Trabajo Práctico 2

Facultad de Ingeniería, Universidad de Buenos Aires
[75.73] Arquitectura de Software

Grupo "Ladrillo"

100029 del Mazo, Federico
102264 Hojman, Joaquin
97112 Kasman, Lucía
97131 Rombolá, Juan Pablo
103409 Schmidt, Agustina

23 de Noviembre de 2022

Trabajo Práctico 2

En este trabajo buscamos hacer un análisis de un sistema en distintas configuraciones, intentando encontrar el cuello de botella que lo limita.

Específicamente, nuestro sistema se compone de un cluster de instancias de aplicaciones de `express` (`node.js`) que a su vez llaman a un servidor de `gunicorn` (`python`).

```
# Pingueamos a nuestro Azure Virtual Machine Scale Set
# Este cluster funciona como load balancer y delega el ping a alguna de las instancias
↪  de node.js
$ curl "http://ladrillo-fdm.eastus.cloudapp.azure.com"
Hello World!

# Pingueamos al VMSS, pero esta vez le pedimos a `/remote` en vez de `/`
# Este endpoint hace un llamado externo al servidor de gunicorn
$ curl "http://ladrillo-fdm.eastus.cloudapp.azure.com/remote"
{"id":1}
```

Las tres configuraciones que queremos analizar son:

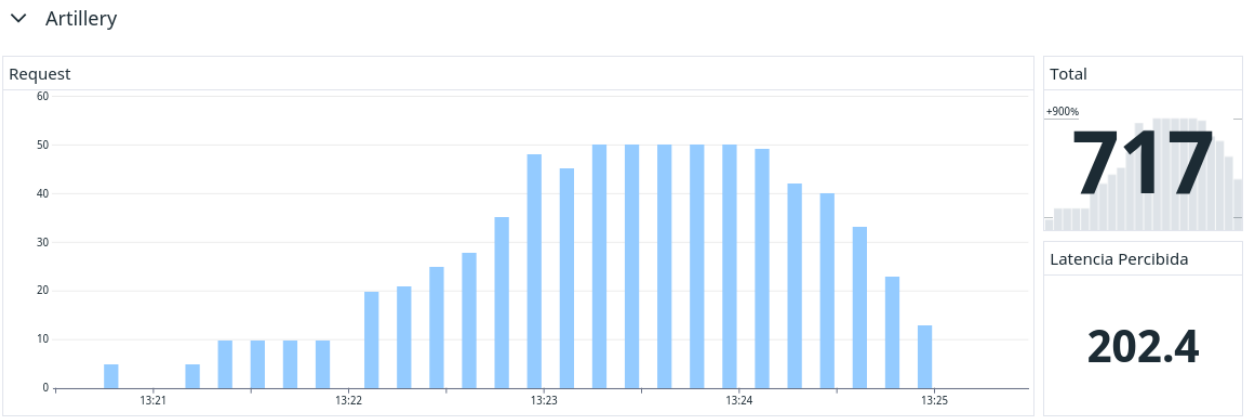
- ¿Qué pasa si solo tenemos una instancia de `node`?
- ¿Qué pasa si el cluster delega a 3 instancias de `node` balanceando la carga?
- ¿Qué pasa si a una única instancia de `node` le agregamos una cache de `Redis`?

Nuestras pruebas consisten de correr el mismo escenario de `artillery` para todas las configuraciones de nuestro sistema, para así tener puntos de comparación.

```
# Llamamos al escenario de artillery sobre el endpoint `/remote`
# El escenario consiste de correr `ping.yaml` el cual es nada más unos llamados a `/`
↪  para ver cuanta latencia estamos manejando actualmente, y luego correr
↪  `scenario.yaml` que contiene el flujo principal de WarmUp + RampUp + Plain +
↪  CleanUp
```

