de la mejor manera.

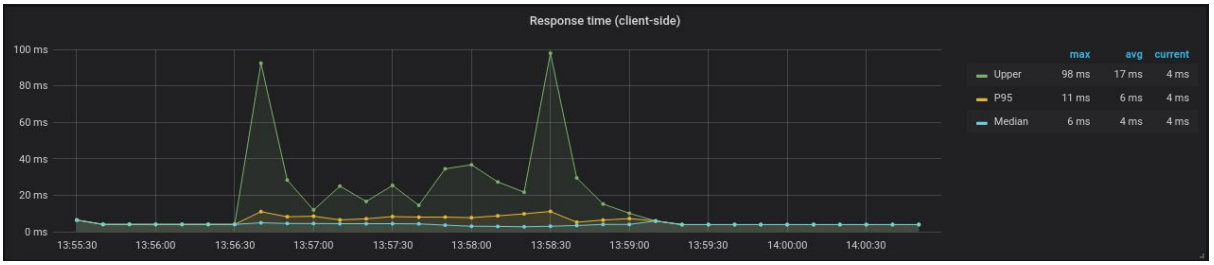
- Ver tiempo de respuesta

node ping:



El tiempo de respuesta es bastante constante, con un promedio de 25 ms. Se ve un pico de 1 segundo cuando empieza la meseta intensa y responde con 500.

gunicorn ping:



El tiempo de respuesta es más variado, pero presenta picos más bajos, ninguno llega al segundo, todos andan en el orden de 100 ms. Esto nos da un tiempo de respuesta promedio de 6 ms.

Conclusiones

En este caso del endpoint de ping, **node** consume **menos CPU** y no deja en pending ninguna request. En cambio, **gunicorn** consume en promedio **0,5% más de CPU** y deja en pending **por lo menos una request,** pero esto puede ser algo aislado de esta corrida en particular**.**

Por otro lado, el consumo promedio de memoria para el caso de **gunicorn** es un 0,5% más alto que el caso de **node**, llegando al 3% constante.

Finalmente, **gunicorn** otorga un tiempo de respuesta **más rápido** (6ms), mientras que **node** tiene un promedio de respuesta un poco más alto (25ms). Esto se debe a que gunicorn es un servidor sincrónico y node es asincrónico.

Exploratory Test Timeout

Objetivo

El objetivo es el mismo que en el *Exploratory Test Ping*, pero en este caso mirando el endpoint del estilo *timeout,* el cual implementa un ‘sleep’ de unos segundos cuando recibe una request.

Resultado esperado

El resultado esperado es el mismo que el test anterior, comparar el comportamiento de ambas tecnologías pero llamando a otro tipo de endpoint.

Ejecución de la prueba

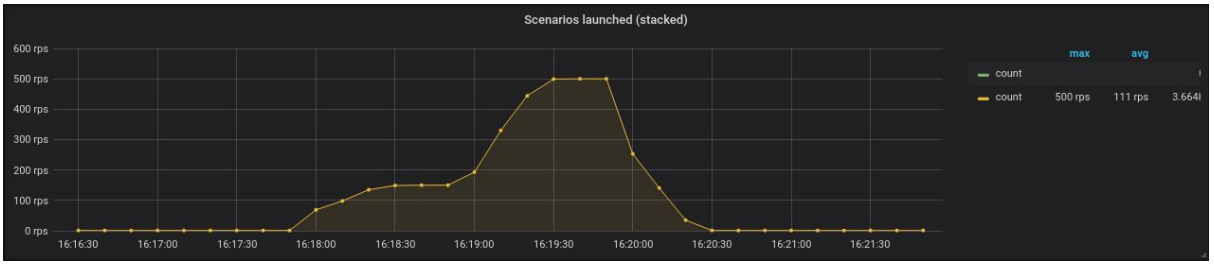
Escenario utilizado: `basic_scenario_timeout.yaml`

Hacer un escenario que empiece con una cantidad fija de requests y hacer una rampa y después mantenerla por otro lapso de tiempo X (probar con el timeout).

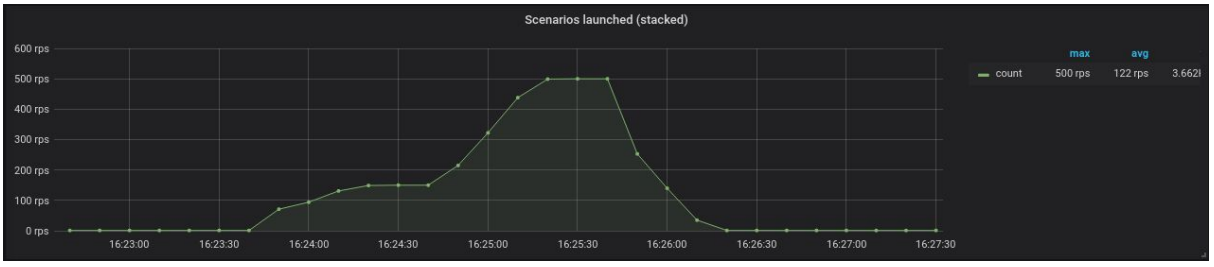
Este escenario es una prueba bastante simple, en la cual tiene:

1. una fase de warm-up mediante un ramp-up liviano,
2. una meseta liviana
3. otra ramp-up un poco más exigente
4. una meseta con la misma exigencia que el ramp-up anterior
5. finaliza con una ramp-down

node timeout: scenarios launched

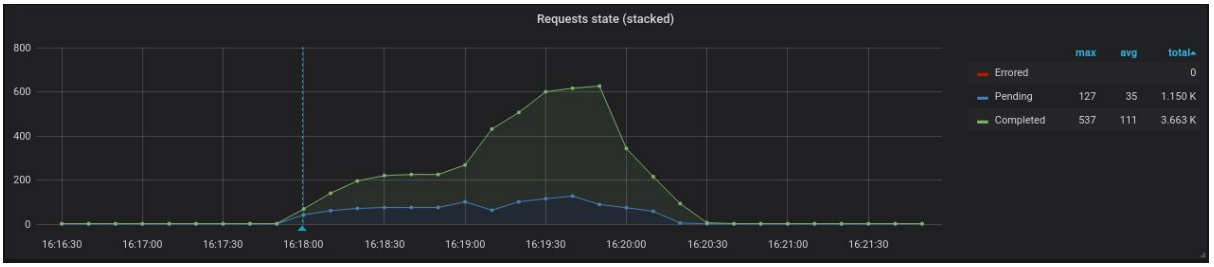


gunicorn timeout: scenarios launched



- Ver que pasa en los primeros **requests** (servidores lazy)

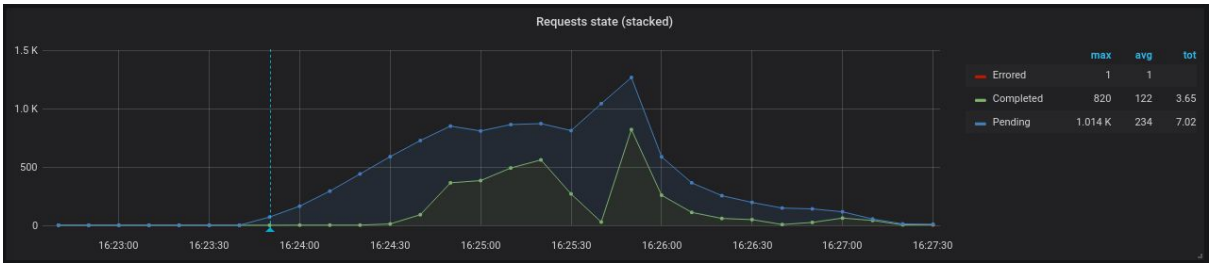
node timeout:




```
Started phase 0 (Ramp - warm up), duration: 30s @ 19:17:51(+0000) 2020-05-25
Report @ 19:18:01(+0000) 2020-05-25
Elapsed time: 10 seconds
```

Desde el principio ya empiezan a encolarse requests en pending, a causa del timeout, pero también hay algunos que se completan. A razón de 50% en pending, antes de llegar al pico de rps.

gunicorn timeout:




```
Started phase 0 (Ramp - warm up), duration: 30s @ 19:23:42(+0000) 2020-05-25
Report @ 19:23:52(+0000) 2020-05-25
Elapsed time: 10 seconds
```

Gunicorn timeout ya arranca con el 100% de requests en pending, típico de un servidor sincronico.

