



# Trabajo Práctico N° 1

Arquitectura de Software  
2do Cuatrimestre 2020

|              |   |
|--------------|---|
| Integrantes: | Daneri, Alejandro - 97839<br>Huang, Yuhong - 102146<br>Klein, Santiago - 102192 |
|--------------|---|

# Índice

|  |           |
|--|-----------|
| <b>1. Sección 1</b>  | <b>2</b>  |
| 1.1. Introducción . . . . .                                  | 2         |
| 1.2. Endpoint: Ping . . . . .                                | 2         |
| 1.3. Endpoint: Sleep . . . . .                               | 5         |
| 1.3.1. Load test . . . . .                                   | 5         |
| 1.3.1.1. Gunicorn . . . . .                                  | 5         |
| 1.3.1.2. Node . . . . .                                      | 8         |
| 1.3.1.3. Node replicado . . . . .                            | 9         |
| 1.3.1.4. Conclusión . . . . .                                | 11        |
| 1.3.2. Spike test . . . . .                                  | 11        |
| 1.3.2.1. Gunicorn . . . . .                                  | 11        |
| 1.3.2.2. Node . . . . .                                      | 13        |
| 1.3.2.3. Node replicado . . . . .                            | 15        |
| 1.3.2.4. Conclusión . . . . .                                | 17        |
| 1.3.3. Endurance test . . . . .                              | 17        |
| 1.3.3.1. Gunicorn . . . . .                                  | 17        |
| 1.3.3.2. Node . . . . .                                      | 19        |
| 1.3.3.3. Node replicado . . . . .                            | 20        |
| 1.3.3.4. Conclusión . . . . .                                | 21        |
| 1.4. Endpoint: Heavy . . . . .                               | 22        |
| 1.4.1. Load test . . . . .                                   | 22        |
| 1.4.1.1. Gunicorn . . . . .                                  | 22        |
| 1.4.1.2. Node . . . . .                                      | 24        |
| 1.4.1.3. Node replicado . . . . .                            | 26        |
| 1.4.1.4. Conclusión . . . . .                                | 28        |
| 1.4.2. Spike test . . . . .                                  | 29        |
| 1.4.2.1. Gunicorn . . . . .                                  | 29        |
| 1.4.2.2. Node . . . . .                                      | 31        |
| 1.4.2.3. Node replicado . . . . .                            | 33        |
| 1.4.2.4. Conclusión . . . . .                                | 34        |
| 1.4.3. Endurance test . . . . .                              | 35        |
| 1.4.3.1. Gunicorn . . . . .                                  | 35        |
| 1.4.3.2. Node . . . . .                                      | 37        |
| 1.4.3.3. Node replicado . . . . .                            | 39        |
| 1.4.3.4. Conclusión . . . . .                                | 41        |
| <b>2. Sección 2</b>  | <b>42</b> |
| 2.1. Sincrónico / Asincrónico . . . . .                      | 42        |
| 2.1.1. Primer servicio . . . . .                             | 43        |
| 2.1.2. Segundo servicio . . . . .                            | 44        |
| 2.1.3. Conclusión . . . . .                                  | 44        |
| 2.2. Cantidad de workers en el servicio sincrónico . . . . . | 45        |
| 2.3. Demora en responder . . . . .                           | 46        |
| <b>3. Sección 3 [OPC]</b>                                    | <b>47</b> |

## 1. Sección 1

### 1.1. Introducción

En esta sección, se utilizarán los siguientes endpoints:

| Endpoint | Funcionalidad   |
|----------|---|
| Root     | Respuesta de un valor constante (rápido y de procesamiento mínimo)    |
| Heavy    | Loop de <b>3 segundos</b> (lento y de alto procesamiento)             |
| Sleep    | Duerme <b>3 segundos</b> y responde (lento y de procesamiento mínimo) |

Cada endpoint se implementará en 3 servidores, que constan de los siguientes servicios:

- *Node* (x1)
- *Gunicorn* (x1)
- *Node replicado* (x3)

Dichos servidores se someterán a distintos escenarios, con el objetivo de realizar 3 tipos de tests:

- *Load Test*: analizar el comportamiento del sistema ante una cantidad de carga (requests) creciente, hasta llegar a un límite en el que se producen fallas y el sistema deja de responder.
- *Spike Test*: analizar el comportamiento del sistema ante altos volúmenes de carga, repentinos, en períodos de tiempo cortos.
- *Endurance Test*: analizar el comportamiento del sistema ante una carga constante, en un período de tiempo largo.

### 1.2. Endpoint: Ping

Para realizar un análisis exploratorio, se sometió a este endpoint a una prueba de carga, para observar las cotas inferiores de los distintos valores. El escenario consiste en aumentar de 10 en 10, cada 10 segundos, la cantidad de requests por segundo (rps en adelante), en un intervalo de 0 a 300 rps. En los siguientes gráficos se observan resultados similares, variando únicamente la utilización de los recursos:



Figura 1: Scenarios launched (stacked) & Requests state (stacked) - Load Test - Gunicorn



Figura 2: Response time (client-side) &amp; Resources usage - Load Test - Gunicorn

```
All virtual users finished
Summary report - Gunicorn
Scenarios launched: 45214
Scenarios completed: 45214
Requests completed: 45214
Mean response/sec: 139.98
Response time (msec):
  min: 1.2
  max: 180.1
  median: 2.3
  p95: 11.5
  p99: 30
Scenario counts:
  Root (/): 45214 (100 %)
Codes:
  200: 45214
```



Figura 3: Scenarios launched (stacked) &amp; Requests state (stacked) - Load Test - Node



Figura 4: Response time (client-side) &amp; Resources usage - Load Test - Node

```
All virtual users finished
Summary report - Node
Scenarios launched: 45247
Scenarios completed: 45247
Requests completed: 45247
Mean response/sec: 140.58
Response time (msec):
  min: 0.8
  max: 71.4
  median: 1.8
  p95: 9.6
  p99: 26.5
Scenario counts:
  Root (/): 45247 (100 %)
Codes:
  200: 45247
```

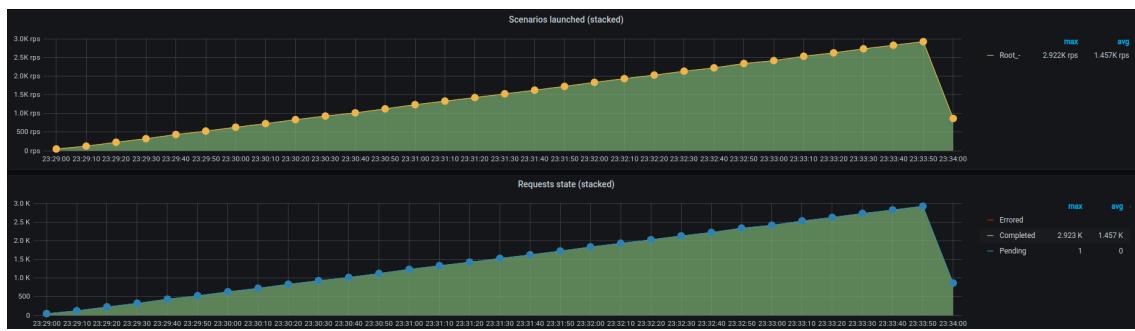


Figura 5: Scenarios launched (stacked) &amp; Requests state (stacked) - Load Test - Node Replicated



Figura 6: Response time (client-side) &amp; Resources usage - Load Test - Node Replicated

```
All virtual users finished
Summary report - Node Replicated
Scenarios launched: 45173
Scenarios completed: 45173
Requests completed: 45173
Mean response/sec: 140.56
Response time (msec):
    min: 0.9
    max: 41.6
    median: 1.8
    p95: 7.5
    p99: 11.7
Scenario counts:
    Root (/): 45173 (100 %)
Codes:
    200: 45173
```

Se puede observar que la mediana del tiempo de respuesta del cliente es menor en *Node* y *Node Replicado* que en *Gunicorn*. El consumo de CPU, por su parte, es significativamente mayor en *Gunicorn*. No obstante, el consumo de memoria de *Node* y *Node Replicado* es mayor al de *Gunicorn*.

A partir de este análisis, se puede deducir que *Node* es más eficiente y performante que *Gunicorn* en cuanto al uso de CPU, pero no lo es en el consumo de memoria.

### 1.3. Endpoint: Sleep

#### 1.3.1. Load test

El escenario *Load Test* consiste en ir incrementando cada 10 segundos, de 10 en 10 y en forma de *rampa*, la cantidad de requests por segundo. Así, comenzando en 0 req/s, se llega a 200 req/s en un lapso de 200 segundos. Para los tres servidores se testeó con este escenario.

##### 1.3.1.1 Gunicorn

Se obtuvieron los siguientes resultados:

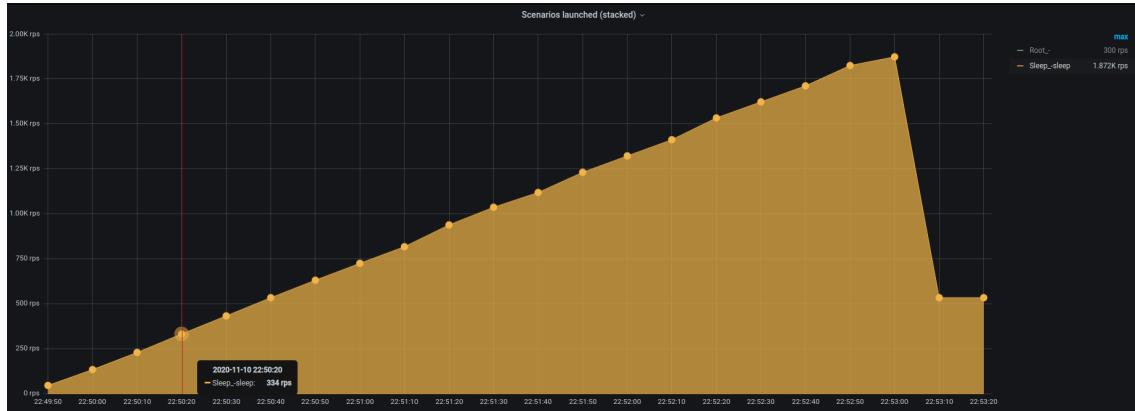


Figura 7: Scenarios launched (stacked) - Load Test - Gunicorn

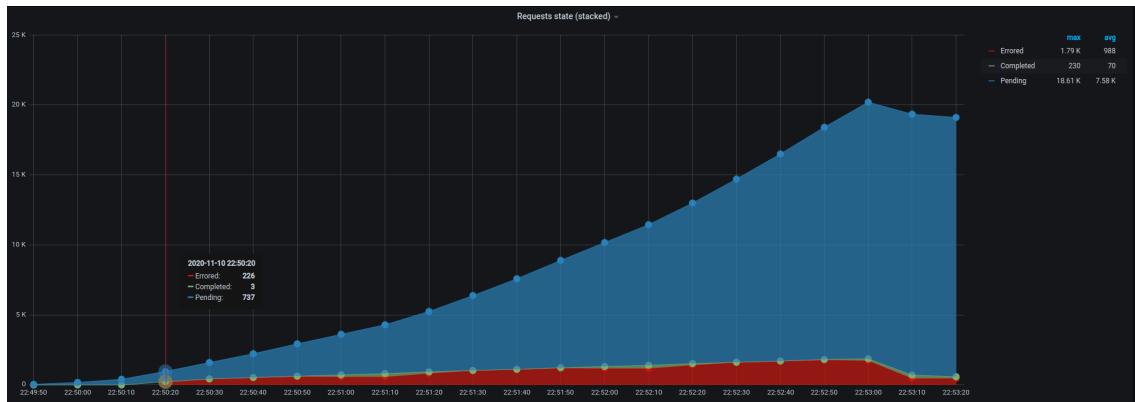


Figura 8: Requests state (stacked) - Load Test - Gunicorn



Figura 9: Resources usage - Load Test - Gunicorn

**Summary report**

Scenarios launched: 20051

Scenarios completed: 1769

Requests completed: 1769

```
Mean response/sec: 76.48
Response time (msec):
  min: 3013.1
  max: 61039.1
  median: 60002.8
  p95: 60008.4
  p99: 61008.8
Scenario counts:
  Sleep (/sleep): 20051 (100%)
Codes:
  200: 21
  504: 1748
Errors:
  ECONNRESET: 18282
```

Considerando que la función *sleep* es **bloqueante** al utilizar esta tecnología, el servidor recibirá un request y lo procesará, bloqueando así el hilo de ejecución. Por tanto, como el único hilo de ejecución está bloqueado, en la llamada de *sleep*, los requests que llegan en ese intervalo de bloqueo se encolarán para ser procesados luego, de manera sincrónica y secuencial. Por este motivo, se ve afectado muy negativamente el tiempo de respuesta percibido por el usuario.

Teniendo en cuenta que por cada request el hilo principal duerme 3 segundos, eventualmente, al ir aumentando paulatinamente la carga, el servidor colapsará, puesto que cada vez llegarán más requests y no podrá responderlos.

Esto se ve claramente reflejado en el gráfico, pronunciándose un desfasaje entre la cantidad de requests encoladas y las efectivamente respondidas.

El servidor, al principio, puede responder a lo sumo 3 requests cada 10 segundos, por lo que el resto se va encolando, y así la curva azul de *pending requests* crece muy rápidamente.

Aproximadamente, a los 40 segundos, se llega a una **carga máxima de 40 req/s**, en la que el servidor comienza a lanzar errores, procesando unas pocas requests nada más. Al seguir aumentando la carga, el servidor finalmente deja de estar disponible, a los 70 segundos, con una carga de 70 req/s aproximadamente.

A fin de cuentas, el endpoint responde satisfactoriamente 21 requests, y lanza errores en 18282 requests. El tiempo de respuesta es pobre, con una mediana de 1 minuto para una request que debería tardar 3 segundos si el servidor estuviera plenamente funcional, sin carga alguna.

