Trabajo Práctico N 1

75-73 Arquitectura de Software

2C 2018

Fecha de Entrega: 11/10/2018

Integrantes

Apellido, Nombre	Padrón
Cabrera, Jorge	93310
Reyero, Félix	92979
Serra, Diego	92354

Índice

Introducción:

Mediante el presente trabajo práctico presentamos una simple arquitectura compuesta por un load balancer (Ngnix) que distribuye carga a un pool de instancias de docker.

El objetivo es estudiar y entender -mediante la simulación de un cluster en un entorno local- el comportamiento de sus nodos en diferentes escenarios, y mediante distintas cargas de tráfico.

Los escenarios fueron realizados bajo dos implementaciones: Python y NodeJs.

Caso	Descripción
Respuesta Rápida (PING)	Imprime un string por consola. No requiere ningún tipo de procesamiento, por lo que se estima que el costo computacional debe ser bajo.
Retardo sin procesamiento (DELAY)	Imprime un texto por consola luego de un tiempo de espera de 10 segundos.
Procesamiento (CALC)	Devuelve el resultado de una operación que utiliza recursos de la CPU. Se tomó como caso una función con números grandes y uso de la función seno en la misma.

Para todos los casos, se definieron una series de fases que generan carga a diferentes tasas de rpm. Las fases se detallan a continuación:

Fase	Arrival Rate	Duración (segundos)
Simple	50	60
Medium	150	60
Ligth Heavy	300	60
Heavy	600	60

	Artillery	Nginx
Timeout (segundos)	120	60

SPECS COMPUTADORA UTILIZADA	Detalle
Núcleos	4
Procesador	

1. Caso Ping

Este escenario realiza una tarea de rápido procesamiento con un costo computacional bajo: imprime por consola un string y termina la ejecución. La tarea ejecutada en . El cual representa un Healthcheck básico.

