ESCUELA SUPERIOR POLITÉCNICA DEL LITORAL

Facultad de Ingeniería en Electricidad y Computación

"PLANIFICADOR EXTENSIBLE PARA LA PLATAFORMA FAAS OPEN LAMBDA"

INFORME DE MATERIA DE GRADUACIÓN

Previa a la obtención del Título de:
INGENIERO(A) EN CIENCIAS COMPUTACIONALES

Presentado por:
Gustavo Gerson Totoy Casal
Gabriel Emilio Aumala Encarnación

GUAYAQUIL - ECUADOR

AÑO: 2018

AGRADECIMIENTO

Agradezco a la Dra. Cristina Abad por ser una excelente guía y darme la oportunidad de trabajar en un proyecto tan interesante y significativo como este.
A mis compañeros de trabajo Mayer Mizrachi, Luis Loaiza, Alberto Vera, Gianni Carlo y Daniel Tigse por apoyar mis estudios mientras cumplia mis responsabilidades laborales.

Gabriel Emilio Aumala Encarnación

AGRADECIMIENTO

Agradezco a la vida y al intrincado número de interacciones que me ayudaron de una u otra manera a culminar esta etapa, la cual en varias ocasiones pensé quedaría inconclusa y dejada atrás para el olvido.

Gustavo Gerson Totoy Casal

DEDICATORIA

Dedico este trabajo a mis padres, Lila Encarnación y Manuel Aumala, por siempre apoyarme en mi educación y fomentar buenos valores.

Gabriel Emilio Aumala Encarnación

DEDICATORIA

Dedico este trabajo a cada persona que pueda encontrar alguna utilidad o interés sobre lo realizado en este proyecto.

Gustavo Gerson Totoy Casal

DECLARACIÓN EXPRESA

“Los derechos de titularidad y explotación, nos corresponde conforme al reglamento de propiedad intelectual de la institución; Gustavo Totoy y Gabriel Aumala damos nuestro consentimiento para que la ESPOL realice la comunicación pública de la obra por cualquier medio con el fin de promover la consulta, difusión y uso público de la producción intelectual”

Gustavo Totoy

Gabriel Aumala

RESUMEN

El presente trabajo considera uno de los mayores problemas de la emergente arquitectura serverless: La latencia de arranque introducida al tener que instalar largas listas de dependencias para cada función lambda. Una posible solución, un nuevo algoritmo de distribución de carga que para cada nuevo requerimiento elige el nodo trabajador dando mayor prioridad al nodo que ya tenga las dependencias necesarias instaladas y así poder saltar el costoso paso de instalación, ya ha sido propuesto y simulado con resultados teóricos favorables por investigadores de la Escuela Politécnica Superior del Litoral (ESPOL). Nosotros implementamos un nuevo planificador con este nuevo algoritmo propuesto, al igual que los algoritmos convencionales de distribución de carga. Para demostrar la utilidad del planificador de tareas implementado, se ejecutaron pruebas en una plataforma abierta sobre un ambiente de producción en la nube, midiendo la latencia de cada requerimiento en cientos de funciones lambda y calculando el rendimiento promedio con cada algoritmo. Nuestros resultados experimentales demuestran que la distribución basada en paquetes instalados es una clara mejora en rendimiento para plataformas serverless sobre los algoritmos de distribución convencionales.

Palabras clave: Serverless, FaaS, Balanceador de carga

ABSTRACT

This work considers one of the main problems of the emerging serverless architecture, the startup latency introduced when installing long dependency lists for each lambda function. A possible solution, a new scheduling algorithm, in which for every new request it chooses the worker node giving higher prority to those that already have installed all necessary dependencies, skipping the expensive installation step, has already been proposed by researchers at ESPOL with positive results. We implement a new scheduler with this proposed algorithm, as well as other popular load balancing algorithms. To prove the effectiveness of the implemented scheduler we ran benchmarks on an open platform on a production cloud environment, measuring the latency of each request in hundreds of lambda functions and calculating the average latency with each algorithm. Our experimental results show that package aware distribution is a clear improvement over conventional load balancing algorithms for serverless platforms.

Keywords: *Serverless, FaaS, Load balancer*

ABREVIATURAS

ESPOL Escuela Politécnica Superior del Litoral

HTTP Hyper Text Transfer Protocol

FaaS Function as a Service

IP Internet Protocol

IPv4 Internet Protocol version 4

IPv6 Internet Protocol version 6

JSON JavaScript Object Notation

PyPI Python Package Index

EC2 Elastic Cloud Computing

SSH Secure Shell

AWS Amazon Web Services

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

CAPÍTULO 1

1. Introducción

En los últimos años, se ha presentado un creciente interés en arquitecturas ”serverless” en la industria de tecnologías de la información [1]. Desde el lanzamiento de AWS Lambda en el 2014 [2], cada vez son más las empresas de tecnología que optan por implementar nuevos productos a través de arquitecturas serverless. Serverless computing es un nuevo paradigma el diseño e implementación de software, principalmente aplicaciones web [3]. Es una abstracción sobre servidores, infraestructura y sistemas operativos. La idea principal de este paradigma es que el desarrollador de la aplicación únicamente se encarga de crear el código que maneje nuevos eventos que recibe la aplicación en tiempo real y deja que el proveededor de servicios en la nube, una empresa grande como Amazon o Microsoft, se encargue de la administración, escalamiento y monitoreo de la aplicación y su infraestructura subyacente. El beneficio de serveless computing no sólo se limita a la disminución de la complejidad del despliegue de la aplicación sino también la rebaja de los costos de servicios en la nube ya que los proveedores de este servicio unicamente cobran por el tiempo de CPU que consume la función.

En el mercado existen varios proveedores de serverless computing como AWS Lambda, Microsoft Azure y Google Cloud Function. Sin embargo, casi todas estas plataformas son software propietario desarrollado de forma privada. Para poder investigar sobre arquitecturas serverless es indispensable una plataforma abierta que permita a cualquier individuo u organización explorar nuevos enfoques sobre la implementación de estos sistemas. Es por esto que la universidad de Wisconsin en el 2016 presentó OpenLambda [4], una plataforma que responda a estas necesidades. En la publicación de OpenLambda se detallan las motivaciones para crear esta plataforma así también

como los varios problemas a investigar como motores de ejecución, soporte para paquetes, balanceadores de carga, manejo de sesiones, entre otros. La manera como esta implementado OpenLambda actualmente es con un programa escrito en Go que administra ”workers” que pueden correr funciones únicamente en Python. Para balancear la carga entre los workers se usa Nginx, un servidor Hyper Text Transfer Protocol (HTTP) usado ampliamente en la industria que ofrece varios algoritmos genéricos para balanceo de cargas.