El consumo de recursos es bajo, debido a la naturaleza del endpoint.

En cuanto a los atributos de calidad se ve afectada la disponibilidad críticamente luego de sobrepasar los 70 req/s.

### 1.3.1.2 Node

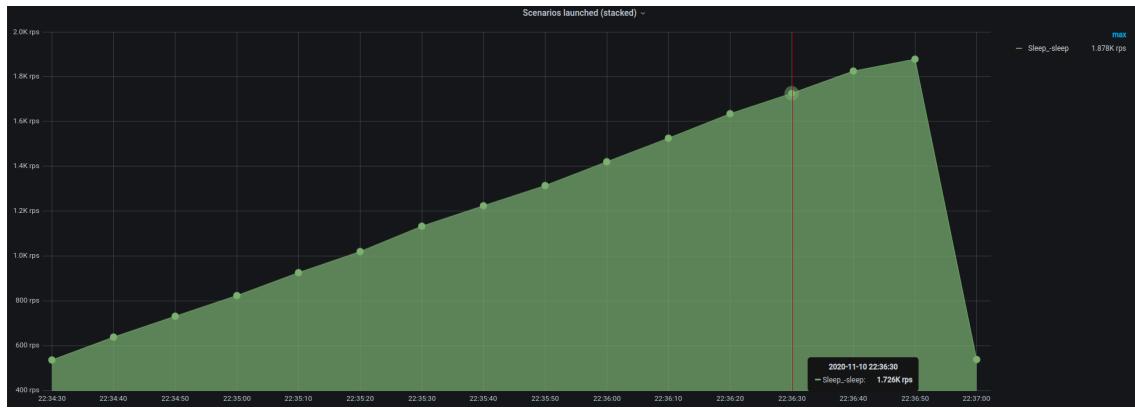


Figura 10: Scenarios launched (stacked) - Load Test - Node

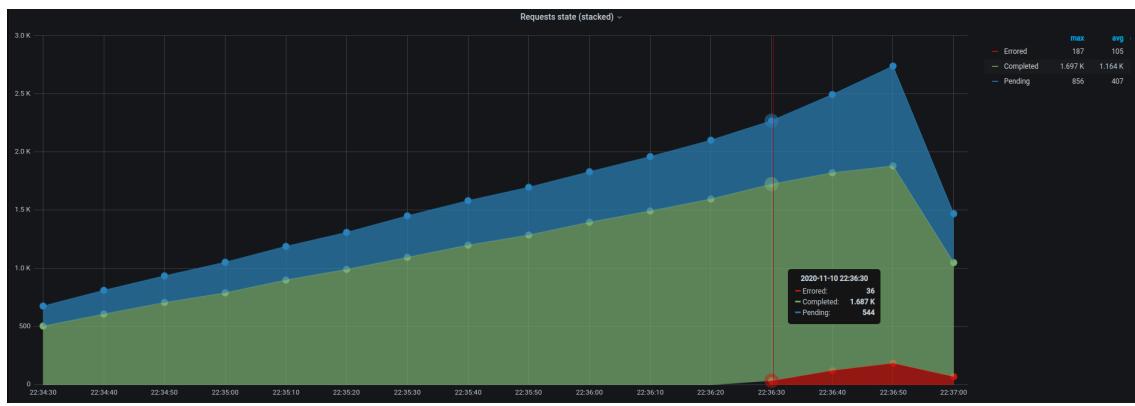


Figura 11: Requests state (stacked) - Load Test - Node



Figura 12: Resources usage - Load Test - Node

Summary report

```

Scenarios launched: 20085
Scenarios completed: 19666
Requests completed: 19666
Mean response/sec: 91.27
Response time (msec):
  min: 3000.2
  max: 3056.8
  median: 3002.7
  p95: 3007.5
  p99: 3011
Scenario counts:
  Sleep (/sleep): 20085 (100 %)
Codes:
  200: 19666
Errors:
  ECONNRESET: 419

```

En este caso, el endpoint utiliza la función `setTimeout` que es **no bloqueante**. Así, el *event loop* de node, al recibir un request, lo ubica en una lista de pending timeouts, pudiendo resolver otros requests. En cada iteración, el event loop verificará si se cumplió el timeout, en cuyo caso ejecutará el callback correspondiente y devolverá finalmente la respuesta. Por este motivo, la mediana del response time del cliente se mantiene casi constante a lo largo del test de carga en el valor esperado, *3 segundos*, y el servidor puede atender múltiples requests de manera concurrente y asincrónica.

Gráficamente, se observa que el servidor responde satisfactoriamente (y en el tiempo estipulado) los requests a medida que aumenta la carga, llegándose al punto límite en el que dispara errores al cabo de casi 3 minutos, soportando una **carga límite de 180 req/s**. Al efectuar un análisis sobre los errores obtenidos, se observa que surgen a partir del límite de requests en cola del load balancer nginx. Node, por su parte, logra atender asincrónicamente los requests que le llegan, en un bajo tiempo de respuesta.

### 1.3.1.3 Node replicado

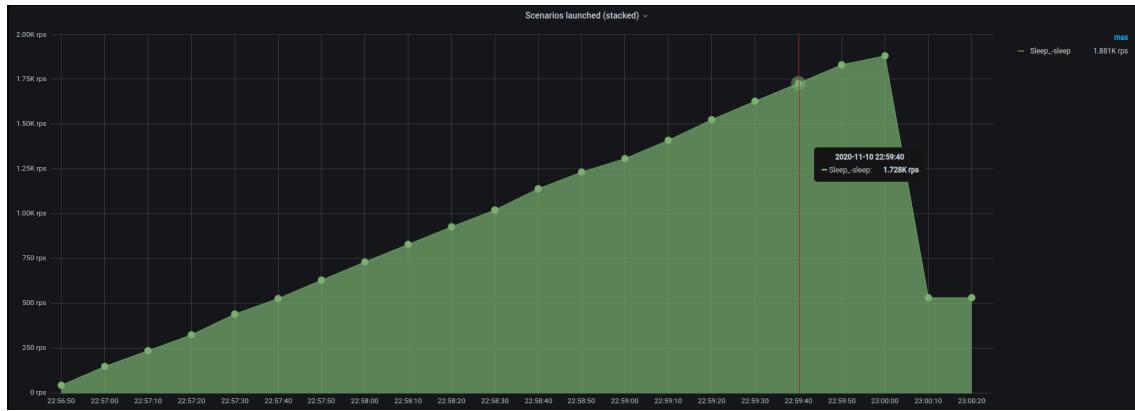


Figura 13: Scenarios launched (stacked) - Load Test - Node Replicated

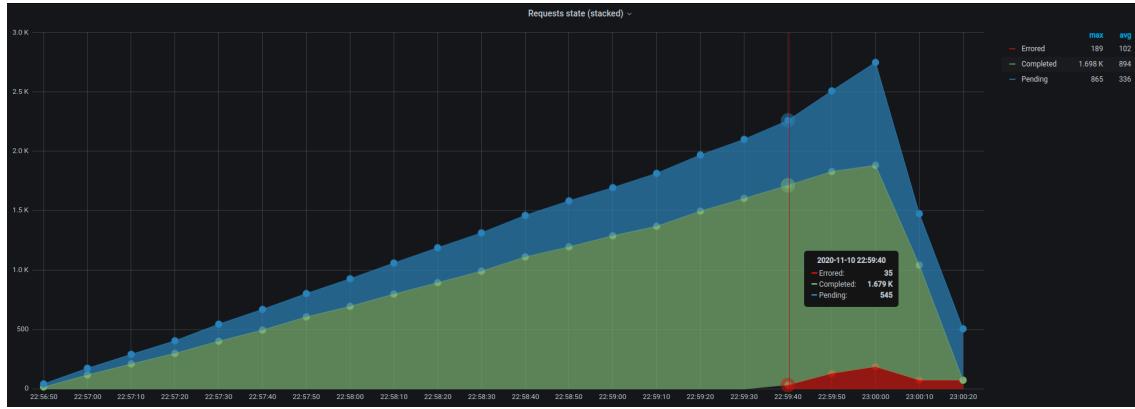


Figura 14: Requests state (stacked) - Load Test - Node Replicated

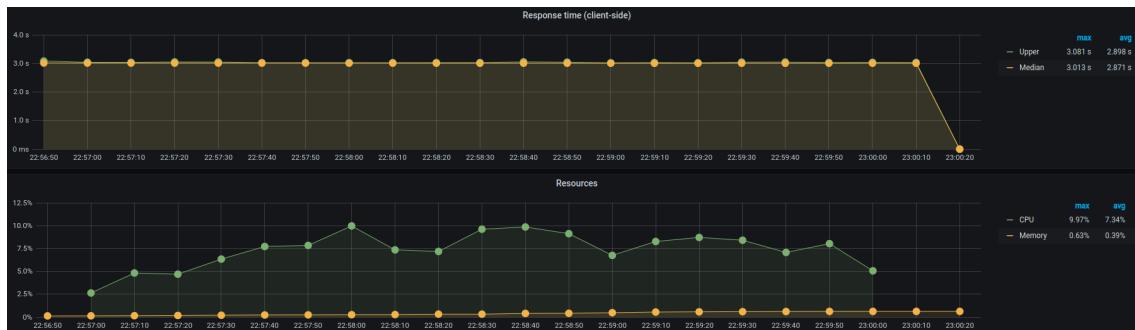


Figura 15: Resources usage - Load Test - Node Replicated

```

Summary report
Scenarios launched: 20089
Scenarios completed: 19657
Requests completed: 19657
Mean response/sec: 90.83
Response time (msec):
  min: 3000.3
  max: 3050.4
  median: 3003
  p95: 3007.6
  p99: 3011.5
Scenario counts:
  Sleep (/sleep): 20089 (100%)
Codes:
  200: 19657
Errors:
  ECONNRESET: 432

```

En este caso, los resultados son casi idénticos al servidor que contiene un solo servicio de node, dado que las curvas son similares y se alcanza el punto de carga máxima, donde comienzan aemerger fallas a la misma cantidad de req/s.

Por este motivo, la *escalabilidad horizontal*, a priori, no se justifica, puesto que no provee mejoras apreciables en el análisis de este escenario. El hecho de que el punto de fallas se alcanza a la misma carga aproximadamente, es un claro indicio de que el cuello de botella no está en el servicio, sino que muy probablemente en el load balancer *nginx* (o en alguna limitación inherente a la configuración establecida). La mediana del response time percibido por el cliente, por su parte, es similar al caso de node no replicado también, así como el consumo de recursos.

#### 1.3.1.4 Conclusión

Con respecto a la **disponibilidad**, el servidor *Gunicorn* presentó sucesivas fallas; en primer lugar se produjeron fallas de *timing*, degradándose de manera muy rápida el *response time*, luego, aparecieron fallas de *omission*, en las que el sistema no responde a ciertos requests, para luego producirse un *crash*, en el que el sistema sufre omisiones repetitivas y deja de responder. Ante las susodichas fallas, el servidor fue incapaz de enmascararlas, afectando ampliamente su disponibilidad. Los servidores *Node* y *Node Replicado*, por su parte, tuvieron una mayor disponibilidad a lo largo de los escenarios, emergiendo omission failures al alcanzarse el punto de inflexión donde las requests encoladas llegan a un máximo. No obstante, dichas fallas provienen de un límite inherente al load balancer *nginx*, el cual es compartido por todos los servidores, por lo que una primera mejora que se le podría hacer al sistema sería aumentar la capacidad de encolar requests del load balancer.

En cuanto a **performance**, *Node* y *Node Replicado* demostraron resultados idénticos, muy superiores a los de *Gunicorn*, más que nada debido a la naturaleza del endpoint y el funcionamiento de las tecnologías, como fue explicado previamente. La *latencia* y el *completion time* de *Node* y *Node Replicado* se mantuvo estable durante todo el escenario, en valores cercanos al ideal, mientras que en *Gunicorn* ambos valores degradaron muy rápidamente.

El consumo de CPU fue ampliamente mayor en el servidor *Node* y *Node Replicado* que en el de *Gunicorn*, lo cual es natural, dado que atendió muchas más requests. Los consumos de memoria fueron bajos en todos los casos.

Por último, cabe mencionar que **escalar** horizontalmente, replicando el servidor de *Node*, no produjo mejoras apreciables en los resultados, por lo que no sería conveniente asumir los costos de ello. Para mejorar los resultados de *Gunicorn*, se podría escalar horizontalmente en cantidad de workers, utilizando más CPUs, para así poder procesar más requests en paralelo.

### 1.3.2. Spike test

#### 1.3.2.1 Gunicorn

Para realizar un *Spike Test* el servidor de *Gunicorn*, se utilizó un escenario que repite dos veces el siguiente ciclo:

- Carga constante de 5 req/s durante 20 segs.
- Rampa ascendente pronunciada (spike up) desde 5 req/s hasta 40 req/s en 10 segs.
- Carga constante de 40 req/s durante 10 segs.
- Rampa descendente pronunciada (spike down) desde 40 req/s hasta 5 req/s en 10 segs.

