Universidad Autónoma de Occidente

Facultad de Ingeniería

BALANCEADOR DE CARGA UTILIZANDO NGINX

Daniel Felipe Ayala, Código:2195574
Universidad Autónoma De Occidente
Ing. Electrónica y Telecomunicaciones

Johan Steven Arboleda:2185826
Universidad Autónoma de Occidente
Ing. Electrónica y telecomunicaciones

Edwin Fabian Ortega:2175732
Universidad Autónoma De Occidente
Ing. Electrónica y Telecomunicaciones

I. Introducción

El balanceo de carga es una técnica fundamental en entornos de servidores para distribuir eficientemente la carga de trabajo entre varios servidores. Nginx es un servidor web y proxy inverso muy utilizado que también puede actuar como un balanceador de carga altamente eficiente. En este informe, proporcionaremos una descripción detallada del proceso de configuración de Nginx como balanceador de carga en un sistema CentOS, destacando los beneficios, los pasos clave y las mejores prácticas.

II. ¿Qué es Nginx?

Nginx, pronunciado "engine-ex", es un servidor web de código abierto que, desde su éxito inicial como servidor web, ahora también se usa como proxy inverso, caché HTTP y equilibrador de carga.

Algunas empresas de alto perfil que utilizan Nginx incluyen Autodesk, Atlassian, Intuit, T-Mobile, GitLab, DuckDuckGo, Microsoft, IBM, Google, Adobe, Salesforce, VMWare, Xerox, LinkedIn, Cisco, Facebook, Target, Citrix Systems, Twitter, Apple , Intel y muchos más (*¿Qué Es Nginx Y Cómo Funciona?*, 2022).

Nginx fue desarrollado originalmente por Igor Sysoev y se lanzó públicamente por primera vez en octubre de 2004. Igor inicialmente concibió el software como una respuesta al problema C10K, que se refiere al problema de rendimiento del manejo de 10.000 conexiones simultáneas.

Con sus raíces en la optimización del rendimiento a pequeña escala, Nginx a menudo supera a otros servidores web populares en las pruebas de rendimiento , especialmente en situaciones con contenido estático y/o una gran cantidad de solicitudes simultáneas.

III. ¿Cómo funciona Nginx?

Nginx está diseñado para ofrecer bajo uso de memoria y alta concurrencia. En lugar de crear nuevos procesos para cada solicitud web, Nginx utiliza un enfoque asíncrono basado en eventos, procesando las solicitudes en un solo hilo.

Con Nginx, un proceso maestro puede controlar varios procesos de trabajo. El proceso maestro gestiona los procesos de trabajo y estos llevan a cabo el procesamiento real.

Funcionamiento del balanceador de carga con Nginx:

El balanceador de carga de Nginx utiliza un algoritmo de equilibrio de carga para distribuir las solicitudes entrantes entre los servidores backend disponibles. Algunos de los algoritmos de equilibrio de carga soportados por Nginx incluyen:

Round Robin (Por turnos): Las solicitudes se envían a los servidores en secuencia, uno tras otro, en el orden en que están definidos en la configuración.

IP Hash: Se utiliza una función hash para asignar las solicitudes a los servidores backend basándose en la dirección IP del cliente. Esto garantiza que las solicitudes del mismo cliente sean siempre dirigidas al mismo servidor backend, lo que puede ser útil para mantener la coherencia en aplicaciones específicas.

Least Connections (Menor cantidad de conexiones): Las solicitudes se envían al servidor backend con la menor cantidad de conexiones activas en ese momento. Esto permite distribuir la carga de manera proporcional en función de la capacidad de cada servidor.

IV. BENEFICIOS DEL BALANCEO DE CARGA CON NGINX

El uso de Nginx como balanceador de carga ofrece una serie de ventajas, que incluyen:

1. Alta disponibilidad: Distribuir la carga entre varios servidores backend garantiza que si uno de ellos falla, otros puedan continuar atendiendo las solicitudes de los clientes.
2. Escalabilidad horizontal: A medida que aumenta la demanda, se pueden agregar nuevos servidores backend para manejar la carga adicional, permitiendo una escalabilidad flexible.
3. Rendimiento mejorado: Nginx es conocido por su capacidad para manejar una gran cantidad de solicitudes simultáneas y mantener un rendimiento óptimo incluso bajo carga pesada.
4. Tolerancia a fallos: Si uno de los servidores backend falla o se vuelve lento, Nginx puede redirigir automáticamente las solicitudes a servidores en funcionamiento, evitando interrupciones en el servicio.
5. Optimización de recursos: Al distribuir la carga, se aprovecha mejor la capacidad de los servidores, lo que permite utilizar eficientemente los recursos disponibles.