Uno de los resultados de investigaciones realizadas sobre OpenLambda que ha llamado la atención de investigadores de ESPOL es Pipsqueak. Pipsqueak es una plataforma serverless basada en OpenLambda que mantiene un grupo de intérpretes de Python cacheados que cargaron paquetes en memoria durante la ejecución de funciones [5]. El objetivo es que cuando se invoca una nueva función que requiere los mismos paquetes de una función anterior, se puede reutilizar ese mismo intérprete y arrancar mucho más rápido. Al estar basado en OpenLambda, Pipsqueak también usa Nginx como balanceador de carga. Al ver que el balanceador de carga de Pipsqueak no maximizaba el uso de los interpretes cacheados, investigadores de ESPOL desarrollaron de forma teórica un algoritmo de planificación que conoce sobre los interpretes cacheados en cada nodo trabajador y distribuye la carga de manera más eficiente ofreciendo únicamente resultados preliminares [6, 7]. Para poder presentar conclusiones útiles, es necesario implementar un balanceador de carga que use este algoritmo y realizar evaluaciones extensas de su rendimiento en OpenLambda.

1.1. Descripción del problema

En el sector privado hay muchas organizaciones realizando investigaciones para mejorar el rendimiento de implementaciones de plataformas serverless para poder abastecer la creciente demanda de servicios en la nube que soporten estas arquitecturas serverless. OpenLambda y Pipsqueak son algunos de los proyectos open source resultantes de los esfuerzos más recientes. El grupo de investigación de Big Data de ESPOL, tras analizar las ventajas de Pipsqueak ha propuesto un nuevo planificador de tareas para OpenLambda, con algoritmos inteligentes que maximizan el uso de la caché de

Pipsqueak. Los resultados preliminares muestran mejoras de rendimiento significativas, sin embargo, esto resultados provienen de simulaciones y no existen experimentos reales en la nube que muestren la disminución de la latencia al invocar funciones en OpenLambda [7].

1.2. Justificación del problema

En una plataforma serverless, se puede esperar que correr una función lambda inicie casi inmediatamente ya que los intérpretes que ejecutan el código ya están cargados en memoria. Sin embargo, si la función depende de varias librerías, se introduce una latencia de arranque muy alta al descargar e instalar estas librerías [4]. En la actualidad no existe un planificador open source para OpenLambda que tome en cuenta la información de los paquetes usados por las funciones para distribuir la carga hacia los nodos trabajadores y así evitar instalar múltiples veces los mismos paquetes. Los investigadores de ESPOL han propuesto algoritmos prometedores que resuelvan este problema, pero no existe una plataforma en donde se pueda implementar los algoritmos propuestos por los investigadores de ESPOL y evaluar su rendimiento en un ambiente de producción.

1.3. Objetivos

1.3.1. Objetivo general

Diseñar e implementar un planificador de funciones en la nube para OpenLambda, que busque afinidad en la asignación de funciones a nodos trabajadores que idealmente tengan pre-cargados los paquetes requeridos por la función.

1.3.2. Objetivos específicos

Los objetivos específicos del presente trabajo son los siguientes:

1. Diseñar un planificador que reciba los requerimientos con la información necesaria

por los algoritmos de planificación y transmita el requerimiento al nodo seleccionado usando las convenciones de OpenLambda.

2. Ofrecer una plataforma open source con algoritmos inteligentes de planificación que mejoren el tiempo de inicio de las llamadas a las funciones en OpenLambda.

3. Ofrecer una plataforma open source donde se puedan implementar nuevos algoritmos de planificación, evaluar el rendimiento de los mismo y contrastarlos entre ellos.

1.4. Marco teórico

Una plataforma en la nube del tipo Function as a Service (FaaS) permite la creación de aplicaciones distribuidas como un conjunto de funciones en la nube que tienen como características ser pequeñas, sin estado y se encargan de una sola tarea o responsabilidad [7], reduciendo la carga a los desarrolladores de manejar servidores, máquinas virtuales y contenedores. De esta manera el desarrollador puede crear aplicaciones elásticas en la nube sin preocuparse de los temas mencionados anteriormente ni en administradores de elasticidad. Algunas plataformas FaaS existentes en la actualidad son OpenLambda, Fission, OpenWhisk, AWS Lambda y Azure Functions.

1.4.1. OpenLambda

OpenLambda es una plataforma FaaS en la cual el usuario debe copiar el código fuente de su función a un registro de código fuente para que dicha función este disponible para ser ejecutada por la plataforma al recibir un requerimiento por HTTP solicitando el resultado de esa función con los parámetros especificados [4]. OpenLambda posee un planificador que selecciona un nodo trabajador en un cluster de tamaño configurable usando el algoritmo establecido en la configuración. Cuando un nodo trabajador recibe el requerimiento, verifica si posee el código fuente de la función solicitada en su registro local, si no es así se solicita una copia al registro de código fuente, instala todas las dependencias necesarias, y procede a la ejecución de la función en un ambiente aislado. En la actualidad OpenLambda permite el uso de contenedores Docker y Sock [8] como

sus ambientes de ejecución aislados para mayor seguridad, ya que siempre existe la posibilidad de que un usuario suba una función maliciosa al sistema [4].

1.4.2. Planificación de funciones en OpenLambda

En OpenLambda la planificación de las funciones es elaborada por Nginx, el cuál actúa como un balanceador de cargas de trabajo en el cluster. Nginx ofrece los siguientes mecanismos de balanceo de carga [9]:

1. Round Robin: La carga se distribuye equitativamente entre los nodos tomando en cuenta pesos configurables para cada nodo.

2. Least Connections: El requerimiento es asignado al nodo con la menor cantidad de conexiones activas tomando en cuenta pesos configurables para cada nodo.

3. IP Hash: El nodo trabajador se escoge usando la dirección Internet Protocol (IP) del cliente, aplicando los primeros tres octetos si la dirección es Internet Protocol version 4 (IPv4) o la dirección completa si es Internet Protocol version 6 (IPv6) a una función de hash. Este método garantiza que todos los requerimientos realizados desde una dirección IP van hacia el mismo nodo trabajador.

4. Generic Hash: El nodo trabajador se escoge usando una clave escogida por el usuario, puede ser alguna cadena de caracteres en el requerimiento, la dirección IP con el puerto o alguna combinación de estos.

Estos mecanismos permiten distribuir la carga de trabajo entre los nodos. Sin embargo, Nginx no ofrece mecanismos que permitan tomar decisiones inteligentes que tengan algo de contexto sobre la naturaleza de la carga o de los nodos trabajadores.

CAPÍTULO 2

2. Metodología

La metodología aplicada al presente proyecto consiste de dos partes: Implementar el software del planificador y correr pruebas para analizar su rendimiento. En este capítulo se detalla como se ejecutarán estas partes para alcanzar el objetivo de tener un mejor planificador para OpenLambda.

