Análisis del impacto en la performance de una web app utilizando un lenguaje compilado y un lenguaje interpretado

Brian Edgar Ré, Michele Philippe Ré, Agustín Olmedo

Facultad de ingeniería. Universidad Austral, Argentina
{brian.re,michele.re,agustin.olmedo}@ing.austral.edu.ar
http://www.austral.edu.ar/ingenieria

Resumen Actualmente se desarrollan muchas aplicaciones web y el tiempo de respuesta de las mismas es un factor muy importante. En este trabajo se presenta un análisis de dos stacks de tecnologías utilizando, por un lado, un lenguaje compilado (Golang) y, por otro lado, un lenguaje interpretado (Python). El aspecto a analizar será desde el punto de vista técnico aplicado a un caso de uso de un *e-commerce* (listar elementos aplicando un filtro). Se analizó principalmente el *throughput*. Las pruebas se realizaron en instancias EC2 de AWS que se encontraban dentro del mismo *availability zone*, haciendo peticiones al servidor a través de una IP privada. Además, se analizó el impacto que tiene considerar la latencia entre el cliente y el servidor al realizar peticiones. En conclusión, los resultados demuestran que Golang tiene un *throughput* mejor que Python con una latencia mínima, pero a medida que aumenta la latencia disminuye la diferencia entre ambos lenguajes.

Keywords: Lenguaje compilado, lenguaje interpretado, Golang, Python, AWS, test de performance, throughput, latencia.

1. Introducción

Cada lenguaje de programación posee ciertas características específicas con sus ventajas y desventajas. Por ejemplo, si el lenguaje está pensado para trabajar eficientemente de forma paralela y concurrente, si utiliza un garbage collector, si compila a un lenguaje intermedio, etc. Es por ello que es de gran utilidad comparar las características entre distintos lenguajes de programación y conocer en que es bueno cada lenguaje para poder tomar una decisión fundamentada a la hora de tener que elegir un lenguaje de programación. Una de las principales diferencias que se presenta entre los lenguajes de programación es que pueden ser compilados o interpretados.

La principal diferencia entre un lenguaje compilado y uno interpretado es que el compilado, al no tener que traducir el programa en tiempo de ejecución, se ejecuta más rápido que el interpretado.

Hoy en día, el enfoque que se tiene a la hora de elegir las tecnologías necesarias para llevar a cabo un proyecto de software es el de elegir las tecnologías de acuerdo a las necesidades del proyecto. Por ejemplo, si se va a desarrollar un servicio de mensajería en tiempo real, se va a elegir un lenguaje que sea concurrente, eficiente y a prueba de fallos, mientras que si se va a desarrollar una aplicación web para una *startup*, se va a priorizar un lenguaje que sea fácil de usar y que pueda realizarse una primera versión funcional en poco tiempo. En este último caso, la performance puede ser o no un requerimiento indispensable. En muchas ocasiones, las empresas eligen como tecnologías el stack de Django para desarrollar aplicaciones web debido a la facilidad que se tiene para desarrollar nuevas funcionalidades en poco tiempo (ya que el framework tiene ciertas configuraciones por defecto que ayudan a reducir el tiempo de desarrollo). Sin embargo, en muchas ocasiones, las empresas que eligen este stack deciden migrar sus aplicaciones a un stack que utilice un lenguaje compilado debido a los problemas de performance que tienen. Estos problemas comienzan a impactar fuertemente en el negocio, lo que se traduce en un impacto en los ingresos de la empresa (se tiene que pagar un elevado precio por el uso de servidores, la aplicación funciona con lentitud, etc.).

Por lo tanto, si las empresas deciden migrar sus aplicaciones web de un lenguaje interpretado a uno compilado, la pregunta que se presenta es: ¿cuánta diferencia de performance hay entre una aplicación web escrita en un lenguaje interpretado respecto a una aplicación escrita en un lenguaje compilado? Para responder a esta pregunta compararemos una aplicación web escrita en Go[6] (lenguaje compilado) y la misma aplicación web escrita en Python[7] (lenguaje interpretado).