Python

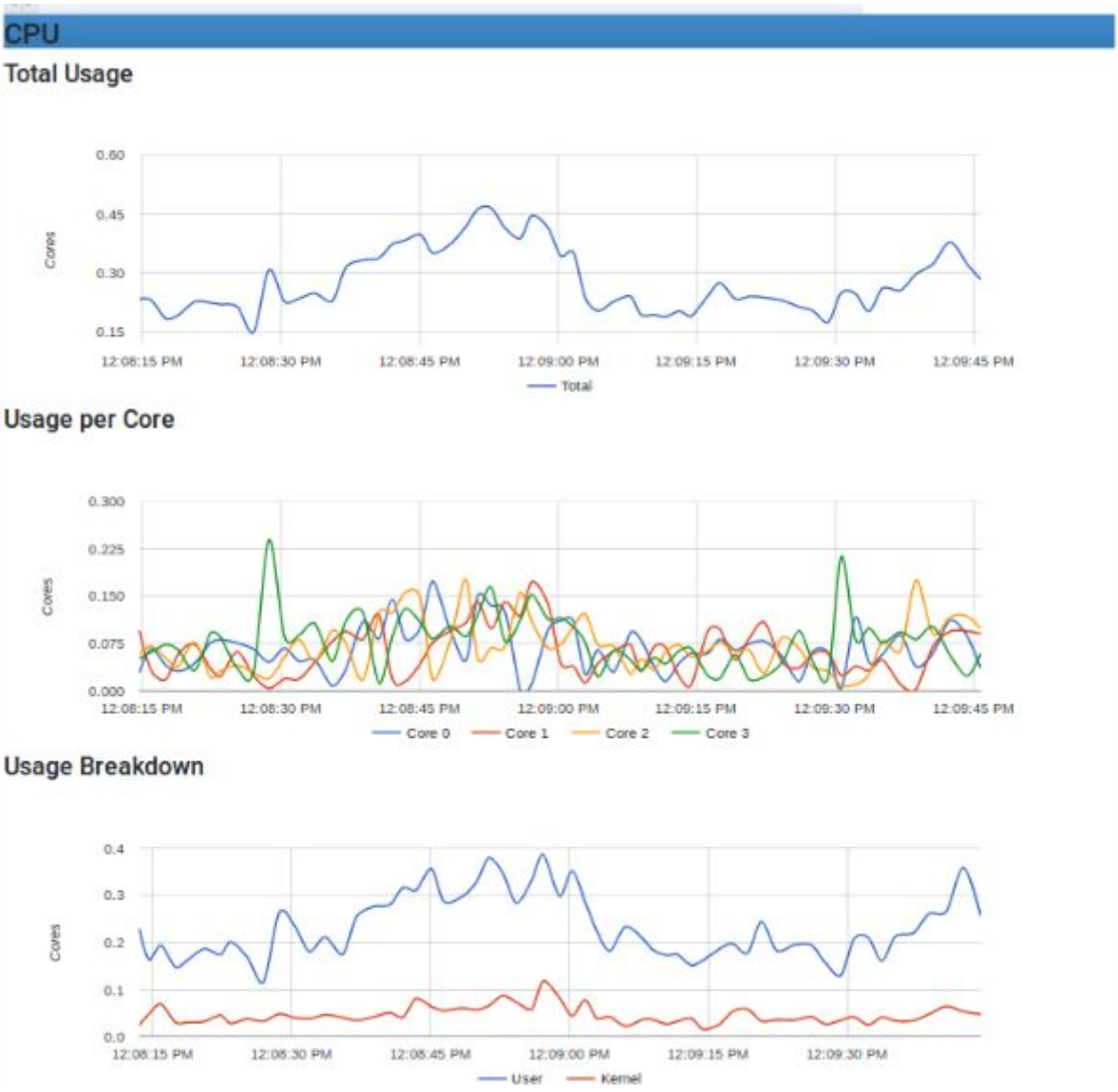


Figura 1.1: recursos utilizados por un container de python

Figura 1.2: procesamiento de un worker de python

Figura 1.3: detalle de procesamiento de un worker de python

Para este caso se ve una relación uno a uno entre llegada de los request y la respuesta a los mismos (los completados).

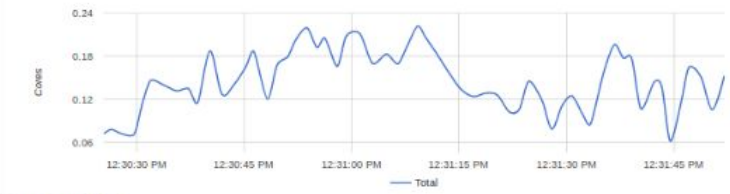
Se observó, en primer lugar que el tráfico que se genera desde artilliery no parece producirse a una tasa constante de rpm. En la medida que llega tráfico a los containers se crean conexiones TCP que generan un aumento en el uso de CPU de la instancia junto con un aumento en la tasa de transferencia de datos. Además se ven picos de response time en los momentos en los que aumenta el throughput.

La memoria total usada por los containers no se ve afectada, esto se debe a que python realiza el procesamiento de los datos en un único thread.

