ESCUELA SUPERIOR POLITÉCNICA DEL LITORAL

Facultad de Ingeniería en Electricidad y Computación

BALANCEO DE CARGA MEJORADO PARA GRANJA DE SERVIDORES WEB

INFORME DE PROYECTO INTEGRADOR

Previo a la obtención del Título de:

INGENIERO EN COMPUTACIÓN

HAROLD RICARDO ARAGÓN INTRIAGO
JONATHAN FERNANDO PARRALES NEIRA

GUAYAQUIL – ECUADOR

AÑO: 2019

AGRADECIMIENTOS

Nuestro más sincero reconocimiento de primera mano a Dios, por la salud, sabiduría y conocimiento otorgado a lo largo de nuestra carrera para cumplir con todas nuestras metas propuestas y lograr cada uno de las consignas que se nos han presentado en nuestra vida académica desde la niñez hasta la actualidad en nuestra madurez, queremos agradecer de manera enfatizada a nuestra tutora Dra. Cristina Abad por guiarnos de la mejor manera en el desarrollo del proyecto, por su predisposición siempre oportuna y por la paciencia que nos ha otorgado como maestra y como tutora de este proyecto, finalmente agradecemos por sus sabios consejos y las críticas constructivas otorgadas a lo largo del desarrollo de este proyecto a nuestro profesor guía en la materia Dr. Boris Veintimilla.

De manera general agradecemos a cada uno de los actores que han intervenido en nuestra etapa académica dentro de esta universidad como es la ESPOL, ya que gracias a sus consejos y enseñanzas nos han inculcado a ser mejores personas y profesionales y de esta manera poder conseguir nuestras metas propuestas.

DEDICATORIA

Dedico de manera sincera y cordial el presentes proyecto a Dios por darme la vida y la sabiduría de poder conseguir mis metas, a mis padres por ser el pilar detrás de mí, para escalar cada peldaño que la vida me ha presentado, por el apoyo moral y económico para nunca desistir de mis estudios y saber sobresalir ante los demás siendo siempre humilde y arraigado en los principios fieles del hogar. – Jonathan Parrales

Dedico este proyecto, a mi madre que me ha guiado durante toda mi vida, a mi padre que siempre me ha apoyado y aconsejado, a mis familiares, amigos, profesionales y compañeros que han estado involucrados en mi formación profesional. – Harold Aragón

TRIBUNAL DE EVALUACIÓN

Ph.D Boris Vintimilla Burgos
PROFESOR DE LA MATERIA

Ph.D Cristina Abad Robalino
PROFESOR TUTOR

DECLARACIÓN EXPRESA

"La responsabilidad y la autoría del contenido de este Trabajo de Titulación, nos corresponde exclusivamente; y 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"

Harold Ricardo
Aragón Intriago

Jonathan Fernando
Parrales Neira

RESUMEN

Los algoritmos balanceadores de carga de hoy en día sufren un estancamiento en mejoras y nuevas producciones, en este proyecto se describe la implementación de dos nuevos algoritmos de políticas de balanceo de carga para el servidor web Caddy con el fin de mostrar mejoras en el rendimiento por encima de los algoritmos tradicionales ya implementados, los nuevos algoritmos: PASch y Consistent Hashing with Bounded Load, los cuales fueron sometidos a pruebas de rendimiento, midiendo la latencia en las respuestas, el número de peticiones respondidas exitosamente, el número de peticiones recibidas, dejando como resultados exitosos que Consistent Hashing with Bounded Load presenta el nivel de rendimiento a la par que los mejores algoritmos de balanceo de carga de la actualidad, mientras que PASch demostró un mejor desempeño que los demás algoritmos mostrando aproximadamente un 10% de mejoras en latencia y peticiones respondidas. Los resultados obtenidos expresan que las nuevas variantes de algoritmos para balancear cargas de trabajo pueden explotar en un mejor nivel el desempeño de una arquitectura en base a una granja de servidores, si se aplica de buena manera la elección de un algoritmo acorde al fin de uso, no siempre un mismo algoritmo va a desempeñar el mejor papel, este puede variar dependiendo el uso y el contexto de trabajo, ayuda mucho implementar algoritmos personalizados mezclando otros algoritmos a fin de conseguir un mejor rendimiento.

Palabras Clave: Balanceador, Algoritmo, Servidor, Aplicación Web, Servidor Web

ABSTRACT

Today's load balancing algorithms suffer from stagnation in improvements and new productions, this project describes the implementation of two new load balancing policy algorithms for the Caddy web server in order to show performance improvements by On top of the traditional algorithms already implemented, the new algorithms: PASch and Consistent Hashing with Bounded Load, which were subjected to performance tests, measuring the latency in the responses, the number of requests answered successfully, the number of requests received, leaving as successful results that Consistent Hashing with Bounded Load presents the level of performance as well as the best load balancing algorithms of today, while PASch demonstrated better performance than the other algorithms showing approximately 10% improvements in latency and requests answered. The results obtained express that the new variants of algorithms for balancing workloads can exploit at a better level the performance of an architecture based on a server farm, if the choice of an algorithm according to the end of use is applied in a good way , not always the same algorithm will play the best role, this may vary depending on the use and working context, it helps a lot to implement custom algorithms by mixing other algorithms in order to achieve better performance.

Keywords: Balancer, Alghoritms, Server, Web Application, Web Server.

ÍNDICE GENERAL

ABREVIATURAS

ESPOL	Escuela Superior Politécnica del Litoral
HTTPS	Protocolo Seguro de Transferencia de Hipertexto
IP	Protocolo de Internet
URI	Identificador Uniforme de Recursos
FAAS	Funciones como un Servicio
ID	Identificación
AIM	Metodología de Implementación de Aplicaciones
GLP	Licencia Pública General
URL	Localizador de Recursos Uniforme
AWS	Amazon Web Service

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

CAPÍTULO 1

1. INTRODUCCIÓN

Actualmente la tecnología gira alrededor de internet, en consecuencia, las empresas basan sus modelos de negocio esencialmente en esta red, puesto que de ahí radica un gran medio de comunicación y difusión de los productos o servicios que son ofertados, se gana la oportunidad de conquistar muchos clientes potenciales, puesto que su negocio está disponible a nivel mundial, sin una gran inversión.

En este capítulo se describe la descripción del problema, objetivos planteados y la descripción del marco teórico, con la descripción de los conceptos básicos sobre los que se trata en el proyecto.

1.1 Descripción del problema

Los servidores web son sistemas que gestionan las aplicaciones, recibiendo solicitudes de los clientes, las procesan y responde con contenido o retroalimentación de la solicitud. Las aplicaciones web que tienen muchos clientes a diario, funcionan con granjas de servidores para poder satisfacer todos los requerimientos que reciben y operar a una capacidad adecuada. Una de las formas de optimizar la eficiencia y la capacidad de los servidores es el uso de los balanceadores de carga de trabajo, que sirven para designar las solicitudes de mejor manera entre los servidores, evitando que se sobrecarguen, y así, ayudando a reducir la latencia y el congestionamiento.

Cada aplicación cuenta con sus propias necesidades al momento de manejar la carga de trabajo, que es de acuerdo con el tráfico que llegue a sus servidores, por eso cada balanceador de carga puede ser configurado con un algoritmo diferente Escoger el algoritmo con el que se desempeñará el balanceador de carga es importante, ya que determinará su rendimiento. Las empresas tienen sus aplicaciones con implementaciones con balanceadores de carga estándares, no obstante, si se lograra mejorar en

algún aspecto su rendimiento traería como efecto que estas empresas escatimen mucho menos en su inversión monetaria.