2. Trabajos relacionados

Existen muchos estudios evaluando y analizando el rendimiento de servidores web. T. Lance et al. [1] evaluaron experimentalmente el impacto en el rendimiento del servidor web de tres tecnologías de contenido dinámico diferentes (Perl, PHP, Java). Los resultados mostraron que los costos de la generación del contenido dinámico puede reducir la tasa máxima de peticiones admitida por un servidor web hasta 8 veces. Mientras tanto, los resultados indicaron que las tecnologías de servidor Java suelen superar a Perl y PHP para la generación dinámica de contenido. T. Suzumura et al. [2] realizaron un estudio exhaustivo de la capacidad de PHP como motor de servicios web tanto en aspectos cualitativos como cuantitativos mientras lo compararon con otros motores de servicios web implementados en Java y C. Los resultados mostraron que PHP como motor de servicio web funciona de manera competitiva con Axis2 Java para servicios web que involucran pequeñas cargas útiles, y lo supera en gran medida para cargas grandes de 5 a 17 veces. Axis2 C funciona mejor que PHP, pero los resultados experimentales demuestran que el rendimiento de PHP está más cerca de Axis2 C con cargas útiles más grandes. N. Togashi et al. [3] implementaron

programas simples de multiplicación de matrices tanto en Go como en Java para analizar las características de concurrencia. La implementación de Java usa Java Thread, y la implementación de Go usa Goroutine y Channel. En los resultados de los experimentos se observó que Go obtuvo un mejor rendimiento que Java en tiempo de compilación y concurrencia. K. Lei et al. [4] compararon Node.js, Python y PHP realizando pruebas sistemáticas objetivas y pruebas realistas de comportamiento de los usuarios especialmente tomando Node.js como punto central a discutir. Llegando a la conclusión que Node.js puede manejar muchos más peticiones que Python y PHP en un periodo de tiempo determinado.

Y. Ueda et al. [5] compararon el lenguaje Go, compilado estáticamente, con dos lenguajes compilados dinámicamente, JavaScript y Java. Los resultados experimentales demostraron que la implementación de Go logró un rendimiento 3.8x y 2.4x mayor que las implementaciones de JavaScript y Java respectivamente, después de un simple ajuste en la configuración del servidor. El análisis detallado indicó que esto se debe principalmente a que Go sufre menos el polimorfismo que JavaScript debido a la escritura estática, y porque el framework web para Go produce menos sobrecarga para procesar los requests de servicio web RESTful que Java. Además argumentan que los lenguajes compilados estáticamente jugarán un papel más importante para las aplicaciones web debido a sus ventajas de rendimiento y también a las nuevas metodologías de integración e instalación continuas que eliminan algunas de las deficiencias en los lenguajes compilados estáticamente sobre los lenguajes compilados dinámicamente y de secuencias de comandos.

Las investigaciones anteriores se centran en la comparación de las tecnologías de desarrollo web. Pero hasta donde sabemos, todos estos estudios no consideran la latencia. Este documento es el primero en comparar el rendimiento de estas dos tecnologías web populares, Golang y Python teniendo en cuenta también el impacto de la latencia.

3. Entorno de pruebas

3.1. Configuración de Hardware

En cada prueba se utilizaron dos máquinas: una como servidor que alojará la aplicación web de uno de los stacks; y otra como cliente que alojará la herramienta K6, la cual realizará las peticiones al servidor.

Las pruebas serán ejecutadas en el cloud de Amazon (AWS) en el mismo *Availability Zone* para que la latencia entre ambas instancias sea mínima, y que las instancias no sean del tipo *burstable* (para evitar que la CPU sea el cuello de botella en las pruebas).

Se utilizaron las siguientes instancias:

- 1 instancia c5.large para el servidor.
- 1 instancia c5.xlarge para el cliente.

Todos los servidores correrán el sistema operativo Ubuntu 18.04. Las instancias que actuarán de servidor ejecutarán una imagen de Docker[1] que contendrá la aplicación web. Cada stack correrá en servidores y clientes distintos.

3.2. Configuración de Software

Este estudio se centra en comparar dos stacks de tecnologías para aplicaciones web. Una utilizando el lenguaje Go y otra utilizando el lenguaje Python. A continuación se describen cada uno de los stacks.