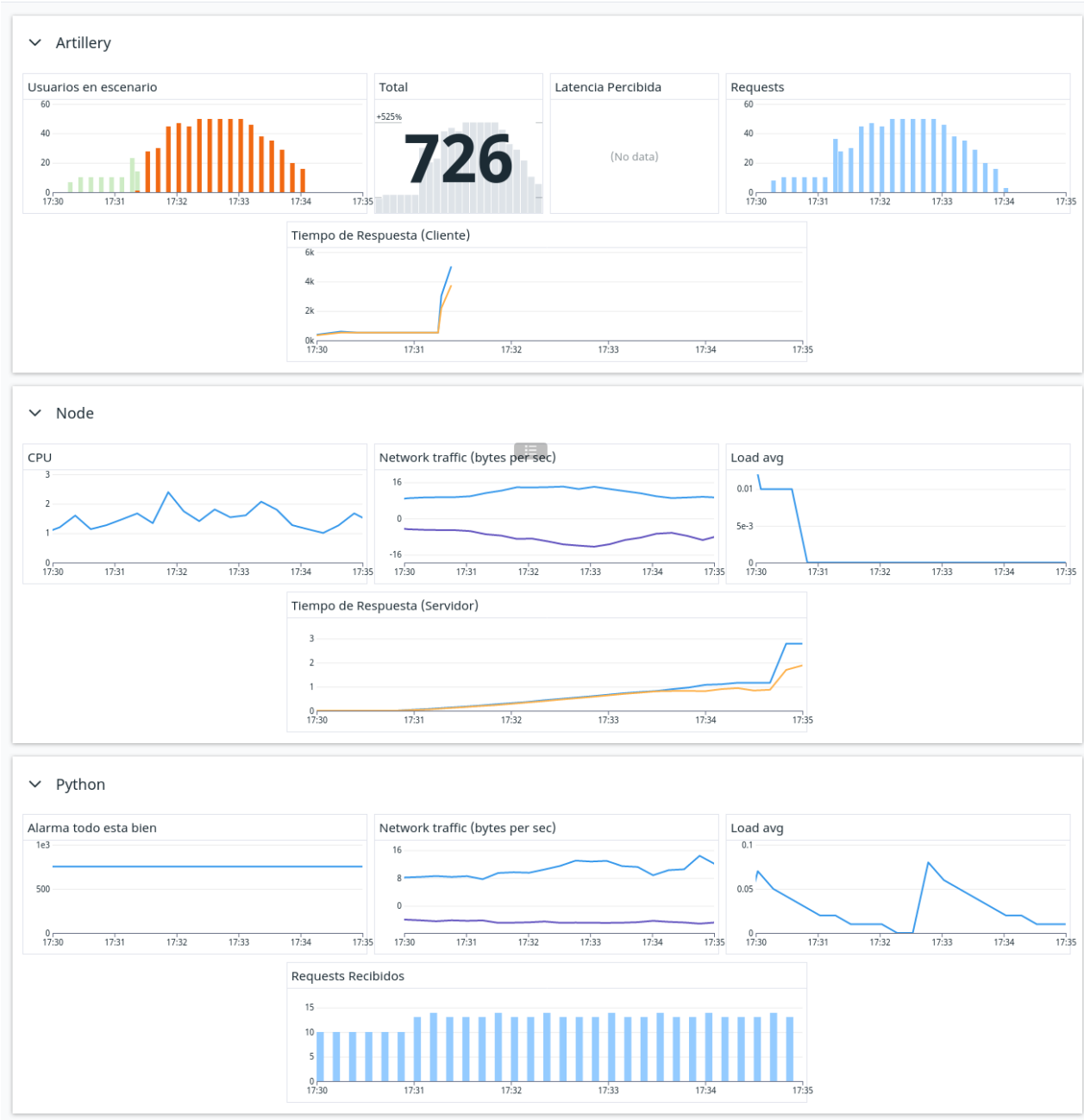
```
$ cd perf
$ ./run.sh "/remote"
All VUs finished. Total time: 4 minutes, 16 seconds

--------------------------------
Summary report
--------------------------------

vusers.created: ..................................................................... 715
http.requests: ...................................................................... 715
http.request_rate: .................................................................. 2/sec
```

El escenario corrido finalmente se refleja en Datadog, ya que todos los componentes de nuestro sistema tienen distintos agentes que reportan métricas. A partir de ello, nos daremos una idea de dónde estan los distintos cuellos de botella que limitan nuestro sistema.



Escenario de artillery a correr sobre todas las configuraciones: `WarmUp` + `RampUp` + `Plain` + `CleanUp`

Para el análisis hicimos un *dashboard*, para ver cómo funciona y se relaciona cada componente:



Dashboard global del sistema

- Localmente, nos interesan las métricas que envía `artillery`:
  - Los usuarios completados y fallidos nos muestran dónde está el punto de quiebre del sistema.
  - Los requests por segundo nos muestran el patrón que armó nuestro escenario.
  - El tiempo de respuesta nos muestra el punto de vista del cliente, que nos sirve para compararlo con el resto del sistema.
  - El número total de requests nos ayuda a confirmar que estamos efectivamente analizando un escenario entero (en vez de uno parcialmente, o el fin de uno y el comienzo de otro).
  - La latencia percibida nos da una gran idea de cuánto estamos perdiendo en el trayecto desde la

computadora local hasta la instancia de la VMSS, porque surge de llamados a `/` y no a `/remote`. Es decir, al restarle este número al tiempo de respuesta a `/remote`, podemos aproximar cuánto está tardando la máquina de `node` en llamar a la máquina de `python`.