2.1. Diseño del planificador

El planificador se llamará “Olscheduler“ y será usado de forma similar a Nginx, como un balanceador de carga para OpenLambda que recibe los requerimientos HTTP de los clientes y usa algún algoritmo especificado para decidir a que nodo trabajador enviar la carga. El módulo Olscheduler tiene las siguientes propiedades:

1. Mantiene un arreglo de estructuras que describen varias propiedades de los nodos trabajadores como el peso asignado, los paquetes cargados, entre otros. La finalidad es poder rápidamente proveer toda la información relevante al algoritmo de distribución de carga especificado. Para tener la lista de paquetes de cada función, es necesario pasar un archivo con estos datos en formato JavaScript Object Notation (JSON).

2. Recibe los requerimientos HTTP de las invocaciones de las funciones lambda, procesa los encabezados del requerimiento lo reenvía en el formato esperado por OpenLambda.

3. Usa un archivo en formato JSON para configurar varias cosas como: el puerto a utilizar, el host a utilizar, las direcciones de los nodos trabajadores, el algoritmo a

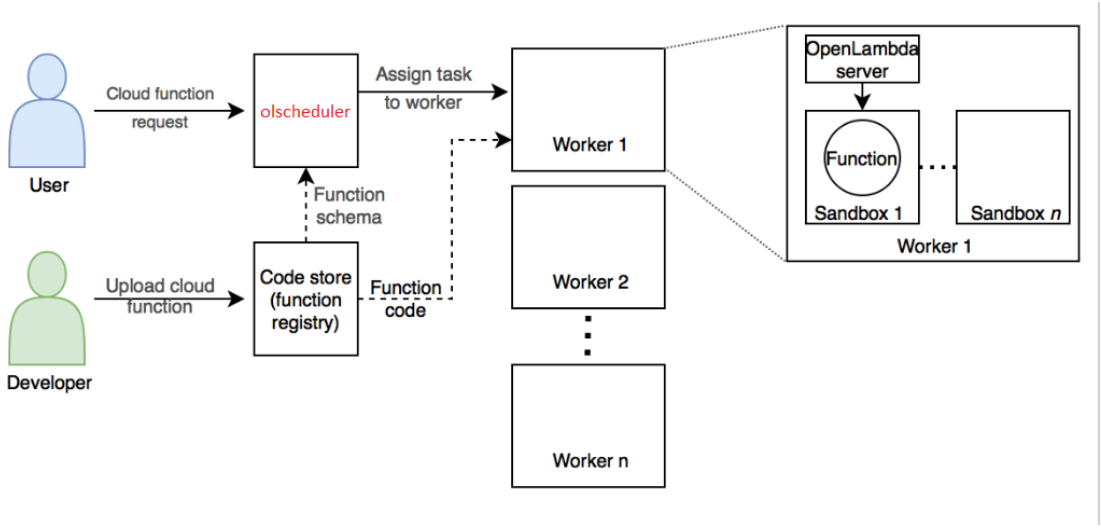

Figura 2.1: Configuración de despliegue de plataforma Faas con olscheduler

usar para seleccionar el nodo trabajador y otras variables en las que se puedan necesitar algo de flexibilidad.

4. Cada requerimiento reenviado hacia un nodo trabajador se maneja de manera asíncrona delegando módulos de la librería estándar de Go como ”net/http/httputil” para manejar las conexiones y la concurrencia.

5. Cada algoritmo de planificación será implementado en su propio archivo con extensión go y será un función dentro del paquete balancer.

2.2. Plan de recolección de datos (módulo olscheduler)

El usuario que consume la plataforma FaaS debe de enviar requerimientos HTTP a la url ”/runLambda/<lambda-name>”, dónde ”<lambda-name>” debe de ser reemplazado por el nombre que identifica a la función que desea correr. Antes de correr la función esta debe de ser subida a la plataforma declarando su lista de dependencias. Para facilitar la automatización de pruebas con cientos de funciones, se provee el índice de funciones con sus dependecias en un archivo JSON al iniciar el planificador.

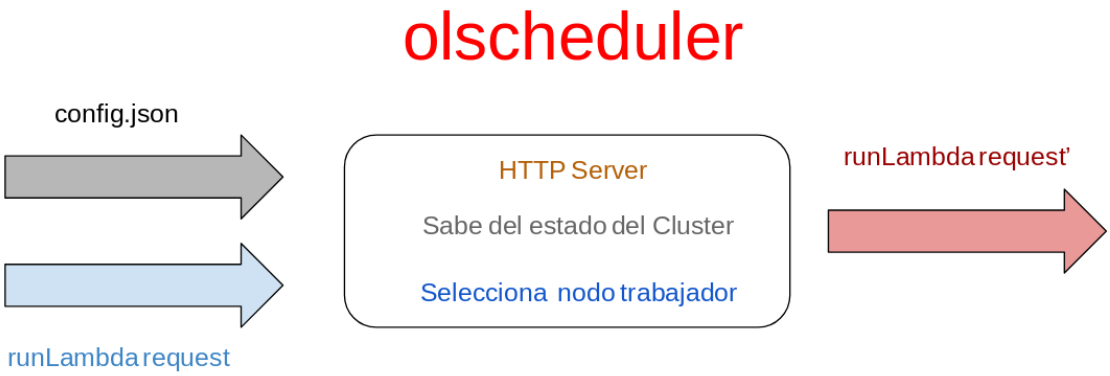

Figura 2.2: Entradas y salidas de olscheduler

2.3. Selección del lenguaje de programación

Se decidió codificar el planificador en el lenguaje de programación Go, ya que la actualidad, Go es uno de los mejores lenguajes para programar servidores de alto rendimiento debido a varias decisiones en el desarrollo de su runtime [10]. El lenguaje usa un garbage collector así que no hay necesidad de manejar manualmente la memoria del programa. Esto permite desarrollar el software de manera rápida y minimizando problemas de seguridad. Go también tiene excelente soporte para concurrencia, algo muy valuable en servidores web ya que es común trabajar con miles de conexiones concurrentes. El lenguaje incluso tiene sintaxis especial para correr código en nuevos hilos y envíar y recibir datos entre diferentes hilos.

En aplicaciones web, es más común usar lenguajes dinámicos como JavaScript, PHP, o Python [11], los cuales ofrecen más facilidades para el desarrollo. Sin embargo, estos lenguajes generalmente no tiene buen soporte para correr código con múltiples hilos y al ser interpretados, en lugar de compilarse a código de máquina, tienen un bajo rendimiento. Tampoco fueron diseñados específicamente para servidores. Como nuestro objetivo es reemplazar Nginx, un servidor web escrito en C optimizado a lo largo de los años, el rendimiento y la concurrencia son muy importantes, por lo cual decidimos usar Go.

2.4. Plan de implementación

El primer paso es crear un servidor HTTP que pueda procesar los requerimientos HTTP requests que envían los clientes para ejecutar sus funciones. Para esto vamos a delegar los módulos de la librería estándar de Go ”net/http” y ”net/http/util” que con pocas líneas de código pueden crear un servidor HTTP listo para producción.