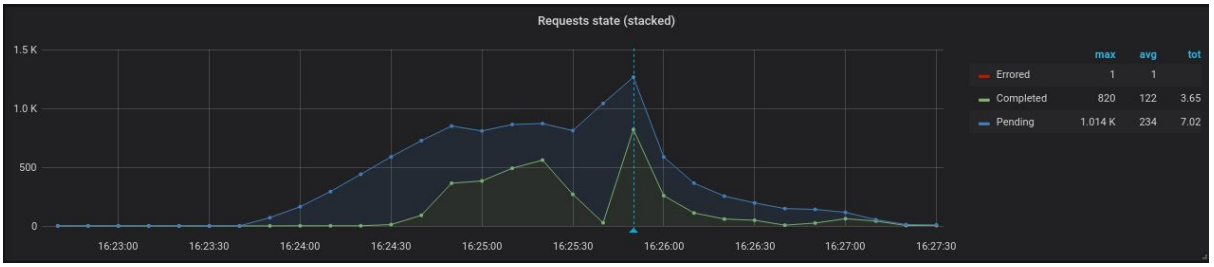
- Ver qué pasa con las **requests en general**: hay picos (completed, pending, errors)?

node timeout:


```
Started phase 2 (Ramp - más violenta), duration: 30s @ 19:18:51(+0000) 2020-05-25
Report @ 19:19:01(+0000) 2020-05-25
Elapsed time: 1 minute, 10 seconds
```

El pico de respuestas se relaciona con 60 segundos luego de empezado la rampa intensa.

gunicorn timeout:




```
Started phase 4, duration: 1.96s @ 19:25:43(+0000) 2020-05-25
Started phase 5, duration: 1.96s @ 19:25:45(+0000) 2020-05-25
Started phase 6, duration: 1.96s @ 19:25:47(+0000) 2020-05-25
```

Gunicorn timeout recién en el comienzo del ramp down empieza a desencolar requests.

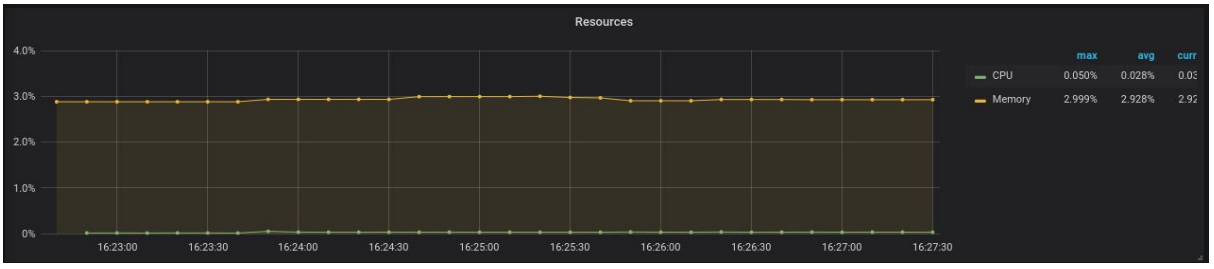
- Ver que pasa en el **procesador** cuando se mantiene la corrida durante el tiempo

node timeout:


```
Started phase 3 (Plain - más violenta), duration: 30s @ 19:19:22(+0000) 2020-05-25
Report @ 19:19:31(+0000) 2020-05-25
Elapsed time: 1 minute, 40 seconds
```

No hay picos en consumo excesivo de CPU, se mantiene muy estable a lo largo de toda la corrida

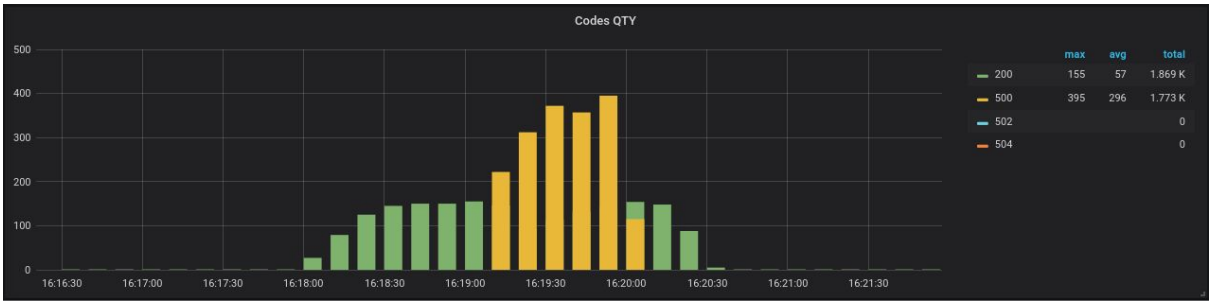
gunicorn timeout:

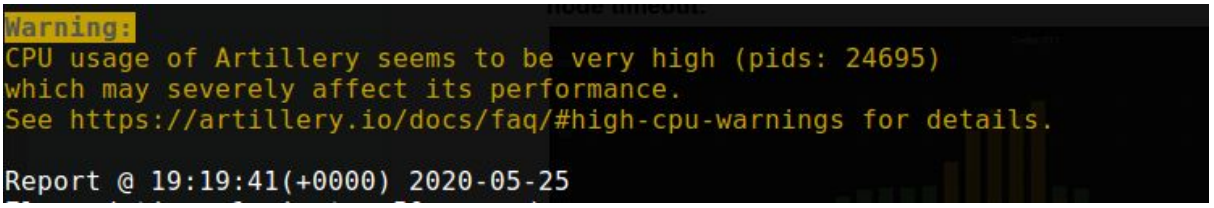


El consumo de CPU en gunicorn es nulo, no llegan a procesarse las requests.

- Ver qué responde el servidor en cada caso

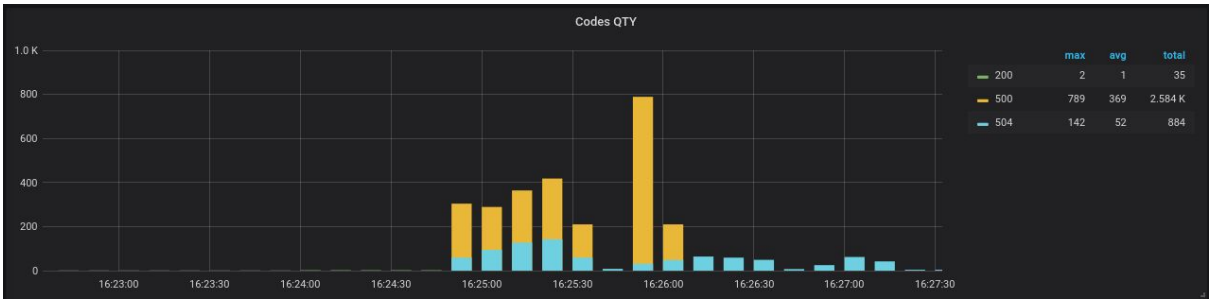
node timeout:





Al principio se responden satisfactoriamente las requests. Cuando empiezan a llegar más rps, el servidor logra responderlas, pero con codigo 500, es decir, no se llegan a procesar debido a la alta demanda. Esto se puede ver en el consumo del CPU, el cual baja de 1% a 0,5% en este intervalo de tiempo.

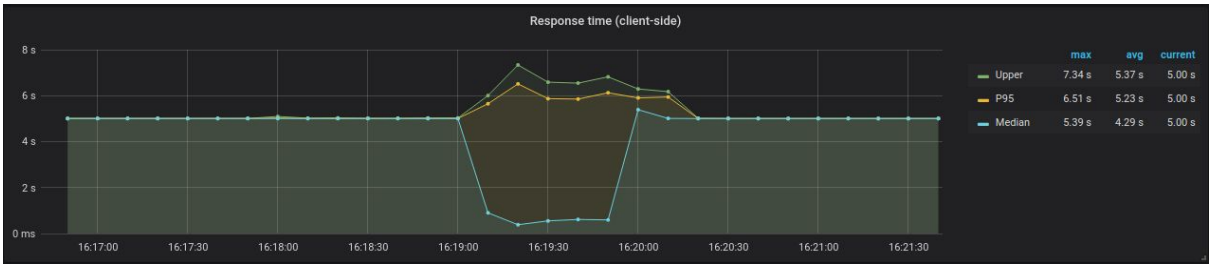
gunicorn timeout:



Solamente llegan a responderse 35 requests de manera satisfactoria. El resto (~ 3.000) devuelven 500 o 504.

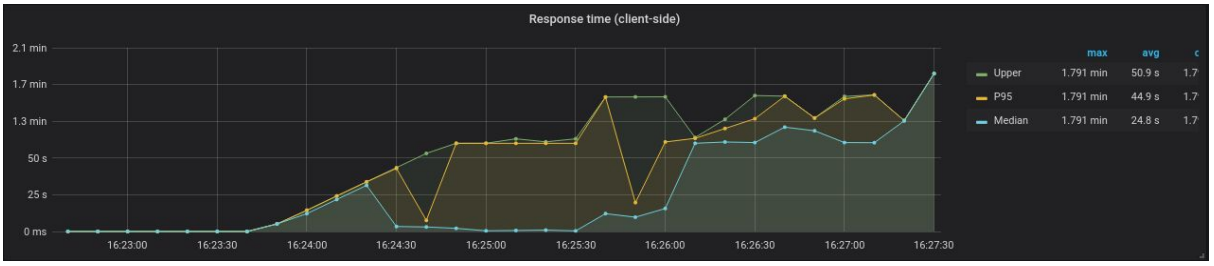
- Ver tiempo de respuesta

node timeout:



El tiempo de respuesta es bastante uniforme durante el tiempo que el servidor no se satura, respondiendo en un promedio de 5 segundos. Esta característica es típica de un servidor asincrónico. Luego, el tiempo de respuesta diverge un poco.

gunicorn timeout:



Podemos ver que el promedio del tiempo de respuesta de Gunicorn timeout es de 45 segundos, mostrando un comportamiento mucho más inestable, independientemente de la fase que se encuentre.