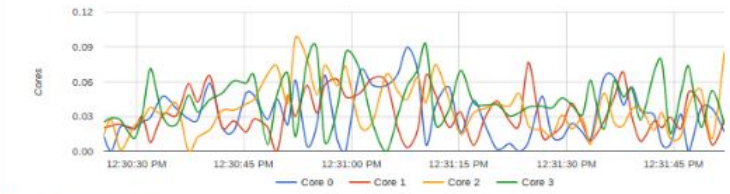
Node

CPU

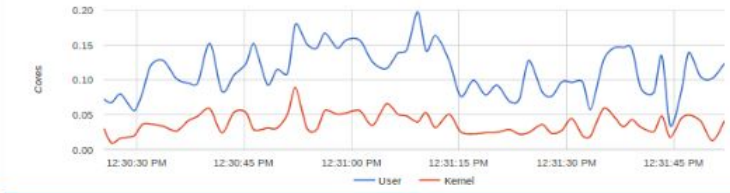
Total Usage



Usage per Core

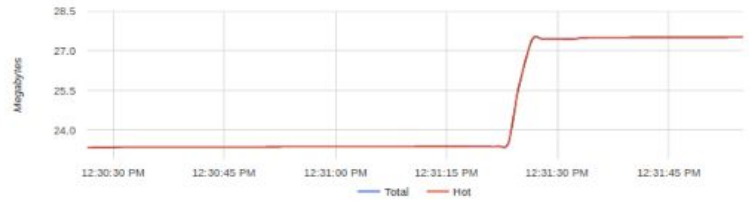


Usage Breakdown



Memory

Total Usage



Usage Breakdown

27.52 MiB / 7.67 GiB (0%)

Network

Interface: **eth0**

Throughput

Errors

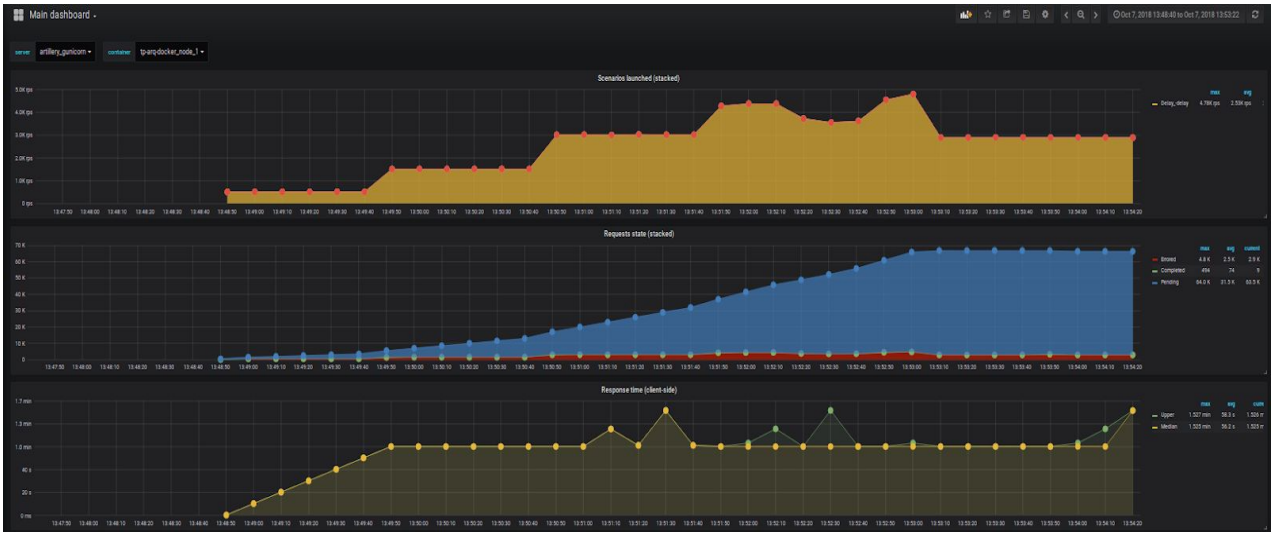
En node, para este primer escenario de prueba la situación tiene algunas similitudes con el escenario visto en python:

- el response time se incrementa de manera directamente proporcional al throughput
- el uso de memoria no se ve afectado de manera significativa.
- a diferencia de python, la latencia para la mayoría de los requests es mucho menor (percentil 99).
- se ve una relación uno a uno entre llegada de los request y procesamiento con éxito de los mismos, es decir la tasa de errores se mantiene en 0[1].
- El uso de CPU también es diferente al de python, teniendo un uso más reducido del mismo.

2. Caso Delay

En este escenario para cada request el webserver[2] ejecutará un sleep de 10 segundos.