Para poder implementar los algoritmos de distribución de carga es necesario que olscheduler tenga acceso a varios datos sobre los nodos trabajadores. Para evitar tener que hacer consultas directamente con los nodos trabajadores, Vamos a crear una estructura ”Worker” que contenga varios atributos como la carga del nodo y el peso asignado a ese nodo, y olscheduler mantendrá un arreglo con instancias de ”Worker” que modelen todos los nodos trabajadores disponibles. Como los atributos de los nodos trabajadores no cambian durante los benchmarks no hay problemas de sincronización entre el modelo y los nodos reales.

El último paso es poder configurar olscheduler. Para esto se usará un archivo en formato JSON que especifica varias variables configurables. Para decodificar este archivo usaremos el modulo ”json” de la librería estándar. Los datos decodificados se usan para crear instancias de ”Worker”. Como olscheduler solo ofrece una interfaz de linea de comandos, el archivo de configuración se debe de pasar como un argumento de línea de comando. Para facilitar esto usaremos la librería ”github.com/urfave/cli” para procesar facilmente los argumentos y brindar una mejor experiencia de usuario.

2.5. Selección de herramienta de benchmarking

Para correr benchmarks sobre servidores HTTP existen varias alternativas de código abierto. Algunos ejemplos son httperf y Siege [12], dos herramientas para simular diferentes tipos de cargas en servidores HTTP y medir las latencias. Estas herramientas son muy generales y se enfocan más en servidores de aplicaciones web y no en balanceadores de carga. Para nuestros benchmarks se usará Pipbench, una herramienta para benchmarking creada por el equipo de OpenLambda. A diferencia de otras alternativas más populares Pipbench no simula cargas genéricas HTTP, sino cargas

específicas de invocación de funciones; Pipbench genera un repositorio local con paquetes de Python artificiales, similares a los que se encuentran en el repositorio Python Package Index (PyPI), y funciones para OpenLambda que dependan de estos paquetes. El repositorio de paquetes artificiales es muy importante ya que mantener una copia del repositorio PyPI no es algo trivial.

2.6. Plan de recolección de datos (pruebas)

Nuestro plan es usar Pipbench para generar el repositorio de paquetes artificiales de Python junto con 100 funciones que dependan de varios de estos paquetes y correr benchmarks que midan la latencia de la ejecución de estas funciones. Estos benchmarks se correrán con todos los mecanismos de balanceo de carga del planificador y también se correrán con Nginx, para poder formular conclusiones sobre el rendimiento del planificador.

2.7. Plan de pruebas

Para correr las pruebas vamos a usar tres instancias tipo m4.xlarge de Amazon Elastic Cloud Computing (EC2) con Ubuntu 16.04 para correr estos tres programas:

1. Olscheduler
2. Cluster de nodos trabajadores de OpenLambda
3. Repositorio de paquetes controlado

El cluster de OpenLambda usará el container Sock, creado por el equipo de OpenLambda [8] en lugar de Docker, debido a las facilidades de cacheo de paquetes y ambientes de ejecución de Python. Los experimentos se escribirán como scripts de Bash. Todo será administrado vía Secure Shell (SSH) y se usará la herramienta Screen de Linux, para manejar varias sesiones en cada servidor con los varios procesos que debe de correr cada uno.

Las pruebas consisten en generar 100 funciones lambda y medir los valores de latencia al correr estas funciones reiteradas veces usando los cuatro algoritmos de planificación

que ofrece Olscheduler. Por cada configuración se generará la misma carga, invocando las mismas funciones varias veces, con intervalos diminutos entre cada requerimiento.

2.8. Análisis de datos

Con los valores de latencia reportados en las pruebas se elaborará un análisis estadístico descriptivo que permita contrastar el rendimiento de los algoritmos entre sí. Se generarán gráficos para visualizar los resultados con Chart.js y Plotly.js, dos librerías de JavaScript populares para crear gráficos en un navegador web.

2.9. Fiabilidad de datos

Para asegurar la confiabilidad de los resultados se tomaron varias medidas:

1. Se decidió correr los experimentos en Amazon Web Services (AWS) ya que esta plataforma es muy común para sistemas en producción y al ser tan popular y accesible es más fácil que otras personas reproduzcan nuestros resultados. Se eligió usar Ubuntu 16.04 en estos servidores por razones similares. Es una distribución popular y estable, además de ser oficialmente soportada por OpenLambda.

2. Pipbench mide la latencia desde el momento en que se hace el pedido de la ejecución de la función lambda hasta el momento en que el resultado devuelto por OpenLambda es recibido [5]. Las configuración de las instancias EC2 minimiza el riesgo de que la latencia de la red sea mucho mayor que la latencia de los algoritmos del planificador.

3. La carga a usar en los experimentos, los requerimientos de las funciones a ejecutar son generados de forma automática.

4. La configuración de los nodos trabajadores para Olscheduler es tomada de la configuración del cluster levantado en OpenLambda para evitar que los nodos trabajen de forma incorrecta.

CAPÍTULO 3

3. Resultados y Análisis

En este capítulo, se explicaran con gráficos los resultados de las pruebas de cada algoritmo. Para cada algoritmo se provee un gráfico de dispersión del número identificador de la función lambda ejecutada versus la latencia medida por Pipbench. Estos gráficos de dispersión usan puntos con una pequeña transparencia, por lo tanto en la zonas con puntos más oscuros hay una mayor concentración de puntos. Finalmente se mostrar gráfico de barras que compara la latencia promedio medida con cada algoritmo.

3.1. Resultados del algoritmo Round Robin

Como se puede ver en las figuras 3.1 y 3.2, con Round Robin, la mayoría de valores de latencia se concentran cerca del cero, es decir son muy bajos. Sin embargo, hay muchas cajas largas visibles, lo cual indica que un porcentaje significativo de requrimientos estan en el orden de magnitud de los 10 segundos o más. Incluso se pueden apreciar varios valores bastante altos, cerca de los 300 segundos. Hay un puñado de funciones cuyo primer cuartil esta por encima de los 50 segundos. Para casi la mitad de funciones, el tercer cuartil está por encima de los 50 segundos, y para la cuarta parte de las funciones, el tercer cuartil está alrededor de los 150 segundos. Round Robin hace un trabajo decente en distribuir la carga, pero claramente podría ser mucho mejor. Este algoritmo se usará como base para comparar los demás.