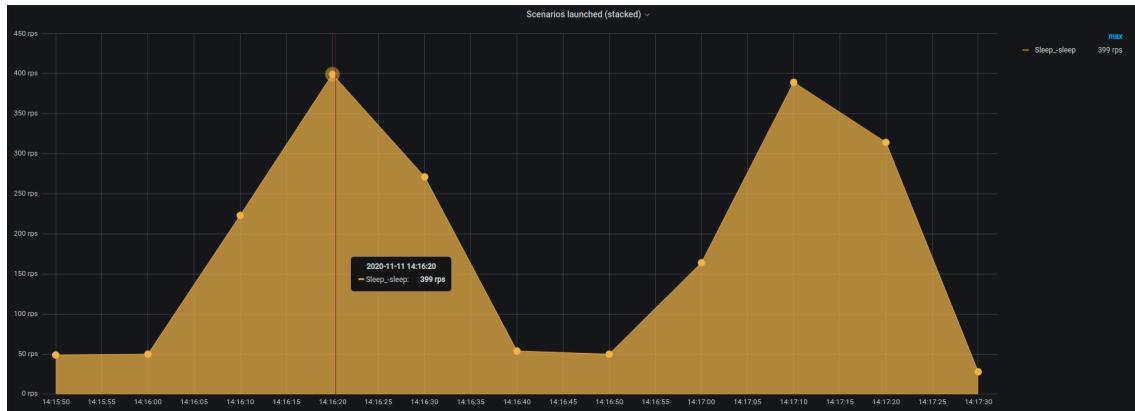


Figura 16: Scenarios launched (stacked) - Spike Test - Gunicorn

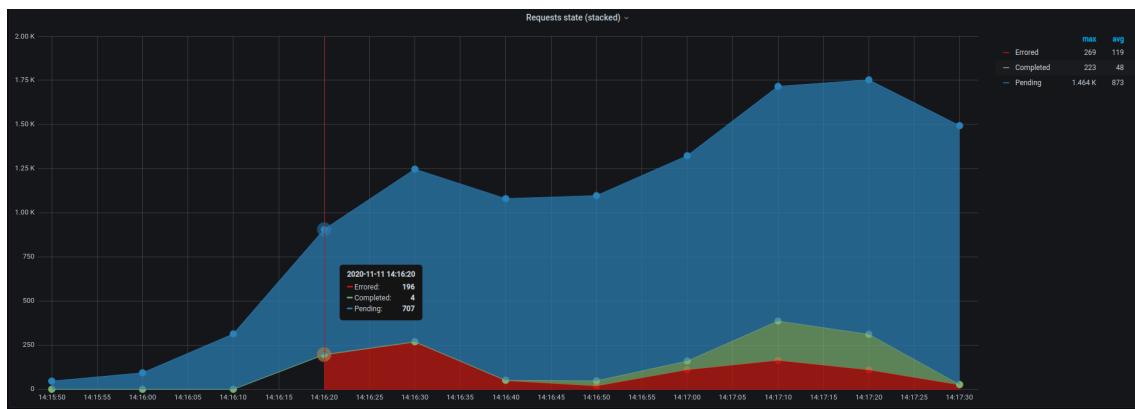


Figura 17: Requests state (stacked) - Spike Test - Gunicorn



Figura 18: Resources usage - Spike Test - Gunicorn

```
All virtual users finished
Summary report
  Scenarios launched: 1991
  Scenarios completed: 1036
```

```

Requests completed: 1036
Mean response/sec: 12.25
Response time (msec):
  min: 3012.8
  max: 60033.8
  median: 60002.8
  p95: 60006.2
  p99: 60007.7
Scenario counts:
  Sleep (/sleep): 1991 (100 %)
Codes:
  200: 21
  504: 1015
Errors:
  ECONNRESET: 955

```

Se puede observar que, ya desde el comienzo, el servidor comienza a encolar los requests debido a la naturaleza del endpoint (bloqueante) y a las características del servidor, que corre en un solo hilo.

Así, se encolan las requests incluso antes de que se llegue a la primera etapa de spike up, donde además de seguir encolándose, comienzan a aparecer errores correspondientes al alcance de la máxima capacidad de requests en la cola del load balancer.

A fin de cuentas, el endpoint responde satisfactoriamente unas pocas requests (21) al principio, y, al comenzar la fase de spike up del segundo ciclo, deja de responder y retorna de ahí en adelante timeout (504) y ECONNRESET (cota superior de requests encoladas), por lo que no hay que dejarse confundir por el crecimiento de la curva verde, que corresponde a respuestas de timeouts (por parte de nginx).

La mediana del tiempo de respuesta percibido por el usuario es muy pobre, degradándose a medida que aumentan la carga.

El consumo de CPU y memoria, por su parte, se mantienen estables a lo largo de todo el escenario.

### 1.3.2.2 Node

Para realizar un *Spike Test* el servidor de *Gunicorn*, se utilizó un escenario que repite dos veces el siguiente ciclo:

- Carga constante de 10 req/s durante 20 segs.
- Rampa ascendente pronunciada (spike up) desde 10 req/s hasta 170 req/s en 10 segs.
- Carga constante de 170 req/s durante 10 segs.
- Rampa descendente pronunciada (spike down) desde 170 req/s hasta 10 req/s en 10 segs.



Figura 19: Scenarios launched (stacked) - Spike Test - Node

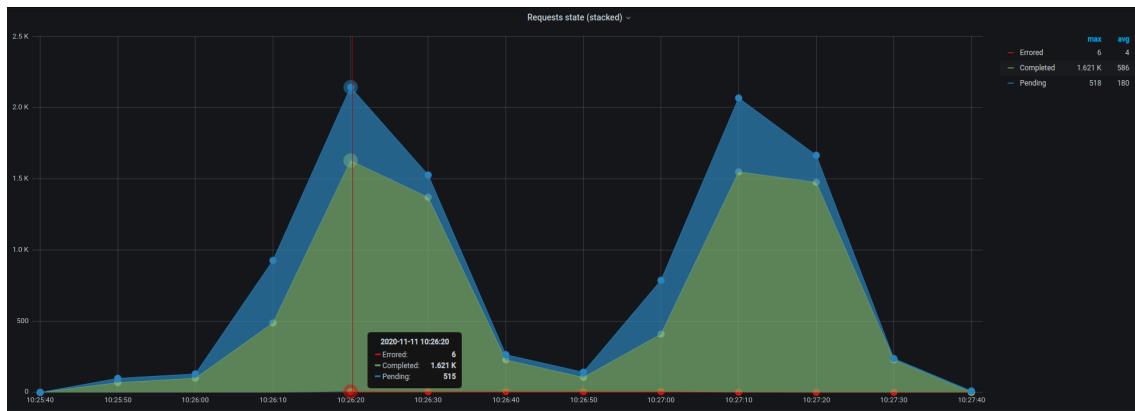


Figura 20: Requests state (stacked) - Spike Test - Node



Figura 21: Resources usage - Spike Test - Node

All virtual users finished  
 Summary report  
 Scenarios launched: 7626

```

Scenarios completed: 7617
Requests completed: 7617
Mean response/sec: 62.47
Response time (msec):
  min: 3000.2
  max: 3043.7
  median: 3002
  p95: 3006
  p99: 3009.7
Scenario counts:
  Sleep (/sleep): 7626 (100%)
Codes:
  200: 7617
Errors:
  ECONNRESET: 9

```

Considerando la naturaleza de Node, resultan esperables los resultados obtenidos. La curva de requests completados acompaña a la de los requests pendientes durante todo el escenario, separándose a medida que aumenta la carga por el tiempo de sleep inherente al endpoint. Esta tecnología es capaz de atender asincrónicamente todas las requests enviadas, puesto que su hilo principal no se bloquea en la ejecución de las mismas. Se producen unos pocos errores debido al límite máximo de requests encolados (ECONNRESET). No obstante de ello, no se produce una degradación en el tiempo de respuesta percibido por el usuario, manteniéndose este en un valor quasi-constante de 3 segundos (que resulta lo esperado) durante todo el escenario.

La curva de consumo de CPU aumenta y disminuye en correspondencia con la carga, no llegando a niveles altos. El consumo de memoria, por su parte, es muy bajo y casi despreciable.

### 1.3.2.3 Node replicado

Se ejecutó el mismo escenario que para Node, y tal como era esperado, se obtuvieron los mismos resultados. El análisis de dicha coincidencia fue realizado en el Load Test y se omite para evitar redundancias.



Figura 22: Scenarios launched (stacked) - Spike Test - Node Replicated

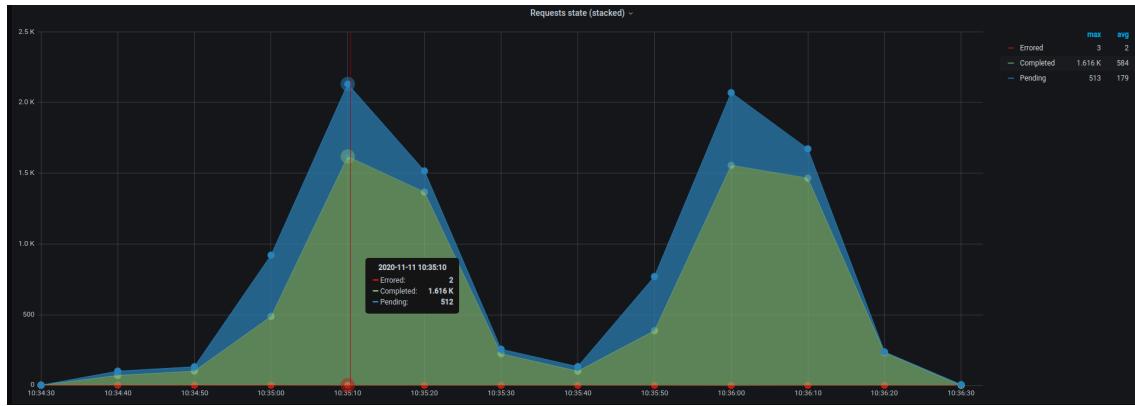


Figura 23: Requests state (stacked) - Spike Test - Node Replicated



Figura 24: Resources usage - Spike Test - Node Replicated

```
All virtual users finished
```

#### Summary report

Scenarios launched: 7599

Scenarios completed: 7594

Requests completed: 7594

Mean response/sec: 62.22

Response time (msec):

min: 3000.2

max: 3034.9

median: 3002.1

p95: 3006.9

p99: 3010.9

Scenario counts:

Sleep (/sleep): 7599 (100%)

Codes:

200: 7594

Errors:

ECONNRESET: 5

#### 1.3.2.4 Conclusión

En el análisis previo se reafirmaron las conclusiones que a las que se había llegado mediante el Load Test. El servidor *Gunicorn* degradó velozmente en disponibilidad, pudiendo atender pocas requests y presentando fallas de crash en la mayor parte del escenario (luego del primer spike up). Los servidores *Node* y *Node Replicado* conservaron latencias y response times cercanos al valor ideal, no viéndose afectados por el súbito incremento de carga, y siendo muy estables en performance y disponibilidad.

#### 1.3.3. Endurance test

##### 1.3.3.1 Gunicorn

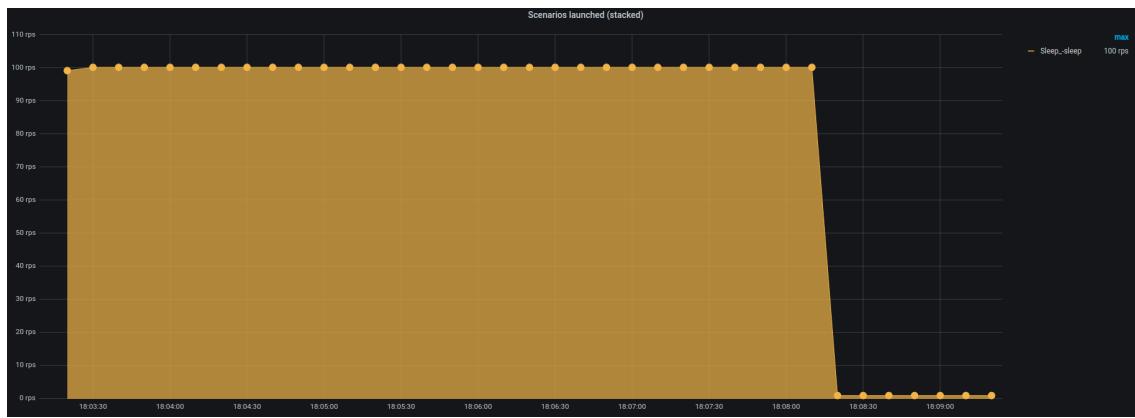


Figura 25: Scenarios launched (stacked) - Endurance Test - Gunicorn

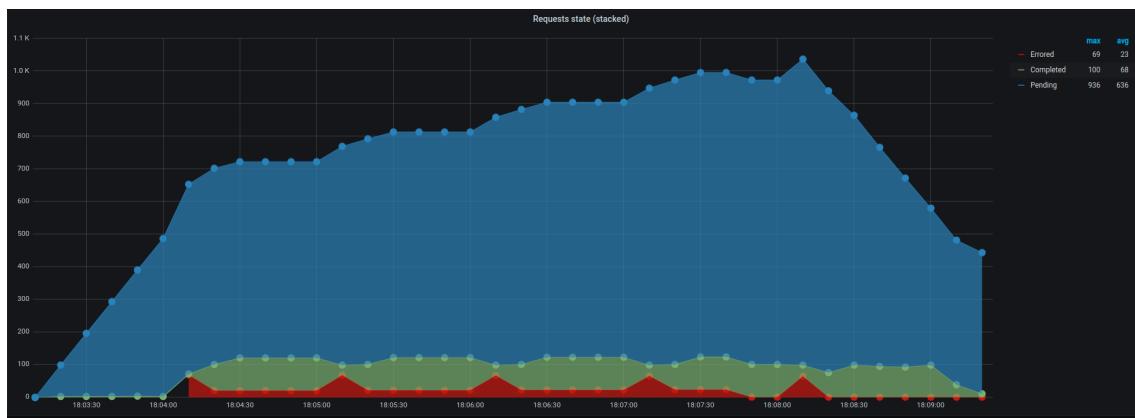


Figura 26: Requests state (stacked) - Endurance Test - Gunicorn



Figura 27: Resources usage - Endurance Test - Gunicorn

```
All virtual users finished
Summary report
Scenarios launched: 3000
Scenarios completed: 2573
Requests completed: 2573
Mean response/sec: 7.71
Response time (msec):
    min: 3014.3
    max: 91628.3
    median: 60001.8
    p95: 60006.5
    p99: 67222
Scenario counts:
    Sleep (/sleep): 3000 (100%)
Codes:
    200: 20
    504: 2553
Errors:
    ECONNRESET: 427
```

A partir de los resultados obtenidos, se observa que el endpoint, ante una carga constante no muy grande, comienza a fallar al cabo de aproximadamente un minuto, primero retornando errores ECONNRESET (debido a la alta cantidad de requests encolados), para luego dejar de responder (504) por el resto del escenario, afectando por completo la disponibilidad del servidor.