V. PASOS PARA CONFIGURAR EL BALANCEO DE CARGA CON NGINX EN CENTOS:

Primero que todo creamos una carpeta y dentro de CMD ingresamos el siguiente comando dentro de la ruta que se desee.

Vagrant init

El anterior comando nos genera un archivo llamado Vagrantfile, dentro de este archivo creamos las máquinas que vayamos a utilizar, les mostraremos un ejemplo con todas las máquinas que en nuestro caso creamos para la realización de este proyecto:

```
Vagrant.configure("2") do |config|

 config.vm.define :cliente do |cliente|
    cliente.vm.box = "generic/centos8"
    cliente.vm.network :private_network, ip: "192.168.50.2"
    cliente.vm.hostname = "cliente"
  end


 config.vm.define :servidor do |servidor|
 servidor.vm.box = "centos/stream8"
 servidor.vm.network :private_network, ip: "192.168.50.3"
 servidor.vm.network :forwarded_port, guest: 80, host:5567
 servidor.vm.network :forwarded_port, guest: 443, host:5568
 servidor.vm.hostname = "servidor"
end



config.vm.define :servidor2 do |servidor2|
 servidor2.vm.box = "centos/stream8"
 servidor2.vm.network :private_network, ip: "192.168.50.4"
 servidor2.vm.hostname = "servidor2"
 end

config.vm.define :balanceador do |balanceador|
 balanceador.vm.box = "centos/stream8"
 balanceador.vm.network :private_network, ip: "192.168.50.5"
 balanceador.vm.hostname = "balanceador"
 end

config.vm.define :servidor3 do |servidor3|
 servidor3.vm.box = "centos/stream8"
 servidor3.vm.network :private_network, ip: "192.168.50.6"
 servidor3.vm.hostname = "servidor3"
 end

end
```

Para instalar Nginx en centOS, puede seguir estos pasos:
Abre la terminal en tu sistema CenOs.

Instalar Nginx utilizando el siguiente comando:

sudo yum install nginx -y

Una vez finalizada la instalación, puedes iniciar el servicio de Nginx utilizando el siguiente comando:

sudo systemctl start nginx

Para asegurarte de que Nginx se inicie automáticamente al arrancar el sistema, ejecuta el siguiente comando:

sudo systemctl enable nginx

Para verificar si Nginx se instaló correctamente, abre un navegador web e ingresa la dirección IP de tu servidor CentOS. Si ves la página de bienvenida de Nginx, significa que la instalación fue exitosa.
Ingresar a la ruta:

vim /etc/nginx/nginx.conf

Por defecto nuestro archivo se va a ver así:

```
server {
    listen       80 default_server;
    listen       [::]:80 default_server;
    server_name  _;
```

```
        root         /usr/share/nginx/html;
        # Load configuration files for the default server block.
        include /etc/nginx/default.d/*.conf;
        location / {
        }
        error_page 404 /404.html;
            location = /40x.html {
        }
        error_page 500 502 503 504 /50x.html;
            location = /50x.html {
        }
    }
```

Al realizar una modificación en nuestro archivo va a quedar:

```
user nginx;
worker_processes auto;
error_log /var/log/nginx/error.log;
pid /run/nginx.pid;
include /usr/share/nginx/modules/*.conf;
events {
    worker_connections 1024;
}
http {
    log_format  main  '$remote_addr - $remote_user [$time_local]
"$request" '
                      '$status $body_bytes_sent "$http_referer" '
                      '"$http_user_agent" "$http_x_forwarded_for"';
    access_log  /var/log/nginx/access.log  main;
    sendfile            on;
    tcp_nopush          on;
    tcp_nodelay         on;
    keepalive_timeout   65;
    types_hash_max_size 2048;
    include             /etc/nginx/mime.types;
    default_type        application/octet-stream;
    include /etc/nginx/conf.d/*.conf;
    server {
        root         /usr/share/nginx/html;
        include /etc/nginx/default.d/*.conf;
        location / {
        }
        error_page 404 /404.html;
            location = /40x.html {
        }
        error_page 500 502 503 504 /50x.html;
            location = /50x.html {
        }
    }
}
```