Conclusiones

En el caso del **timeout**, el único que arroja resultados positivos es **node**, ya que es un server asincrónico, lo cual nos da más dinamismo en el procesamiento. El *sleep* que hace el endpoint internamente, node detecta más rápido cuando el mismo terminó, debido al uso compartido del tiempo de procesamiento. En cambio, **gunicorn tiene que esperar a que termine el tiempo de *sleep* de una request**, para recién empezar a procesar la siguiente, razón por la cual vemos encolarse muchas requests desde el principio.

Además, por esta causa, **gunicorn** casi no responde ninguna request con código 200.

Endurance Test

Objetivo

Analizar cómo se comportan los endpoints dada una carga alta constante mantenida en el tiempo. Analizar mitigación de los problemas escalando.

Resultado esperado

Usando un único contenedor de node, se van a terminar encolando requests tarde o temprano, habiendo escalado debería reducirse la cantidad de requests encolados.

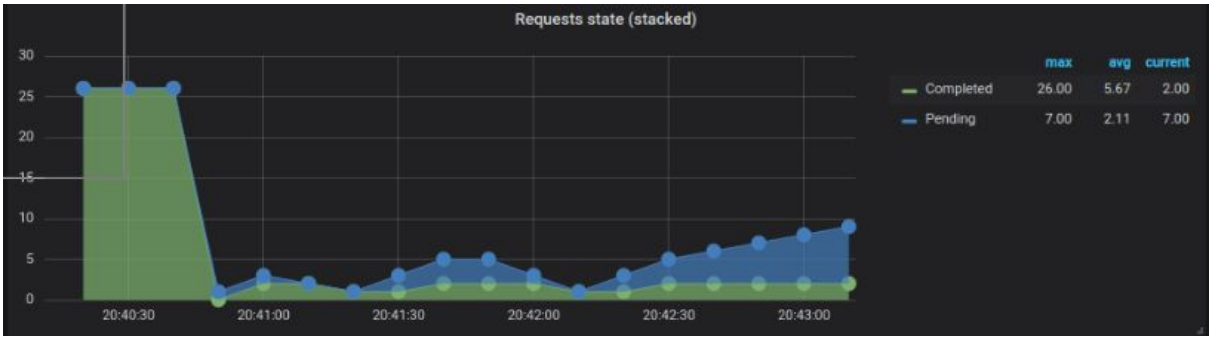
Ejecución de la prueba

Endpoint Heavy

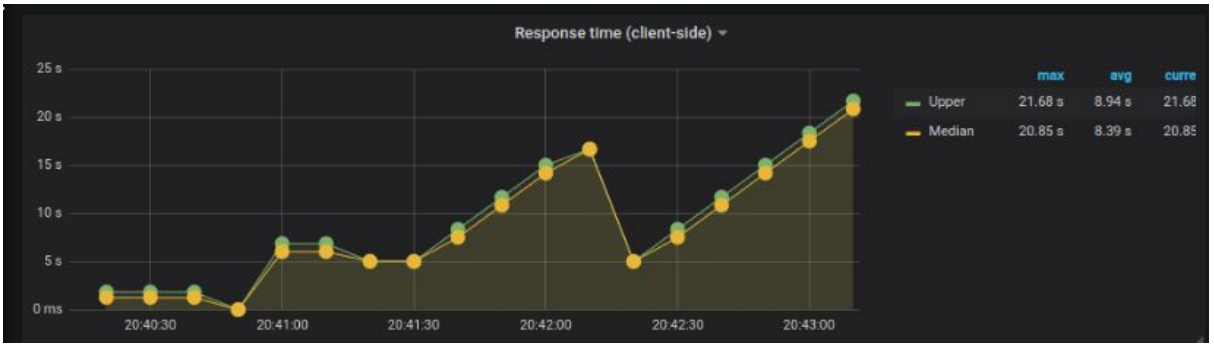
Se probó este endpoint con una carga constante de 0.3 rps, primero para el servidor **node** y luego para el servidor **node escalado** con 3 contenedores. La intención es ver si escalando, se logran superar los obstáculos que presenta el caso para node.

Node

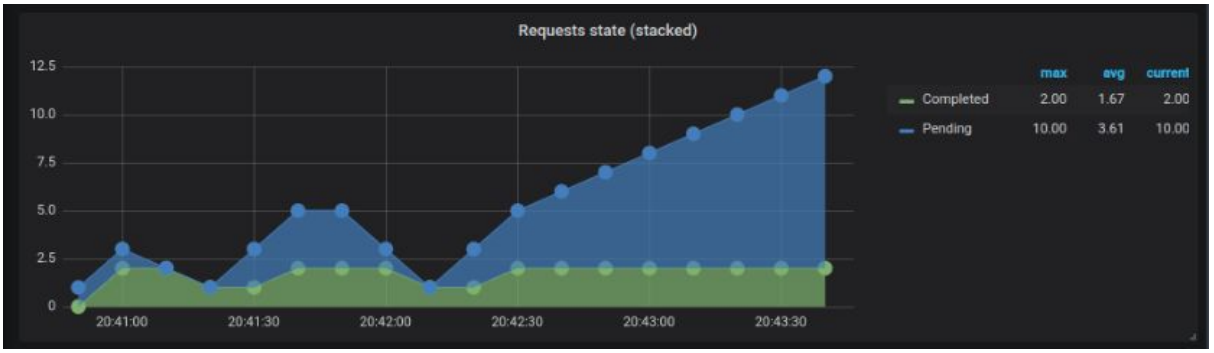
Desde el comienzo del test, se empiezan a encolar requests, lo que tiene sentido dado que cada una lleva un tiempo no trivial de procesamiento, pero se ve que cíclicamente el servidor puede hacer un catch-up de las requests y “ponerse al día”



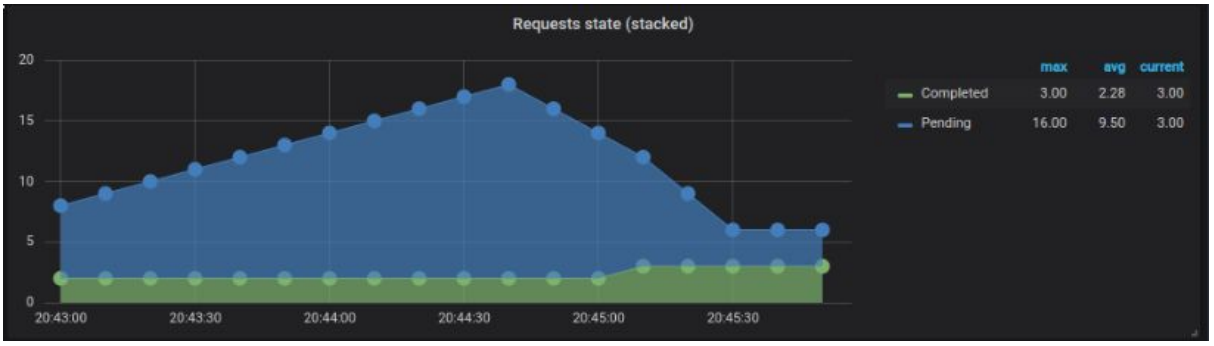
A los 2 minutos 40 de comenzado el test, ya la cantidad de encolados empieza a subir con más pendiente y el servidor no puede resolverlas.



Cosa que se condice con el response time, aparece un primer gran pico cuando se acumulan unas 5 requests, baja una vez que se logra responderlas pero una vez que empieza la curva ascendente final, el response time sigue subiendo a la par, dado que cada vez se consiguen menos responses.