El tiempo de respuesta percibido por el usuario se degrada rápidamente a medida que se encolan las requests, con valores muy lejanos a los ideales, mostrando una muy mala performance.

El consumo de recursos es bajo y estable durante todo el escenario, dada la naturaleza del endpoint, con escaso procesamiento.

### 1.3.3.2 Node



Figura 28: Scenarios launched (stacked) - Endurance Test - Node

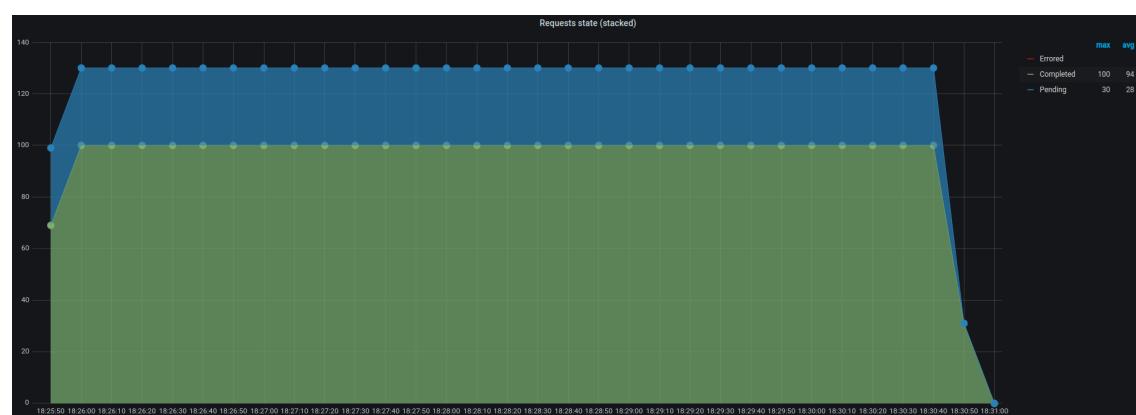


Figura 29: Requests state (stacked) - Endurance Test - Node

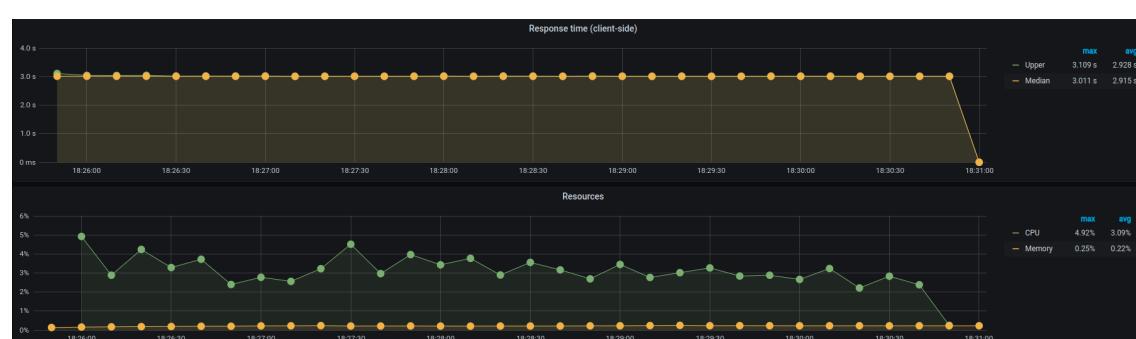


Figura 30: Resources usage - Endurance Test - Node

```
All virtual users finished
Summary report
Scenarios launched: 3000
Scenarios completed: 3000
Requests completed: 3000
Mean response/sec: 9.36
Response time (msec):
  min: 3000.6
  max: 3036.7
  median: 3003.8
  p95: 3006.4
  p99: 3009.9
Scenario counts:
  Sleep (/sleep): 3000 (100%)
Codes:
  200: 3000
```

En este caso, el servidor muestra un comportamiento muy estable y sólido durante todo el escenario, siendo capaz de responder todas las requests en tiempo y forma, lo cual era esperable dada la naturaleza de la tecnología (asincrónica y concurrente, no bloqueándose en las requests de timeout).

En este escenario, su disponibilidad es plena, y el rendimiento es muy bueno, dado que el tiempo de respuesta percibido por el usuario se mantiene constante en los valores esperados, y todos los requests lanzados son respondidos adecuadamente.

El consumo de recursos, por su parte, es estable, si bien presenta ciertas variaciones menores, sobre todo en el de CPU.

### 1.3.3.3 Node replicado

Tanto los resultados como el análisis son análogos al de Node. A continuación, los resultados obtenidos:

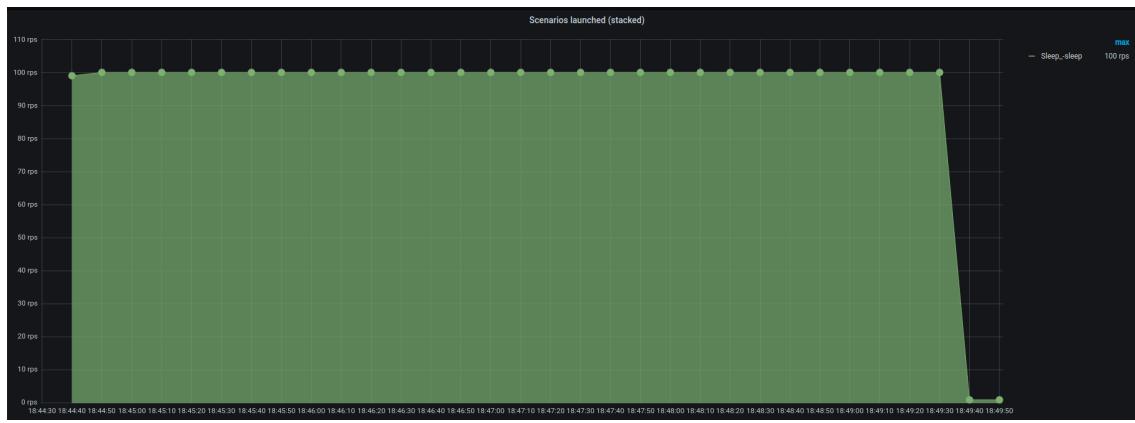


Figura 31: Scenarios launched (stacked) - Endurance Test - Node Replicated

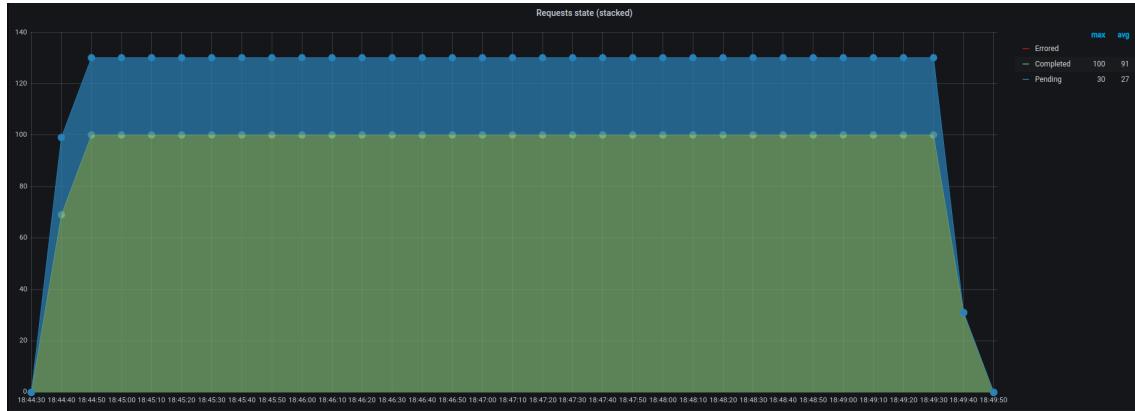


Figura 32: Requests state (stacked) - Endurance Test - Node Replicated



Figura 33: Resources usage - Endurance Test - Node Replicated

```
All virtual users finished
Summary report
Scenarios launched: 3000
Scenarios completed: 3000
Requests completed: 3000
Mean response/sec: 9.36
Response time (msec):
  min: 3000.8
  max: 3038.3
  median: 3003.9
  p95: 3006.4
  p99: 3011
Scenario counts:
  Sleep (/sleep): 3000 (100%)
Codes:
  200: 3000
```

### 1.3.3.4 Conclusión

Nuevamente, los servidores *Node* y *Node Replicado* fueron muy estables, mientras que Gunicorn degradó rápidamente en disponibilidad y performance. Se reafirman las conclusiones alcanzadas

en los previos tests.

## 1.4. Endpoint: Heavy

### 1.4.1. Load test

El escenario *Load Test* consiste en ir incrementando cada 10 segundos, de 10 en 10 y en forma de *rampa*, la cantidad de requests por segundo. Así, comenzando en 0 req/s, se llega a 100 req/s en un lapso de 100 segundos.

#### 1.4.1.1 Gunicorn

Al ejecutar este escenario, se obtuvieron los siguientes resultados:

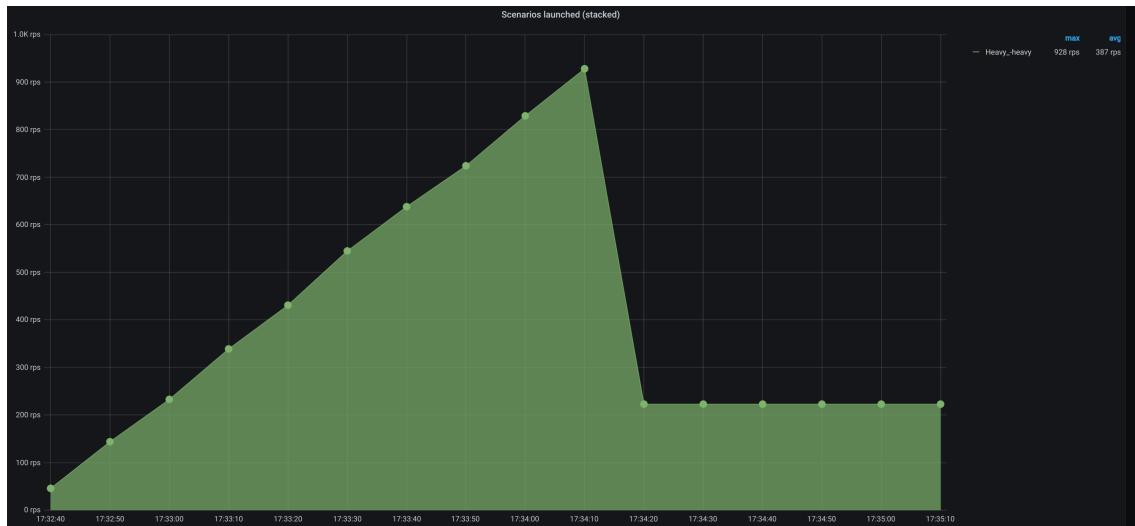


Figura 34: Scenarios launched (stacked) - Load Test - Gunicorn

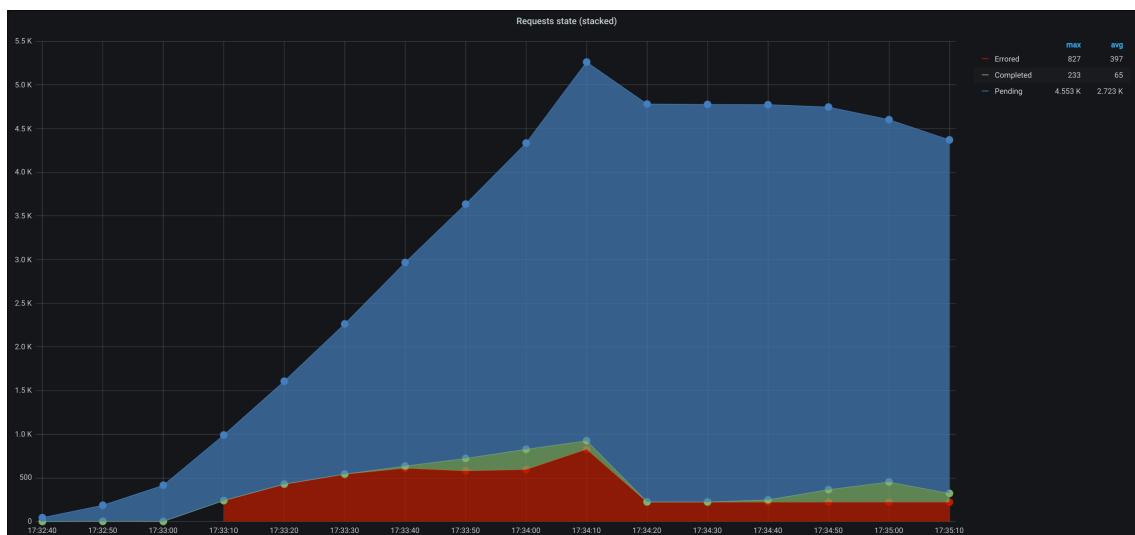


Figura 35: Requests state (stacked) - Load Test - Gunicorn



Figura 36: Resources usage - Load Test - Gunicorn

```
Summary report @ 17:35:08(-0300) 2020-11-10
Scenarios launched: 5080
Scenarios completed: 1035
Requests completed: 1035
Mean response/sec: 31.89
Response time (msec):
  min: 3006.4
  max: 60053.2
  median: 60002
  p95: 60007
  p99: 60015.4
Scenario counts:
  Heavy (/heavy): 5080 (100%)
Codes:
  200: 22
  504: 1013
Errors:
  ECONNRESET: 4045
```

El endpoint *Heavy* simula un procesamiento intensivo, por lo que el único worker de *Gunicorn* se bloquea, procesando secuencialmente una request a la vez. Durante el tiempo de procesamiento, en este caso, 3 segundos, los requests se encolarán, debido a que el worker se bloquea procesando en este tiempo, por lo que los requests pendientes se van encolando, hasta que en algún momento el servidor colapsa, y no llega a responderlos.