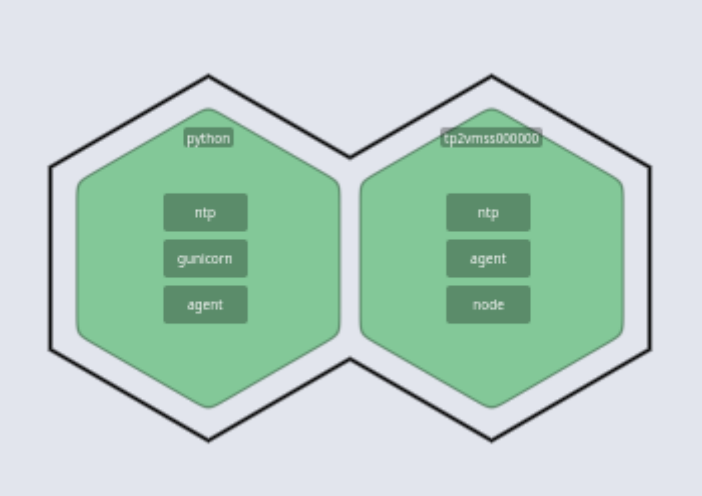
- Luego tenemos las métricas de una de las instancias de `node`:
  - El tráfico de red, el consumo de CPU y el *load average* nos sirven para ver cómo está trabajando la máquina, y así poder buscar dónde se producen los picos y se está saturando.
  - El tiempo de respuesta ahora podemos verlo también desde el servidor, y compararlo con el punto de vista del cliente.
- Finalmente tenemos los gráficos de la máquina de `gunicorn`, de la cual teóricamente no tenemos información ni acceso, pero que aun así nos es funcional al análisis:
  - La "alarma todo está bien" nos muestra que el servicio está funcionando correctamente. Viendo el código sabemos que todos los pedidos tienen un `sleep(0.75)`, es por eso que este gráfico *siempre* debe ser una línea de 750 milisegundos, con o sin cortes intermedios.
  - El tráfico de red y el *load average* de esta máquina cumplen el mismo propósito de las instancias de `node`.
  - Los requests recibidos nos sirven para ver si efectivamente hubo un llamado a esta máquina: ya que nuestro escenario envía request frecuentemente y sin pausa, un corte nos significaría que nunca hubo un llamado y que el solicitante resolvió el pedido por sí mismo (¡con una cache!).

TO DO: Entonces... ¿Qué pretendemos ver?

- Verde al principio, rojo despues en single
- etc etc etc

Estudio 1 - Node Singular

La configuración a analizar consiste de una sola instancia de la máquina de `node`, sin ningún tipo de cache, que recibe todo lo que se le pregunte a la VMSS y, al ingresar un pedido a `/request`, hace un llamado remoto a la máquina de `gunicorn`.



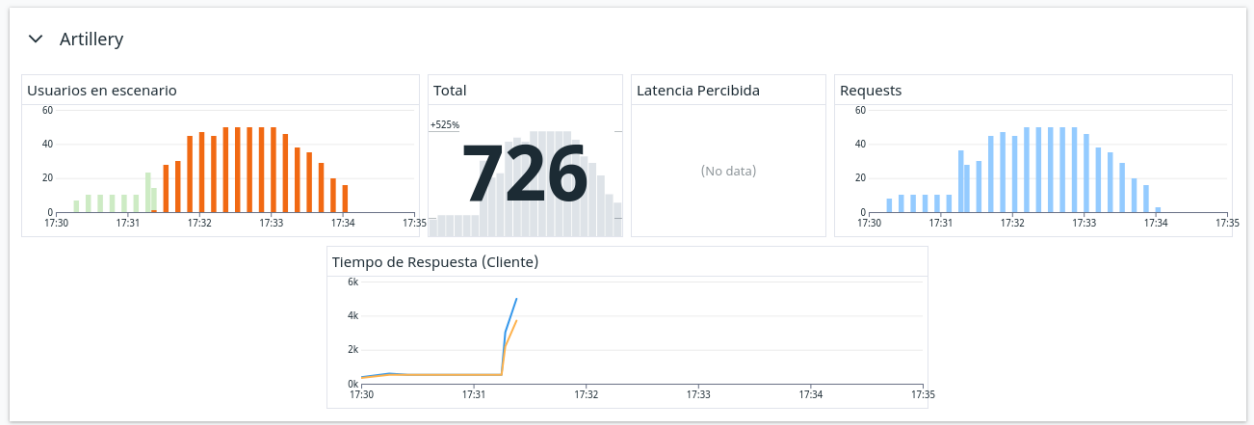
Hosts al tener sólo una instancia de node