Python



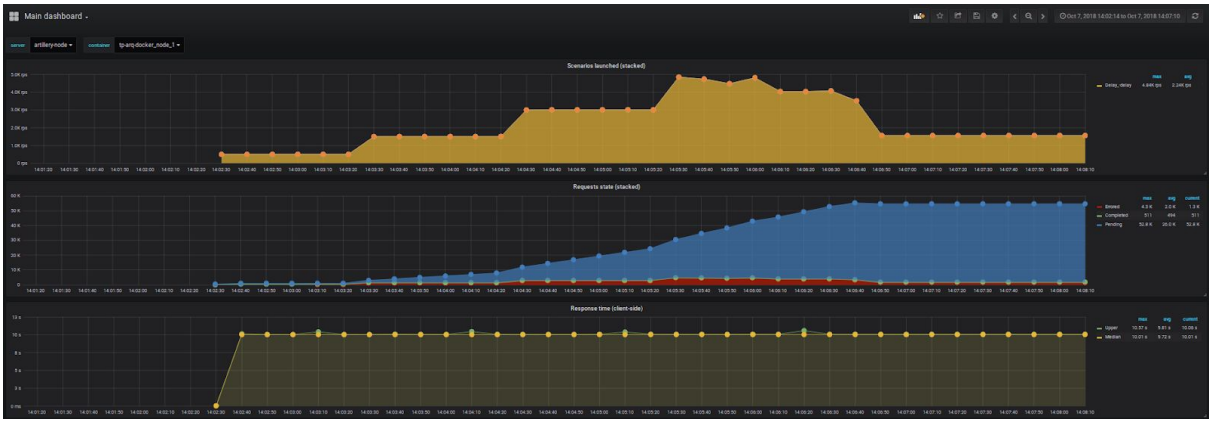
[1] ver figura x.x
[2] cada container de docker puede ser entendido como un webserver

A partir de los resultados obtenidos podemos observar como las request se van encolando en la medida que van llegando al webserver. Esto es una consecuencia directa de contar con una única instancia que atiende la totalidad del tráfico, y potenciado por una tecnología, en este caso python, que no soporta procesamiento concurrente.

El uso total de la memoria es constante, y esto puede deberse a que por más que el tráfico sea irregular y creciente, siempre es el mismo proceso el encargado de atender las solicitudes. Esto provoca que el garbage collector de python siempre esté liberando memoria a una tasa constante.


Memory
Total Usage
28.0
27.5
27.0
26.5
Megabytes
1:51:00 PM
1:51:15 PM
1:51:30 PM
1:51:45 PM
1:52:00 PM
1:52:15 PM
Total
Hot
Usage Breakdown
27.63 MiB / 7.67 GiB (0%)
Network
Interface: eth0
Throughput
16,000
12,000
8,000
4,000
0
Bytes per second
1:51:00 PM
1:51:15 PM
1:51:30 PM
1:51:45 PM
1:52:00 PM
1:52:15 PM
Tx bytes
Rx bytes
Errors
1.0
0.5
0.0
-0.5
-1.0
Errors per second
1:51:00 PM
1:51:15 PM
1:51:30 PM
1:51:45 PM
1:52:00 PM
1:52:15 PM
Tx
Rx

Node


Main dashboard
Scenarios launched (stacked)
Requests state (stacked)
Response time (client-side)

En este caso la diferencia notoria en relación a python, se puede ver en el gráfico que representa el uso de los cores: naturalmente node, que es una tecnología con soporte de procesamiento concurrente, hace un mejor uso de los múltiples core de la instancia.

Otra diferencia está en el uso de memoria utilizada, que ahora deja de ser una recta constante con pendiente nula, para crecer de manera lineal en las diferentes etapas de rafagas de trafico. Como mencionamos anteriormente python tiene la capacidad de atender múltiples request, esto hace que vaya reservando mayor cantidad de memoria en la medida que procesa más solicitudes. Por este mismo motivo en las fases de mayor tráfico tenemos mayor uso de memoria.

Comparativa:

A diferencia del caso de ping para las dos arquitecturas, aquí se observa que la relación entre la tasa de request entrantes y la cantidad de respuestas con status “200” deja de ser 1 a 1. En otras palabras comienzan a aparecer errores.

En el caso de python al tener un delay de 10 segundos (bloqueante), por cada petición entrante al webserver, muchas requests no pueden ser atendidas y comienzan a ser encoladas. Muchas de estas no son atendidos en menos de 60 seg, y el loadbalancer corta la ejecución respondiendo con timeout.