Al principio, el servidor puede responder a lo sumo 3 requests cada 10 segundos, por lo que el resto de los requests se acumulan, lo cual se puede ver con el crecimiento de la curva de pending requests (azul). Aproximadamente, a los 34 segundos, se llega a una **carga máxima de 34 req/s**, en la que el servidor empieza a lanzar errores. Luego, al alcanzar los **70req/s**, ya no hay respuesta exitosa por parte del servidor.

En el reporte, se puede ver que el endpoint responde únicamente 22 requests exitosamente, y lanza 4045 errores, producto de alcanzar la máxima cantidad de requests encolados por parte del nginx. Asimismo, hay 1013 de respuestas de tipo 504, lo cual significa que el nginx llega el tiempo máximo de espera para una respuesta (timeout).

El proceso consume todo el CPU durante todo el escenario, lo cual era esperable dado que se trata de un escenario de procesamiento intensivo. El consumo de memoria, por su parte, se mantiene en niveles bajos y estables.

El tiempo de respuesta percibido por el cliente se degrada rápidamente a medida que se van encolando los pending requests.

En cuanto a los atributos de calidad se ve afectada la disponibilidad luego de sobreponer los **70 reqs/s**.

A grandes rasgos, el escenario presenta similitudes con el del endpoint sleep para la misma tecnología, dado que en ambos el worker se bloquea y causa el encolamiento de muchos requests, hasta que se llega al límite. La diferencia principal entre ambos radica en el consumo de CPU.

#### 1.4.1.2 Node

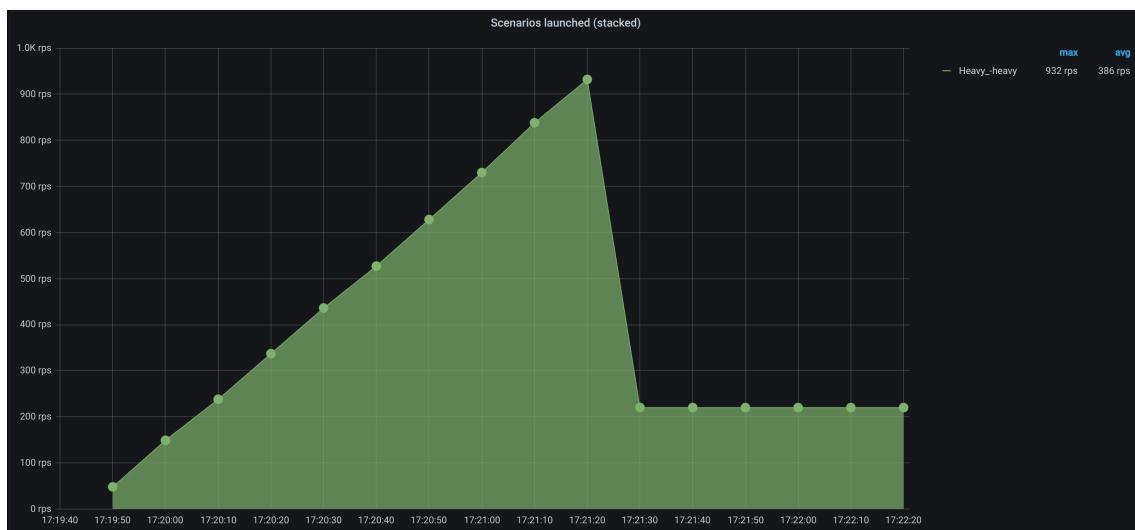


Figura 37: Scenarios launched (stacked) - Load Test - Node

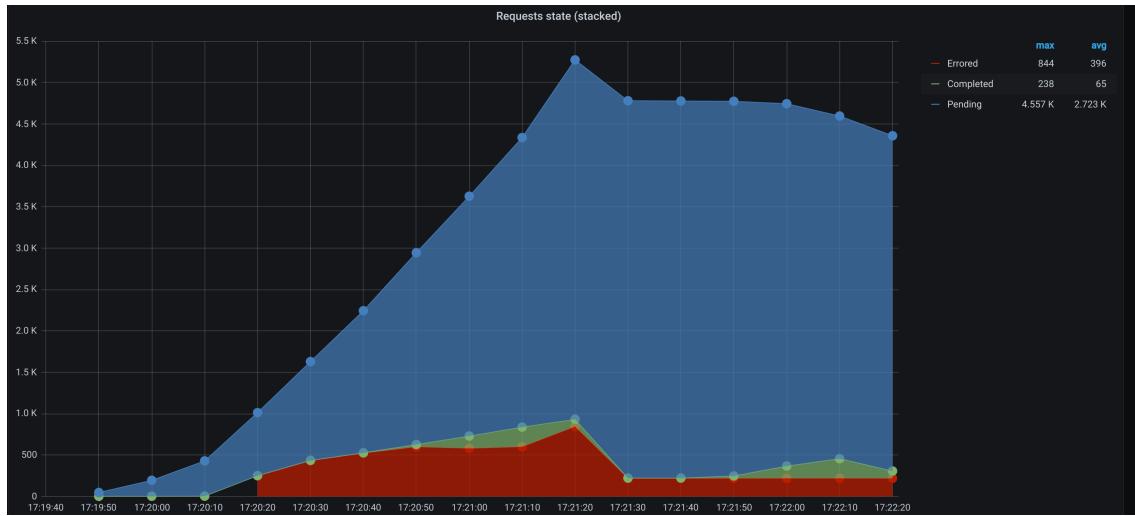


Figura 38: Requests state (stacked) - Load Test - Node



Figura 39: Resources usage - Load Test - Node

```
All virtual users finished
Summary report @ 15:24:42(-0300) 2020-11-11
Scenarios launched: 5042
Scenarios completed: 1035
Requests completed: 1035
Mean response/sec: 31.65
Response time (msec):
    min: 3036.7
    max: 60036.7
    median: 60000.5
    p95: 60005.3
    p99: 60007.5
Scenario counts:
```

```

Heavy (/heavy): 5042 (100%)
Codes:
 200: 21
 504: 1014
Errors:
 ECONNRESET: 4007

```

Gráficamente, se obtienen resultados muy similares a los de Gunicorn. Análogamente, al principio, el servidor puede responder a lo sumo 3 requests cada 10 segundos, por lo que el resto de los requests se van acumulando (creciendo la curva azul de pending requests).

Aproximadamente, a los 34 segundos, se alcanza una **carga máxima de 34 req/s**, en la que el servidor empieza a lanzar errores. En el alrededor de **70 req/s** ya no hay respuesta exitosa por parte del mismo.

En el reporte, se puede ver que el endpoint responde únicamente 21 requests exitosamente, lanza 4007 errores debido a que nginx llega a encolar la cantidad máxima de requests, y hay 1014 errores de tipo 504 (timeout).

El consumo de CPU se mantiene al máximo durante todo el escenario, lo cual es natural en este tipo de escenario de procesamiento intensivo. El consumo de memoria, por su parte, se mantiene en niveles bajos y estables.

El tiempo de respuesta percibido por el cliente degrada rápidamente a medida que se van encolando los requests.

En cuanto a los atributos de calidad, se ve seriamente afectada la disponibilidad luego de sobrepasar los **70 rqs/s**.

#### 1.4.1.3 Node replicado



Figura 40: Scenarios launched (stacked) - Load Test - Node Replicated

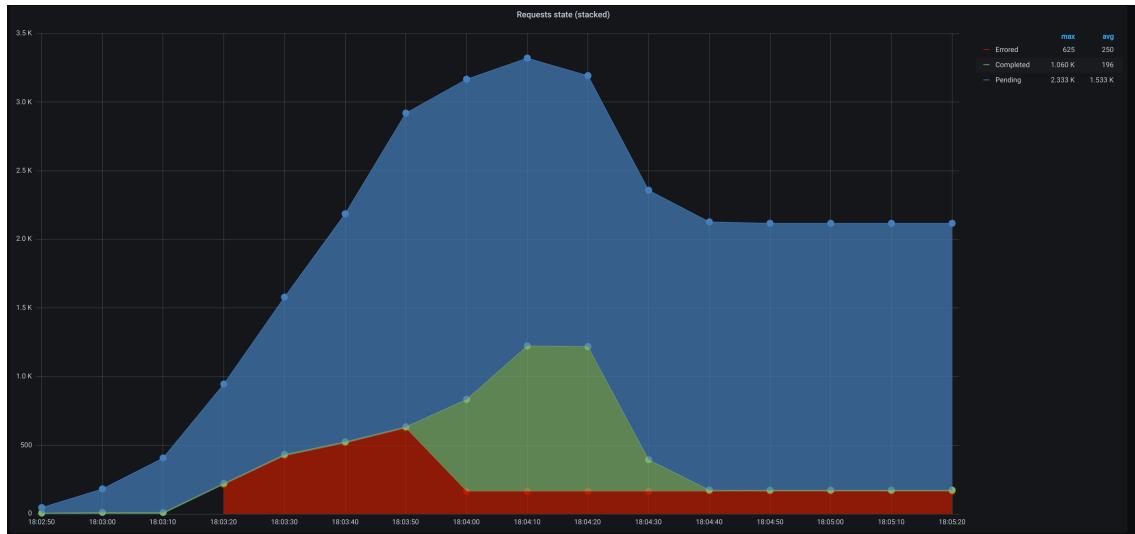


Figura 41: Requests state (stacked) - Load Test - Node Replicated



Figura 42: Resources usage - Load Test - Node Replicated

```
All virtual users finished
Summary report @ 18:04:46(-0300) 2020-11-10
Scenarios launched: 5045
Scenarios completed: 3104
Requests completed: 3104
Mean response/sec: 38.3
Response time (msec):
  min: 0.4
  max: 60115.4
  median: 0.7
  p95: 60001.7
  p99: 60004.5
Scenario counts:
  Heavy (/heavy): 5045 (100%)
```

```

Codes:
 200: 69
 502: 3035
Errors:
 ECONNRESET: 1939
 ESOCKETTIMEDOUT: 2

```

En el gráfico, a diferencia con el de Node (único), se puede ver que el servidor al principio puede responder a lo sumo 9 requests, lo cual es tres veces más. Esto era esperable, ya que Node Replicado está formado por tres servicios de Node, triplicando así la capacidad de procesamiento. No obstante, sucede el mismo efecto que antes pero de forma más tardía: los requests se van acumulando, la curva de pending requests va creciendo, pero la cantidad de pending requests es menor en comparación a los casos anteriores. Aproximadamente, a los 34 segundos, se llega a una **carga máxima de 35 req/s**, en la que el servidor empieza a lanzar errores. Cuando se alcanza una carga de alrededor de **80req/s** se produce una caída en el servidor.

Con respecto al response time percibido por el usuario, si bien va degradando, es llamativo que durante 30 segundos, su mediana es 0. Esto coincide con la caída del servidor, y su respectiva aparición de errores 502. Luego surgen errores de socket. Nuevamente es clara la degradación del mismo, proporcional a la cantidad de requests que se van encolando.

En el reporte, se puede ver que el endpoint responde 69 requests exitosamente (tres veces más que en los otros servidores), lanza 1939 errores por haber alcanzado la máxima cantidad de requests encolados, y 3035 de errores de tipo 502, que aparecen una vez el servidor está caído. Luego de correr el escenario el servidor quedó completamente inutilizable.

El consumo de CPU es máximo en todo el escenario, y el consumo de memoria se mantiene estable en niveles bajos, al igual que en los otros servidores.

En cuanto a los atributos de calidad se ve afectada totalmente la disponibilidad luego de sobrepasar los **80 rqs/s**.

#### 1.4.1.4 Conclusión

Comparando los resultados obtenidos bajo el escenario de **Load test**, se puede ver que los tres comienzan a tirar errores en una carga de aproximadamente **35 rqs/s**. Las curvas de *Gunicorn* y *Node* son muy parecidas, y son similares asimismo con la curva de *Gunicorn* del endpoint *Sleep*. Como ya fue previamente explicado, todo esto se produce debido a que el hilo principal se bloquea, en este caso, procesando, no pudiendo atender nuevas requests de forma concurrente, lo cual sí sucede en *Node Replicado*.

En cuanto a la **disponibilidad**, los tres servidores lanzaron sucesivas fallas después de realizar **Spike Down**. Van apareciendo timing failures, que degeneran en omission failures, para culminar en crash failures, en el que el sistema sufre omisiones repetitivas y deja de responder. Particularmente, en el caso de *Node Replicado*, el servidor se cae y queda en estado no funcional.

Con respecto a la **performance**, los tiempos de respuesta degradan a medida que se van encolando los requests en todos los casos.

Al ser los requests de procesamiento intensivo, el consumo de CPU fue pleno en todos los escenarios, mientras que el consumo de memoria fue muy bajo y estable.

Por último, cabe destacar que la **escalabilidad horizontal** en *Node Replicado*, permitió aplazar el punto de fallas, y responder más request al mismo tiempo (y por ello, se encolan un

poco menos de requests), en particular, el triple, mejorando significativamente la performance.