La cantidad de encolados supera ampliamente las que el server puede responder



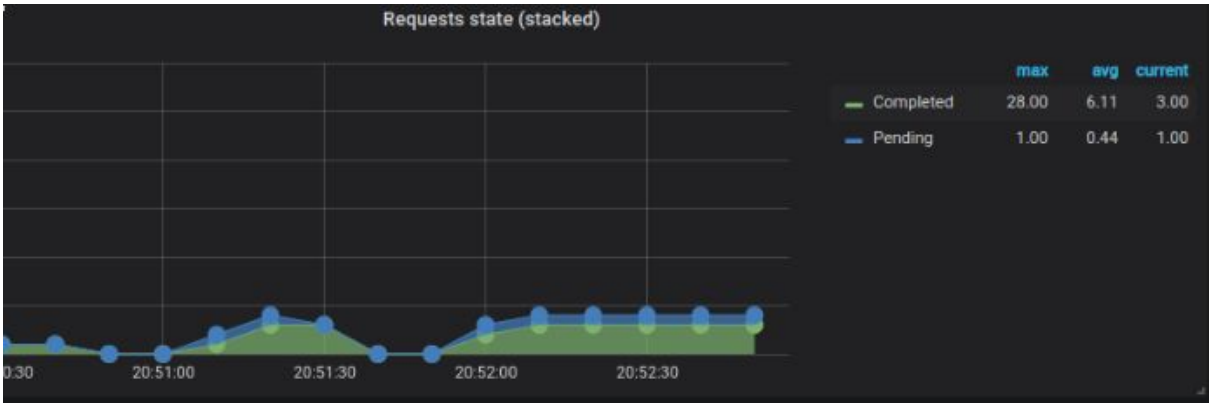
4 minutos después de comenzado el test, comienzan a desencolarse las requests, pero ya no devolviendo 200, sino 504

Se obtiene un 21% de 504 y p95 de 60008 msec al final del escenario.

Node Replicated

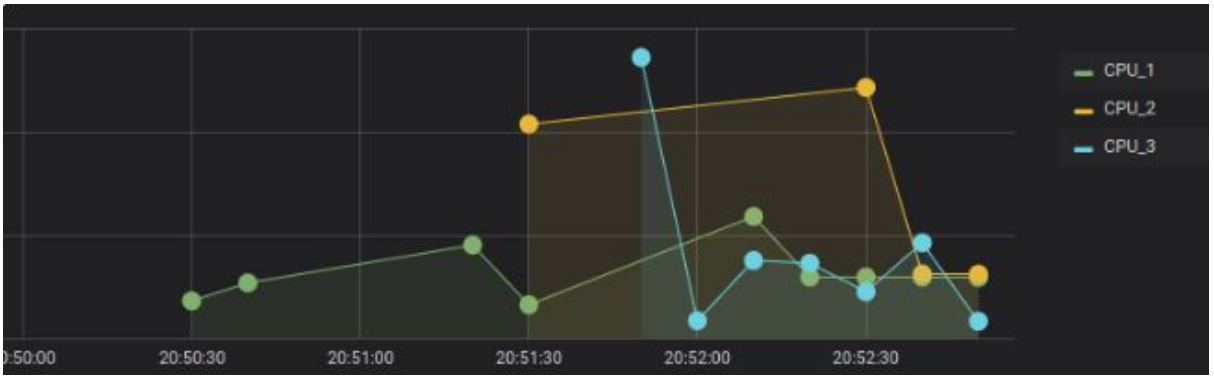
En este caso se obtiene una mucho mejor performance respecto a tiempos de respuesta y códigos de respuesta.

Pasado el primer minuto del escenario, se empiezan a encolar requests, ya acá tenemos la primera mejora ya que en el caso anterior, se encolaba desde el principio



Aparecen los mismos ciclos de encolado - desencolado, con la diferencia que la cantidad de encolados termina por mantenerse constante en el tiempo y nunca se dispara.

Lo mismo sucede con el tiempo de respuesta, se mantiene constante.



Lo que llama la atención, es la aparición tardía de dos de los contenedores de node, podemos ver que empiezan a procesar al minuto / minuto y medio de haber comenzado el escenario

Durante todo el experimento la carga se dividió de forma pareja entre los contenedores, dado el load balancer, y ninguno disparó su porcentaje de consumo de CPU

Habiendo escalado, se obtienen 0 códigos 5xx y un tiempo de respuesta p95: 5009

Conclusiones

En este caso, escalar horizontalmente el servicio rinde frutos para este escenario de carga alta mantenida.

Mientras que con un solo contenedor se puede dar buena respuesta durante unos minutos, rápidamente empiezan a aparecer errores. Con 3 contenedores de node, se logra que el encolado de requests no se dispare, que el tiempo de respuesta sea más de 10 veces más bajo y no se devuelve ningún timeout error.

Concurrent users Test

Objetivo

Determinar el punto de quiebre del servicio con respecto a cuántos usuarios concurrentes (request / Virtual Users) puede atender por un período de tiempo acotado antes de empezar a responder con algún tipo de error.

Resultado esperado

Se asume que el “cuello de botella” estará en el endpoint Heavy, por lo que se centran en él los esfuerzos de la prueba. Se espera ver que las implementaciones en Node y Python se comportan de manera similar, mientras que al escalar uno de los servicios la cantidad de requests que el endpoint puede aceptar antes de dar error se multiplican por la cantidad de copias del servicio.

Se espera además que el tiempo de respuesta aumente linealmente a medida que crece la cantidad de requests encolados.

Ejecución de la prueba

Se utilizó para esta prueba el escenario `concurrent_users.yaml`. Este escenario incrementa paulatinamente en cada fase la cantidad de requests, dejando una pausa entre fase y fase suficientemente grande como para garantizar que al iniciar una fase todos los requests de la anterior tuvieron tiempo de ser contestados. Es decir, en cada fase se empieza “de cero”.

Se puede observar en esta imagen, extraída como ejemplo de la corrida de Node sin escalar, que al momento de iniciar una fase, ya fueron contestados todos los requests (Pending) de la anterior

Además de los gráficos de líneas, se utilizaron gauges para medir el máximo de requests procesados sin que hubiera ocurrido un error en el lapso de 10 segundos. Estos gauges se

hicieron aplicando la idea que conceptualmente se muestra en la siguiente función de Graphite:

```
multiplySeries(offset(scale(isNonNull(stats.gauges.$server.codes.504), -1), 1), stats.gauges.$server.rps.count)
```

Con esta función, se multiplica la cantidad de requests ejecutados por una variable que vale 0 si hubo un error o 1 en caso contrario. De esta manera, se obtiene la máxima cantidad de requests ejecutados sin error, ya que a partir de ese valor la serie empezará a valer 0.

NodeJS

Se pudo observar que la cantidad máxima de requests que el servicio puede procesar en 10 segundos antes de dar error es de entre 12 y 14 (es decir, 1.2 ~ 1.4 rps)

El tiempo de respuesta promedio en este momento de corte es de 60 segundos (ver conclusiones):

```
Report @ 04:16:00(+0000) 2020-05-28
Elapsed time: 7 minutes, 50 seconds
  Scenarios launched:  0
  Scenarios completed: 2
  Requests completed:  2
  Mean response/sec: NaN
  Response time (msec):
    min: 51014.1
    max: 55614.9
    median: 53314.5
    p95: 55614.9
    p99: 55614.9
  Codes:
    200: 2

Report @ 04:16:10(+0000) 2020-05-28
Elapsed time: 8 minutes, 0 seconds
  Scenarios launched:  0
  Scenarios completed: 13
  Requests completed:  13
  Mean response/sec: NaN
  Response time (msec):
    min: 59999.5
    max: 60009.4
    median: 60001.4
    p95: 60009.3
    p99: 60009.4
  Codes:
    504: 13
```

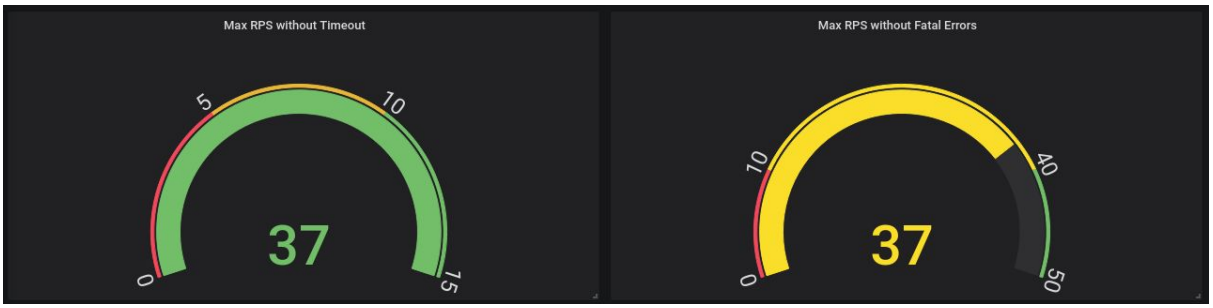
GUnicorn

```
Started phase 18 (Ph2.5), duration: 10s @ 04:43:01(+0000) 2020-05-28
Report @ 04:43:05(+0000) 2020-05-28
Elapsed time: 6 minutes, 50 seconds
  Scenarios launched:  10
  Scenarios completed: 9
  Requests completed:  9
  Mean response/sec: 2.56
  Response time (msec):
    min: 2.3
    max: 160.7
    median: 39.4
    p95: 160.7
    p99: 160.7
  Codes:
    404: 9
```

```
gunicorn_1  | [2020-05-27 15:35:06 +0000] [10] [ERROR] Socket error processing request.
gunicorn_1  | Traceback (most recent call last):
gunicorn_1  |   File "/usr/local/lib/python3.6/site-packages/gunicorn/workers/sync.py", line 133, in handle
gunicorn_1  |     req = next(parser)
gunicorn_1  |   File "/usr/local/lib/python3.6/site-packages/gunicorn/http/parser.py", line 41, in __next__
gunicorn_1  |     self.mesg = self.mesg_class(self.cfg, self.unreader, self.req_count)
gunicorn_1  |   File "/usr/local/lib/python3.6/site-packages/gunicorn/http/message.py", line 186, in __init__
gunicorn_1  |     super().__init__(cfg, unreader)
gunicorn_1  |   File "/usr/local/lib/python3.6/site-packages/gunicorn/http/message.py", line 53, in __init__
gunicorn_1  |     unused = self.parse(self.unreader)
gunicorn_1  |   File "/usr/local/lib/python3.6/site-packages/gunicorn/http/message.py", line 235, in parse
gunicorn_1  |     self.headers = self.parse_headers(data[:idx])
gunicorn_1  |   File "/usr/local/lib/python3.6/site-packages/gunicorn/http/message.py", line 73, in parse_headers
gunicorn_1  |     remote_addr = self.unreader.sock.getpeername()
gunicorn_1  | OSError: [Errno 107] Transport endpoint is not connected
```