Ingresar a la ruta:

cd /etc/nginx/conf.d/

Crear un archivo llamado loadbalancer.conf:

touch loadbalancer.conf

Luego agregar en el archivo lo siguiente:

```
upstream backend {
        server 192.168.50.3;
        server 192.168.50.4;
        server 192.168.50.6;
    }
    server {
        listen      80 default_server;
        listen      [::]:80 default_server;
        server_name frontal.bitsandlinux.com;
        location / {
            proxy_redirect      off;
            proxy_set_header    X-Real-IP $remote_addr;
            proxy_set_header    X-Forwarded-For
$proxy_add_x_forwarded_for;
            proxy_set_header    Host $http_host;
        proxy_pass http://backend;
    }
}
```

Finalmente le damos los siguientes comandos

sudo systemctl restart nginx
setenforce 0

Para reiniciar el servicio guardamos todos los cambios realizados anteriormente y ya podremos ingresar la ip de nuestro balance de carga en un buscador como por ejemplo google chrome y al recargar la página realizaremos un redireccionamiento.

Ahora vamos a explicar como utilizar Artillery para saturar el balanceador y ver cómo reaccionan los servidores. Primero en el servidor creado llamado cliente, vamos a instalar vim para poder editar los archivos.

sudo yum install vim

Como en esta ocasión en el cliente estamos utilizando el centos8/generic, tendremos que apagar el firewall utilizando el siguiente comando:

service firewalld stop

Necesitaremos instalar el lenguaje de python dentro de nuestro cliente, entonces vamos a utilizar los siguientes comandos para la instalación:

curl -sL https://rpm.nodesource.com/setup_14.x | sudo bash -
sudo yum install -y nodejs

Ya tenemos todo lo que necesitamos para instalar el artillery, por lo tanto usamos lo siguiente:

sudo npm install -g artillery

Verificamos que haya sido instalado correctamente:

artillery --version

Luego creamos un archivo YAML dentro de nuestro cliente:

touch test.yaml

Y le agregamos a nuestro archivo las configuraciones deseadas para el funcionamiento de artillery con los respectivos servidores que se vayan a utilizar, el siguiente código es un ejemplo:

```
config:
  target: "http://192.168.50.5"
  phases:
    - duration: 60
      arrivalRate: 10
scenarios:
  - name: "Mi escenario de prueba"
    flow:
      - get:
          url: "/"
```

Finalmente utilizamos el siguiente comando para el funcionamiento de Artillery:

artillery run test.yaml

Artillery es una herramienta de código abierto utilizada para realizar pruebas de carga y pruebas de estrés en aplicaciones y servicios web. Está diseñado para simular el comportamiento de un gran número de usuarios concurrentes y medir el rendimiento y la capacidad de respuesta de una aplicación bajo condiciones de carga intensiva

```
config:
  target: "http://192.168.50.30"
  phases:
    - duration: 60
      arrivalRate: 100
scenarios:
  - name: "Mi escenario de prueba"
    flow:
      - get:
          url: "/"
```

En este caso para la primera prueba con el uso de artillery, lo configuramos de esta manera, en el target tenemos nuestro máquina de balanceo, en duration, lo tenemos configurado a que la prueba se ejecute en un periodo de tiempo de 60 segundos y en arrivalRate agregamos el número de solicitudes por segundo que queremos que el artillery realice a nuestros servidores es de 100 solicitudes por segundo.

```
upstream backend {

        server 192.168.50.10;
        server 192.168.50.20;
        server 192.168.50.40;
    }

    server {
        listen      80 default_server;
        listen      [::]:80 default_server;
        server_name frontal.bitsandlinux.com;

        location / {
            proxy_redirect      off;
            proxy_set_header    X-Real-IP $remote_addr;
            proxy_set_header    X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header    Host $http_host;
        proxy_pass http://backend;
    }
}
```