Caddy Server es un servidor web en el que nos hemos enfocado, ya que es nuevo en comparación con grandes en la industria y en nuestra búsqueda por mejorar su balanceador de carga hemos propuesto incluir dos nuevos algoritmos, uno es *Consistent Hashing with Bounded Loads* y el otro es *PASch*. Agregar nuevos algoritmos al balanceador de carga de Caddy traerá consigo mejoras de rendimiento para las aplicaciones que se ejecutan usando esta plataforma, que se constata a través de pruebas de rendimiento al servidor web, con distintos escenarios en los que se podría estar encontrando el servidor y además con algunas configuraciones, que incluyen las nuevas implementaciones.

Debido a que Caddy es de código abierto, provee la facilidad de proponer nuevos algoritmos para su balanceador de carga que se ajusten a su funcionamiento.

1.2 Objetivos

1.2.1 Objetivo general

Mejorar la distribución de carga de trabajo usando el servidor web Caddy, con la implementación de dos algoritmos de balanceo de carga para incrementar el rendimiento en aplicaciones web.

1.2.2 Objetivos específicos

- Implementar los algoritmos *Consistent Hashing with Bounded Loads* y *PASch* en el servidor web Caddy, añadiendo el algoritmo en el balanceador de carga para la optimización de la distribución de carga de trabajo en una granja de servidores web.
- Analizar el comportamiento de las aplicaciones, mediante pruebas de rendimiento usando los nuevos algoritmos de balanceo de carga, para identificar los cambios conseguidos en Caddy.

- Agregar una nueva actualización con la implementación de los nuevos algoritmos al repositorio máster de Caddy para que sea de acceso público.

1.3 Marco teórico

Caddy Server [1], como su nombre lo indica es un servidor web reciente, el cual está tomando mucha aceptación, gracias a sus muchos beneficios que le otorgan un realce ante su competencia más directa que vendría a ser *NGINX*. Su diferencia radica en la fácil legibilidad del código y a su implementación por defecto de *HTTPS*, lo cual beneficia mucho en la seguridad de las aplicaciones web, cosa que no se puede hacer de manera tan fácil en implementaciones de otras plataformas.

Sin embargo, Caddy no es el mejor en la industria ya que no aporta muchos beneficios que despunten su uso y es inferior en algunos aspectos si lo comparamos con *NGINX*. Los servidores web generalmente implementan balanceadores de carga para mejorar su desempeño en la administración de muchos servidores, actualmente las políticas para el balanceo de carga que implementa Caddy Server son las siguientes:

- *Aleatorio* (Predeterminado), asigna aleatoriamente las cargas de trabajo a uno de los servidores.
- *Least_Connection*, asigna carga al servidor con menos conexiones activas.
- *Round_Robin*, asigna las cargas de trabajo a todos los servidores de tal manera en que no se repitan las solicitudes que se envían.
- *First*, asigna la carga de trabajo al primer servidor disponible en un orden definido en el *Caddyfile*.
- *IP_Hash*, Asigna las cargas de trabajo basado en un hash usando la dirección IP del cliente.
- *URI_Hash*, aplica la misma lógica que *IP-Hash*, la diferencia es que en vez de usar la dirección *IP*, utiliza una dirección *URI*.

- *Header*, asigna la carga de trabajo mediante un valor de hash sobre un encabezado determinado que generalmente son *HTTP*, especificado después del nombre de la política.

A continuación, se detallan características de los algoritmos *Consistent Hashing with Bounded Loads* y *PASch*.

1.3.1 Consistent Hashing with Bounded Loads

Normalmente el manejo de la solicitud en un servidor web, como por ejemplo *HAProxy*, que antes de implementar el "*Consistent Hashing*", sucedía de que al momento que llega una solicitud se calcula un hash de lo que puede ser una parte del url, por ejemplo, el id y utilizando ese hash se elige un servidor disponible. El hash es un número muy grande y si se hace la operación módulo con el número de servidores disponibles, se obtiene el índice del servidor que se usará. Y eso funcionará mientras la cantidad de servidores sea estable, pero cuando aumentemos o disminuyamos servidores, la mayoría de las solicitudes que antes iban al mismo servidor, ahora irán a otro, desaprovechando la información que tenían cacheada en el anterior servidor.

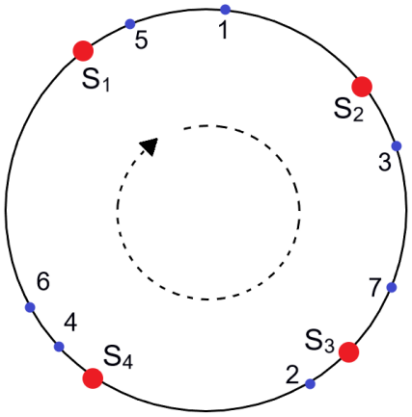

Figura 1 Consistent Hashing [2]

Es muy común que en las aplicaciones durante un período de tiempo se agreguen servidores y durante otro momento se apaguen algunos. Cuando se usa "*Consistent Hashing*", se resuelve el problema de añadir o quitar servidores [2], ya que como se puede ver en la figura 1 a cada servidor (**S_i**) se le da un rango de valores del hash. La representación en forma de anillo es porque así se ubican todos los valores, donde a cada uno de los S_i se le asigna una porción del anillo. Las solicitudes (**n**) serán

atendidas por el S_i que tenga a cargo la porción del anillo a la que pertenece el hash de la solicitud. A cada S_i se le asignan todas las solicitudes que le siguen a favor de las manecillas del reloj, hasta encontrar otro S_i. Como resultado la mayoría de las solicitudes llegan al mismo servidor al que habían sido asignadas antes. Aunque con esto logramos un mejor balanceo de la carga, también podemos llegar a un grave problema. Existe contenido con alta popularidad o muy solicitado en un corto periodo de tiempo, como normalmente suele ser en internet, es lo que pasa con un video viral, descuentos en tiendas virtuales o compras de boletos de un estreno, en general ocurre un pico en la carga. Si usamos "*Consistent Hashing*", en el momento que sucede algún caso de pico, un servidor va a recibir mucha carga, causando sobrecargas, ya que las solicitudes se mantendrán enviando al mismo servidor.

Para resolver este problema podemos hacer uso de "*Bounded Loads*", que agrega un límite o cota superior en la carga de cualquier servidor, en relación con la carga promedio de toda la granja de servidores. Para aplicarlo definimos un factor de equilibrio "c", que sea mayor que 1, "c" controla la cantidad de desequilibrio permitido entre los servidores. Por ejemplo, si c = 1.5, ningún servidor debe obtener más del 150% de la carga promedio. Se tienen otros valores relacionados, tales como:

- "m" es la cantidad de solicitudes pendientes, incluida la última
- n es la cantidad de servidores disponibles
- t es la carga de destino.

Cuando llega una solicitud:

- Se calcula la carga promedio = m/n
- Se calcula la carga de destino, t = c*m
- Se asignan capacidades a los servidores, así como una capacidad total, que es cm y la de cada servidor cm/n

Y así ningún servidor puede superar la parte de la carga que le toca por más de una solicitud [2].

Cuando se implementa el algoritmo *Consistent Hashing with Bounded Loads,* se espera que todo funcione de una manera parecida hasta antes de repartir la carga. Al recibir una solicitud se calcula su hash y el servidor más cercano o al que se elige para ese hash. Una vez que sucede esto, existen dos posibles escenarios, en el primero, tenemos al servidor elegido por debajo de su capacidad, es decir, no ha rebasado el límite superior, por lo tanto, se le asigna la solicitud. Si no es así, se elige al siguiente servidor en el anillo de hash, y se vuelve a verificar la capacidad hasta encontrar alguno. Tiene que haber uno, ya que es imposible que la carga de todos los servidores esté por encima de la media.