En el caso de node se observa[3] que el 99% de los request fueron respondidos en 10 segundos, por lo que se deduce que no se producen bloqueos en esta arquitectura, aunque si se observa una cantidad significante de request con conexiones rechazadas.

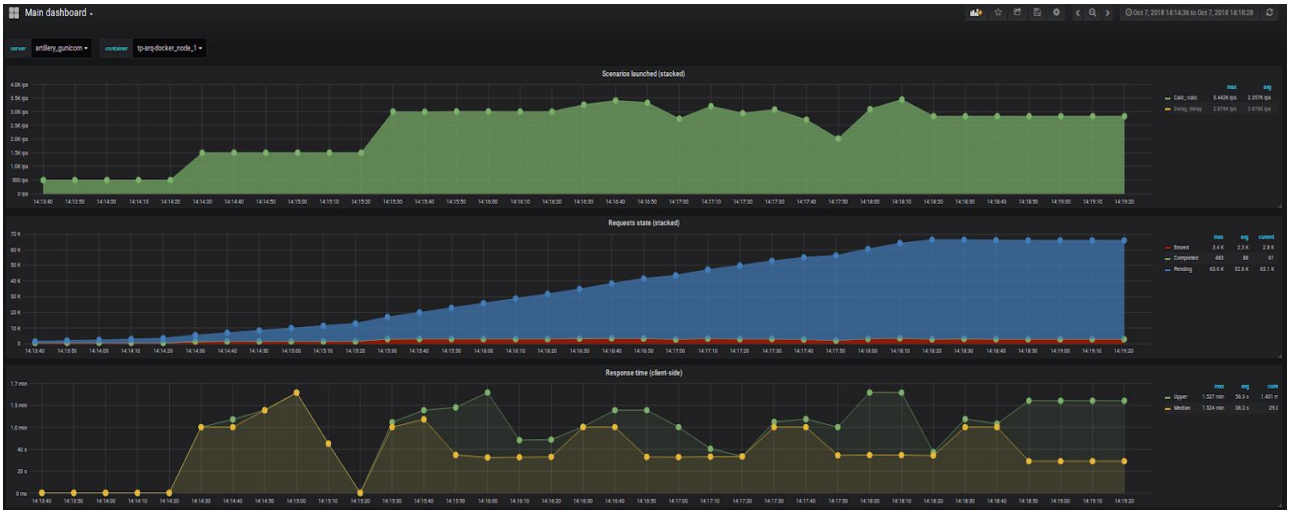
La gran diferencia entre las dos plataformas, se evidencia en la cantidad de request procesados exitosamente siendo que Node acepta solo 15479 del total de request, respondiendo exitosamente a la totalidad de ellos, mientras que python solo acepta 3841 y solo responde exitosamente 5 de ellos.

El costo de la capacidad de Node para procesar múltiples request se ve reflejado en un mayor uso de memoria en comparación con Python, mientras que el beneficio es el mencionado anteriormente: para un mismo escenario de prueba Node logra procesar aproximadamente 4 veces más.

3. Caso calc

Realizamos el cálculo de la función seno para valores entre 5 y 6 millones con una suma adicional para lograr un cálculo con un costo computacional considerable.

Gunicorn



[3] ver en anexo, tabla comparativa de resultados

Lo que se observa es un mayor uso de los core del webserver, y esto se debe a que el container está realizando una operación más compleja.

El uso de memoria es constante y esto se debe a lo mencionado anteriormente, el garbage collector de python va liberando memoria en una proporción constante.

En este caso los request se van encolando porque la proporción de solicitudes entrantes es mucho mayor a la cantidad de solicitudes procesadas: resolver la petición del cliente es muy costosa.