Se observa que a partir del décimo request, el servicio se cae y empieza a responder con un HTTP 404

NodeJS Replicado



Con tres réplicas la cantidad de request procesados correctamente hasta que aparezca un error triplica aproximadamente el valor obtenido con una única réplica

```
Report @ 05:33:22(+0000) 2020-05-28
Elapsed time: 20 minutes, 12 seconds
  Scenarios launched:  0
  Scenarios completed: 1
  Requests completed:  1
  Mean response/sec: NaN
  Response time (msec):
    min: 117250.7
    max: 117250.7
    median: 117250.7
    p95: 117250.7
    p99: 117250.7
  Codes:
    200: 1
  Errors:
    ESOCKETTIMEDOUT: 2
```

Conclusiones

Para el caso de NodeJS, se obtuvieron resultados que no se tuvieron en cuenta al principio, pero luego de analizarlos terminan siendo evidentes.

Por un lado, los requests terminan mucho antes de lo previsto. La justificación encontrada es que, aunque haga uso intensivo de CPU, cada request usa el procesador por 5 segundos, que no es lo mismo que usar 5 segundos de procesador. La diferencia es muy sutil: debido a la implementación que se hizo del endpoint, en la que se toma registro de un timestamp y se ejecuta un bucle hasta que pasan 5 segundos de ese momento, lo que termina ocurriendo es que la suma de los tiempos de ejecución de cada instrucción termina siendo mucho menor a esos 5 segundos. Los requests comparten procesador según el scheduler de Node, que parece ser bastante equitativo.

A lo anterior se le suma que el cuello de botella termina siendo el timeout configurado en el balanceador de carga, que es de un minuto. Luego de ese tiempo, el request deja de ser considerado “pendiente” y es respondido con un HTTP 504.

Se podría haber extendido este tiempo de timeout para encontrar el verdadero punto de quiebre de la implementación en Node, pero resulta más criterioso considerar que a partir de un tiempo de respuesta de 1 minuto, desde el punto de vista del cliente que está esperando una respuesta, puede dejar de considerarse que el servicio está funcionando correctamente.

En el caso de la implementación con Python, dado que se está utilizando un único worker que procesa los requests de forma sincrónica, era de esperarse que no resistiera demasiados requests. Una implementación en Producción requeriría varios workers haciendo un mejor uso de los múltiples cores de los procesadores modernos.

Por último, lo que sí sucedió como se predijo en los resultados esperados es que aumentar la cantidad de réplicas del servicio de Node multiplica por esa cantidad los requests que se pueden atender sin problemas.

Spike Test

Objetivo

Evaluar cómo la capacidad de recuperación del servicio ante picos consecutivos de alta carga con el servicio de Timeout

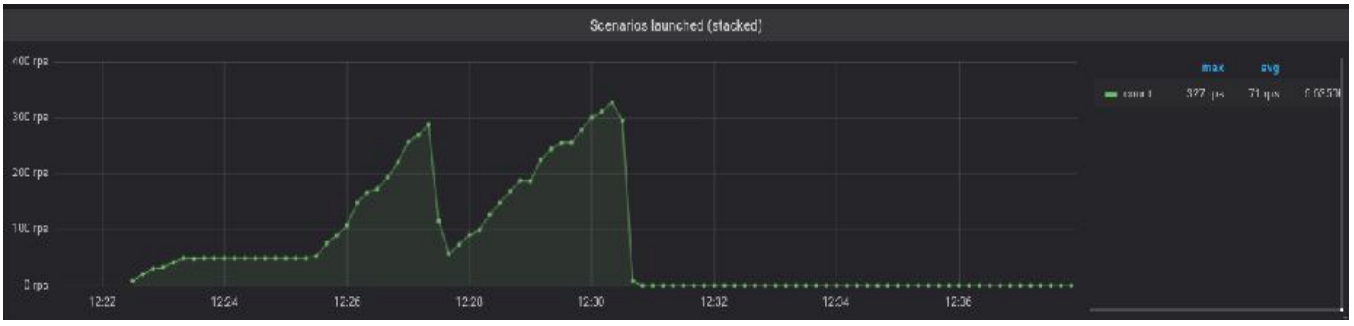
Resultado esperado

Que se vayan completando los requests luego de cada pico y el sistema vuelva a estar estable, también analizar si el escalado horizontal mejora el desempeño.

Ejecución de la prueba

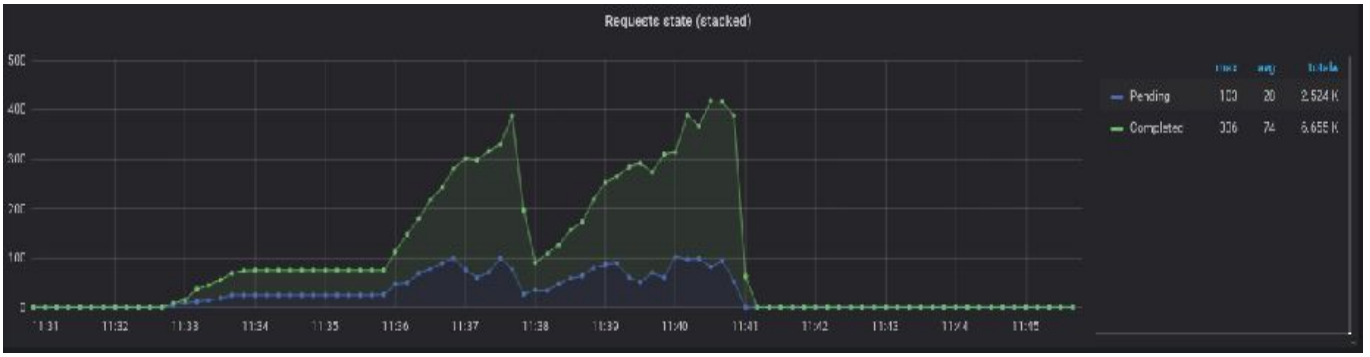
Se realizó un escenario con el artillery donde las fases son:

- Un warmup con un pequeño RampUp de 1 minuto
- Una meseta que permita acomodar los recursos utilizados a medida que se resuelven las llamadas
- Una alta carga de 2 minutos, buscando llegar al punto donde el servidor apenas puede aguantarla
- Una caída brusca de requests
- Otro pico de alta carga de 2 minutos
- WarmDown



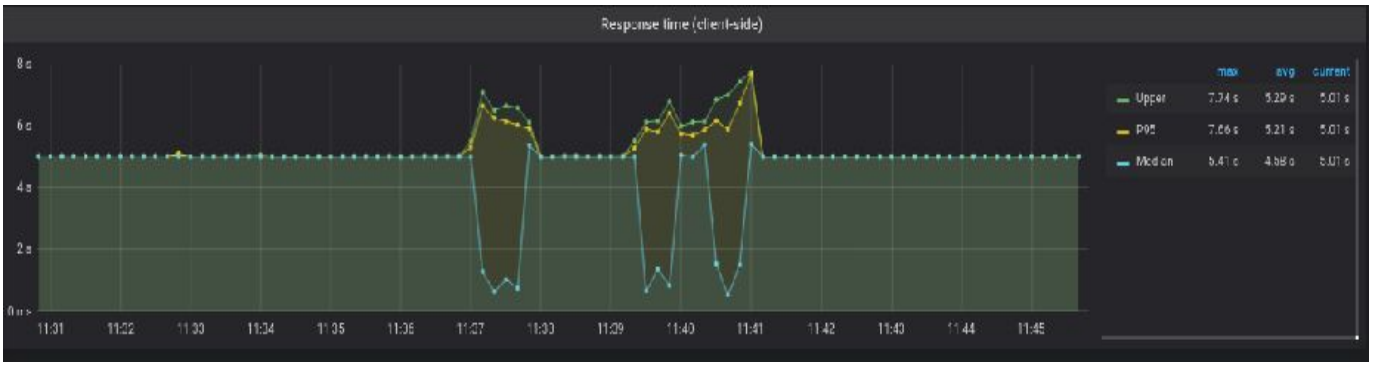
Escenario corrido con las fases descritas más arriba, este escenario se corrió para el node, gunicorn y el node_replicated

node_timeout



Dada la naturaleza asincrónica de JS y del servicio de timeout, se puede ver que los requests realizados desde el artillery se van procesando a medida que la event_queue va completando la llamada

node_timeout



se pueden ver como los picos en el servicio, alteran el comportamiento del tiempo de respuesta que perciben los clientes

node_timeout

Dado que el servicio de timeout no es un proceso que requiere mucho procesamiento el máximo que se alcanzó fue menos al 1.25%

node_timeout

Como se lo sometió a un escenario de alta carga muy cerca del punto de quiebre del servidor, este comienza a devolver errores, dificultando la posibilidad de recuperarse después de un escenario un pico de carga.