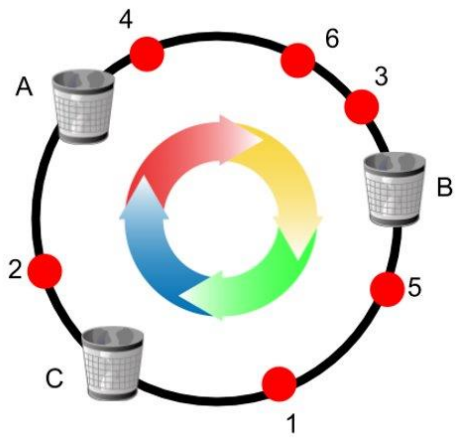

Figura 2 Consistent Hashing with Bounded Load [3]

Otra forma de ver en funcionamiento este algoritmo de una manera sencilla [3], es como en la figura 2, donde se muestran cestos y pelotas, suponiendo que estamos en el anillo de hash, donde tenemos ubicados unos cestos y unas pelotas. Los cestos son los servidores y las pelotas son las solicitudes y estas son ubicadas con funciones hash diferentes. Tenemos 6 solicitudes pendientes, la capacidad de los cestos se define en 2 y las pelotas se empiezan a ubicar en un orden específico, por ejemplo, su id. Entonces, tenemos que primero la pelota 1 entra en el cesto C, la 2 en el A, la 3 en el B, la 4 en el B, la 5 en el C y la 6 debería ubicarse en B, pero como está lleno sigue avanzando y como el C también está lleno, la pelota 6 termina llegando a la cesta A. Con la implementación de este algoritmo se evitan las sobrecargas y se logra que la memoria cache tenga muchos aciertos, independientemente de si cambiamos la cantidad de servidores disponibles en cualquier momento.

1.3.2 PASch

El algoritmo de *PASch* [4] se basa en la optimización en la importación de paquetes dentro de servidores con arquitectura *FaaS*, en especial debido a que trabaja con un enfoque en la importación de librerías que son dependencias de funciones.

Durante un estudio realizado sobre el *framework OpenLambda*, el cuál ejerce el papel de un servidor web basado en una arquitectura *FaaS*, se implementó el algoritmo *PASch* para optimizar la importación de paquetes. En la Figura 3 se observa un diagrama de la arquitectura modular de *OpenLambda*, básicamente usa un balanceador de carga (*NGINX* por defecto), también consta de una base de datos con funciones que son cargadas dependiendo del fin para que se use. El balanceador se encarga de distribuir las cargas de trabajo a un conjunto de máquinas virtuales, que ejecutan una función o varias acorde a su carga de trabajo.

Una gran desventaja que se tiene con este *framework* es que el balanceador de carga usa algoritmos clásicos, que no solventan el problema de latencia de las respuestas de las aplicaciones web, y desperdiciando recursos cuando un servidor tiene que cargar en memoria alguna dependencia de sus funciones internas, muchas veces este tipo de funciones llaman librerías que son muy grandes y consecuentemente el tiempo de carga va a ser grande.

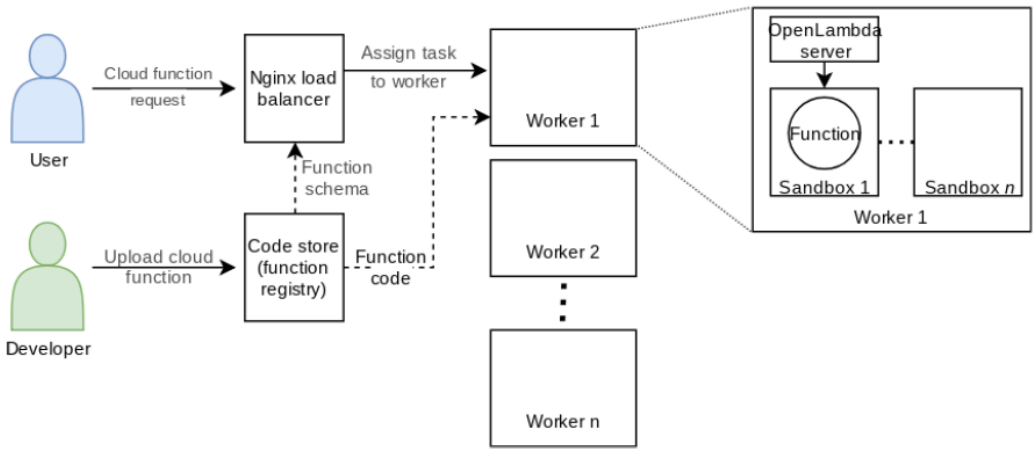

Figura 3 Arquitectura de Open Lambda con NGINX como balanceador de carga [4]

Para solventar esto se realizó la implementación de un algoritmo que está enfocado a mejorar la latencia en la carga de paquetes de dependencias dentro de los trabajadores.

PASch presenta una secuencia lógica de ejecución como se muestra en la Figura 4, en donde se aplica una función de afinidad entre dos trabajadores. Se sabe el límite de carga de trabajo que puede tener cada trabajador, esto ayuda a la función de afinidad a decidir a los mejores dos candidatos. Luego se verifica que el trabajador seleccionado tenga la mejor carga de trabajo posible y se cerciora nuevamente que la carga de trabajo de ese trabajador no exceda de su límite de carga.

Algorithm 1: Package-aware scheduler algorithm (PASch)

Global data: List of workers, $W = \{w_1, ..., w_n\}$, and their load thresholds, $T = \{t_1, ..., t_n\}$, mapping function M

Input: Function, f, largest required package, p

```
1  if (p is not null)then
       /* Get affinity workers                          */
2      ⟨a1, a2⟩ = M(p)
       /* Select target with least load                 */
3      if (load(w_a1) < load(w_a2))then
4          A := a1
5      else
6          A := a2
       /* If target is not overloaded, we
          are done                                      */
7      if (load(w_A) < t_A)then
8          Assign f to w_A
9          return
   /* Balance load                                      */
10 Assign f to least loaded worker, w_i
```

Figura 4 Pseudocódigo del Algoritmo PASch [4]

La función de mapeo realiza un papel muy importante respecto a la ejecución del algoritmo *PASch*, debido a que la afinidad de elección entre todos los trabajadores disponibles en el servidor conociendo previamente los límites de carga que posee cada trabajador. Esto garantiza que la elección devuelva dos candidatos que no vayan a sobrecargar su límite y por ende llegasen a provocar cuellos de botella en la ejecución de las tareas, básicamente la función analiza de entre todos los trabajadores escogiendo dos, a los cuales se les aplica una implementación de una función de *hashing consistente* acorde a un *ID* del paquete de trabajo que se pretende importar como se observa en la Figura 5.

Algorithm 2: Mapping function, M; given a package, returns two affinity nodes.

Global data: A consistent hash implementation, *consistent*, and value to be added to the package ID to map a second affinity worker to it, *salt*

Input: Package id, p

Output: Affinity workers for p, $\langle a1, a2 \rangle$

```
/* Get two affinity workers                    */
```

1 $a1 = \text{consistentHash.get}(p)$

2 $a2 = \text{consistentHash.get}(p + salt)$

3 **return** $\langle a1, a2 \rangle$

Figura 5 Función de Mapeo del Algoritmo PASch [4]

Luego de la implementación del nuevo algoritmo al balanceador de carga se notaron mejoras con el hit en el cacheo un 51.2% aproximadamente y la carga de trabajo del planificador a 64.1%, lo que garantiza un impacto positivo implementando este algoritmo con respecto al uso de balanceo de cargas con los algoritmos clásicos [4].

CAPÍTULO 2

Este capítulo describe la metodología utilizada en el proyecto, detallando cada una de las fases, de una manera general general.

2. METODOLOGÍA

Para llevar a cabo la solución de este proyecto se usó como base la Metodología de *Implementación de Aplicaciones de Oracle (AIM)* [5], donde se llevan a cabo 6 etapas: definición, análisis operacional, diseño de la solución, construcción, transición y producción.