Vamos a analizar una corrida de 4 minutos y 16 segundos, de la cual (sin contar los 5 usuarios creados para el `ping` inicial) se completaron exitosamente 93 de los 726 usuarios creados en el escenario.

```
$ ./run.sh "/remote"
All VUs finished. Total time: 4 minutes, 16 seconds

--------------------------------
Summary report @ 17:34:02(-0300)
--------------------------------

errors.ETIMEDOUT: ..................................................... 633
http.codes.200: ....................................................... 93
http.request_rate: .................................................... 3/sec
http.requests: ........................................................ 726
http.responses: ....................................................... 93
vusers.completed: ..................................................... 93
vusers.created: ....................................................... 726
vusers.failed: ........................................................ 633
```

Los usuarios fallidos son todos por el mismo motivo: `ETIMEDOUT`. Por defecto, `artillery` tiene un tiempo de espera de 10 segundos antes de salir con error. Este número nos parece apropiadamente elegido: si un pedido tarda más de 10 segundos, vamos a tomarlo como fallido, y vamos a considerar que estamos acercándonos al punto de quiebre del sistema.

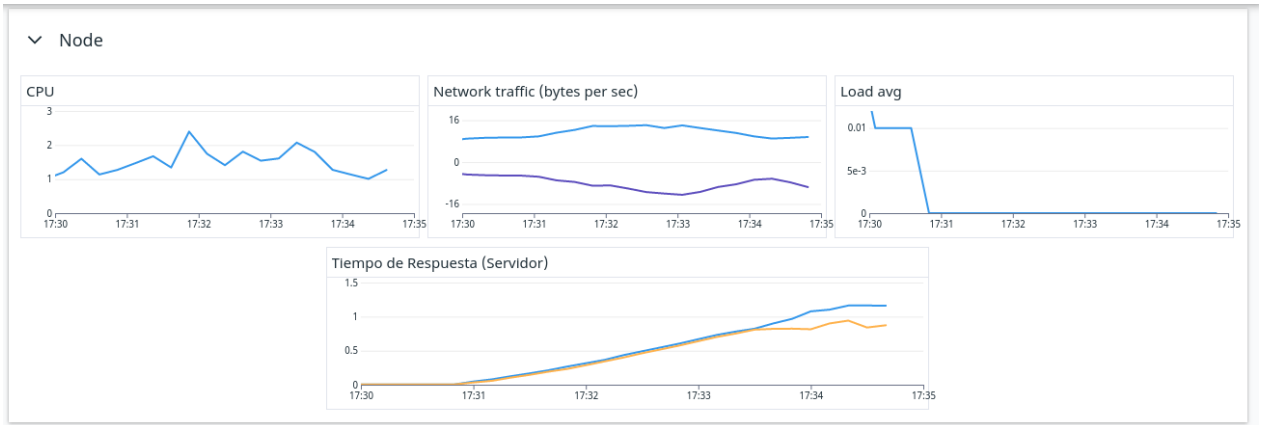
Como aclaración, esto no significa que la instancia de `node` haya dejado de funcionar, sino que simplemente tardó mucho en respondernos. Desde su lado, tranquilamente `node` sigue recibiendo pedidos y llamando al servicio externo. ¡Esto lo podremos chequear más adelante!



Node Singular - Local

Claramente se puede ver que el sistema comienza a fallar en la fase de `RampUp`, de manera casi instántanea: se disparan los tiempos de respuesta, y los usuarios fallidos comienzan a superar enteramente a los usuarios completados.

En esta fase es en donde apenas empiezan a aumentar la cantidad de requests a más de una por segundo. Es decir: **con sólo una instancia, el sistema no tolera más de 2 usuarios por segundo**. Intentar decir "está alrededor de 1.7 requests por segundo" es un análisis complejo: no tiene sentido en la vida real decir que mandamos 1 request y fracción de otro[1], es decir no se pueden fraccionar los requests. Para obtener un valor exacto y que tenga sentido podríamos hablar de cada cuántos segundos mandamos un nuevo request.