Se repitió el mismo análisis para el caso de **gunicorn** sin escalar, con los siguientes resultados:

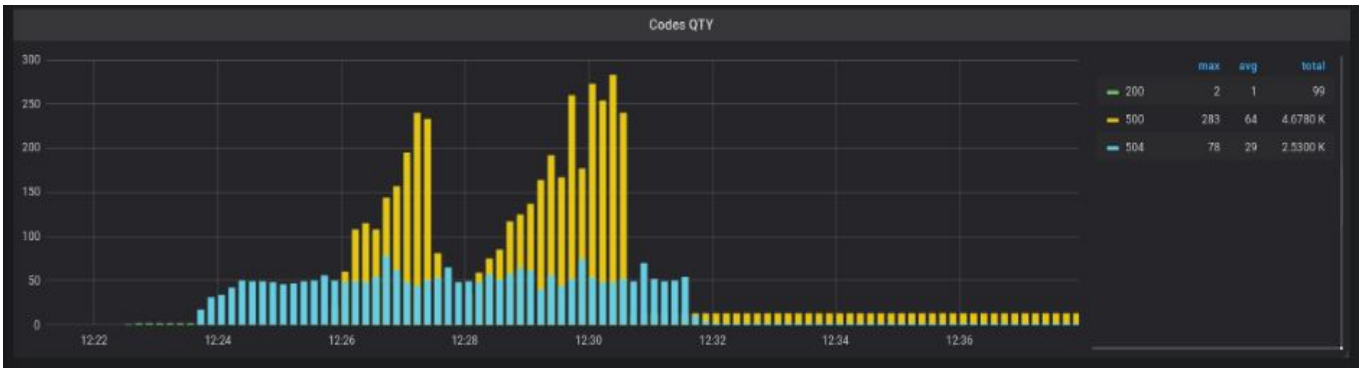
gunicorn_timeout

Dada la naturaleza sincrónica de python, el servicio empieza a encolar requests, logrando una throughput mucho más bajo que el logrado con node para el mismo escenario.

gunicorn_timeout

El tiempo de respuesta se modifica mucho dado el bajo througput que tiene el servicio en gunincorn

gunicorn_timeout



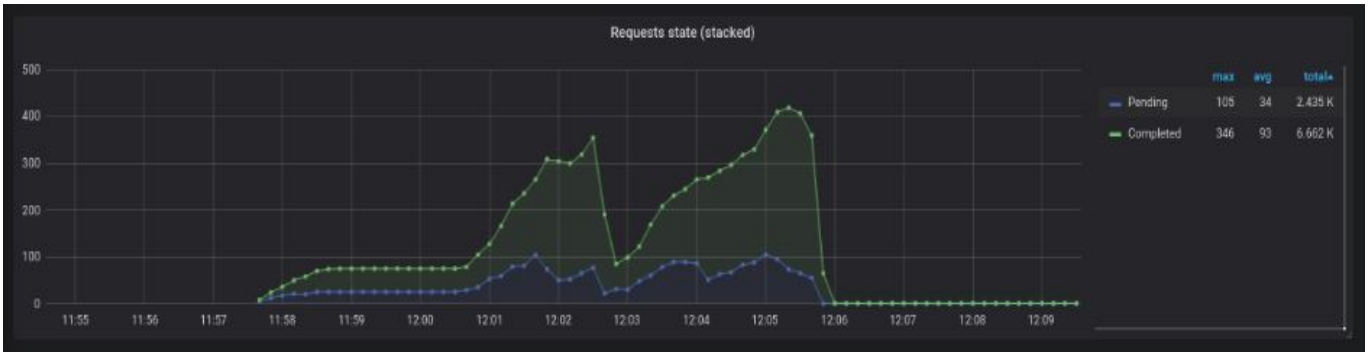
Como se acumulan muchos requests y dada la capacidad del servidor de poder procesar pocas requests por segundo, el servicio comienza a tirar múltiples errores incluyendo timeout

gunicorn_timeout

Nuevamente, como se había marcado anteriormente Python, consume más memoria que node y mucha menos % de CPU

Finalmente, para analizar cómo performa un servicio escalado con node con 3 contenedores, para analizar cómo afecta escalar horizontalmente el servicio.

node_replicated timeout



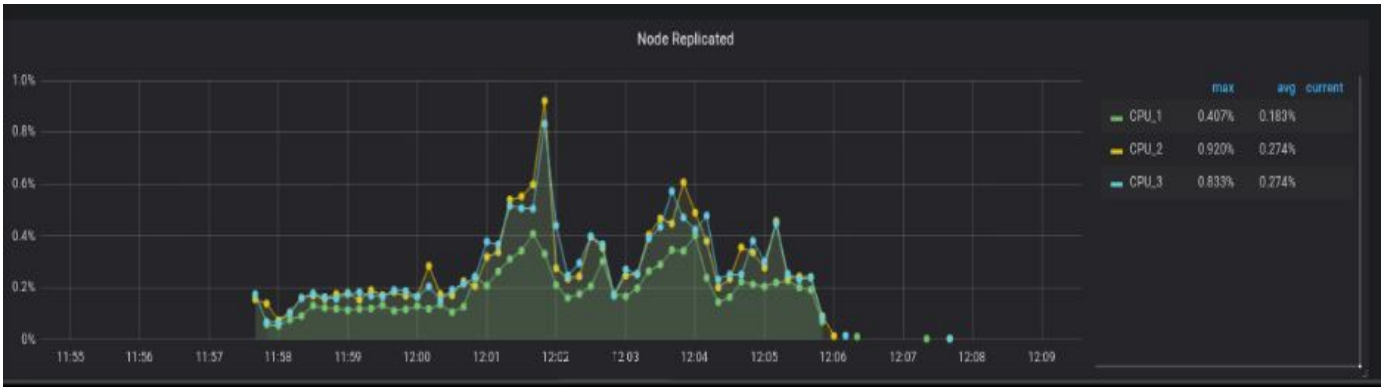
node_replicated timeout

Escalar el servidor implicó en una reducción muy alta en los tiempos de respuesta que percibió cada usuario virtual, pasando de casi 1 minuto y medio para el caso de una sola instancia, a medias que no superaron los 5 segundos

node_replicated timeout

El servicio escaldo presento una forma más uniforme, aunque se pueden ver los picos, sin embargo se reduce la cantidad de errores que devuelve el servidor.

node_replicated timeout



Escalar el servicio horizontalmente, permitió una mayor distribución de uso de los procesadores, cuyo pico fue menor a 1% (comparado al 1.25% del sin escalar)

Conclusiones

Decidimos realizar la prueba de picos, con el fin de poder analizar la capacidad que podría tener el servicio de recuperarse después de un escenario donde muchos usuarios deciden consultar recursos. Nos llamó la atención cómo la influencia de la tecnología puede impactar tanto en el desempeño, donde pudimos ver que el servicio en **node** encolaba menos requests y devolvía muchos menos errores que su equivalente en **python**. Sin embargo, node consume mucho más procesador, suponemos que en un escenario donde el recurso a consultar resulta computacionalmente costoso, Node podría performar mucho peor que Python.

Concluimos que para este tipo de servicios es bueno poder utilizar un **load balancer**, pues los beneficios que monitoreamos con el mismo servicio escalándolo horizontalmente, rendía mejor ante escenarios reales como picos de carga en un servicio.