2.1 Definición

Con el fin de cumplir nuestro objetivo de la manera más viable, se analizaron distintos escenarios, con los cuales definimos pequeñas metas a corto plazo, que llevaron hasta el análisis operacional. Para comprobar que el entorno de trabajo funcione correctamente, es importante ejecutar una o varias aplicaciones web con Caddy, usando varios servidores, automatizar el envío de solicitudes y verificar como se reparte la carga.

Una vez que se comprobó el funcionamiento del entorno de trabajo, se ejecutó una aplicación, con los distintos algoritmos de balanceo de carga que tiene el servidor web. Ya que el camino que se busca recorrer es el más corto y con mejores resultados, desde un inicio se hicieron pequeñas modificaciones al balanceador de carga, es decir, a los algoritmos *Consistent Hashing with Bounded Loads* y *PASch*. No es importante que las modificaciones iniciales al balanceador de carga giren en torno a la solución final, pero si se tiene que apreciar, que los cambios surgen efecto, se puede provocar una caída o enviarle toda la carga al mismo servidor.

Finalmente, hicimos pequeñas pruebas a los algoritmos de balanceo, se cambiaron los algoritmos obteniendo resultados que luego se podrán usar en comparaciones. Con estos objetivos que tenemos definidos a corto plazo y sabiendo los objetivos a largo plazo, se pudo empezar con la fase analítica y tener el camino libre para determinar cómo quedaría la solución final.

2.2 Análisis operacional

En la configuración de Caddy el eje principal sobre el cual se hace un manifiesto de las directivas que va a incorporar, así como todos los parámetros necesarios para personalizar el funcionamiento del servidor web acorde a las necesidades del usuario se realizan dentro del archivo llamado *CaddyFile*. Dentro del diseño de Caddy yace la idea de que todo sea un complemento o *plug-in*, teniendo así una semejanza a un rompecabezas, el objetivo de esto es hacer el código muy sostenible, debido a que solo se usa lo necesario para el funcionamiento de algún sistema y si a futuro se necesita alguna característica extra, ésta se acople a lo ya existente sin afectar lo demás, sino más bien sumando funcionalidad [6].

Del análisis que hicimos en el entorno de trabajo, obtuvimos ciertas características relevantes [6] [7]:

- Casi todo en Caddy es un complemento.
- Cada una de las directivas del archivo de configuración o “CaddyFile”, son complementos.
- Tiene un plugin de gancho de evento, que ejecuta una función cuando Caddy emita algún evento en particular.
- Caddy posee un proxy con soporte de backend múltiple.
- El balanceador de cargas se encuentra en el proxy.
- Cuenta con varios parámetros para ayudar al balanceador de carga, como pruebas de errores, verificadores de estado o conexiones.

Teniendo en cuenta estas características, se analizó el funcionamiento que tiene Caddy antes de realizar cualquier modificación, se inició con las metas a corto plazo definidas previamente. Una vez, que ejecutamos las aplicaciones con diferentes algoritmos tenemos algunas revelaciones:

- Para cambiar de algoritmo balanceador de carga, se modifica el parámetro llamado política del proxy.
- Agregar un nuevo algoritmo es agregar una nueva política.

- Se tiene que especificar que verificar o probar en la configuración del proxy.

Como parte del análisis también se revisó, cómo está estructurado el código fuente tal como viene del repositorio y se cumplió el último objetivo a corto plazo que es modificar una parte del balanceador de carga. Por otro lado, el proxy está dividido en una serie de componentes, entre los cuales constan las políticas de los algoritmos de balanceo de carga que se encuentran en el archivo "*policy.go*".

Las revelaciones finales del análisis operacional son:

- Los algoritmos están registrados como políticas.
- Los servidores se representan como un objeto y todos se guardan en una lista.
- Cada política cuenta como una función que recibe la lista de servidores y la solicitud, y retorna el servidor asignado.
- Modificar la política que está seleccionada sí afecta el comportamiento del balanceo de la carga.

2.3 Diseño de la solución

En la Figura 6, se puede ver un diagrama general de una aplicación ejecutándose, usando Caddy, donde se muestra el balanceador de cargas que es la ubicación en la que se centró para cumplir los objetivos. Teniendo más claro que parte se modificó, ahora se detallará el proceso a seguir para agregar un nuevo algoritmo.

1. Se crea el objeto del algoritmo a agregar, con sus parámetros.
2. Registrar la nueva política en la función de inicialización.
3. Si es necesario, definir funciones que pueden ser útiles, por ejemplo, calcular un hash.
4. Definir la función "*Select*", usando el objeto del algoritmo.
5. Realizar las operaciones necesarias para escoger un servidor de la lista.

6. Asegurarse que retornemos un tipo “UpstreamHost”.

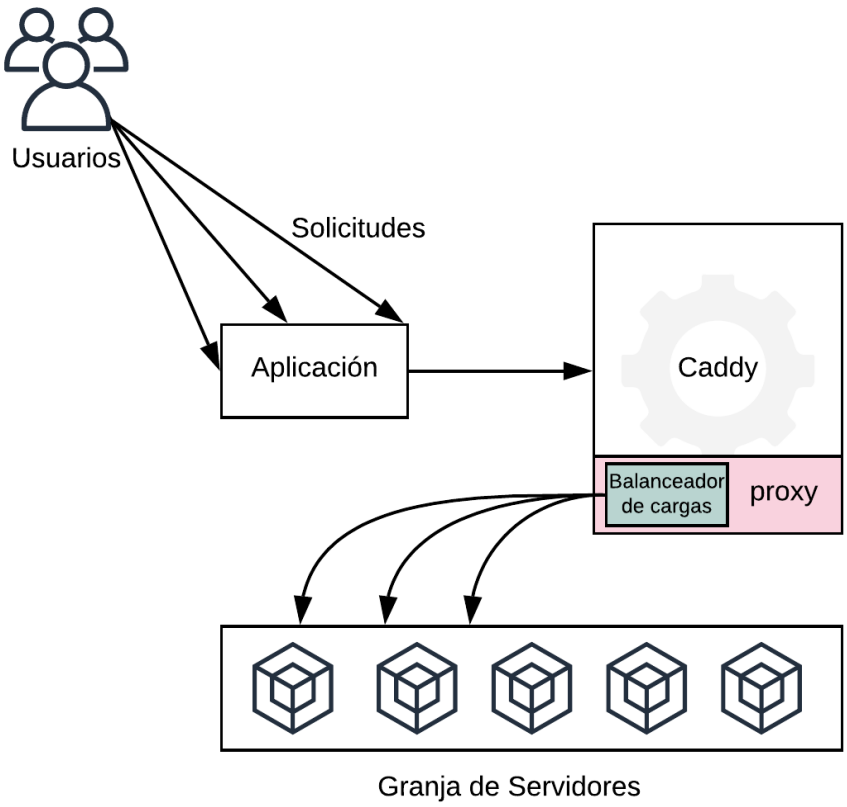

Figura 6 Aplicación general con Caddy [Elaboración Propia]