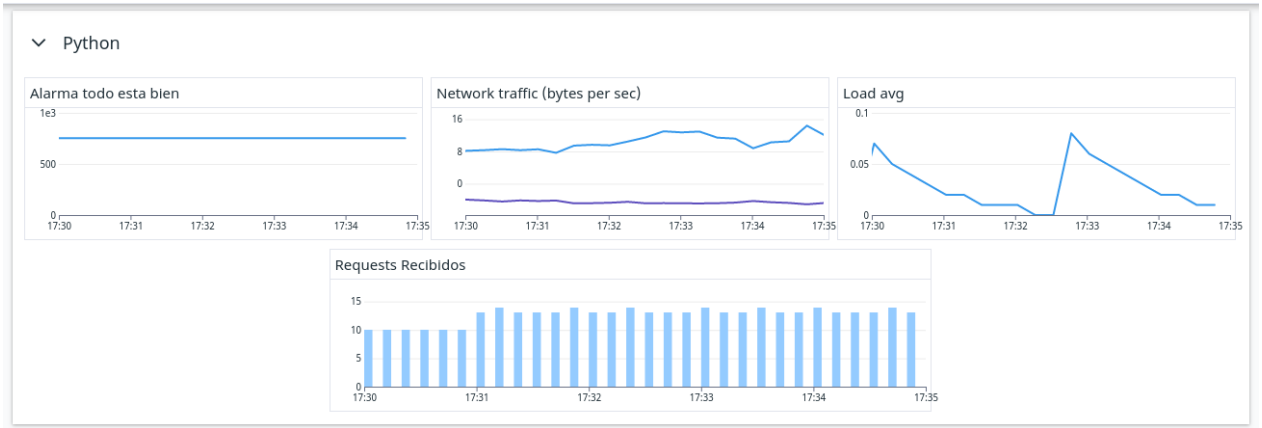
Node Singular - Servidor Node

Desde el punto de vista del servidor, también vemos que el punto de quiebre se encuentra alrededor de las `17:31:30hs`. Es en donde comienza el `RampUp` y se tiene el pico global de uso de CPU.

Lo interesante del tiempo de respuesta (tanto desde el cliente como desde el servidor) es que se puede ver que es lineal en relación al `RampUp`. Comienza constante, y al empezar a recibir más de un usuario, crece linealmente.

El gráfico de tráfico de red nos muestra que la instancia sigue recibiendo información: todavía está llamando al servicio externo, incluso después de haber pasado el `RampUp`. El servicio externo no estaría pareciendo ser el factor limitante, ¡sigue funcionando! El factor limitante parece ser solamente cómo nuestra instancia de `node` maneja pedidos en simultáneo.

[1] Como bien nos enseña el fundador de artillery en la sección de issues



Node Singular - Servicio Externo

Finalmente, podemos ver que desde el servicio externo todo funciona correctamente. Se recibieron todos los requests, se manejaron correctamente,y siempre se mantuvo constante el tiempo de demora de 750ms.

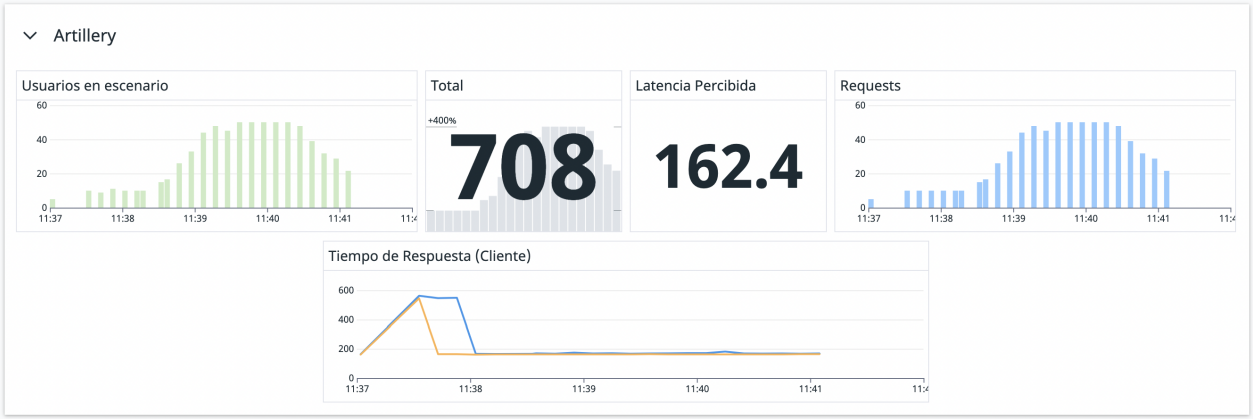
Estudio 2 - Node Singular con Redis

En este escenario, analizamos la configuración de una sola máquina de `node`, pero con un cache de Redis intermedio entre `node` y el servicio externo en `python`. En el Estudio 1 vimos que el cuello de botella se encontraba en el servicio externo, por lo tanto nuestra hipótesis es que agregando una cache intermedia los tiempos de respuesta van a mejorar sustancialmente.