Memory

Total Usage

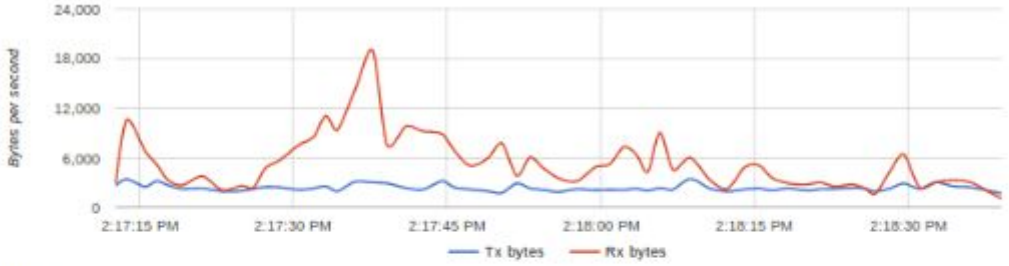
Usage Breakdown

29.36 MiB / 7.67 GiB (0%)

Network

Interface: **eth0**

Throughput

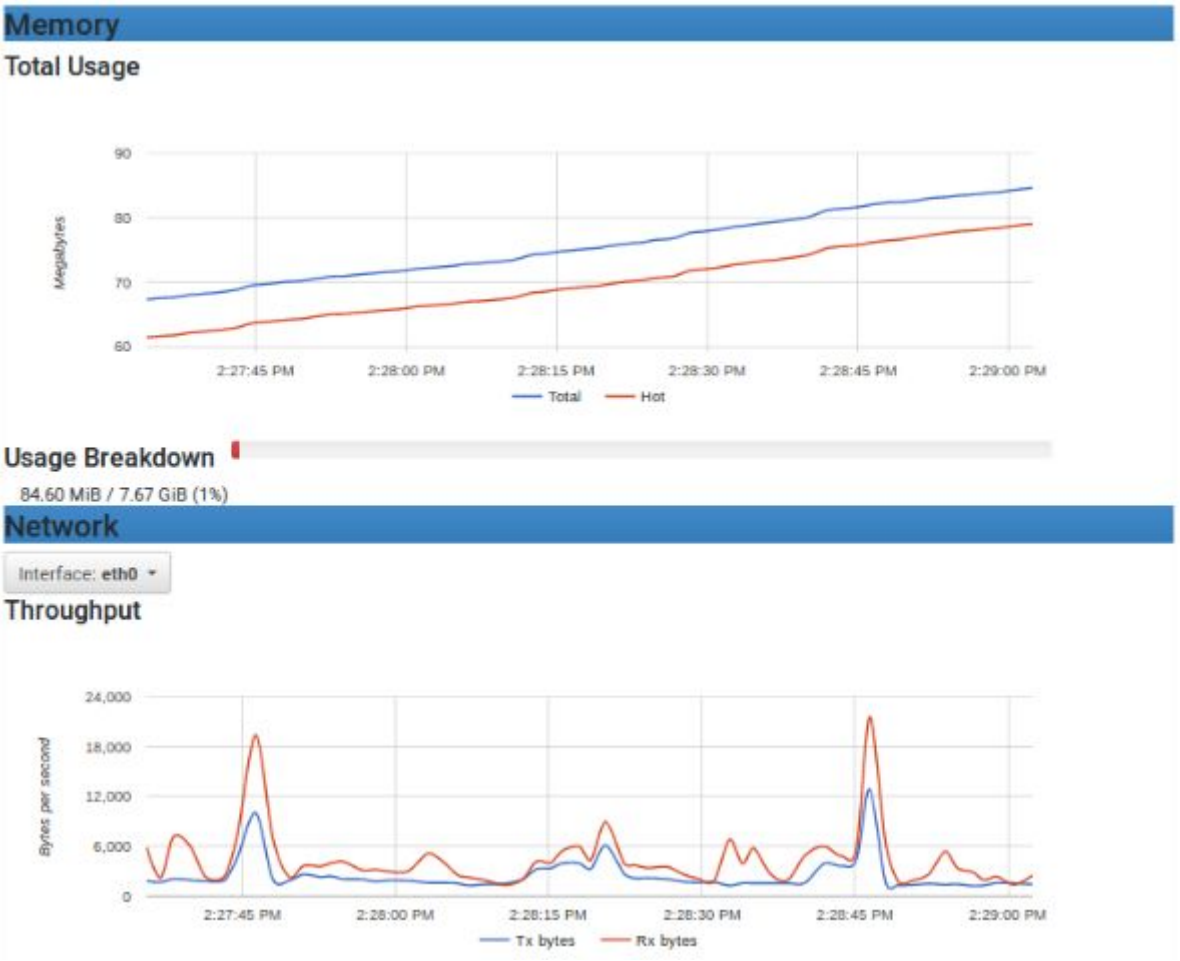


Errors

Node


Main dashboard
Scenarios launched (stacked)
Requests state (stacked)
Response time (client-side)


CPU
Total Usage
Cores
1.08
1.05
1.02
0.99
2:27:30 PM
2:27:45 PM
2:28:00 PM
2:28:15 PM
2:28:30 PM
2:28:45 PM
Total
Usage per Core
Cores
1.2
0.9
0.6
0.3
0.0
Core 0
Core 1
Core 2
Core 3
Usage Breakdown
User
Kernel

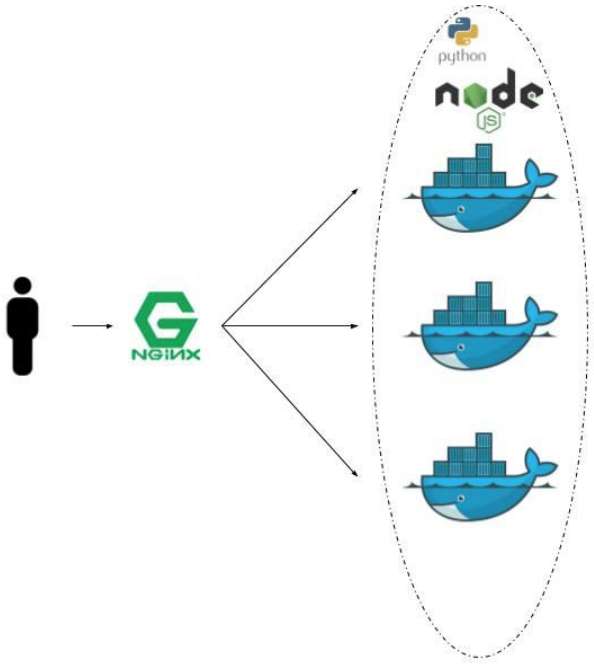


Comparativa:

- Diferencias de consumo de cpu en endpoint calc (node 83% vs python 11%), y para la memoria (90MB node vs 30MB python).
- a diferencia del escenario “delay”, python tiene una mejor performance en relación a node (en cálculo intensivo matemático).
- tienen una tasa de rechazos similar. Pero de los que llegan, python tiene más responses con status 504 (gateway timeout) en comparación a node.

Balanceada

En esta ocasión buscamos simular el comportamiento de un cluster de instancias. El load balancer conoce las distintas instancias que forman parte del pool y distribuye la carga con una política round robin.



Esquema de arquitectura propuesta, para un pool de 3 instancias de docker

Se decidió balancear el escenario Delay en lugar del escenario Ping, ya que para este siempre se obtuvieron respuestas con status 200, por lo que no sería necesario compensar la arquitectura para que mejore.

No se presentan capturas de pantalla para estos casos, solo la salida del artillery, ya que con múltiples containers resulta complicado monitorear el comportamiento de todas las instancias en simultáneo.

Delay Python:

Se verificó una mejora en los siguientes indicadores[4]:

- Para python se pasa de 5 casos exitosos a 36.
- Una disminución de la media de 60015 a 5.4
- Disminución en rechazo del container (econnreset) muy significativa, pero la gran mayoría de estos casos resultan en timeout por parte del load balancer.

[4] ver tabla comparativa en Anexo

Delay Node:

- Disminución en el número de casos exitosos
- Min, median, max y los percentiles se mantuvieron en valores similares..