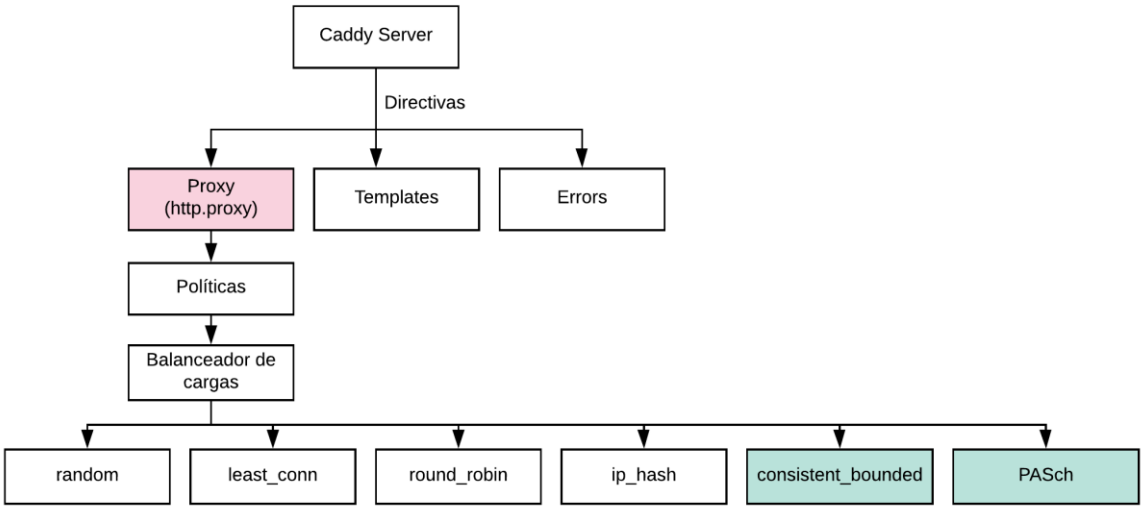

Figura 7 Diagrama Jerárquico de Caddy Server [Elaboración Propia]

Por cada nuevo algoritmo implementado, se añadió una hoja al árbol jerárquico de Caddy Server como muestra la Figura 7. Comprobamos el funcionamiento de los algoritmos a través de pruebas de rendimiento. Se ejecutó pruebas en distintos escenarios, para verificar si existen cambios con respecto a los otros algoritmos y con respecto al balanceo de carga en general de Caddy.

Debido a la necesidad de realizar pruebas de rendimiento del funcionamiento del sistema usando las políticas de balanceo de carga nativas, y extrapolar los resultados obtenidos con la implementación de los dos nuevos algoritmos, se usaron tres herramientas para medir el rendimiento de los servidores, las cuales son:

- *Bombardier*, Herramienta de benchmark, desarrollada en Go, utiliza una implementación de HttpFast para el tráfico http, que es mucho más rápida que la implementación nativa de Go, el análisis de resultados puede ser almacenado en archivos según sea la conveniencia del usuario [8].
- *Httperf*, Herramienta para crear carga de trabajo y medirla en servidores web, su sistema se caracteriza por ser robusto y mantener constantes las cargas de trabajo sobre servidores web, puede generar archivos log con los resultados generados [9].

- *Siege,* Herramienta para medir la carga de trabajo de servidores web, se caracteriza por poder implementar el uso de peticiones concurrentes hacia el servidor simulando de mejor manera el escenario de carga de trabajo real, puede generar archivos csv con los resultados de las cargas de trabajo [10].

2.4 Construcción, transición y producción

Después de tener claro cuál es el camino que, a tomar para la solución, se empezó con la construcción, tomando como un guion a seguir lo planteado anteriormente. Con la implementación y testeo de funcionamiento de un nuevo algoritmo, para que la solución sea aceptada como parte de la rama principal de Caddy, es necesario cumplir con los siguientes requerimientos:

- Compatibilidad con las licencias que maneja Caddy (licencias GLP).
- Si se necesita modificar el *CaddyFile*, hay que hacer una solicitud extra.
- Presentar documentación pertinente sobre el funcionamiento, instalación y pruebas sobre el nuevo complemento desarrollado.

La implementación de los algoritmos de *Consistent Hashing with Bounded Loads* y *PASch*, buscan demostrar que el rendimiento del balanceador de carga de Caddy Server va a incrementar su eficiencia, tal como se ha demostrados en otros estudios realizados como el caso de Vimeo y la implementación en *OpenLambda*. Con esto se espera que las aplicaciones puedan hacer pruebas y si consideraran que tienen mejoras notables con estos algoritmos, usen las políticas en producción, además se quiere aumentar el uso de Caddy en nuevas aplicaciones, ya que tendrán más opciones de balanceo de carga.

CAPÍTULO 3

En este capítulo se detalla la fase de construcción y transición que conlleva la ejecución de todas las pruebas sobre la infraestructura en *AWS*, ejecución de pruebas de funcionamiento, implementación de algoritmos en el servidor Caddy, la implementación detallada de cada uno de los algoritmos utilizados, *PASch* y *Consistent Hashing with Bounded Loads*.

3. SOLUCIÓN

3.1.1 Infraestructura en Amazon Web Service

Máquina	CPU	Ram	Almacenamiento	SO
Caddy Balancer	8	32GB	20GB	Ubuntu 18.04LTS
Slave 1	1	2GB	20GB	Ubuntu 18.04LTS
Slave 2	1	2GB	20GB	Ubuntu 18.04LTS
Slave 3	1	2GB	20GB	Ubuntu 18.04LTS
Slave 4	1	2GB	20GB	Ubuntu 18.04LTS
Slave 5	1	2GB	20GB	Ubuntu 18.04LTS

Tabla 1 Disposición de Máquinas Virtuales en AWS [Elaboración Propia]

La implementación del proyecto se realizó de manera local usando una distribución Linux como lo es Ubuntu 18.04 LTS, que provee el escenario necesario para el desarrollo deseado. En este ambiente local se realizó la instalación del lenguaje Go 1.12.7 como base sobre el cual está desarrollado Caddy Server.

Dentro del ambiente de Amazon se usó el servicio de *Amazon Elastic Compute Cloud (Amazon EC2)*, dentro del cual se instanciaron 6 máquinas virtuales según se describe en la Tabla 1.

Todas las solicitudes serán distribuidas por Caddy, que está en la máquina "*Caddy Balancer*" y serán procesadas por las máquinas que tienen la aplicación que son los "*Slave*", la configuración básica para una máquina de clase esclava se detalla a continuación:

- Instalar apache2 dentro de cada máquina virtual.
    - *sudo apt install apache2*
- Instalar Go 1.12.7
    - *sudo snap install go --classic*
- Clonar repositorio de la aplicación Ran, que es un servidor web de archivos estático escrito en Go, dentro del directorio "/var/www/".
    - sudo git clone https://github.com/m3ng9i/ran
- Descargar paquete de imágenes de prueba, como archivos a servir
    - *wget --no-check-certificate –r 'https://docs.google.com/uc?export=download&id=1Q4FJxX74qVQ2-Zt0r93vSXc8XXYU6aUm' -O images.zip*
- Descomprimir archivo en una carpeta de nombre "*images*", para fines prácticos del entrenamiento.
- Crear servicio de inicio al arranque de la máquina virtual dentro del directorio "/etc/apache2/sites-available/ran.conf"
    - *<VirtualHost *:80>*
        ServerAdmin ubuntu@localhost
        ServerName ran.test.net
        DocumentRoot /var/www/ran
        ErrorLog ${APACHE_LOG_DIR}/ran-error.log
        CustomLog ${APACHE_LOG_DIR}/ran-access.log combined
        ProxyPass / http://localhost:8000/
        ProxyPassReverse / http://localhost:8000/

</VirtualHost>

- Ingresar la configuración de Apache2 para que trabaje Ran en el host de localhost.

[Unit]
Description=Testing Ran web server
[Service]
ExecStart=/var/www/ran/ran -p=8000
WorkingDirectory=/var/www/ran
StandardOutput=journal
StandardError=inherit
SyslogIdentifier=ran
User=ubuntu
Type=simple
Restart=always
RestartSec=10
[Install]
WantedBy=multi-user.target

- Reiniciar servicios de apache2

La máquina virtual principal es la que va a soportar el tráfico de Internet, en esta máquina se configuró el balanceador de carga usando a Caddy. Para la configuración, solo basta tener un archivo *Caddyfile* que contenga las directivas de funcionamiento de Caddy.

```
:80
proxy / ec2-52-207-153-236.compute-1.amazonaws.com:80 ec2-34-204-48-192.compute-
1.amazonaws.com:80 ec2-3-91-71-224.compute-1.amazonaws.com:80 ec2-54-221-185-
90.compute-1.amazonaws.com:80 ec2-54-234-214-237.compute-1.amazonaws.com:80 {
  policy uri_hash
}
```

Figura 8 Archivo de Configuración de Caddy Server [Elaboración Propia]

Como se observa en la Figura 8, el *Caddyfile* contiene la directiva en el puerto por default que va a funcionar. Con el objetivo de usar a Caddy

como balanceador, se especifican las url hacia dónde se va a redirigir el tráfico.