La prueba de Artillery realizada fue la misma que en el Estudio 1, con las fases de ping, warm up, rump up y plain y usando el endpoint `remote/cached`.

Además, cabe aclarar que para estos escenarios utilizamos una configuración de `cacheKeyLength` de 10. Esto define el tamaño de la cache y por tanto la cantidad de keys que va a soportar Redis, por lo que mientras más grande sea, más requests serán necesarias para llenarla. Al ser de tamaño 10, los primeros 10 requests no hacen hit en la cache y siguen de largo hacia el servicio externo, pero a partir del request 11, se empieza a usar la cache.

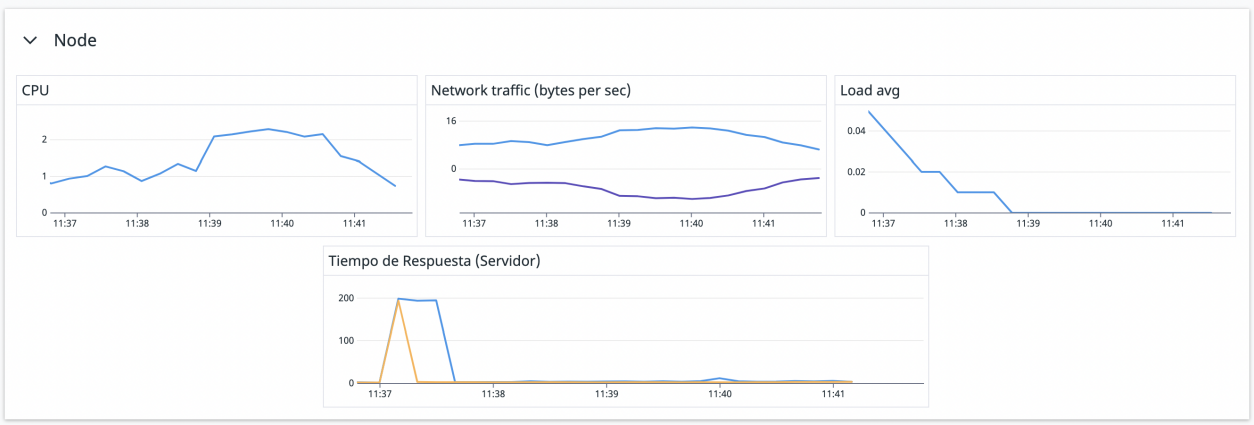
Hosts al tener sólo una instancia de node, con cache de Redis



Node Cache - Local

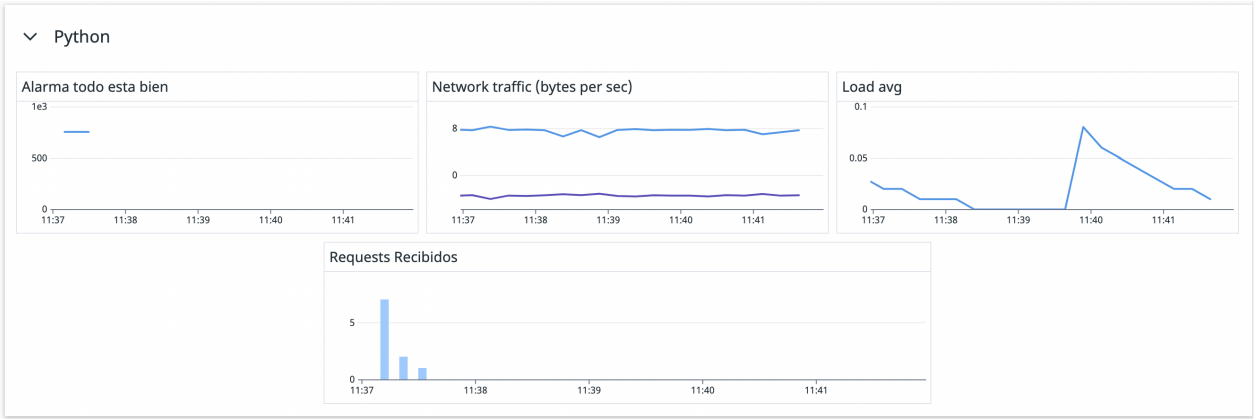
En las métricas del lado de Artillery (usuario), podemos apreciar, en primer lugar, que todos los requests del escenario fueron completados con éxito sin fallas, esto nos da un indicio de que la mejora aplicada fue de utilidad.