Stack de Go

- Go (v1.12): es un lenguaje compilado, fácil de utilizar y de instalar en producción. Fue desarrollado pensando en la concurrencia, por lo que a la hora de realizar las pruebas se podría llegar a obtener una performance mucho mayor que con un lenguaje interpretado. Se utilizó el servidor HTTP que trae Go en la librería estándar. Además, no se utilizó ningún framework ya que no existe alguno lo suficiente maduro que se utilice para el desarrollo de páginas web (como puede ser Django para el stack de Python). Según la encuesta[2] realizada por Jetbrains en el año 2018, el 49 % de los desarrolladores no utiliza frameworks web.

Stack de Python

- Python (v3.6): es un lenguaje interpretado ampliamente utilizado en la industria debido a la baja curva de aprendizaje y a la gran comunidad que tiene.
- Django (v2.1.1): es uno de los frameworks de Python más utilizados para desarrollar aplicaciones web debido a la gran cantidad de funcionalidades que ofrece y la productividad que se tiene trabajando con el mismo.
- Gunicorn (v19.9.0): es el servidor de aplicación utilizado para Django (WSGI server) que servirá para procesar las peticiones que son recibidas.

Además para ambos stacks se utilizará la base de datos PostgreSQL (v11)[3] (que será ejecutado dentro de un Docker) y también un agrupador de conexiones de base de datos (PGBouncer v1.7.2)[4].

[1] https://docs.docker.com
[2] https://www.jetbrains.com/research/devecosystem-2018/go/
[3] https://www.postgresql.org/
[4] https://pgbouncer.github.io/

3.3. Herramienta utilizada para obtener las métricas

En cuanto a la herramienta utilizada para realizar las mediciones, se utilizó K6[5], que es una herramienta escrita en Go y que utiliza Javascript para escribir las pruebas.

4. Metodología de las pruebas

Las métricas utilizadas para medir la performance de las aplicaciones web son las siguientes:

- **Throughput:** cantidad de peticiones que el servidor responde por segundo.
- **Throughput teórico:** cálculo estimado de la cantidad de peticiones que el servidor responde por segundo teniendo en cuenta la latencia de red.

El **Throughput teórico** lo definimos como:

$$Tiempo\ de\ procesamiento\ de\ respuesta\ =\ \frac{1000\ milisegundos}{(throughput\ /\ cantidad\ de\ virtual\ users)}$$

$$Tiempo\ de\ procesamiento\ con\ latencia\ =\ Tiempo\ de\ procesamiento\ de\ respuesta\ +\ latencia$$

$$Throughput\ teórico\ =\ (\frac{1000\ milisegundos}{Tiempo\ de\ procesamiento\ con\ latencia})\ *\ cantidad\ de\ virtual\ users$$

Si se quiere estimar el *throughput real*, se deberá definir previamente un determinado valor de latencia y hacer el cálculo del *throughput* basándonos en este tiempo adicional. En la vida real, los clientes (usuarios de la aplicación) se encuentran por fuera del *datacenter* en donde se encuentra alojado el servidor (que aloja la web app). Como los clientes pueden ser de cualquier parte del mundo (y pueden a veces encontrarse a muchos kilómetros de distancia del servidor), la latencia puede variar entre unos pocos milisegundos (10 ms) hasta varias decenas de milisegundos (200 ms). Para poder calcular el *throughtput teórico*, tomaremos como valor de latencia 100 ms.

En el caso del *throughtput* la prueba se realiza utilizando dos servidores que se encuentran en el mismo *Availability Zone* y se utiliza la ip privada de la red para realizar los peticiones, por lo que el cliente y el servidor se encuentran separados a muy poca distancia. En este caso, se asumirá que la latencia es cero y el tiempo de respuesta registrado en la prueba será considerado como el tiempo que el servidor demora en procesar y responder a una petición.