Dentro del archivo de configuración llamado en nuestro caso loadbalancer.conf, tenemos toda la configuración necesaria para que nuestro balanceador de carga con el uso de nginx balancee la carga de red entrante y la carga de trabajo, en esta ocasión estamos utilizando el método básico llamado round-robin que distribuye las solicitudes a los servidores de aplicaciones de forma redonda.

```
http.codes.200: ..... 6000
http.request_rate: ..... 100/sec
http.requests: ..... 6000
http.response_time:
  min: ..... 0
  max: ..... 56
  median: ..... 2
  p95: ..... 4
  p99: ..... 10.9
http.responses: ..... 6000
vusers.completed: ..... 6000
vusers.created: ..... 6000
vusers.created_by_name.Mi escenario de prueba: ..... 6000
vusers.failed: ..... 0
vusers.session_length:
  min: ..... 2.2
  max: ..... 85.1
  median: ..... 3.2
  p95: ..... 8.6
  p99: ..... 22.4
```

Estas métricas proporcionan información sobre el rendimiento y el comportamiento de la aplicación durante la prueba. A continuación, se explica el significado de cada una de las métricas:

- http.codes.200: Indica que se recibieron 6000 respuestas con el código de estado HTTP 200.
- http.request_rate: Indica que la tasa de solicitudes fue de 100 solicitudes por segundo.
- http.requests: Indica que se realizaron un total de 6000 solicitudes durante la prueba.
- http.response_time: Proporciona información sobre los tiempos de respuesta de las solicitudes.
- min: El tiempo de respuesta mínimo fue de 0 milisegundos.
- max: El tiempo de respuesta máximo fue de 56 milisegundos.
- median: La mediana de los tiempos de respuesta fue de 2 milisegundos.
- p95: El percentil 95 (95%) de los tiempos de respuesta fue de 4 milisegundos.
- p99: El percentil 99 (99%) de los tiempos de respuesta fue de 10.9 milisegundos.
- http.responses: Indica que se recibieron un total de 6000 respuestas durante la prueba.
- vusers.completed: Indica que se completaron exitosamente 6000 usuarios virtuales durante la prueba.
- vusers.created: Indica que se crearon un total de 6000 usuarios virtuales durante la prueba.
- vusers.created_by_name.Mi escenario de prueba: Indica que se crearon 6000 usuarios virtuales utilizando el escenario de prueba llamado "Mi escenario de prueba".

- ➢ vusers.failed: Indica que no hubo usuarios virtuales que fallaron durante la prueba.
- ➢ vusers.session_length: Proporciona información sobre la duración de las sesiones de los usuarios virtuales.
- ➢ min: La duración mínima de las sesiones fue de 2.2 segundos.
- ➢ Max: La duración máxima de las sesiones fue de 85.1 segundos.
- ➢ median: La mediana de la duración de las sesiones fue de 3.2 segundos.
- ➢ p95: El percentil 95 (95%) de la duración de las sesiones fue de 8.6 segundos.
- ➢ p99: El percentil 99 (99%) de la duración de las sesiones fue de 22.4 segundos.

```
upstream backend {
        least_conn;
        server 192.168.50.10;
        server 192.168.50.20;
        server 192.168.50.40;
    }

    server {
        listen      80 default_server;
        listen      [::]:80 default_server;
        server_name frontal.bitsandlinux.com;

        location / {
            proxy_redirect      off;
            proxy_set_header    X-Real-IP $remote_addr;
            proxy_set_header    X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header    Host $http_host;
        proxy_pass http://backend;
    }
}
```

En esta segunda prueba estamos utilizando el método de Selección del menos ocupado: asigna la siguiente solicitud a un servidor menos ocupado (el servidor con el menor número de conexiones activas.

```
--------------------------------
Summary report @ 06:19:34(+0000)
--------------------------------

http.codes.200: ..................................................... 6000
http.request_rate: .................................................. 100/sec
http.requests: ...................................................... 6000
http.response_time:
  min: .............................................................. 0
  max: .............................................................. 38
  median: ........................................................... 2
  p95: .............................................................. 3
  p99: .............................................................. 10.9
http.responses: ..................................................... 6000
vusers.completed: ................................................... 6000
vusers.created: ..................................................... 6000
vusers.created_by_name.Mi escenario de prueba: ...................... 6000
vusers.failed: ...................................................... 0
vusers.session_length:
  min: .............................................................. 2.1
  max: .............................................................. 57.9
  median: ........................................................... 3.2
  p95: .............................................................. 7.8
  p99: .............................................................. 19.5
```