3.2. Resultados del algoritmo Random

Como se puede ver en las figuras 3.3 y 3.4, los resultados de Random son muy similares a los de Round Robin. Una vez más se observa que la mayoría de valores está cerca del

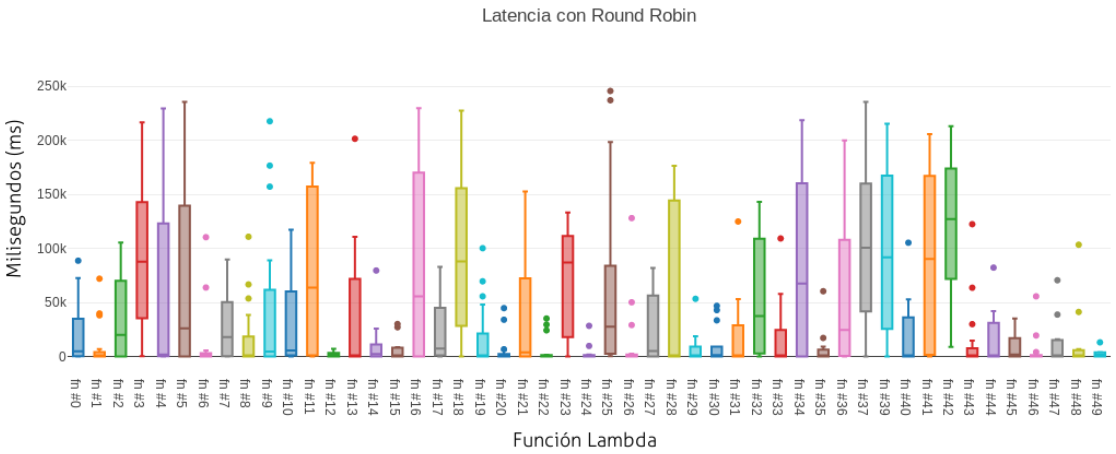

Figura 3.1: Valores de latencia en milisegundos para cada requerimiento enviado a las funciones entre #0 y #49 con algoritmo Round Robin

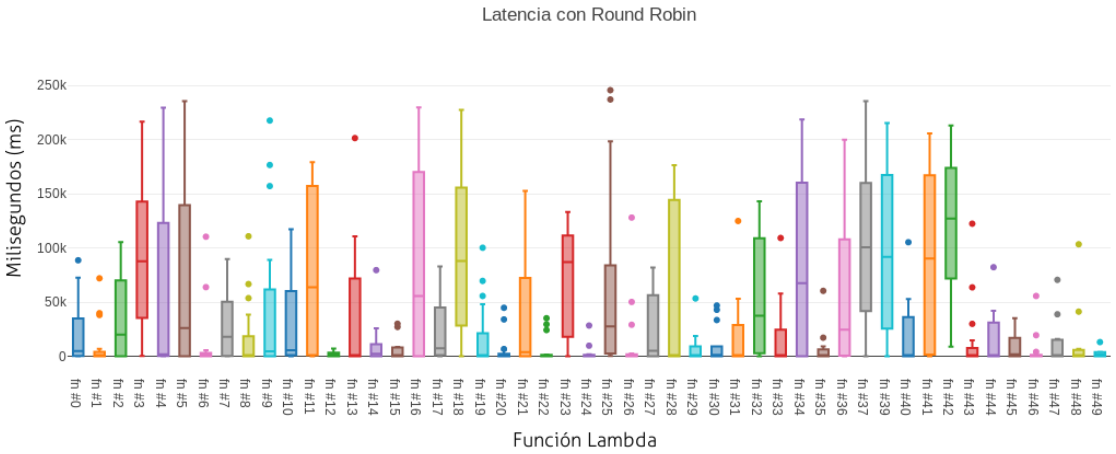

Figura 3.2: Valores de latencia en milisegundos para cada requerimiento enviado a las funciones entre #50 y #99 con algoritmo Round Robin

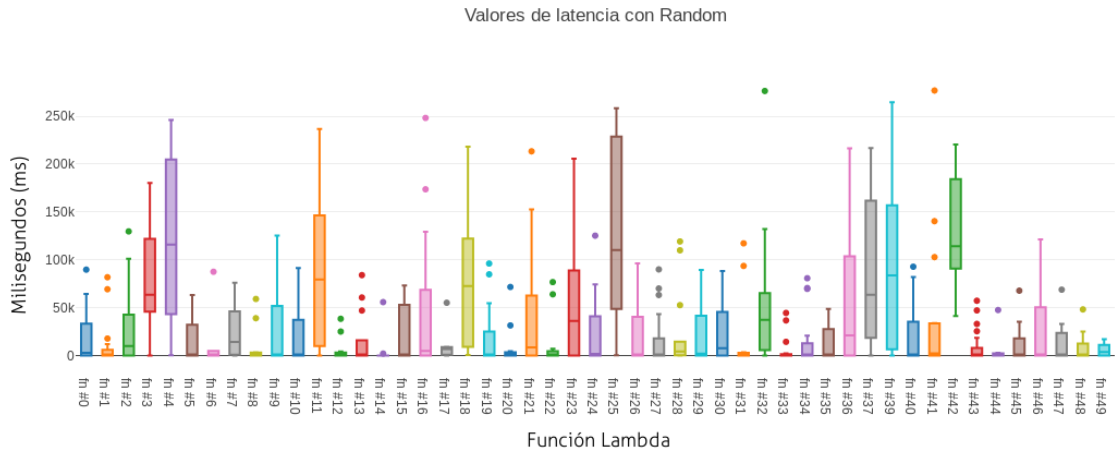

Figura 3.3: Valores de latencia en milisegundos para cada requerimiento enviado a las funciones entre #0 y #49 con algoritmo Random

Figura 3.4: Valores de latencia en milisegundos para cada requerimiento enviado a las funciones entre #50 y #99 con algoritmo Random

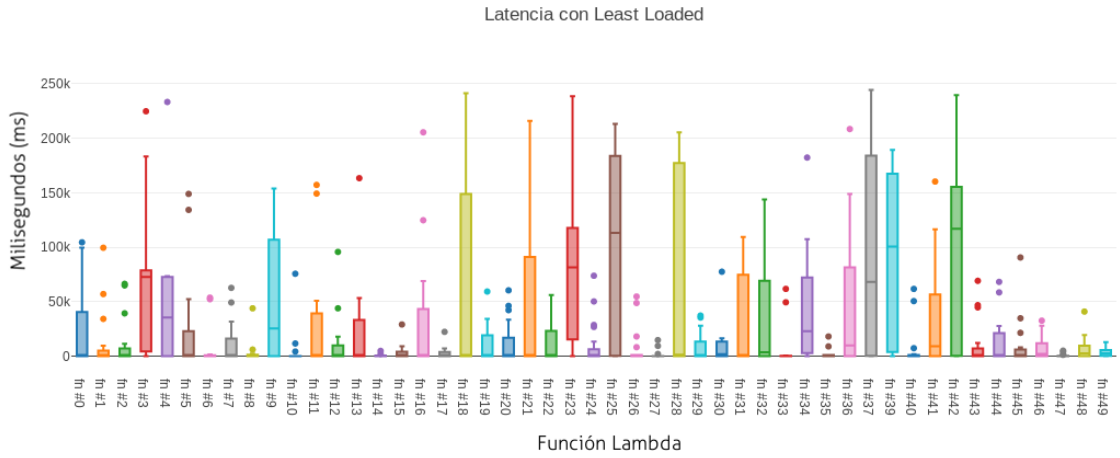

Figura 3.5: Valores de latencia en milisegundos para cada requerimiento enviado a las funciones entre #0 y #49 con algoritmo Least Loaded