[5] https://k6.io/

5. Resultados

5.1. Test Nº1 - Test de performance de los stacks sin sistema de templating ni base de datos

Esta prueba consiste en realizar peticiones al servidor HTTP y que este devuelva en la respuesta un *string* fijo, sin utilizar ningún tipo de sistema de *templating*. El objetivo de la prueba es la de conocer la cantidad máxima de respuestas que puede servir cada servidor HTTP. De esta forma, se tendrá una buena aproximación de las capacidades que tiene cada servidor HTTP sin utilizar componentes de software adicionales.

Throughput del servidor

En la prueba notamos que, a medida que el tamaño de la respuesta aumenta, la diferencia de *throughput* de ambos servidores se reduce, haciendo que la diferencia de performance entre ambos stacks pase del 65 % a un 10 %. Esto se debe a que, para respuestas pequeñas, el tiempo que tarda en transmitir la información por la red (desde el primer bit que se envió hasta el último) es pequeño respecto al tiempo que tarda el servidor en procesar la respuesta. Por lo tanto, del tiempo total que tarda el servidor en responder la petición recibida, la mayor parte está compuesta por el tiempo que tardó el servidor en procesar la respuesta, y el tiempo restante está compuesto por la transmisión de la información por la red. Por el contrario, para respuestas grandes, el tiempo que tarda en transmitir la información por la red es grande respecto al tiempo que tarda el servidor en procesar la petición. Como el tiempo que tarda en transmitir la información es aproximadamente la misma en ambos stacks (ya que depende de la red y no de los stacks), entonces el tiempo total que demora en responder un request va a ser similar. Por ende, el *throughput* de ambos servidores va a ser parecido.

Además, se puede observar que hay mucha diferencia de *throughput* cuando se hace la prueba con varios usuarios virtuales (también llamados *virtual users* o VU, que es el equivalente a cantidad de peticiones simultáneas) respecto a cuando se hace la prueba con un solo usuario virtual. Por ejemplo, para el caso de 1 VU y 1000 caracteres, la diferencia de *throughput* entre el stack de Go y el de Python es de aproximadamente 2.75 a 1. En el caso en que se realice la prueba con 128 VUs, la diferencia de *throughput* entre ambos stacks se incrementa a 8 a 1 (Python tuvo un rendimiento inferior que Go).

Si bien se podría pensar que el motivo de esta diferencia se deba a que el lenguaje interpretado es más lento, luego de investigar el caso se concluyó que se debe a que Python tiene un *Global Interpreter Lock* (o GIL) [8] que es un *mutex* que protege el acceso a los objetos de Python, previniendo que múltiples *threads* ejecuten *bytecode* de Python simultáneamente. Este candado, según la

documentación de Python[6], es necesario porque la gestión de la memoria de Python no es *thread-safe*. El problema que resuelve el GIL es la condición de carrera que se puede producir en el procesador cuando varios procesos quieren acceder al mismo sector de memoria al mismo tiempo. Como consecuencia del uso del GIL, la performance del sistema disminuirá.

Por lo tanto se puede concluir que el factor que más peso tiene en el *throughput* del servidor es el modelo de concurrencia que tiene el lenguaje y no debido a que Python sea interpretado.

Throughput teórico del servidor

Considerando el **throughtput teórico** definido en la sección 4, se pudo observar que la diferencia de *throughput* para 1 VUs era menos del 1 %, independientemente de la longitud de la respuesta. Para el caso de 128 VUs, la diferencia de *throughput* fue del 26 % (Python tuvo un rendimiento menor que Go) cuando la longitud de la respuesta era pequeña, mientras que para respuestas con mayor cantidad de caracteres (millón de caracteres) la diferencia fue del 5 %.

Por lo tanto, si se tuviera que elegir entre un stack u otro, no van a haber diferencias significativas de performance. La única variación que puede haber es en el uso de recursos en el lado del servidor.

5.2. Test N°2 - Test de performance de los stacks utilizando un sistema de templating y realizando operaciones dentro del mismo

Esta prueba consiste en realizar peticiones al servidor HTTP y que éste devuelva en la respuesta un determinado contenido en donde se haya realizado una serie de operaciones dentro del *template*, haciendo uso del sistema de *templating*. Para ello, se listan 50 strings y se hará uso de un condicional para simular alguna condición dentro de la aplicación. El objetivo es conocer la cantidad de respuestas que puede servir cada servidor HTTP realizando operaciones dentro del *template*. De esta forma, se podrá comparar esta prueba con el **Test N°1** y saber si existe algún *overhead* en el caso en que se utilice un sistema de *templating*.