Estas métricas proporcionan información sobre el rendimiento y el comportamiento de la aplicación durante la prueba. A continuación, se explica el significado de cada una de las métricas:

- ➢ http.codes.200: Indica que se recibieron 6000 respuestas con el código de estado HTTP 200 durante la prueba.
- ➢ http.request_rate: Indica que la tasa de solicitudes fue de 100 solicitudes por segundo.
- ➢ http.requests: Indica que se realizaron un total de 6000 solicitudes durante la prueba.

```
upstream backend {
        ip_hash;
        server 192.168.50.10;
        server 192.168.50.20;
        server 192.168.50.40;
    }

    server {
        listen      80 default_server;
        listen      [::]:80 default_server;
        server_name frontal.bitsandlinux.com;

        location / {
            proxy_redirect      off;
            proxy_set_header    X-Real-IP $remote_addr;
            proxy_set_header    X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header    Host $http_host;
        proxy_pass http://backend;
    }
}
```

En la tercera prueba usamos el método de IP Hash a nuestros tres servidores que utiliza la función hash para determinar que servidor se debe utilizar.

```
--------------------------------
Summary report @ 06:23:10(+0000)
--------------------------------

http.codes.200: ..................................................... 6000
http.request_rate: .................................................. 100/sec
http.requests: ...................................................... 6000
http.response_time:
  min: .............................................................. 0
  max: .............................................................. 47
  median: ........................................................... 2
  p95: .............................................................. 3
  p99: .............................................................. 10.1
http.responses: ..................................................... 6000
vusers.completed: ................................................... 6000
vusers.created: ..................................................... 6000
vusers.created_by_name.Mi escenario de prueba: ...................... 6000
vusers.failed: ...................................................... 0
vusers.session_length:
  min: .............................................................. 1.9
  max: .............................................................. 54.6
  median: ........................................................... 3.2
  p95: .............................................................. 6.9
  p99: .............................................................. 16.6
```

Estas métricas proporcionan información sobre el rendimiento y el comportamiento de la aplicación durante la prueba. A continuación, se explica el significado de cada una de las métricas:

- ➢ http.codes.200: Indica que se recibieron 6000 respuestas con el código de estado HTTP 200 (OK) durante la prueba.
- ➢ http.request_rate: Indica que la tasa de solicitudes fue de 100 solicitudes por segundo.
- ➢ http.requests: Indica que se realizaron un total de 6000 solicitudes durante la prueba.

```yaml
config:
  target: "http://192.168.50.30"
  phases:
    - duration: 60
      arrivalRate: 500
scenarios:
  - name: "Mi escenario de prueba"
    flow:
      - get:
          url: "/"
```

Ahora tenemos el caso#2 en el que le configuramos al artillery que le realice al balanceador de nginx 500 solicitudes por segundo en un periodo de 30 segundos.

```
upstream backend {

       server 192.168.50.10;
       server 192.168.50.20;
       server 192.168.50.40;
   }

   server {
       listen      80 default_server;
       listen      [::]:80 default_server;
       server_name frontal.bitsandlinux.com;

       location / {
           proxy_redirect      off;
           proxy_set_header    X-Real-IP $remote_addr;
           proxy_set_header    X-Forwarded-For $proxy_add_x_forwarded_for;
           proxy_set_header    Host $http_host;
       proxy_pass http://backend;
   }
}
```

y volvemos a utilizar el método de round robin para ver que resultados nos arroja el artillery para una cantidad de solicitudes mucho mayor.

```
--------------------------------
Summary report @ 06:27:30(+0000)
--------------------------------

errors.ECONNRESET: ..... 28
errors.ETIMEDOUT: ..... 12517
http.codes.200: ..... 15283
http.codes.500: ..... 2172
http.request_rate: ..... 386/sec
http.requests: ..... 30000
http.response_time:
  min: ..... 1
  max: ..... 6638
  median: ..... 507.8
  p95: ..... 2231
  p99: ..... 3828.5
http.responses: ..... 17455
vusers.completed: ..... 17455
vusers.created: ..... 30000
vusers.created_by_name.Mi escenario de prueba: ..... 30000
vusers.failed: ..... 12545
vusers.session_length:
  min: ..... 2.3
  max: ..... 9888.5
  median: ..... 3134.5
  p95: ..... 8186.6
  p99: ..... 9416.8
```