En la Figura 9, observamos al balanceador de carga Caddy y a sus esclavos.

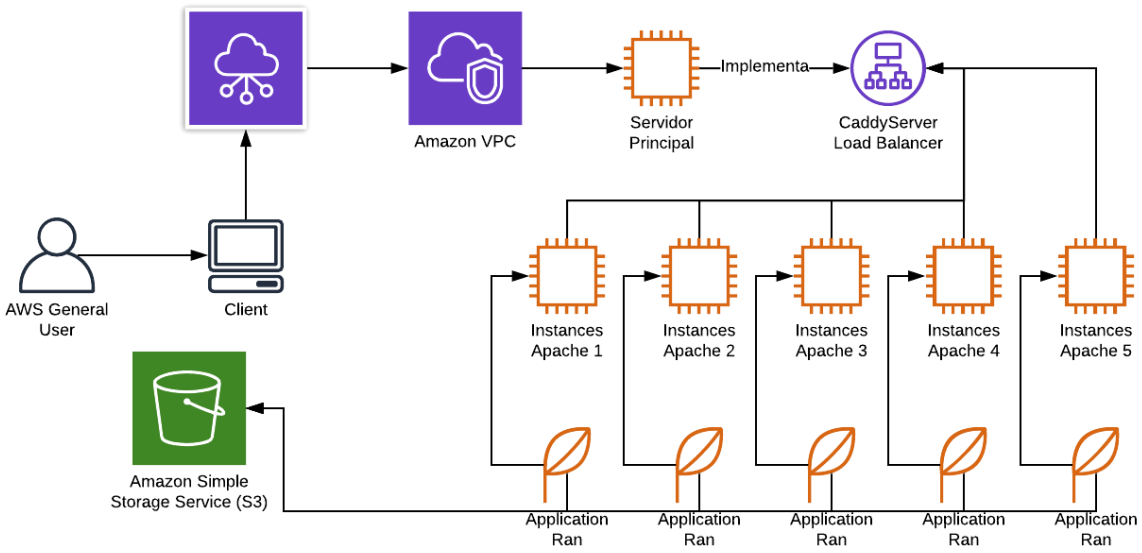

Figura 9 Diagrama del Ambiente de AWS [Elaboración Propia]

3.1.2 Pruebas de funcionamiento de Caddy

Las pruebas de funcionamiento se realizaron, usando *Bombardier, Httperf* y *Siege* como las primeras formas de medir el desempeño de las máquinas virtuales y verificar si se necesita escalar. Para ello se realizó un pequeño caso de prueba en donde se ejecutó un script que realiza la petición de archivos al servidor principal de Caddy y que, a su vez, este nos redirige hacia los esclavos.

Después de ejecutar las debidas pruebas y verificar que el funcionamiento de Caddy se mantiene a lo largo de las distintas pruebas. Se puede dar por finalizada la etapa de instalación y configuración del servicio. Cada vez que se ejecuta una prueba se tiene que reiniciar todo, para que la caché sea borrada.

3.1.3 Implementación de un algoritmo de balanceo en Caddy

La implementación de cada uno de los algoritmos que sirven dentro de la función de balanceador de carga, están dentro del archivo "*policy.go*", el cual como característica principal posee una interfaz como se puede apreciar en la Figura 11, en donde cada uno de los métodos que se vayan a implementar para balancear cargas de servidores deben incorporar esta interfaz, en la cual se describen todas las funciones que deben recibir una referencia a un objeto *HostPool* que es una estructura tipo arreglo de objetos *UpstreamHost*, que básicamente son la clase que enmascara a un servidor, esta abstracción ayuda a mantener a los servidores como objetos, de los cuales podemos obtener diversas características, como el número de conexiones disponibles, que se permiten utilizar dentro de los algoritmos balanceadores para optimizar las distribución de cargas de trabajo, el segundo parámetro que se recibe es la referencia a la petición que el usuario demanda, como respuestas todas funciones deben devolver un objeto del *UpstreamHost*, que es la referencia hacia el servidor que va a responder la solicitud.

```
// HostPool is a collection of UpstreamHosts.
type HostPool []*UpstreamHost

// Policy decides how a host will be selected from a pool.
type Policy interface {
        Select(pool HostPool, r *http.Request) *UpstreamHost
}
```

Figura 10 Interfaz de Caddy para Políticas de Balanceo de Carga [Elaboración Propia]

3.1.4 Implementación de PASch

En la implementación realizada de *PASch* se efectuaron modificaciones que involucraron ciertos cambios en la estructura definida para poder incorporar este algoritmo a Caddy, se definió una estructura como se observa en la Figura 12 que posee tres atributos, el primero es una referencia a la clase Consistent, que nos da acceso a características de

hashing, el segundo atributo nos muestra un parámetro personalizable entero que representa la carga máxima que puede tener un servidor, y como último parámetro observamos un diccionario clave, valor en donde la clave es la url de cada servidor y el valor es la referencia *UpstreamHost* hacia el servidor.

Entonces hablando plenamente de la funcionalidad lo que realiza la función es que llegada una petición si el anillo hash con los servidores está vacío lo inicializa, así mismo inicializa el diccionario con el nombre de cada servidor como clave y la referencia de cada servidor como valor, luego se escoge el servidor acorde a la petición que llega, usa *hashing consistente*, si este servidor escogido tiene menos conexiones que el umbral definido previamente, entonces se aplica una función para seleccionar al servidor con menos conexiones como se observa en la Figura 11, y retorna el servidor menos ocupado para ese instante de tiempo como se observa en la Figura 12.

```
//Return worker least loaded
func (r *PackageAware) selectLeastConnHost(pool HostPool) *UpstreamHost {
        targetIndex := 0
        for i := 1; i < len(pool); i++ {
                if pool[i].Conns < pool[targetIndex].Conns {
                        targetIndex = i
                }
        }
        return pool[targetIndex]
```

Figura 11 Función para Seleccionar el Host con Menos Conexiones [Elaboración Propia]

```
type PackageAware struct {
        hashRing      *consistent.Consistent
        loadThreshold int64
        workerNodeMap map[string]*UpstreamHost
}

//Select selection to worker most free
func (r *PackageAware) Select(pool HostPool, request *http.Request) *UpstreamHost {
        if r.hashRing == nil {
                r.hashRing = consistent.New()
                r.workerNodeMap = make(map[string]*UpstreamHost)
                r.loadThreshold=60 //To-Do JP:Parametrizar
                for _, host := range pool {
                        if host.Available() {
                                r.hashRing.Add(host.Name)
                                r.workerNodeMap[host.Name] = host
                        }
                }
        }
        bestHost, err := r.hashRing.Get(request.RequestURI)
        if err != nil {
                log.Println("[ERROR] There are no hosts in the Hash Ring: ", err)
        } else {
                host := r.workerNodeMap[bestHost]
                if host.Conns >= r.loadThreshold { // Find least loaded
                        host = r.selectLeastConnHost(pool)
                }
                return host

        }
        return nil

}
```

Figura 12 Implementación de PASch [Elaboración Propia]

3.1.4.1 Pruebas de funcionamiento

Para las pruebas de funcionamiento se realizaron dos pruebas pequeñas que garantizaron que algunos escenarios se cumplan tal y como se espera, además de esto también se comprobó mediante el proceso de pruebas de rendimiento que la aplicación funcione como debe. A continuación, se muestra el código de testeo que se le realizó al algoritmo en la Figura 13.

```
func TestPASch(t *testing.T) {
        pool := testPool()
        paschPolicy := &PackageAware{}
        req := httptest.NewRequest(http.MethodGet, "/", nil)

        h := paschPolicy.Select(pool, req)
        h.Unhealthy = 1
        if h == pool[0] {
                t.Error("Expected pasch policy host to be a host different of first.")
        }

        j := pool[0]
        j.Unhealthy = 3
        k := pool[1]
        k.Unhealthy = 2
        l := pool[2]
        l.Unhealthy = 1
        h = paschPolicy.Select(pool, req)
        if h != pool[2] {
                t.Error("Expected pasch policy host to be third host.")
        }
}
```

Figura 13 Implementación Test para PASch [Elaboración Propia]

El primer escenario describe que, si un servidor está sobrecargado, entonces el algoritmo debería de tratar de buscar entre los otros servidores y devolver otro, caso contrario la aplicación debería notificar que el servidor seleccionado no es el más apropiado.