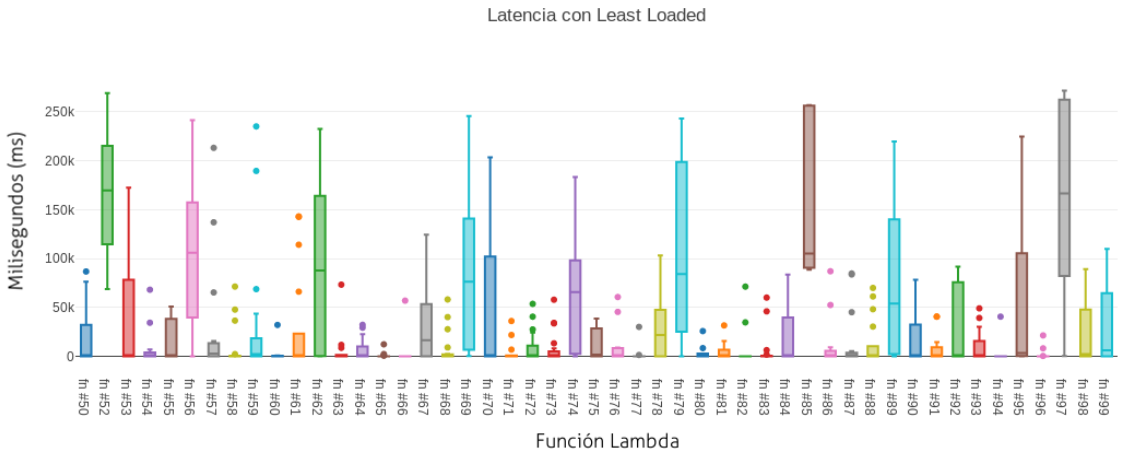

Figura 3.6: Valores de latencia en milisegundos para cada requerimiento enviado a las funciones entre #50 y #99 con algoritmo Least Loaded

cero, pero nuevamente se pueden ver bastantes cajas largas y algunos máximos cerca de los 300 segundos. También hay un puñado de funciones cuyo primer cuartil esta por encima de 50 segundos y el número de funciones cuyo tercer cuartil esta por encima de los 50 segundos es ligeramente menor que el de Round Robin. No es sorpresa que una distribución de carga aleatoria sea tan homogénea como la de Round Robin.

3.3. Resultados del algoritmo Least Loaded

Como se puede ver en las figuras 3.5 y 3.6, una vez más los resultados son parecidos a los de Round Robin. Nuevamente la mayoría está cerca del cero, y se puede ver varias

cajas largas valores máximos cerca de los 300 segundos. Sin embargo, sí se puede notar una pequeña diferencia. Tan sólo el 30 % de las funciones tienen su tercer cuartil por encima de 50 segundos. Sigue habiendo un puñado de funciones cuyo primer cuartil esta por encima de 50 segundos. El número de requerimientos con valores extremadamente altos es un poco menor. Debido a que Least Loded trata de evitar que se sobrecargen los servidores, se pueden apreciar más valores pequeños de latencia. Least Loaded es una pequeña mejora sobre Round Robin y Random.

3.4. Resultados del algoritmo Package Aware

A diferencia de los demás algoritmos, la latencia promedio de las funciones planificadas con este algoritmo depende de un valor configurable: El umbral de carga de trabajo del nodo trabajador. El planificador guarda en memoria el número de requerimientos concurrentes que procesa cada nodo, así que si ese numero se pasa del umbral configurado, se utiliza otro nodo que este menos cargado. Si el umbral es muy pequeño, la latencia promedio aumenta ya que se desperdician las caches de Pipsqueak puesto a que los intérpretes cargados en memoria no se reutilizan al máximo. Por otro lado, si el valor es muy grande, se puede correr el riesgo de sobrecargar los nodos trabajadores, aumentando la latencia de cada requerimiento. Evidentemente hay un valor que minimiza la latencia promedio, y antes de comparar Package Aware es necesario encontrar este valor. Como se puede ver en la figura 3.7, con un load threshold de 100 se reportó la mínima latencia promedio, por lo tanto se usó esta configuración para las demás pruebas.

Como se puede observar en las figuras 3.8 y 3.9, Package Aware es una clara mejora sobre los otros algoritmos. El número de cajas largas es mucho más bajo. El valor máximo de latencia reportado es sólo 218153 milisegundos, mucho menor que los otros algoritmos, cuyos valores máximos llegaban cerca de los 280 segundos. No existe ninguna función cuyo primer cuartil este por encima de 50 segundos, lo cual si ocurría un par de veces con los otros algoritmos. Claramante son pocos los requerimientos que demoran mas de 50 segundos, y la gran mayoría esta cerca de cero.

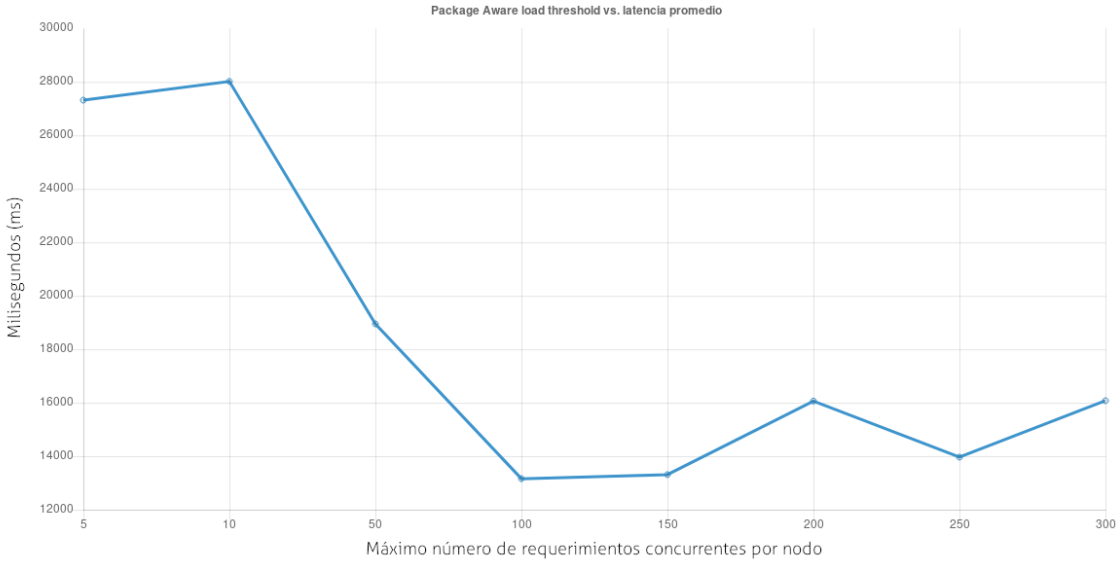

Figura 3.7: Latencia promedio del algoritmo Package Aware en función del valor de load threshold