Si observamos el tiempo de respuesta, vemos como hay un pico al inicio y después se desciende y se mantiene constante en un nivel muy bajo. El pico se condice con los primeros 10 requests que `node` hace a `python` y no están cacheados aún. El descenso y planicie comienzan partir del siguiente request, donde todos empiezan a ser hit en la caché y no hay necesidad de tener que llegar al servicio externo.



Node Cache - Servidor Node

Observando las métricas del lado del servidor de `node`, vemos que el tiempo de carga promedio y el tiempo de respuesta tienen sus máximos al comienzo, mientras se llena la cache, y luego van en descenso una vez que la cache está completa. Esto es coherente con las métricas de Artillery analizadas anteriormente.



Node Cache - Servicio Externo

Por último, si vemos del lado de `python`, notamos que la cantidad de requests recibidos son 10 en total (7, 2 y 1). Esto confirma el hecho de que estamos usando una cache de tamaño 10 y que una vez llena, todos los requests que saldrían de `node` hacia `python` no se terminan haciendo porque su respuesta ya se encuentra en Redis.

Estudio 3 - Node Replicado x2

Explicar cómo cambió el dashboard: - Hacer un grupo “Cluster de nodes” que tenga las 3 líneas de requests recibidos, cosa de que podamos ver a quién le está delegando el load balancer. “Un gráfico de todas los requests de todas las instancias de `node` nos muestra cómo está funcionando el *load balancer* y si alguna instancia en particular se está saturando más que el resto.”

Hosts al tener tres instancias de node

Para este analisis se utilizaran dos instancias de node, debido a que nuestra cuenta de azure tiene un limite de 4 maquinas virtuales, entonces usaremos mgmt + python + node1 + node2.

VMS de azure

En este caso seguimos utilizando pruebas de artillery, con escenarios de pause, warm up, ramp up y plain y usando el endpoint /remote/. Inicialmente lo que esperariamos ver es un escenario mas parecido al caso 1, donde efectivamente veamos que nuevamente se sobrecargue el sistema debido a la ausencia de cache.

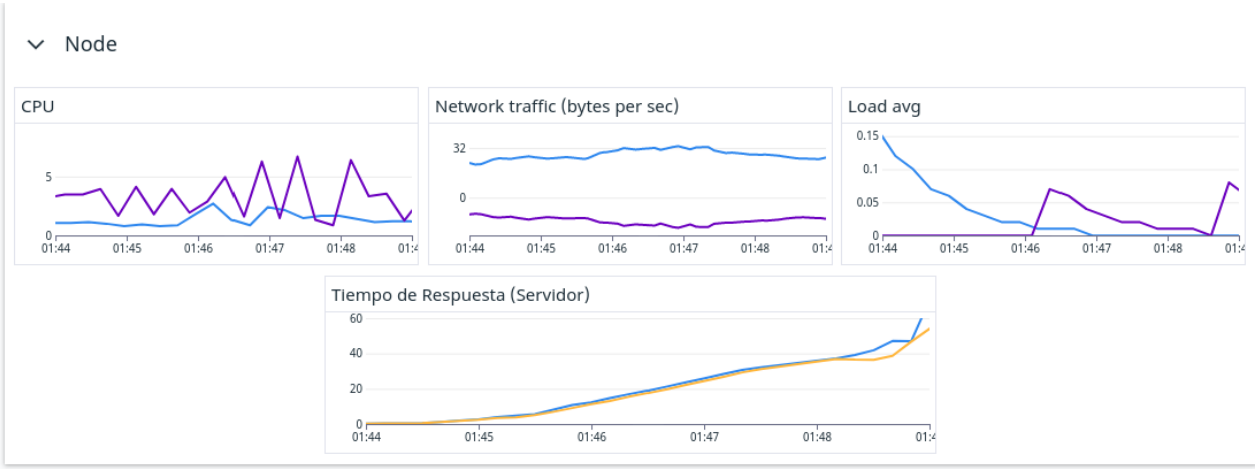
Node Replicated - Artillery

Viendo inicialmente las metricas provistas por artillery, podemos ver que efectivamente se sobrecarga el sistema. Esto podemos notarlo en el grafico de “Usuarios en escenario” ya que durante lo que es la fase de ramp up comienzan a verse usuarios fallidos hasta superar totalmente a los usuarios completados.

A su vez en el grafico de “Tiempo de respuesta” podemos ver como el mismo comienza a aumentar bastante en el mismo momento que comienza a sobrecargarse el sistema con usuarios.