Calc Python:

- la mínima bajó de 252.8 a 7.5, la máxima se elevó un poco y la median bajo de 60009 a 41015.7
- ambos percentiles se elevaron
- incremento de casos exitosos de 1667 a 2299

Calc Node:

- se elevan casos exitosos de 1033 a 4238

Conclusiones

El presente trabajo práctico nos permite comprender el comportamiento de las instancias de un pool para diferentes escenarios. A partir de los resultados obtenidos podemos ver que en ocasiones escalar el tamaño de un cluster puede mejorar el rendimiento del mismo, principalmente en aquellas situaciones en donde sus nodos están muy saturados. Es importante destacar que el rendimiento va a mejorar siempre y cuando el load balancer distribuya la carga de manera uniforme para todas las instancias que forman parte del pool. Como se mencionó antes el rendimiento mejoró notablemente para python en el escenario delay, cuándo pasamos de 1 container a 6 containers, y lo mismo ocurrió para node en el escenario de “calc”. La razón por la cual python mejora el rendimiento en el escenario “delay” cuándo agregamos instancias, es porque el principal motivo de su mal desempeño es su incapacidad en resolver múltiples request al mismo tiempo.

Escalar un cluster no siempre es la solución, incluso cuándo el uso del CPU de los nodos es intensivo. Esto queda claro para el caso de Node en el escenario “Delay” donde se disminuyó la cantidad de respuestas con estatus 200. En otros casos la mejora percibida puede ser insignificante e injustificada, en relación al costo asociado al mismo, como es para el caso de ping.

Por último otro resultado arrojado del análisis está relacionado con la/s tecnología utilizada. Como dijimos anteriormente en algunas ocasiones agregar instancias a un cluster puede solucionar inconvenientes, pero por otro lado la tecnología utilizada sin dudas influye

en los resultados, e incluso un cambio de la misma puede abaratar costos. Pudimos ver que para cálculos matemáticos python parece desempeñarse mejor, y que para procesamiento concurrente node js parece ser una mejor opción. Por ejemplo si vemos el desempeño de node cuándo escalamos el cluster vemos que logramos procesar más request con status 200, pero si lo que se busca es una mejora en el response time claramente no se consiguió, e incluso se empeoro. Esto es así porque al estar menos saturados los nodos pudieron recibir mayor cantidad de request. El costo computacional de procesar una operación matemática compleja en node es siempre la misma, ya sea para un nodo o para mil, escalar el cluster en este contexto no será una solución viable. Habiendo dicho esto concluimos en que el cambio de tecnologías es sin dudas un parámetro a tener en cuenta para mejorar el desempeño de un cluster.

Anexo: Tabla comparativa resultados de Artillery

		Sin balanceo (1 container)						Con balanceo (6 containers)					
		Node			Phyton			Node			Phyton		
endpoint		ping	delay	calc	ping	delay	calc	ping	delay	calc	ping	delay	calc
Request latency	min	0.8	10000.2	252.8	1.2	10030	216.6	-	10000.3	531.6	-	0.4	7.5
	max	579.7	10573.9	91650.1	1631.8	91636.7	91663	-	10472.4	117531	-	120014	119830.3
	median	8.2	10009.8	60010.5	8.2	60015.5	60009	-	10002.7	54709.9	-	5.4	41015.7
	p95	29.1	10028.2	83495.5	33.3	60054.1	67120.1	-	10011.6	59072.3	-	22.7	107639.8
	p99	76.1	10061	91477.6	404.9	75453.6	75485.2	-	10026.9	59805.9	-	60001.5	118277.8
Scenario counts		66015	66015	66015	66015	66015	66015	-	66015	66015	-	66015	66015
Codes	200	66015	15479	1033	66015	5	1667	-	13269	4238	-	36	2299
	502	-			-	-	34	-		-	-	57557	543
	504	-		2133	-	3836	2592	-		-	-	-	
Errors		-			-			-		-	-		
	ECONNRESET	-	50536	62849	-	62174	61722	-	52746	61777	-	8386	61628
	ESOCKETTIMEDOUT											36	1545