#### 1.4.2. Spike test

Se utilizó un escenario para tres servidores que repite dos veces el siguiente ciclo:

- Carga constante de 2 req/s durante 10 segs.
- Rampa ascendente pronunciada (spike up) desde 2 req/s hasta 30 req/s en 20 segs.
- Rampa descendente pronunciada (spike down) desde 30 req/s hasta 1 req/s en 20 segs.

##### 1.4.2.1 Gunicorn

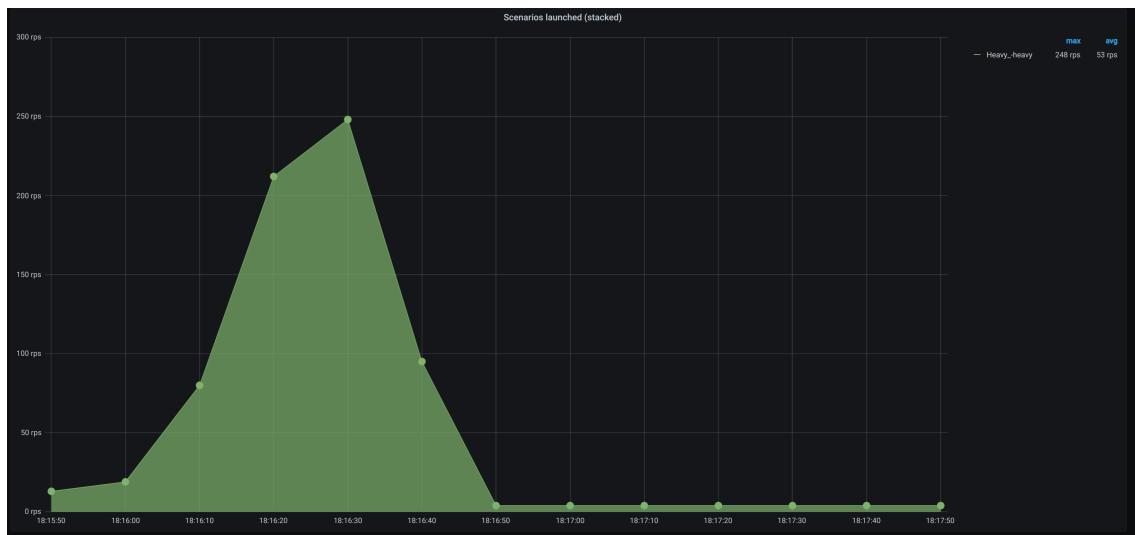


Figura 43: Scenarios launched (stacked) - Spike Test - Gunicorn

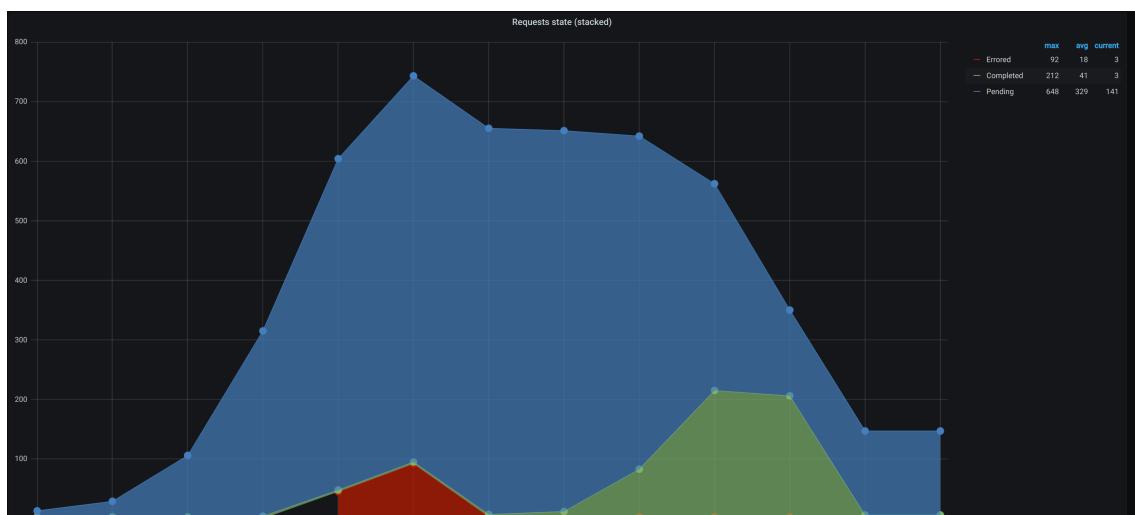


Figura 44: Requests state (stacked) - Spike Test - Gunicorn



Figura 45: Resources usage - Spike Test - Gunicorn

```
All virtual users finished
Summary report @ 18:17:40(-0300) 2020-11-10
Scenarios launched: 671
Scenarios completed: 531
Requests completed: 531
Mean response/sec: 5.56
Response time (msec):
    min: 3006.5
    max: 60036.6
    median: 60001
    p95: 60004.3
    p99: 60006.6
Scenario counts:
    Heavy (/heavy): 671 (100 %)
Codes:
    200: 25
    504: 506
Errors:
    ECONNRESET: 140
```

En el gráfico, se puede ver que mientras incrementa la cantidad de request por segundo, el tiempo de respuesta que percibe el usuario se va degradando. Al comienzo de la fase *Spike Down*, el servidor empieza a lanzar errores. Después de haber finalizado dicha fase, el servidor deja de responder los requests acumulados, retornando error 504, por lo que no se debe confundir el crecimiento en la curva verde.

En el reporte, se puede ver que el endpoint responde únicamente 25 requests exitosamente, lanza 140 errores debiendo que nginx llega a encolar la cantidad máxima de requests, y hay 506 errores de tipo 504 (timeout).

El consumo de CPU se mantiene al máximo durante todo el escenario, y el de memoria es bajo y estable.

### 1.4.2.2 Node

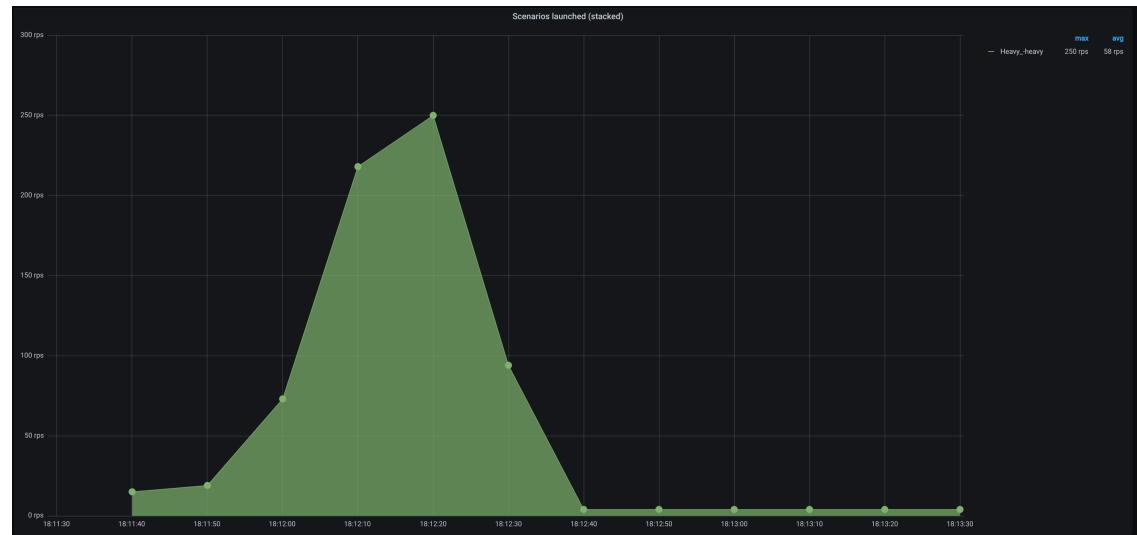


Figura 46: Scenarios launched (stacked) - Spike Test - Node

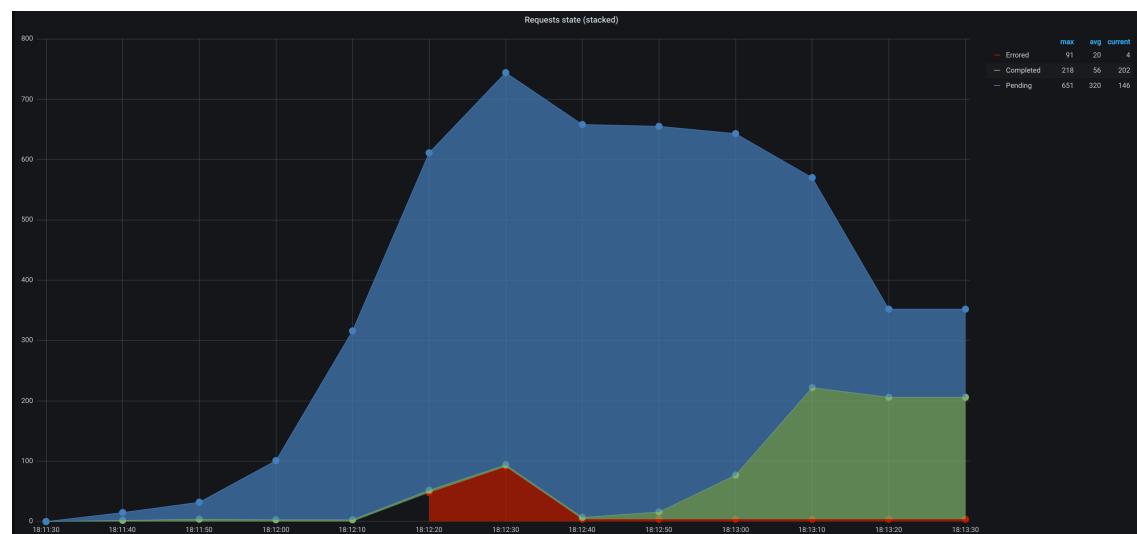


Figura 47: Requests state (stacked) - Spike Test - Node



Figura 48: Resources usage - Spike Test - Node

```
All virtual users finished
Summary report @ 18:13:33(-0300) 2020-11-10
Scenarios launched: 673
Scenarios completed: 530
Requests completed: 530
Mean response/sec: 5.74
Response time (msec):
    min: 3024.1
    max: 60035.7
    median: 60000.9
    p95: 60004.1
    p99: 60015
Scenario counts:
    Heavy (/heavy): 673 (100 %)
Codes:
    200: 24
    504: 506
Errors:
    ECONNRESET: 143
```

Los gráficos son, una vez más, casi idénticos a los de *Gunicorn*. A medida que aumenta la carga, los requests se van encolando (pendientes) cada vez más porque el hilo principal está bloqueado procesando.

En el reporte, se puede observar que el endpoint responde 24 requests exitosamente, lanza 143 errores debido a que nginx llega a encolar la cantidad máxima de requests, y 506 errores 504 (timeout).

El proceso consume el máximo de CPU durante todo el escenario, y el consumo de memoria es bajo y estable.

### 1.4.2.3 Node replicado

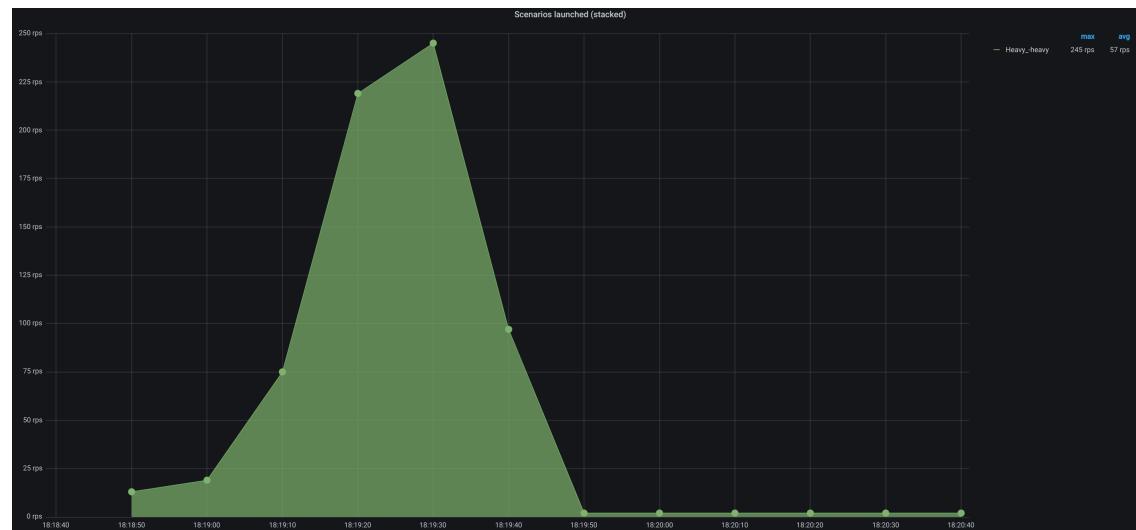


Figura 49: Scenarios launched (stacked) - Spike Test - Node Replicated



Figura 50: Requests state (stacked) - Spike Test - Node Replicated



Figura 51: Resources usage - Spike Test - Node Replicated

```
All virtual users finished
Summary report @ 18:21:07(-0300) 2020-11-10
Scenarios launched: 670
Scenarios completed: 526
Requests completed: 526
Mean response/sec: 4.57
Response time (msec):
    min: 3089.7
    max: 120040.5
    median: 60005.5
    p95: 60065
    p99: 120005.7
Scenario counts:
    Heavy (/heavy): 670 (100 %)
Codes:
    200: 56
    502: 470
Errors:
    ECONNRESET: 121
    ESOCKETTIMEDOUT: 23
```

En este caso, el comportamiento es similar, a excepción de que se pueden procesar más requests al comienzo, debido a la concurrencia que introduce la *escalabilidad horizontal*. Al cabo de 100 segundos, el servidor se cae y deja de responder los requests.