El segundo escenario describe que, si existen varios servidores que se encuentran ocupados y su carga de trabajo no se encuentra distribuida de

manera ascendente correspondientemente a cada servidor de los disponibles, el algoritmo debe responder al servidor con menos carga.

3.1.5 Implementación de Consistent Hashing with Bounded Load

En la implementación de este algoritmo, se define una estructura como se observa en Figura 14, que posee el anillo de *Hash* sobre el cual trabajaremos para el hashing consistente. La función *Select* que es la función principal de todos los algoritmos de balanceo de carga, primero pregunta si el anillo de hash está inicializado, si no es así lo inicializa, con los servidores que estén disponibles, luego se obtiene un servidor usando una librería que ya implementa este algoritmo de una manera eficiente, para que solo se tenga que enviar la url y este retorna un servidor utilizando *consistent hashing with bounded loads*, a este servidor se le incrementa la carga dentro del anillo del hash y retorna el servidor escogido. Cuando se tenga la respuesta de la solicitud, se disminuye la carga de este servidor, dentro del anillo de hash para tener la referencia.

```
type Consistent_Hashing_Bounded struct {
  hashRing *consistent.Consistent
}

func (r *Consistent_Hashing_Bounded) HostDone(hostName string) {
  r.hashRing.Done(hostName)
  return
}

func (r *Consistent_Hashing_Bounded) Select(pool HostPool, request *http.Request) *UpstreamHost {
  if r.hashRing == nil {
    r.hashRing = consistent.New()
    for _, host := range pool {
      if host.Available() {
        r.hashRing.Add(host.Name)
      }
    }
  }
  bestHost, err := r.hashRing.GetLeast(request.RequestURI)
  if err != nil {
    log.Println("[ERROR] There are no hosts in the Hash Ring: ", err)
  } else {
    r.hashRing.Inc(bestHost)
    for _, host := range pool {
      if host.Name == bestHost {
        return host
      }
    }
  }
```

Figura 14 Implementación de Algoritmo Consistent Hashing with Bounded Loads [Elaboración Propia]

3.1.6 Pruebas de rendimiento a los algoritmos de balanceo de cargas

Para el desarrollo de las pruebas de rendimiento se implementó un pequeño script escrito en *Python 3*, que ayuda al envió de solicitudes usando *Bombardier* y posteriormente guardando los resultados en archivos *json* para luego obtener los resultados finales.

A la par de lo mencionado, se usó una base de datos *S3*, en la cual se encuentra una lista de imágenes, con las que posteriormente se genera una lista con sus nombres respectivamente, entonces se crea un archivo que contiene esta lista de nombres. Usando este archivo se ejecuta un lazo repetitivo, en donde usando *Bombardier* se envían solicitudes al servidor Caddy para que las balancee entre los servidores esclavos, a fin de que ellos ejecuten una carga de trabajo más o menos equitativa.

CAPÍTULO 4

Esta sección describe los resultados de la implementación de *Caddy Server* con los dos nuevos algoritmos: *PASch* y *Consistent Hashing with Bounded Loads*, también se compartirán las conclusiones a las que se llegó y se darán algunas recomendaciones con respecto a escoger un algoritmo de balanceo de cargas y a la implementación de un algoritmo, adicionalmente se mencionarán los trabajos futuros.

4. ANÁLISIS DE RESULTADOS

Los resultados obtenidos en este estudio se tabulan a continuación, primeramente, cabe recalcar que las pruebas se las realizó conservando los mismos parámetros para los escenarios de prueba, por lo que el desenvolvimiento de cada uno de los algoritmos es indiferente de la aplicación y si un algoritmo tiene mejor rendimiento que otro es debido a su naturaleza y por ende que está mejor adecuado a los escenarios de pruebas que se ejecutaron.

4.1 Resultados de escenarios de pruebas

- En las pruebas, con los algoritmos *Ip Hash* y *First*, se obtuvieron resultados que no tenían comparación con los demás, están muy por detrás en estos escenarios, así que solo se considerarán los 6 algoritmos restantes.
- *Bombardier*, aplicación utilizada para hacer la prueba de rendimiento arrojó los resultados de latencia que se pueden observar en la Figura 15, los tiempos expresados en segundos van desde 5.8s con *Uri Hash* hasta 4.8s con *PASch*, el resto de los algoritmos se ejecutaron en latencias dentro del rango especificado.

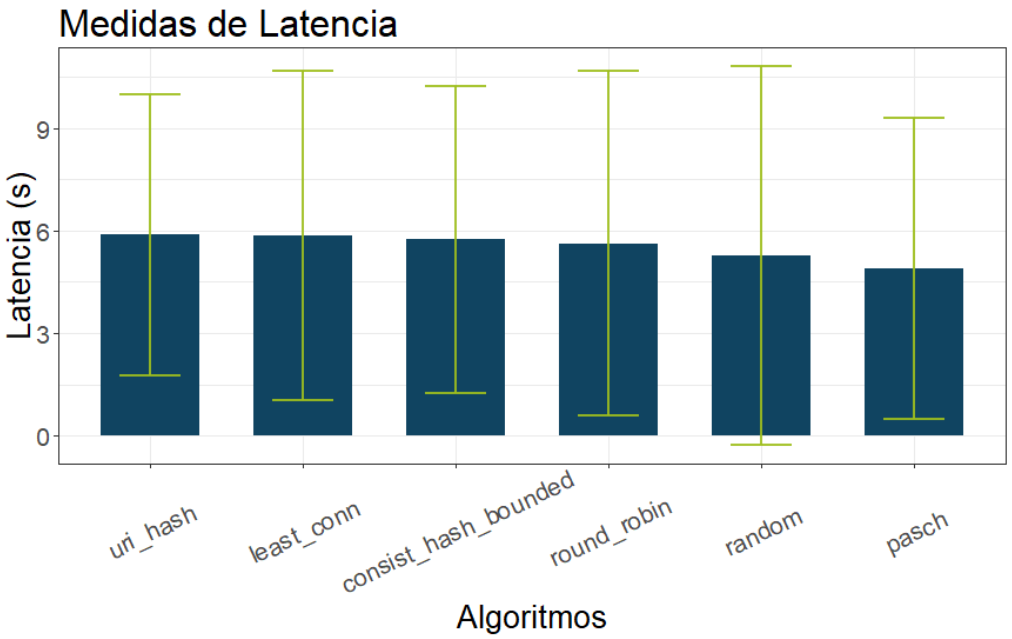

Figura 15 Medidas de Latencia de 6 Algoritmos Implementados en Caddy

[Elaboración Propia]

- Dentro del test se realizaron una serie de peticiones con el fin de sacarle el máximo rendimiento a cada servidor, dejando como resultados para cada algoritmo como se observa en la Figura 16, un máximo de 22124 solicitudes atendidas para *PASch* y 18014 solicitudes como mínimo para *Least Connections*.