Estas métricas proporcionan información sobre el rendimiento y el comportamiento de la aplicación durante la prueba. A continuación, se explica el significado de cada una de las métricas:

- errors.ECONNRESET: Indica que se produjeron 28 errores de tipo ECONNRESET durante la prueba. Este error ocurre cuando se restablece la conexión de forma inesperada.
- errors.ETIMEDOUT: Indica que se produjeron 12517 errores de tipo ETIMEDOUT durante la prueba. Este error ocurre cuando se agota el tiempo de espera de una conexión.
- http.codes.200: Indica que se recibieron 15283 respuestas con el código de estado HTTP 200 (OK) durante la prueba.
- http.codes.500: Indica que se recibieron 2172 respuestas con el código de estado HTTP 500 (Error interno del servidor) durante la prueba.
- http.request_rate: Indica que la tasa de solicitudes fue de 386 solicitudes por segundo.
- http.requests: Indica que se realizaron un total de 30000 solicitudes durante la prueba.
- http.response_time: Proporciona información sobre los tiempos de respuesta de las solicitudes.
- min: El tiempo de respuesta mínimo fue de 1 milisegundo.
- max: El tiempo de respuesta máximo fue de 6638 milisegundos.
- median: La mediana de los tiempos de respuesta fue de 507.8 milisegundos.
- p95: El percentil 95 (95%) de los tiempos de respuesta fue de 2231 milisegundos.
- p99: El percentil 99 (99%) de los tiempos de respuesta fue de 3828.5 milisegundos.
- http.responses: Indica que se recibieron un total de 17455 respuestas durante la prueba.
- vusers.completed: Indica que se completaron exitosamente 17455 usuarios virtuales durante la prueba.
- vusers.created: Indica que se crearon un total de 30000 usuarios virtuales durante la prueba.
- vusers.created_by_name.Mi escenario de prueba: Indica que se crearon 30000 usuarios virtuales utilizando el escenario de prueba llamado "Mi escenario de prueba".
- vusers.failed: Indica que 12545 usuarios virtuales fallaron durante la prueba.

```
upstream backend {
      least_conn;
       server 192.168.50.10;
       server 192.168.50.20;
       server 192.168.50.40;
   }

   server {
       listen      80 default_server;
       listen      [::]:80 default_server;
       server_name frontal.bitsandlinux.com;

       location / {
           proxy_redirect      off;
           proxy_set_header    X-Real-IP $remote_addr;
           proxy_set_header    X-Forwarded-For $proxy_add_x_forwarded_for;
           proxy_set_header    Host $http_host;
       proxy_pass http://backend;
   }
}
```

Ahora usamos el método de least connections para ver cómo variarán los resultados de las pruebas.

```
--------------------------------
Summary report @ 06:30:41(+0000)
--------------------------------

errors.ETIMEDOUT: ..... 12010
http.codes.200: ..... 15500
http.codes.500: ..... 2490
http.request_rate: ..... 499/sec
http.requests: ..... 30000
http.response_time:
  min: ..... 1
  max: ..... 7121
  median: ..... 497.8
  p95: ..... 2018.7
  p99: ..... 3828.5
http.responses: ..... 17990
vusers.completed: ..... 17990
vusers.created: ..... 30000
vusers.created_by_name.Mi escenario de prueba: ..... 30000
vusers.failed: ..... 12010
vusers.session_length:
  min: ..... 2
  max: ..... 10003.5
  median: ..... 2725
  p95: ..... 8186.6
  p99: ..... 9230.4
```

Estas métricas proporcionan información sobre el rendimiento y el comportamiento de la aplicación durante la prueba. A continuación, se explica el significado de cada una de las métricas:

- errors.ETIMEDOUT: Indica que se produjeron 12010 errores de tipo ETIMEDOUT durante la prueba. Este error ocurre cuando se agota el tiempo de espera de una conexión.
- http.codes.200: Indica que se recibieron 15500 respuestas con el código de estado HTTP 200 (OK) durante la prueba.
- http.codes.500: Indica que se recibieron 2490 respuestas con el código de estado HTTP 500 (Error interno del servidor) durante la prueba.
- http.request_rate: Indica que la tasa de solicitudes fue de 499 solicitudes por segundo.
- http.requests: Indica que se realizaron un total de 30000 solicitudes durante la prueba..
- vusers.failed: Indica que 12010 usuarios virtuales fallaron durante la prueba.
- median: La mediana de la duración de las sesiones fue de 2725 segundos.

```
upstream backend {
       ip_hash;
        server 192.168.50.10;
        server 192.168.50.20;
        server 192.168.50.40;
    }

    server {
        listen      80 default_server;
        listen      [::]:80 default_server;
        server_name frontal.bitsandlinux.com;

        location / {
            proxy_redirect      off;
            proxy_set_header    X-Real-IP $remote_addr;
            proxy_set_header    X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header    Host $http_host;
        proxy_pass http://backend;
    }
}
```

Y Finalmente utilizamos el método de ip hash.

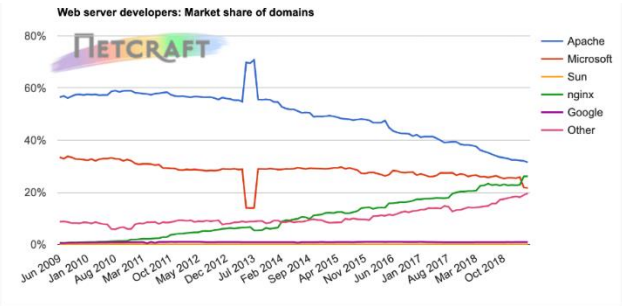
```
--------------------------------
Summary report @ 06:33:18(+0000)
--------------------------------

errors.ETIMEDOUT: ..... 13080
http.codes.200: ..... 15026
http.codes.500: ..... 1894
http.request_rate: ..... 378/sec
http.requests: ..... 30000
http.response_time:
  min: ..... 0
  max: ..... 7538
  median: ..... 507.8
  p95: ..... 2322.1
  p99: ..... 3905.8
http.responses: ..... 16920
vusers.completed: ..... 16920
vusers.created: ..... 30000
vusers.created_by_name.Mi escenario de prueba: ..... 30000
vusers.failed: ..... 13080
vusers.session_length:
  min: ..... 2.1
  max: ..... 9979.9
  median: ..... 3197.8
  p95: ..... 8186.6
  p99: ..... 9230.4
```

Después de todas estas pruebas podemos comprobar que entre más solicitudes se generen en la red a una serie de servidores en específico, se van a generar una mayor cantidad de errores, en nuestras pruebas, al realizar con el Artillery una cantidad de 100 solicitudes por segundo en un periodo de 60 segundos, el resumen de análisis de datos nos demostró que durante ese minuto de prueba no se generaron errores de espera de conexión, al cambiar la configuración de artillery a 500 solicitudes por segundo durante 1 minuto que en total serian 30.000 solicitudes en el minuto se generaron una media de 12.250 errores de espera de conexión, que variaron al método de balanceo utilizado en cada una de las tres pruebas.

VI. ALTERNATIVAS DE SOLUCION DIFERENTES AL BALANCEO DE CARGA CON NGINX:

Apache HTTP Server con mod_proxy_balancer: Apache HTTP Server es un servidor web popular que también puede funcionar como un balanceador de carga utilizando el módulo mod_proxy_balancer. Este módulo permite distribuir la carga entre varios servidores backend y ofrece una configuración flexible y opciones avanzadas de equilibrio de carga.