Throughput del servidor

Al realizar la comparación de *throughput* entre ambos stacks, se observó que, para 1 VUs y para 1000 caracteres, la diferencia de Go vs Python fue de 4.5 a 1. Para una respuesta de 1 millón de caracteres, Python superó el *throughput* de

[6] https://wiki.python.org/moin/GlobalInterpreterLock

Go, realizando un throughput 50 % mayor que Go.

En el escenario de 128 VUs, para 1000 caracteres, la diferencia fue de 8 a 1 (Go vs Python), mientras que para el millón de caracteres, la diferencia de Python vs Go fue aproximadamente un 50 % mayor.

Si realizamos la comparación de los stacks con respecto al **Test N°1**, observamos que para Python, tanto para 1 VUs como para 128 VUs, para 1000 caracteres, la performance disminuyó un 65 %, mientras que para el millón de caracteres y 1 VUs disminuyó un 50 %, y para 128 VUs disminuyó un 65 %.

Para el caso de Go, para 1 VU y 1000 caracteres, hubo una reducción del 45 % en el *throughput* del servidor, mientras que para tamaños de respuesta grandes, el *throughput* disminuyó aproximadamente un 70 %. En el caso de 128 VUs, la performance se redujo por lo menos un 70 %.

En este test concluimos que, por más pequeña que sea la operación que se realiza en el *template*, el impacto que tiene es alto. Por lo tanto, cuanto menos operaciones deba realizar el servidor para renderizar una respuesta, mejor. Es por ello que hoy en día se intenta desacoplar el *front-end* del *back-end* (utilizando frameworks front-end como ser React, Vue.js, etc.), ya que el *front-end* se encarga de renderizar la información desde el lado del cliente, haciendo que el servidor se libere de esta responsabilidad y que el procesamiento de la información que se muestra la lleve a cabo el cliente y no el servidor.

Throughput teórico del servidor

Al comparar Python y Go para 1 VUs, la diferencia entre ambos stacks es mínima (tanto para respuestas con poca cantidad de caracteres como para mucha cantidad de caracteres, la diferencia entre ambos stacks es menor al 5 %).

En el caso de 128 VUs, la diferencia entre ambos stacks es menor que si no hubiera latencia. Aún así, podemos ver que, para poca cantidad de caracteres, Python tiene la mitad de *throughput* que Go. Mientras que para respuestas más largas, Python supera a Go en un 60 %.

A partir de los datos recolectados, se puede concluir que utilizar un sistema de *templating* y realizar operaciones dentro del mismo impacta enormemente en el *throughput* del servidor. Aún así, se puede observar que la latencia juega un rol muy importante en la performance general de ambos stacks.

Por lo tanto, para obtener la mayor performance posible de un servidor, es necesario realizar la menor cantidad de operaciones con el sistema de *templating*. Además, es recomendable reducir al máximo la latencia entre el cliente y el servidor y, de ser posible, delegar la tarea de renderizado al cliente.

5.3. Test Nº3 - Test de performance simulando un entorno real (utilizando base de datos y sistema de templating)

Esta prueba consiste en realizar peticiones al servidor HTTP y que éste devuelva en la respuesta una determinada cantidad de elementos que fueron buscados en la base de datos (filtrado por algún criterio) y renderizados utilizando un sistema de *templating*. El objetivo de esta prueba es el de conocer la cantidad de respuestas que puede servir cada servidor HTTP, el cual renderiza resultados que fueron buscados en la base de datos. Lo que se busca es realizar las mismas operaciones que realiza un *e-commerce* cuando se muestran productos según un criterio (a nivel *templating* y base de datos, no en cuanto a estética). Dicha prueba consta de buscar 50 elementos de una base de datos (simulando ser publicaciones de productos), renderizar el resultado (*template*), y devolver dicha página al usuario que realizó la petición. El tamaño de la respuesta a devolver será de 1.000.000 de caracteres (cantidad de caracteres promedio que devuelve una página de Amazon.com).