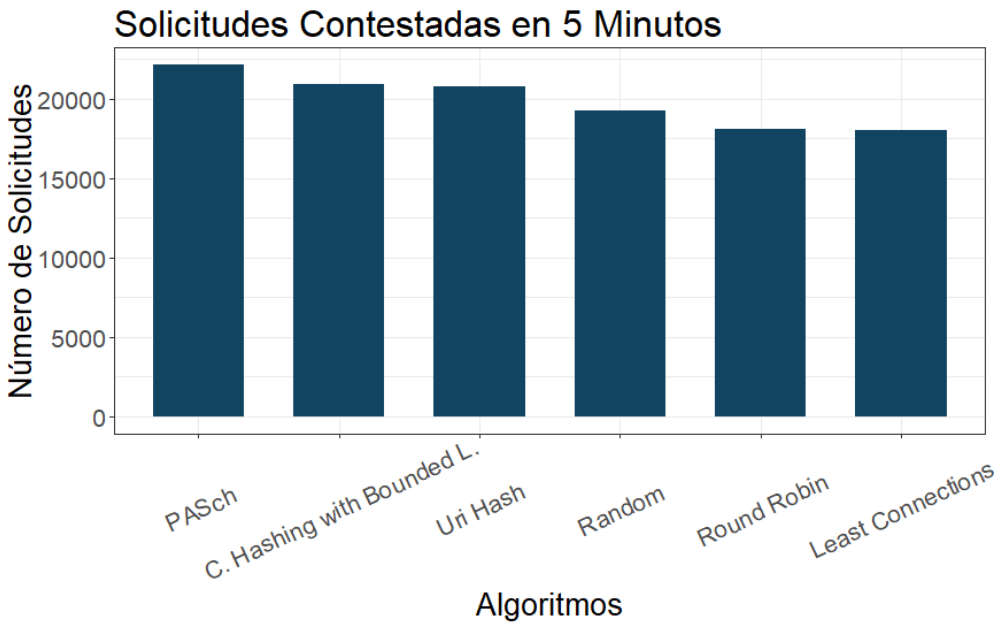

Figura 16 Solicitudes Contestadas cada Algoritmo [Elaboración Propia]

4.2 Revelaciones

Con los resultados de las pruebas, obtuvimos las siguientes revelaciones:

- Los Algoritmos *First* e *Ip_hash*, no tienen comparación con los otros algoritmos en cuanto a resultados, ya que *First* le envía todo al mismo servidor, que es el primero configurado, ya que lo encuentra disponible, causando que existan muchos tiempos de espera para enviar más solicitudes. Ip hash necesita tener diferentes orígenes para funcionar correctamente.
- Como se aprecia en la Figura 15 y la Figura 16, el algoritmo *PASch* es el que tuvo un mejor rendimiento, ya que consta con la menor latencia, en cada prueba y fue con el que se atendieron más solicitudes entre todos los algoritmos.
- El algoritmo *Random* es menos consistente en sus resultados, mientras que los demás algoritmos mantienen resultados similares entre cada prueba.

4.3 Conclusiones y recomendaciones

Conclusiones

- La implementación de 2 nuevos algoritmos de balanceo de carga en Caddy da más opciones para personalizar las aplicaciones, en muchos casos, estos algoritmos tendrán un mejor rendimiento que los otros ya implementados.
- El algoritmo *PASch* mantiene la menor latencia, con diferentes cargas de trabajo, es consistente entre los diferentes escenarios de pruebas.
- El algoritmo *Consistent hashing with Bounded Loads* tiene un buen rendimiento entre las diferentes cargas de trabajo y su mejor rendimiento se da cuando se prioriza aprovechar la caché.
- Escoger el algoritmo de balanceo de carga correcto para un caso en particular, depende de los recursos disponibles, la caché, el origen de las solicitudes y de la complejidad la aplicación.

Recomendaciones

- Antes de escoger un algoritmo de balanceo de carga para trabajar en producción, se tienen que realizar varias pruebas, ya que lo que sirve para una aplicación puede que no sirva para otra, así que hay que buscar el algoritmo que encaje en conjunto con todo el sistema.

- Cuando se hagan las pruebas de rendimiento se debe procurar tomar cargas de trabajo reales usando aplicaciones que ya se encuentren en producción, o si es para otras aplicaciones, tratar de emular cargas de trabajo reales que tengan coherencia con el funcionamiento de la aplicación.

- Si se quiere realizar la implementación o una mejora de un algoritmo de balanceo de carga, se debe procurar usar todos los recursos que provee el servidor, a fin de obtener un rendimiento real de las capacidades del algoritmo junto con el uso de recursos completos del servidor.

4.4 Trabajos futuros

Como parte de los futuros trabajos se considera necesario realizar pruebas de rendimiento que involucren escenarios variados, diferentes de los tratados en este proyecto, para que de esta manera quede completamente cubierto los diversos casos en donde se pueda sacar el máximo provecho del balanceo de carga usando una granja de servidores web.

Una posible mejora que se considera factible es realizar modificaciones a los algoritmos usados para que se puedan modificar sus parámetros dinámicamente, con la finalidad de mejorar su rendimiento, puesto que se observó de manera específica que variando el parámetro *loadthreshold* del algoritmo *PASch*, se puede variar la latencia de este algoritmo. Del mismo modo realizar optimizaciones al código nunca está demás con miras a conseguir mejoras en los tiempos de respuesta y en el asertividad del balanceo de las cargas de trabajo de los servidores web.

También se plantea cubrir nuevos servidores web que fomenten la fácil implementación de los algoritmos descritos en este proyecto, se considera con mayor posibilidad realizar el desarrollo en *HAProxy*, que es un servidor web que contiene características amigables y que además mantiene un prestigio de buen rendimiento dentro de la comunidad, lo que da un realce de valor ante los usuarios y mejora de forma positiva el uso que pueda tener posterior a realizar el estudio.

5. BIBLIOGRAFÍA

[1] Light Code Labs (2019, Marzo). Caddy Features (1.0.0) [Online]. Disponible en: https://caddyserver.com/features

[2] Mirrokni, V., Thorup, M. y & Zadimoghaddam, "Consistent Hashing with Bounded Loads", Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2018), pp. 587-604, Enero, 2018.

[3] Vahab Mirrokni y Morteza Zadimoghaddam (2017, Julio, 27). Consistent Hashing with Bounded Loads [Online]. Disponible en: https://ai.googleblog.com/2017/04/consistent-hashing-with-bounded-loads.html

[4] Cristina L. Abad, Edwin F. Boza, Gabriel Aumala, Gustavo Totoy, "Beyond Load Balancing Package-Aware Scheduling for Serveless Platforms" ICPE '18 Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, pp. 101-106, Abril 2018.

[5] Niu, N., Jin, M., & Cheng, J. R. C., "A case study of exploiting enterprise resource planning requirements", Enterprise Information Systems, vol. 5, p. 183-206, 2011.

[6] Siva Chegondi (22, Abril 2019). Caddy Extensible [Online]. Disponible en: https://github.com/mholt/caddy/wiki/Extending-Caddy

[7] Light Code Labs (2019, Marzo). Caddy User Guide (1.0.0) [Online]. Disponible en: https://caddyserver.com/docs

[8] Github(2019, Julio). Bombardier (1.2.4) [Online]. Disponible en: https://github.com/codesenberg/bombardier

[9] Github (2019, Julio). Httperf (1.0.0) [Online]. Disponible en: https://github.com/httperf/httperf

[10] Github (2019, Julio). Siege (4.0.4) [Online]. Disponible en: https://github.com/JoeDog/siege.git