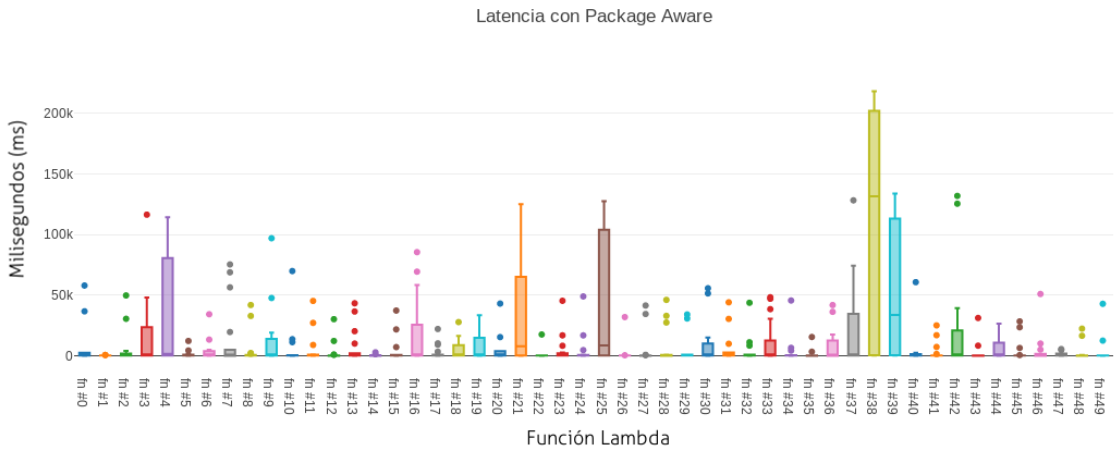

Figura 3.8: Valores de latencia en milisegundos para cada requerimiento enviado a las funciones entre #0 y #49 con algoritmo Package Aware

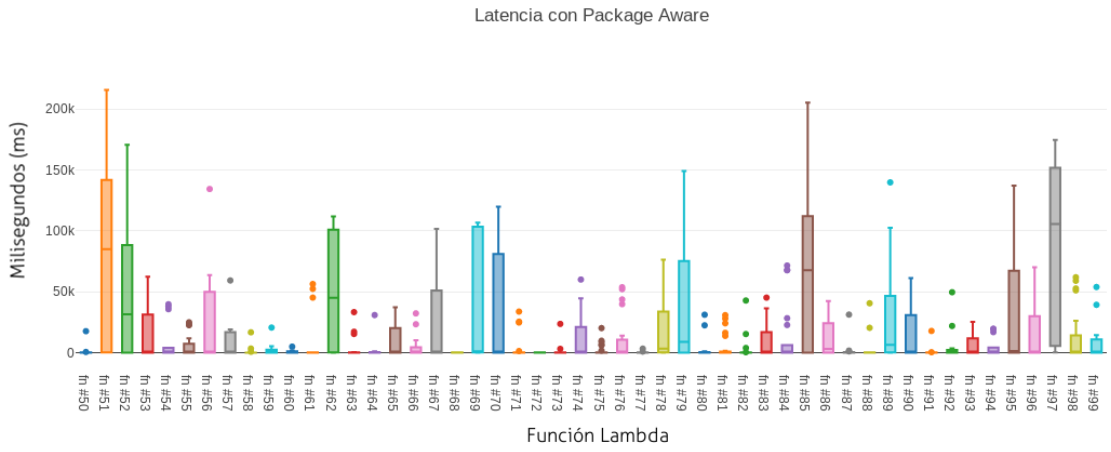

Figura 3.9: Valores de latencia en milisegundos para cada requerimiento enviado a las funciones entre #50 y #99 con algoritmo Package Aware

3.5. Comparación de los algoritmos

Como se puede observar en la figura 3.10, Package Aware, maximizando los hits en la cache de Pipsqueak, tiene el rendimiento más alto de todos. Los algoritmos convencionales todos tienen casi el mismo rendimiento, considerablemente menor que el de Package Aware. Adicionalmente, en la figura 3.11 se puede observar que la latencia promedio de Package Aware es sólo 13160 milisegundos, muy por debajo del más alto, Round Robin con 34719 milisegundos. Round Robin y Random tienen resultados muy similares y son los menos eficientes. Least Loaded es una significativa mejora en cuanto a latencia promedio con 27976 milisegundos, pero aún asi esta lejos de tener la eficiencia de Package Aware.

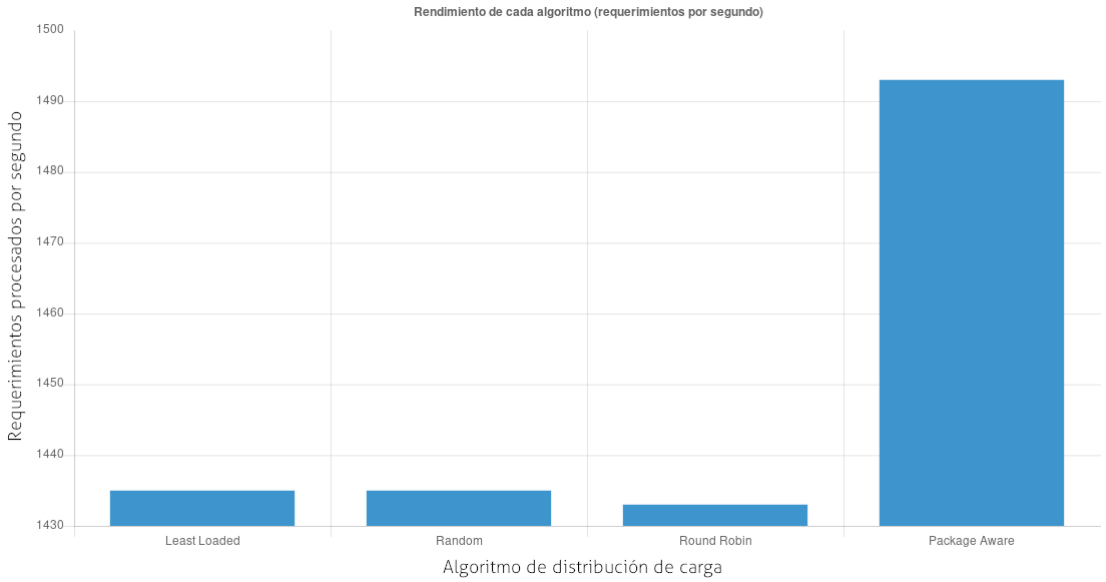

Figura 3.10: Rendimiento de cada algoritmo reportado por Pipbench (requerimientos por segundo)