En el reporte, se puede ver que el endpoint responde 56 requests exitosamente, lanza 121 errores debiendo que nginx llega a encolar la cantidad máxima de requests, y hay 470 errores de tipo 502, que aparecen una vez que el servidor se cae.

Es llamativo que nuevamente este servidor se caiga, cuando el resto no lo hace (simplemente deja de responder).

#### 1.4.2.4 Conclusión

Con respecto a la **disponibilidad**, tanto *Gunicorn* como *Node* presentaron un punto de inflexión en el que se alcanza la máxima cantidad de requests encoladas y dejan de responder. *Node*

Replicado, además, se cae y queda disfuncional. Podría introducirse una mejora al aumentar la capacidad máxima de requests encoladas del load balancer, que es, esencialmente, lo que provoca que los servidores dejen de responder.

En cuanto a **Performance**, la *latencia* y el *completion time* se degradan muy rápidamente en los tres casos. Podría mejorarse escalando en mayor profundidad, tanto horizontal como verticalmente.

#### 1.4.3. Endurance test

Para los tres servidores, se implementa un escenario de mandar la cantidad de requests constantes (10req/s) durante 300 segundos.

##### 1.4.3.1 Gunicorn

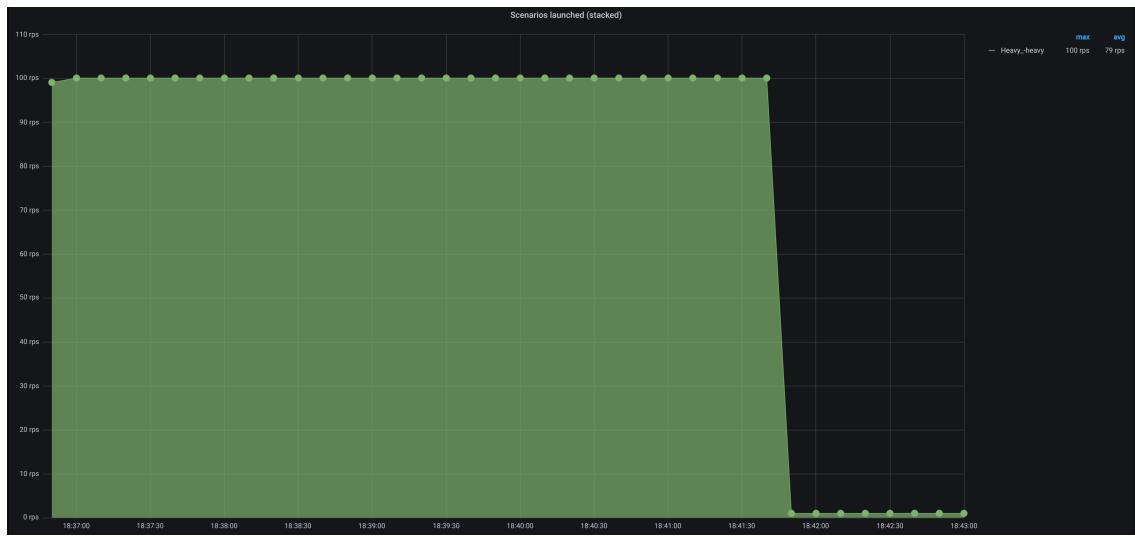


Figura 52: Scenarios launched (stacked) - Endurance Test - Gunicorn

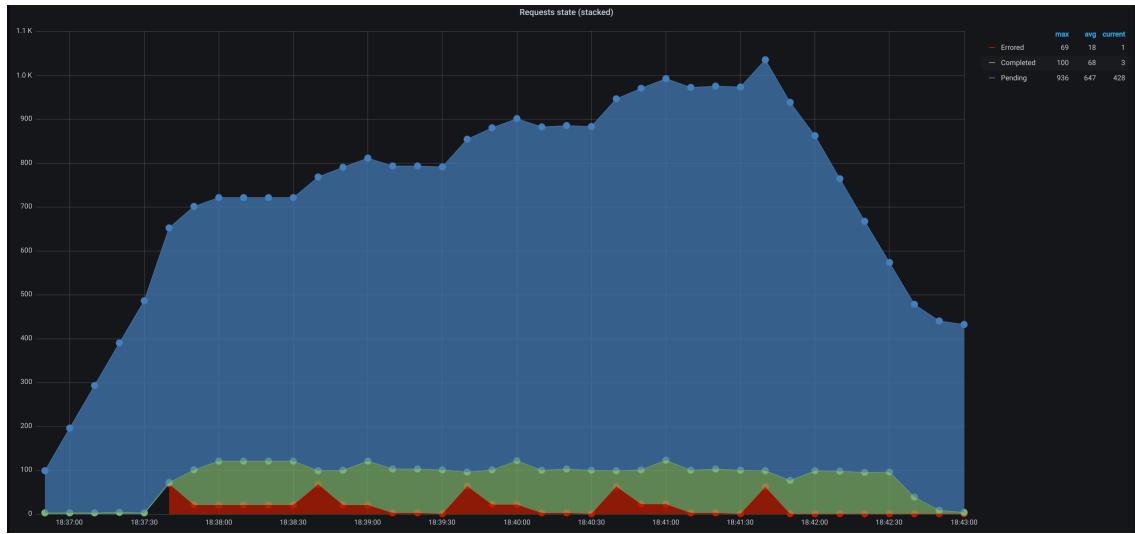


Figura 53: Requests state (stacked) - Endurance Test - Gunicorn



Figura 54: Resources usage - Endurance Test - Gunicorn

```
All virtual users finished
Summary report @ 18:42:54(-0300) 2020-11-10
Scenarios launched: 3000
Scenarios completed: 2574
Requests completed: 2574
Mean response/sec: 7.86
Response time (msec):
  min: 3006.9
  max: 91588.8
  median: 60001.9
  p95: 60009.1
  p99: 63031.4
```

```

Scenario counts:
  Heavy (/heavy): 3000 (100 %)
Codes:
  200: 20
  504: 2554
Errors:
  ECONNRESET: 426

```

Nuevamente, se observa una clara analogía con el *Endurance Test* del endpoint *Sleep*. El servidor responde algunos requests satisfactoriamente durante 1 min 10 segundos, momento en el cual lanza errores de que no puede encolar más requests, y 10 segundos después, el servidor no responde (504, timeout).

Al fin y al cabo, el servidor responde únicamente 20 requests satisfactoriamente, y lanza 2554 errores de tipo 504, y 425 ECONNRESET.

Respecto al tiempo de respuesta percibido por el usuario, va degradando hasta que comienzan a aparecer errores de timeout, donde se mantiene a un valor constante correspondiente justamente al timeout..

Como en los escenarios anteriores, el consumo de CPU es del 100 % y se mantiene estable. Se consume muy poca memoria.

#### 1.4.3.2 Node

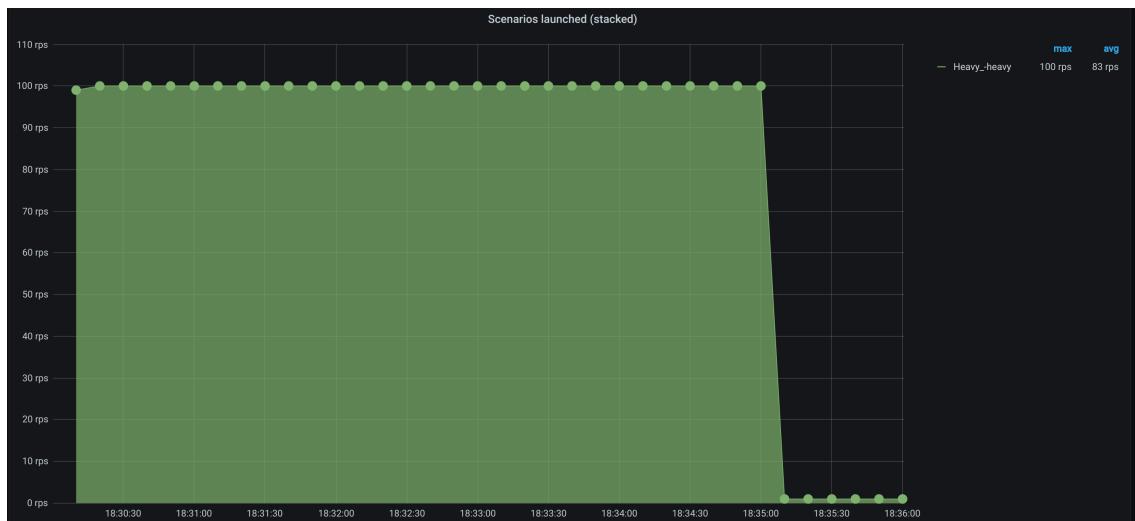


Figura 55: Scenarios launched (stacked) - Endurance Test - Node

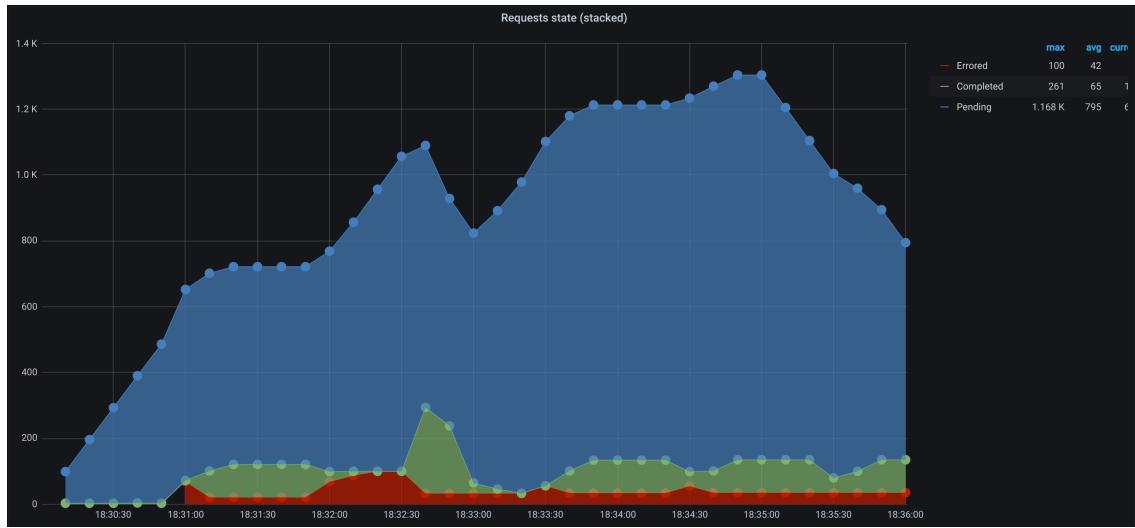


Figura 56: Requests state (stacked) - Endurance Test - Node



Figura 57: Resources usage - Endurance Test - Node

```
All virtual users finished
Summary report @ 18:35:59(-0300) 2020-11-10
Scenarios launched: 3000
Scenarios completed: 2342
Requests completed: 2342
Mean response/sec: 8.34
Response time (msec):
  min: 3011.8
  max: 91626.8
  median: 60000.9
```

```

p95: 91418.1
p99: 91585.1
Scenario counts:
  Heavy (/heavy): 3000 (100%)
Codes:
  200: 20
  504: 2322
Errors:
  ECONNRESET: 658

```

A partir de los resultados obtenidos, se observa que el endpoint, ante una carga constante no muy grande, comienza a fallar al mismo tiempo que gunicorn, primero retornando errores ECONNRESET (debido a la alta cantidad de requests encolados), para luego dejar de responder (504) por el resto del escenario, afectando por completo la disponibilidad del servidor.

El tiempo de respuesta percibido por el usuario se degrada rápidamente a medida que se encolan las requests, con valores muy lejanos a los ideales, mostrando una muy mala performance. También, se puede ver que hay dos intervalos de tiempos en que el servidor no completa requests en absoluto.

El consumo de CPU es máximo y se mantiene en ese nivel estable durante todo el escenario.