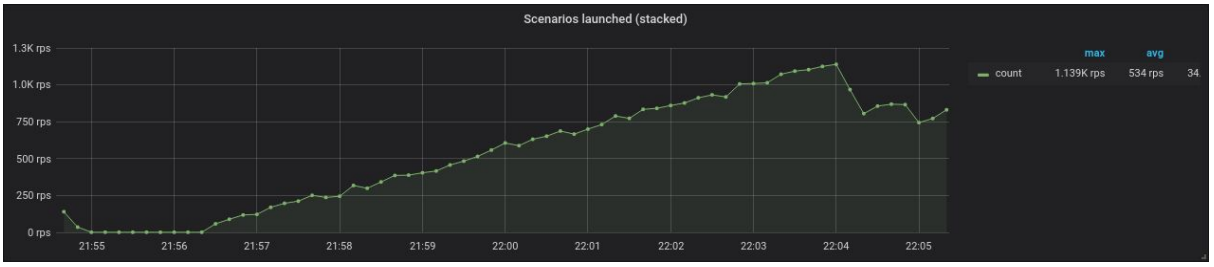
Sección 2:

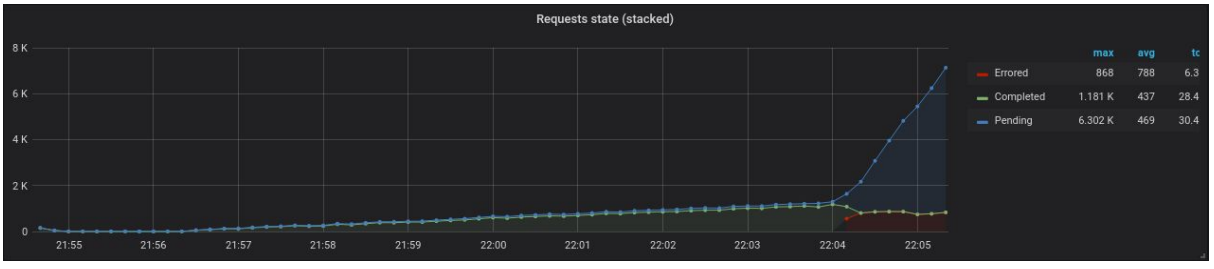
Caracterización de 2 Web services provistos por la cátedra, con propiedades desconocidas. El objetivo de esta etapa es caracterizar a los mismos, indicando cuál es sincrónico y cuál asincrónico, cantidad de workers en el caso del sincrónico y tiempo de respuesta de cada uno.

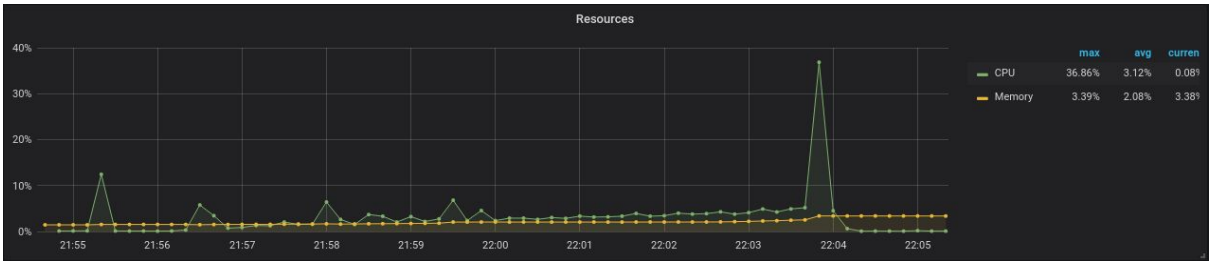
Server 1: http://localhost:9090/

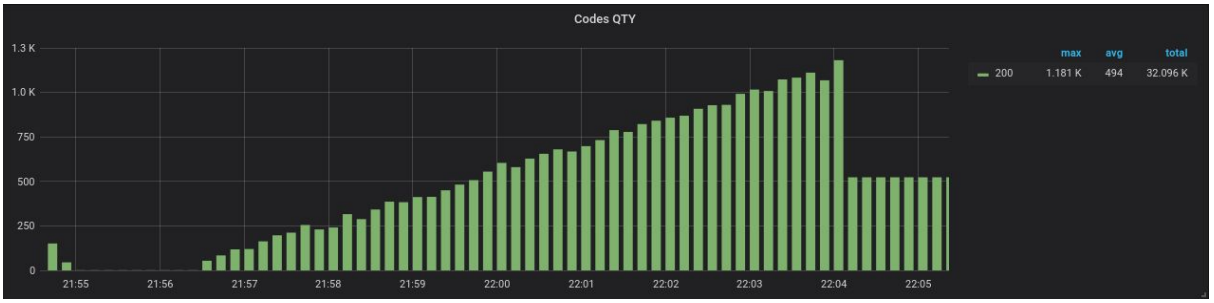
El escenario que se le pasa con Artillery es bastante simple, una rampa creciente que vaya desde 5rps hasta 150 rps al cabo de 10 minutos. De esta manera vamos a ver el comportamiento del web service en distintos momentos.

```
phases:
  -
    name: Ramp up - única
    duration: 600
    arrivalRate: 5
    rampTo: 150
  -
    name: Ramp down - inico
    arrivalRate: 28
    duration: 1.96
  -
    arrivalRate: 26
    duration: 1.96
```









Podemos ver que a medida que aumentan los rps, el tiempo de respuesta se mantiene constante. Esta es una característica típica de los servidores **asincrónicos**, ya que las requests comparten el tiempo de procesamiento, y no se van resolviendo por orden de llegada. Además, mirando la cantidad de respuestas con código 200, durante el tiempo que las rps no saturan al server, las mismas van creciendo al mismo ritmo que el throughput.

El tiempo de respuesta tiene una media de aproximadamente 950 ms. Esto lo podemos ver de 2 maneras: primero en el chart de *Response Time* desde Grafana, aprovechando la información de la leyenda; y sino también desde los logs de Artillery, el cual cada 10 segundos muestra un reporte con algunos stats.

Con respecto al consumo de recursos tanto CPU o Memoria, ninguno tuvo picos mientras el server no se saturó.

Finalmente, luego de más de 1000 escenarios ejecutados en 10 segundos, el server empezó a saturarse, empezando a encolar las requests, generando un pico del consumo de CPU (casi 40%) y dejando más de 700 requests en estado pending. La siguiente es una captura del momento antes y después de la saturación.

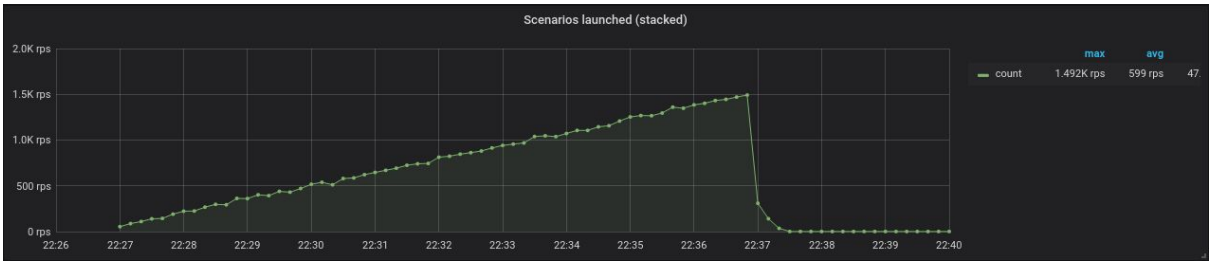
```
Report @ 22:04:00(-0300) 2020-05-24
Elapsed time: 7 minutes, 40 seconds
  Scenarios launched:  1139
  Scenarios completed: 1181
  Requests completed:  1181
  Mean response/sec: 114.01
  Response time (msec):
    min: 950.7
    max: 1956.6
    median: 951.1
    p95: 954.3
    p99: 1950.5
  Codes:
    200: 1181

Report @ 22:04:10(-0300) 2020-05-24
Elapsed time: 7 minutes, 50 seconds
  Scenarios launched:  968
  Scenarios completed: 523
  Requests completed:  523
  Mean response/sec: 96.7
  Response time (msec):
    min: 950.7
    max: 976.5
    median: 951.3
    p95: 968.9
    p99: 972.2
  Codes:
    200: 523
  Errors:
    EADDRNOTAVAIL: 558

Warning:
CPU usage of Artillery seems to be very high (pids: 9552)
```

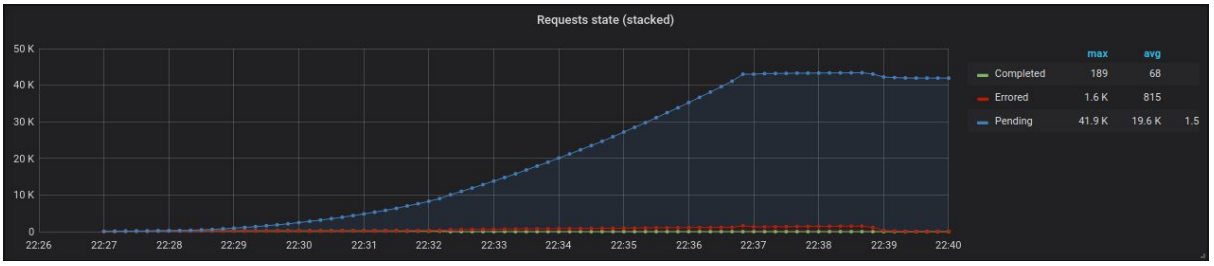
Server	Tipo	Tiempo de respuesta	Events loop
server1 (9090)	Asincrónico	~ 950 ms	-