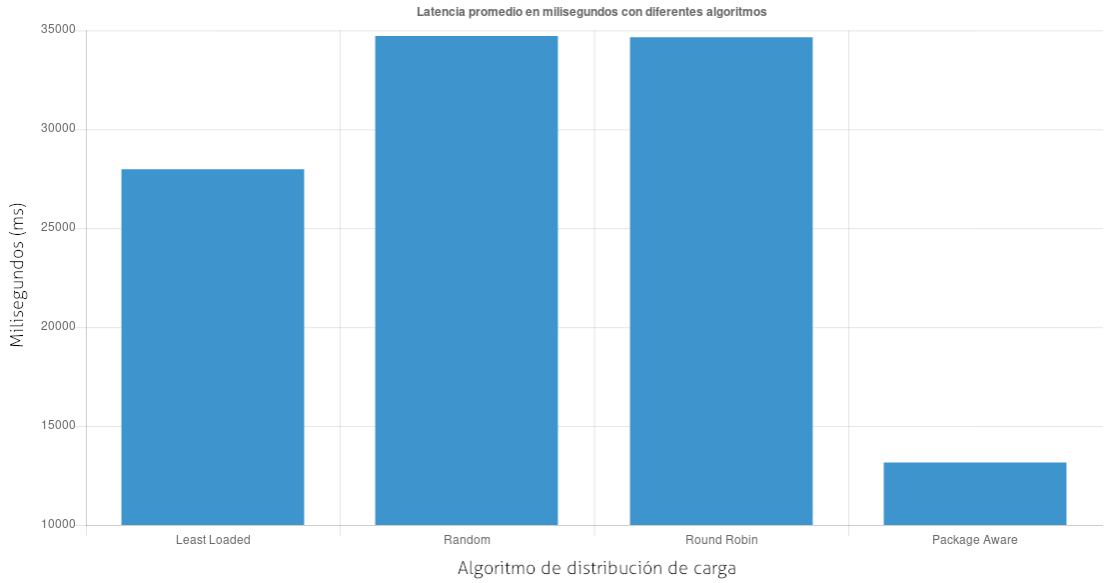

Figura 3.11: Latencia promedio reportada por Pipbench (milisegundos)

CAPÍTULO 4

Conclusiones y Recomendaciones

1. En una plataforma serverless, instalar las dependencias de cada función introduce una latencia de arranque significativa que puede perjudicar considerablemente a los clientes de la plataforma. Mantener estas dependencias en una cache y distribuir la carga de tal forma que se reutilicen las dependencias de la cache en lugar volver a descargarlas incrementa el rendimiento de la plataforma en un factor mayor que cualquier algoritmo de distribución de carga convencional como los que provee Nginx. Sin embargo, esta distribución no es tan sencilla, hay que encontrar un valor máximo de de carga para cada nodo lo suficientemente grande para maximizar el uso de la cache, pero lo suficientemente pequeño para no sobrecargar los nodos trabajadores. La determinación de este valor máximo depende de la configuración de los nodos trabajadores.

2. Entre los algoritmos convencionales de distribución de carga Least Loaded es ligeramente superior a los demás ya que si el cuello de botella en la plataforma serverless el la descarga e instalación de dependencias, este algoritmo tiene más probabilidad de enviar la carga a un nodo que no este ya ocupado descargando dependencias, evitando tener que esperar más tiempo para ejecutar la función. No obstante, usar un algoritmo inteligente que tome en cuenta el estado de las caches de paquetes en cada nodo es mucho mejor.

Recomendaciones

1. Al envíar requerimientos al planificador para realizar las pruebas, es necesario controlar el numero de requerimientos que se envían concurrentemente al planificador. En cada requerimiento, el planificador tienen que mantener abiertos dos

sockets, uno para esperar la respuesta del nodo trabajador y otro para reenvíar la respuesta al cliente. Hasta que el nodo trabajador no termine no se pueden cerrar esos sockets. Por lo tanto, si hay muchos requerimientos concurrentes, el servidor se puede quedar sin sockets disponibles y falla el envío de la carga al nodo trabajador. En nuestro caso, con una demora de 20 milisegundos entre cada requerimiento pudimos evitar este problema. En el futuro podríamos modificar el planificador para que mantenga una cola de requerimientos cuando se queda sin sockets disponibles y volver a intentar cuando se libere uno.

2. Los resultados generados por Pipbench no permiten fácilmente extraer los resultados para procesarlos o generar gráficos. Estos son escritos en un archivo de texto con un formato arbitrario en lugar de exponer un API que te permita escoger solo los datos que necesitas. Al trabajar con miles de resultados no es recomendable copiar estos valores a mano ya que es propenso a cometer errores que lleven a conclusiones incorrectas. Es mejor procesar los logs generados por Pipbench con una herramienta como Awk que pueda procesar las lineas del log y extraer los datos que se necesiten a través de scripts sencillos. En el futuro podríamos modificar Pipbench para poder configurar la forma en que se reportan los resultados.

BIBLIOGRAFÍA

[1] James Maguire. (2018, abril 11). Serverless Cloud Adoption At a Turning Point [online]. Disponible en: https://www.datamation.com/cloud-computing/serverless-cloud-adoption-at-a-turning-point.html.

[2] Gladys Rama. (2014, noviembre 13). Amazon Launches Docker Container Service and 'AWS Lambda' [online]. Disponible en: https://virtualizationreview.com/articles/2014/11/13/amazon-docker-container-service.aspx.

[3] Tomasz Janczuk. (2016, junio 9). What is Serverless? [online]. Disponible en: https://auth0.com/blog/what-is-serverless/.

[4] Scott Hendrickson *et al*, "Serverless computation with OpenLambda", Usenix login;, vol 41, pp. 14-19, 2016.

[5] Edward Oakes *et al*, "Pipsqueak: Lean lambdas with large libraries", IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW), pp. 395–400, 2017.

[6] Gustavo Totoy, Edwin F. Boza, y Cristina L. Abad, "An extensible scheduler for the OpenLambda FaaS platform", Min-Move workshop (co-located with ASPLOS), 2018.

[7] Cristina L. Abad, Edwin F. Boza, y Erwin van Eyk. "Package-aware scheduling of FaaS functions". Companion of the ACM/SPEC International Conference on Performance Engineering (ICPE), 2018.

[8] Edward Oakes *et al*, "SOCK: Rapid task provisioning with serverless-optimized containers", USENIX Annual Technical Conference (USENIX ATC), 2018.

[9] NGINX Inc. Nginx load balancing - HTTP load balancer [online]. Disponible en: https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/#method.

[10] David Cheney. (2015, julio 22). High performance servers without the event loop [online]. Disponible en: https://conferences.oreilly.com/oscon/open-source-2015/public/schedule/detail/41475.

[11] Martin. (2015. junio 11). Top Programming Languages Used in Web Development [online]. Disponible en: https://www.cleverism.com/programming-languages-web-development/.

[12] Dan Nanni. (2015, diciembre 27). What are good web server benchmarking tools for Linux [online]. Disponible en: http://xmodulo.com/web-server-benchmarking-tools-linux.html.