#### 1.4.3.3 Node replicado

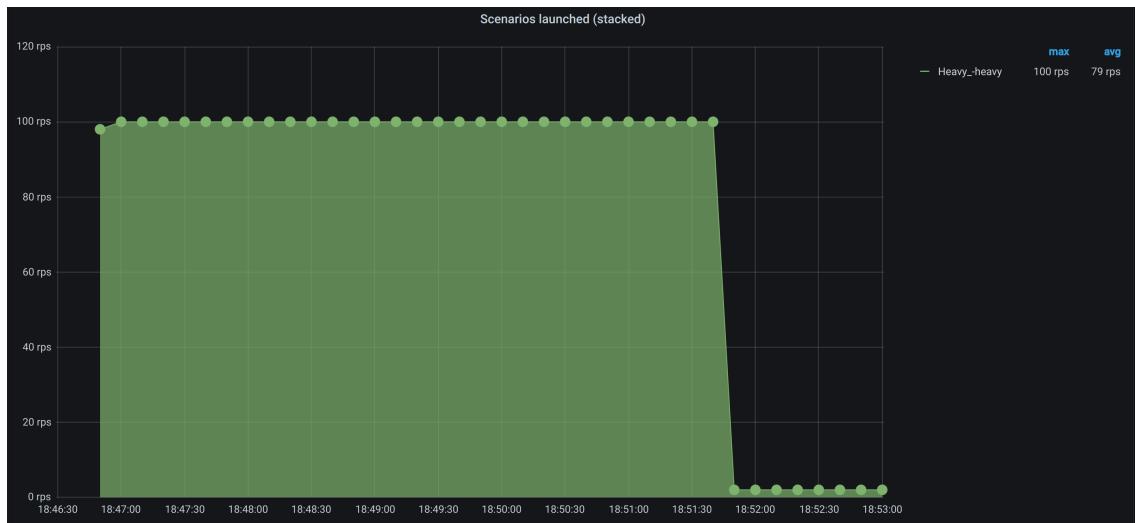


Figura 58: Scenarios launched (stacked) - Endurance Test - Node Replicated

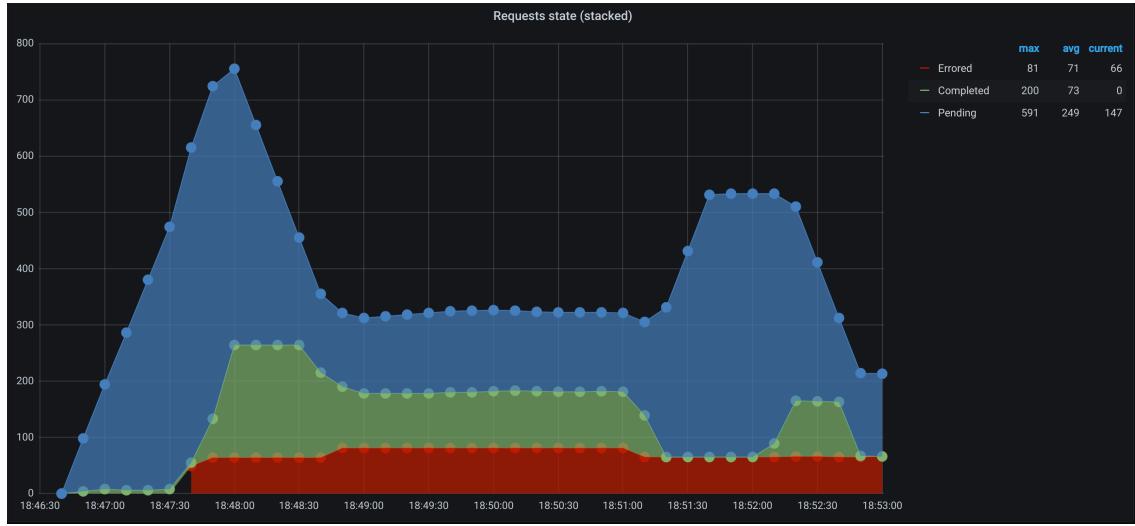


Figura 59: Requests state (stacked) - Endurance Test - Node Replicated



Figura 60: Resources usage - Endurance Test - Node Replicated

```
All virtual users finished
Summary report @ 18:53:35(-0300) 2020-11-10
Scenarios launched: 3000
Scenarios completed: 2853
Requests completed: 2853
Mean response/sec: 7.23
Response time (msec):
  min: 0.6
  max: 120065.4
  median: 3.6
  p95: 60010.3
  p99: 60045.1
```

```
Scenario counts:  
  Heavy (/heavy): 3000 (100 %)  
Codes:  
  200: 42  
  502: 2811  
Errors:  
  ECONNRESET: 113  
  ESOCKETTIMEDOUT: 34
```

Nuevamente, el servidor se cae al desbordarse la cantidad de requests encoladas. El resto del análisis se repite y se omite por redundancia.

#### 1.4.3.4 Conclusión

Después de haber analizado los tres casos, se puede concluir que en todos los casos emergen las primeras fallas y el servidor deja de responder los requests aproximadamente al mismo tiempo.

Se puede notar que la cantidad de pending requests de node replicados es menor que lo de node sin replicar, al comparar ambas curvas azules.

Con respecto a la **disponibilidad**, los tres servidores presentaron sucesivas fallas, en primer lugar se produjeron fallas de timing, degradándose de manera muy rápida el tiempo de respuesta, luego, aparecieron fallas de omisión, en las que el sistema no responde a ciertos requests, para luego producirse un crash, en el que el sistema sufre omisiones repetitivas y deja de responder.

En cuanto a **performance**, los tres servidores resultados idénticos, todos degradando el tiempo de respuesta percibido por el usuario a medida que se encolan más requests. Node y Node Replicated tuvieron dos intervalos de tiempo en que el servidor no respondió en absoluto.

Los consumos de CPU fueron muy altos y estables en todos los casos. En lo que respecta al uso de memoria, estos fueron bajos en todos los casos.

## 2. Sección 2

### 2.1. Sincrónico / Asincrónico

Para determinar la naturaleza sincrónica o asincrónica de los servicios brindado se ha utilizado el siguiente escenario de artillery.

```
phases:  
- name: Reset  
duration: 20  
arrivalRate: 0  
- name: Ramp  
duration: 120  
arrivalRate: 0  
rampTo: 20  
- name: Reset  
duration: 20  
arrivalRate: 0
```

El mismo cuenta con una sola rampa de 2 minutos que va desde 0 a 20 req/s

### 2.1.1. Primer servicio



Figura 61: Requests y tiempo de respuesta para el primer servicio

### 2.1.2. Segundo servicio

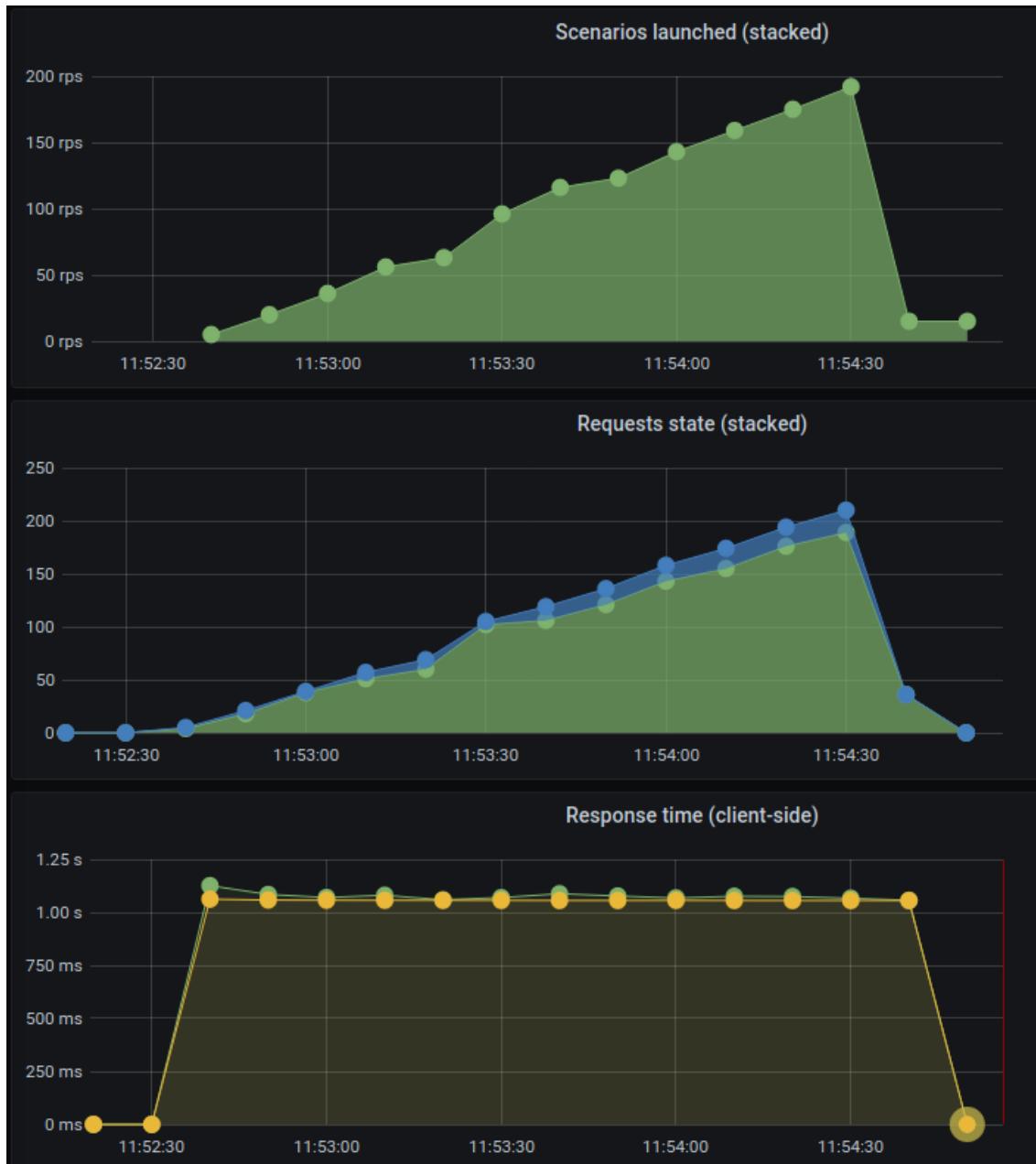


Figura 62: Requests y tiempo de respuesta para el segundo servicio

### 2.1.3. Conclusión

Se puede observar que el primer caso es el servidor sincrónico, mientras que el segundo caso es el servidor asincrónico, ya que se puede notar como en el caso sincrónico se llega a un punto que las request no llegan a ser procesadas en un tiempo relativamente constante, esto se debe a que los workers no pueden aceptar mas request y estas empiezan a quedar en espera para ser procesadas mas tarde. No es así el caso del servicio asincrónico donde se puede ver que las peticiones

son procesadas correctamente por el event loop, el cual no está siendo bloqueado y puede seguir atendiendo nuevos requests.

## 2.2. Cantidad de workers en el servicio sincrónico

Para averiguar el numero de workers del primer servicio se utilizó Apache Bench.

Se comenzó subiendo la cantidad de request concurrentes de 1 en 1, notandose que en general los valores devueltos son similares hasta los 16 request concurrentes.

```
Concurrency Level: 16
Time taken for tests: 1.406 seconds
Complete requests: 16
Requests per second: 11.38 [#/sec] (mean)
Time per request: 1406.075 [ms] (mean)
Time per request: 87.880 [ms] (mean, across all concurrent requests)

Connection Times (ms)
    min mean[+/-sd] median max
Connect: 0 0 0.1 0 1
Processing: 1402 1404 0.9 1403 1405
Waiting: 1402 1404 0.9 1403 1405
Total: 1403 1404 0.9 1404 1406

Percentage of the requests served within a certain time (ms)
 50% 1404
 66% 1404
 75% 1405
 80% 1405
 90% 1406
 95% 1406
 98% 1406
 99% 1406
100% 1406 (longest request)
```

En cambio, al probar 17 request concurrentes obtuvimos lo siguiente:

```
Concurrency Level: 17
Time taken for tests: 2.804 seconds
Complete requests: 17
Requests per second: 6.06 [#/sec] (mean)
Time per request: 2803.643 [ms] (mean)
Time per request: 164.920 [ms] (mean, across all concurrent requests)

Connection Times (ms)
    min mean[+/-sd] median max
Connect: 0 0 0.2 0 1
Processing: 1402 1486 339.2 1404 2803
Waiting: 1402 1486 339.2 1404 2803
Total: 1403 1487 339.2 1405 2803

Percentage of the requests served within a certain time (ms)
 50% 1405
 66% 1405
 75% 1405
 80% 1405
```

```

90% 1406
95% 2803
98% 2803
99% 2803
100% 2803 (longest request)

```

Notamos que al incrementar en 1 la cantidad de request concurrentes el tiempo de respuesta máximo aumenta considerablemente respecto de la mediana, y mirando los valores del percentil 90 y 95 podemos decir que una sola de las requests fue la que no se mantuvo en los mismos valores que las demás. Por lo que se concluye que la cantidad de workers que se está utilizando son **16**.

### 2.3. Demora en responder

```

Server Hostname: localhost
Server Port: 9090

Document Path: /
Document Length: 12 bytes

Concurrency Level: 17
Time taken for tests: 79.826 seconds
Complete requests: 900
Failed requests: 0
Total transferred: 69300 bytes
HTML transferred: 10800 bytes
Requests per second: 11.27 [#/sec] (mean)
Time per request: 1507.819 [ms] (mean)
Time per request: 88.695 [ms] (mean, across all concurrent requests)
Transfer rate: 0.85 [Kbytes/sec] received

Connection Times (ms)
    min mean[+/-sd] median max
Connect: 0 0 0.2 0 2
Processing: 1400 1488 328.9 1402 2801
Waiting: 1400 1488 328.9 1402 2801
Total: 1401 1488 328.9 1402 2802

Percentage of the requests served within a certain time (ms)
 50% 1402
 66% 1404
 75% 1404
 80% 1405
 90% 1410
 95% 2754
 98% 2770
 99% 2777
100% 2802 (longest request)

```

Podemos ver que en este servicio la mediana de respuesta es de **1402 ms**, pero con una desviación estándar de 328.9 ms cuando superamos en 1 la cantidad de request máximo paralelos que puede resolver, lo que hace que algunas request demoren hasta aproximadamente 3000 ms.

```

Server Hostname: localhost
Server Port: 9091

```

```
Document Path: /
Document Length: 12 bytes

Concurrency Level: 1000
Time taken for tests: 53.838 seconds
Complete requests: 50000
Failed requests: 0
Total transferred: 3850000 bytes
HTML transferred: 600000 bytes
Requests per second: 928.70 [#/sec] (mean)
Time per request: 1076.770 [ms] (mean)
Time per request: 1.077 [ms] (mean, across all concurrent requests)
Transfer rate: 69.83 [Kbytes/sec] received

Connection Times (ms)
      min mean[+/-sd] median max
Connect: 0 47.7 0 1025
Processing: 1050 1058 35.7 1053 2083
Waiting: 1047 1058 35.2 1053 2083
Total: 1050 1062 60.7 1054 2089

Percentage of the requests served within a certain time (ms)
 50% 1054
 66% 1056
 75% 1058
 80% 1059
 90% 1064
 95% 1097
 98% 1129
 99% 1155
100% 2089 (longest request)
```

Para este caso, se pudo manejar hasta 1000 conexiones concurrentes sin mayores variaciones, se mantuvo una mediana de **1053 ms**. Sólo 100 request tardaron más del 1155 ms.

### 3. Sección 3 [OPC]