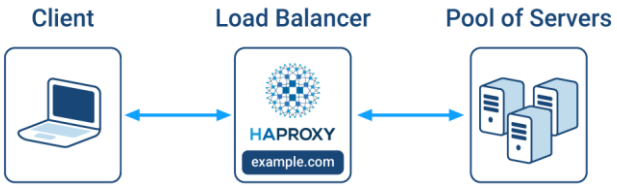


– Desarrolladores de servidores web: cuota de mercado de dominios (Fuente de la imagen Netcraft)

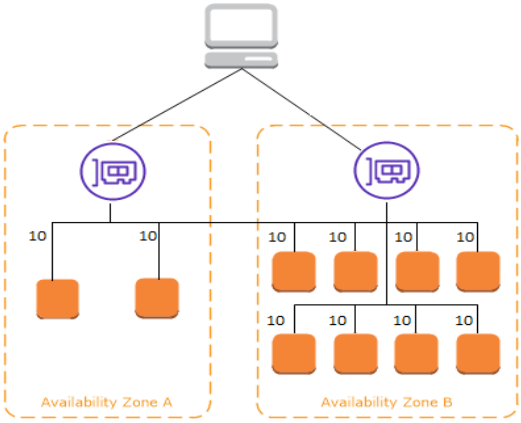
Apache es la opción más popular, Nginx es, en realidad, el servidor web más popular entre los sitios web con mucho tráfico.

Cuando se analizan las tasas de uso por tráfico, Nginx impulsa el: 67.1% de los 10,000 sitios más populares (en comparación con 63.2% en 2018)

HAProxy: es una solución de balanceo de carga de código abierto y alta disponibilidad. Proporciona un equilibrio de carga eficiente y admite múltiples algoritmos de balanceo, como round-robin, ponderación y el algoritmo de detección de comportamiento más adecuado. (Ramirez, 2022)



Amazon Elastic Load Balancer (ELB): Si estás utilizando servicios en la nube de Amazon Web Services (AWS), puedes aprovechar los servicios de balanceo de carga administrados que ofrece AWS, como Elastic Load Balancer (ELB). ELB proporciona equilibrio de carga automático y escalado horizontal para tus aplicaciones basadas en la nube.



Los nodos del balanceador de carga distribuyen las solicitudes provenientes de los clientes a los objetivos registrados. Cuando el balanceo de carga entre zonas está habilitado, cada balanceador de carga distribuye el tráfico a los destinos registrados en todas las zonas de disponibilidad habilitadas. Cuando el balanceo de carga entre zonas está deshabilitado, cada balanceador de carga distribuye el tráfico solo a los destinos registrados en su zona de disponibilidad. (Cómo Funciona Elastic Load Balancing - Elastic Load Balancing, 2022)

VII. Conclusiones

El uso de Nginx como balanceador de carga en CentOS es una solución efectiva para distribuir la carga de trabajo entre múltiples servidores backend. Mediante una configuración adecuada y el seguimiento de las mejores prácticas, se puede lograr una mayor disponibilidad, escalabilidad y rendimiento en entornos de servidores. El balanceo de carga con Nginx es una herramienta valiosa para garantizar un servicio confiable y eficiente en aplicaciones web y entornos de alto tráfico.

La efectividad del método de balanceo de carga en NGINX depende del escenario específico y de los requisitos de la aplicación. Cada método tiene sus ventajas y desventajas.

Referencias

[1] nginx news. (2023). Nginx.org. https://nginx.org/

[2] ¿Qué Es Nginx y Cómo Funciona? (2022, February 21). Kinsta®. https://kinsta.com/es/base-de-conocimiento/que-es-nginx/

[3] Fernández, L. (2020, March 29). Balanceadores de Carga: Así puedes mejorar el rendimiento de tu web. RedesZone; RedesZone. https://www.redeszone.net/tutoriales/servidores/balanceador-carga-load-balancer-que-es-funcionamiento/

[4] Rubén Aguilera Díaz-Heredero. (2018, February 22). Tests de rendimiento con Artillery - Adictos al trabajo. Adictos al Trabajo. https://www.adictosaltrabajo.com/2018/02/22/tests-de-rendimiento-con-artillery/#:~:text=Se%20trata%20de%20una%20herramienta,el%20percentil%2095%20y%20el

[5] davidochobits. (2020, February 24). Nginx: Balanceo de carga HTTP en Linux - ochobitshacenunbyte. Ochobitshacenunbyte. https://www.ochobitshacenunbyte.com/2020/02/24/nginx-balanceo-de-carga-http-en-linux/

[6] Ramirez, N. (2022, January 17). What Is Load Balancing - HAProxy Technologies. HAProxy Technologies. https://www.haproxy.com/blog/what-is-load-balancing/