Throughput del servidor

Luego de haber realizado las pruebas, observamos que:

- Cuanto más grande sea el *template*, más tarda en generar la respuesta. Para ambos stacks, a medida que la cantidad de VUs aumenta, la performance del servidor va disminuyendo.
- El uso de una base de datos degrada enormemente el *throughput* de ambos servidores.
- Para los casos en que se probaron con 64 y 128 VUs, el consumo de recursos de Python era máximo, por lo que comenzaba a afectar la performance del servidor y comenzaba a responder de forma no satisfactoria a las peticiones.
- Para el caso de Go, también se detectaron anomalías. Se cree que el motivo por el cual ocurre este comportamiento es debido a cómo Go gestiona la concurrencia al renderizar templates. Por lo tanto, se excluirán del análisis los resultados obtenidos para 64 y 128 VUs.

Si comparamos Python vs Go para tamaños de respuestas grande (1 millón de caracteres) para 1 VU, la diferencia es de un 42 %, mientras que para 32 VUs es de un 58 %.

Throughput teórico del servidor

Al comparar Python y Go, observamos que la diferencia de *throughput* es mínima. Para el caso en que se haga la prueba con 1 VU, la diferencia de *throughput* es menor al 2 %. En cambio, si se hacen de a 32 VUs, la diferencia de *throughput* llega aproximadamente al 30 %. Esto se debe a que la latencia reduce drásticamente la performance del servidor. Por ende, a medida que se tenga más latencia, menor será el *throughput* del mismo. Esto se corrige teniendo el servidor lo más cerca posible del cliente.

6. Conclusión

A partir de los resultados obtenidos de las pruebas realizadas en el presente trabajo, se puede llegar a la conclusión de que si bien un lenguaje compilado es más rápido que un lenguaje interpretado, esta diferencia de performance no es tan grande si consideramos la latencia.

Por lo tanto, se puede concluir que el problema de performance de una aplicación web termina siendo mayormente un problema de arquitectura y no un problema del tipo del lenguaje utilizado. Es decir, que un lenguaje sea compilado o interpretado no será determinante para la performance de una aplicación web.

Es importante destacar que, si bien podría ser mejor utilizar un lenguaje compilado en lugar de uno interpretado, es más importante tener una buena arquitectura. A continuación mencionaremos algunos puntos a tener en cuenta:

- Si se utiliza una base de datos en el sistema, tratar de acceder a la misma la menor cantidad de veces posible (de ser posible, utilizar una caché).
- Separar el front-end del back-end, ya que de no hacerlo, la performance del servidor se va a ver afectada enormemente.
- Tener el servidor lo más cercano posible al cliente, ya que la latencia hace que la performance de la aplicación web se degrade enormemente.

Referencias

1. T.Lance, A.Martin and W.Carey, Performance Comparison of Dynamic Web Technologies, ACM SIGMETRICS Performance Evaluation Review, Volume 31 Issue 3, December 2003.
2. Suzumura, Toyotaro & Trent, Scott & Tatsubori, Michiaki & Tozawa, Akihiko & Onodera, Tamiya. (2008). Performance Comparison of Web Service Engines in PHP, Java and C. 385-392.
3. N. Togashi and V. Klyuev, “Concurrency in go and java: Performance analysis,” in 2014 4th IEEE International Conference on Information Science and Technology, pp. 213–216, April 2014.
4. K. Lei, Y. Ma, and Z. Tan, “Performance comparison and evaluation of web development technologies in php, python, and node.js,” in Computational Science and Engineering (CSE), 2014 IEEE 17th International Conference on, pp. 661–668, Dec 2014.
5. Y. Ueda and M. Ohara, ”Performance competitiveness of a statically compiled language for server-side Web applications,”2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Santa Rosa, CA, 2017, pp. 13-22.
6. The Go Programming Language. https://golang.org/.
7. https://www.Python.org/.
8. https://wiki.python.org/moin/GlobalInterpreterLock