Respecto a los gaficos que habiamos observado cuando usamos una unica instancia de node, podemos ver que la performance no mejora practicamente nada. El motivo de esto es que el cuello de botella que genera la caída de rendimiento y la perdida de requests esta en el servicio externo! Por eso, a pesar de agregar una nueva instancia de node no vemos mejoras, porque el problema nunca fue nuestra instancia de node.

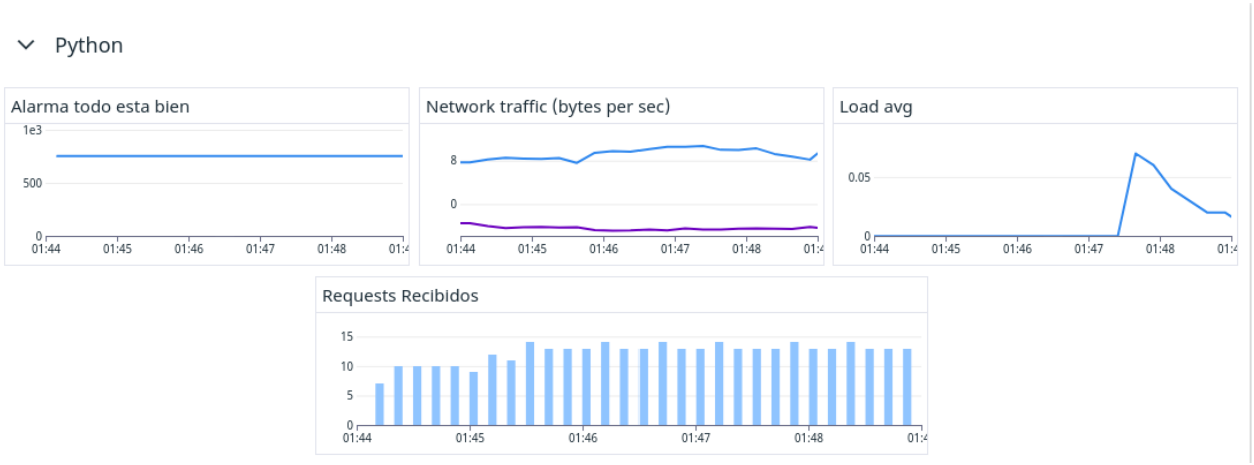
Esta ultima afirmación constata con el caso anterior, donde vimos que al agregar una cache entre nuestra instancia de node y el servicio externo de python, si vimos una gran mejora en el rendimiento.



Node Replicated - Node

Desde el punto de vista del servidor, también vemos que el punto de quiebre se encuentra aproximadamente en el momento en que comienza el RampUp.

Nuevamente el tiempo de respuesta del servidor tiene una forma lineal en relación al RampUp. Comienza constante, y al empezar a recibir más de un usuario, crece linealmente.



Node Replicated - Servicio Externo

En esta imagen, podemos apreciar que en el servicio externo todo funciona correctamente. Se recibieron todos los requests, se manejaron correctamente, y siempre se mantuvo constante el tiempo de demora de 750ms.

Si bien replicar el servidor de node no es la solución al problema analizado en este trabajo, si tiene utilidades. Supongamos el sistema con cache que mostramos en el item anterior, si a ese sistema se le enviaran muchisimas requests por segundo es probable que el mismo colapse pero no por la cache, sino porque la unica replica de node que funciona alli no puede manejar tantos requests (antes de enviarselos al servicio externo). En ese caso, si seria util tener mas replicas de node para poder distribuir la carga entre ellas antes de enviar sus respectivos requests al servicio externo con cache en el medio.

Conclusiones

En el presente trabajo practico pudimos comprender y poner en practica diversos conceptos y herramientas como fueron, servicios cloud, herramientas de monitoreo cloud, una gama de servicios de Azure, y demas.

Nos fue posible entender que se nos estaba pidiendo y como lograrlo, ademas de formular hipotesis sobre los resultados que creimos que ibamos a obtener. Luego tuvimos la oportunidad de verificar dichas hipótesis contra con las metricas que obtuvimos, y conseguimos comprobar algunas y descartar otras.

Pudimos comprender como resolver el problema planteado, lo cual resulto muy provechoso para entender como funcionan los servicios cloud y como se pueden utilizar para resolver problemas de esta naturaleza. Vimos que si bien uno podria suponer que agregar replicas de node parecia la forma de llevar a cabo el problema propuesto, lo que habia que hacer realmente era determinar donde estaba efectivamente el problema, que en este caso era el servicio externo, para poder dirimir asi la mejor forma de solucionar dicha situación.