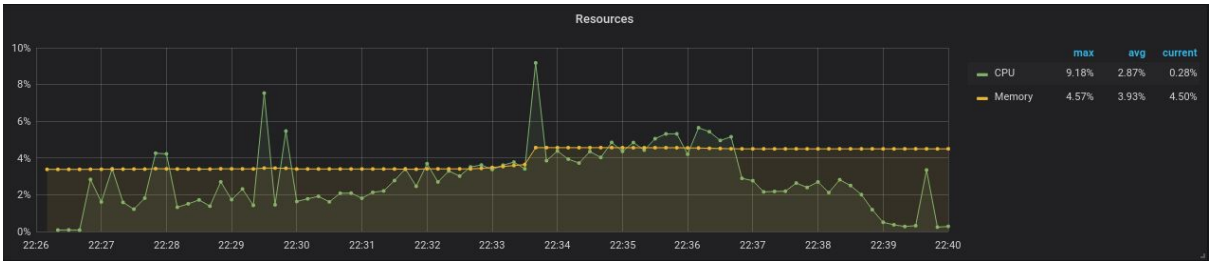
Server 2: http://localhost:9091/

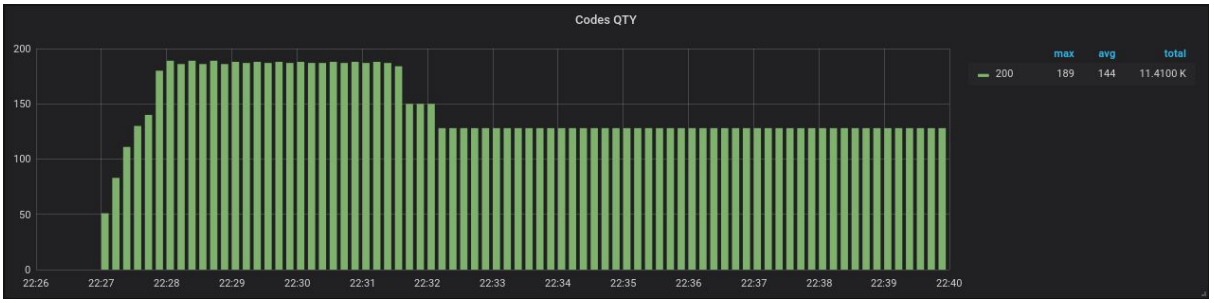
El escenario que se le pasa con Artillery es el mismo que para el Server 1: una rampa creciente que vaya desde 5rps hasta 150 rps al cabo de 10 minutos. De esta manera vamos a ver el comportamiento del web service en distintos momentos, y ver las diferencias entre este server y el server 1.

```
phases:
  -
    name: Ramp up - única
    duration: 600
    arrivalRate: 5
    rampTo: 150
  -
    name: Ramp down - inico
    arrivalRate: 28
    duration: 1.96
  -
    arrivalRate: 26
    duration: 1.96
```





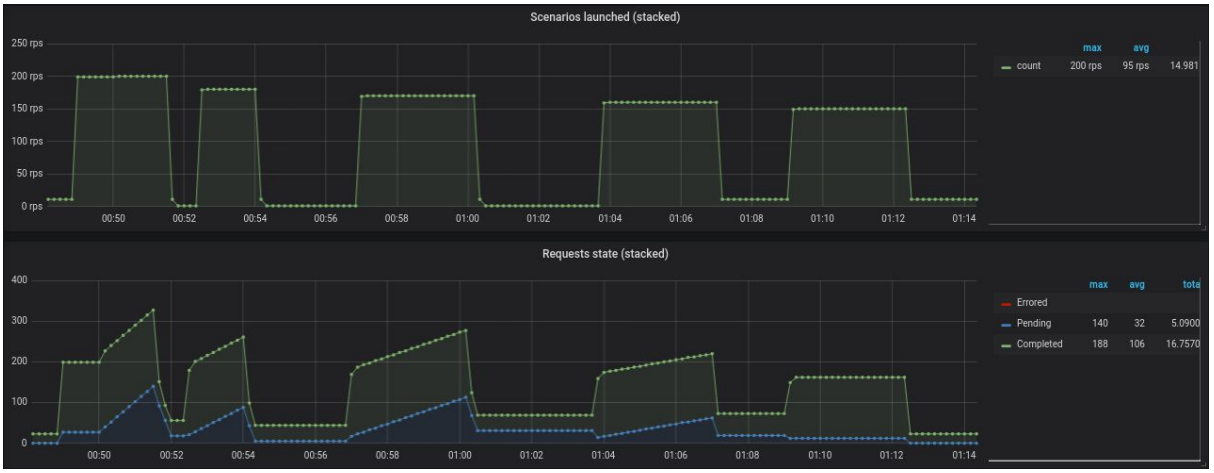




En este caso podemos ver que el comportamiento es bastante distinto al del Server 1. En este caso, podemos ver que a medida que aumentan las requests, también aumenta el tiempo de respuesta, y también aumenta la cantidad de requests pendientes.

En este caso estamos frente a un servidor **sincrónico** por lo mencionado en el párrafo anterior, y además hay un punto de quiebre en el cual la cantidad de rps satura a la cantidad de workers del servidor. En la siguiente imagen, marcamos el instante anterior a la saturación:

El instante anterior a la saturación es de 140 rps cada 10 seg, es decir, 14 requests por segundo. Vamos a ir probando con valores cercanos a este número para ver cuál es la saturación real.



Como podemos ver en el gráfico de arriba, probamos con diferentes rps, hasta que encontramos uno que **no aumente la cantidad de requests pendientes**. La cantidad de rps límite para que este servidor no se sature es 15 rps.

Entonces corrimos un escenario más extenso de tiempo con este valor, para evaluar su comportamiento y obtener una mejor representación del tiempo de respuesta promedio:

Con 15 rps también el tiempo de respuesta se mantiene constante, en un promedio de ~800 ms, teniendo un percentil p95 con un promedio de 860 ms. Con lo cual, con ambos datos podemos estimar la cantidad de workers disponibles en este server (tomaremos el p95 para estar un poco más tranquilos probabilísticamente):

$$\text{cantidad de workers} = \frac{\#\ rps}{tiempo\ de\ respuesta} = \frac{15\ req.\ por\ segundo}{0{,}86\ segundos} = 17{,}44$$

Entonces podemos decir que el este servidor sincrónico tiene al menos 17 workers.

Server	Tipo	Tiempo de respuesta	Workers
server2 (9091)	Sincrónico	~ 800 ms	17

Conclusiones generales del TP:

Como conclusiones globales de todo el TP podemos decir varias cosas:

- Aprendimos muchas tecnologías y herramientas nuevas para la mayoría de nosotros. Si bien alguna vez pudimos haber usado Docker o Nginx para algo, herramientas como cAdvisor, Graphite, Grafana y Artillery no las habíamos usado anteriormente. Además de entender el uso de cada una por separado, es importante haberlas utilizado juntas: caso de cAdvisor, Graphite y Grafana bastante acopladas, y Artillery para generar carga simulando Usuarios Virtuales.
- También nos ayudó a entender más los conceptos de sincrónico y asincrónico, y que no solamente quede como 'característica' de una tecnología, sino ponerlas en uso y ver el impacto de cada una dependiendo el escenario.
- En la mayoría de los casos vimos que **NodeJS** aporta mejores resultados, justamente por su comportamiento asincrónico y la capacidad para procesar varias request a la vez, compartiendo el tiempo de procesamiento. A su vez, esto no quiere decir que **Gunicorn** sea malo por ser sincrónico. Lo importante es saber las fortalezas de cada uno y tener en claro cuándo conviene usarlos dependiendo nuestra necesidad: si necesitamos rapidez y podemos controlar el acceso al servicio, **Gunicorn** seria buena alternativa; pero si necesitamos alto throughput en paralelo para soportar demanda muy variable, **Node** nos serviría más.
- Estas diferencias no solo las vimos en el throughput de cada tecnología, sino que también pudimos observar cómo difieren ambas tecnologías en lo que concierne a uso de recursos, donde Python consume mas memoria RAM pero mucho menos CPU que su equivalente en Node. Esto nos permite pensar que si queremos realizar un servicio que necesite de bastante capacidad de procesamiento, Python podría considerarse como mejor alternativa sobre NodeJS.
- También a veces con poner un servidor asincrónico con Node no es suficiente a causa de la alta demanda. Entonces aprendimos lo fácil y rápido que es escalar un servicio con Docker, creando múltiples réplicas y dejando que Nginx maneje la carga a través de un **load balancer**.
- Por otro lado, con todas las herramientas aprendidas, pudimos caracterizar un servicio web desde afuera, sin tener pistas de cómo está implementado internamente. Esto es muy poderoso de lograr, evaluando tiempos de respuestas y cómo responde ante distintos niveles de carga.
- Por último, pudimos ver lo importante que es conocer los atributos de calidad vistos en la teoría, como pueden ser la disponibilidad, performance, tiempo de respuesta, etc.