UNIVERSIDAD LUTERANA SALVADOREÑA

FACULTAD DE CIENCIAS DEL HOMBRE Y LA NATURALEZA

LICENCIATURA EN CIENCIAS DE LA COMPUTACIÓN

CICLO: II-2024

COORDINADOR: Lcdo. Erick Mejía.

DOCENTE: Lic. Rafael Diaz

TEMA: Balanceador de Carga Nivel de código fuente.

INTEGRANTES:

Nombre	Carnet
Odalys Abigail Gomez Linares.	**GL23262**
Daniela Alejandra Reyes Garcia.	**RG23261**
Gresia Elizabeth Martinez Aguiluz.	**MA23263**
Gabriela Virginia Velasquez Turcios.	**VT2255**
Cristian Alexander Parada López.	**PL24273**
Diego Alejandro rajo Ramírez	**RR23403**

Noviembre 2024

Introducción.

Uno de los principales problemas de los mayores sitios web en Internet es cómo gestionar las solicitudes de un gran número de usuarios. Se trata de un problema de escalabilidad que surge con el continuo crecimiento del número de usuarios activos en el sistema.

El equilibrio de carga, balance de carga es un concepto usado en administración de sistemas informáticos que se refiere a la técnica usada para compartir el trabajo a realizar entre varios ordenadores, procesos, discos u otros recursos

La implementación de un balanceador de carga utilizando Node.js y Express, distribuye un balanceo de carga que redirija las solicitudes HTTP a varios servidores, cada uno de los cuales responderá con una página HTML diferente. permitiendo así una demostración de cómo se puede gestionar la carga de trabajo en aplicaciones web. A través de este sistema, se busca no solo mejorar la disponibilidad y la escalabilidad de las aplicaciones, sino también ofrecer una experiencia de usuario fluida y eficiente.

Este enfoque no solo demuestra la funcionalidad del balanceador de carga, sino que también ilustra cómo se pueden gestionar múltiples instancias de servidores para servir contenido dinámico o estático.

Objetivo general

Diseñar un sistema de balanceo de carga basado en código fuente, capaz de distribuir de manera eficiente el tráfico de red entre múltiples servidores, optimizando el uso de recursos, mejorando el rendimiento del sistema y garantizando la alta disponibilidad y estabilidad de los servicios en entornos distribuidos.

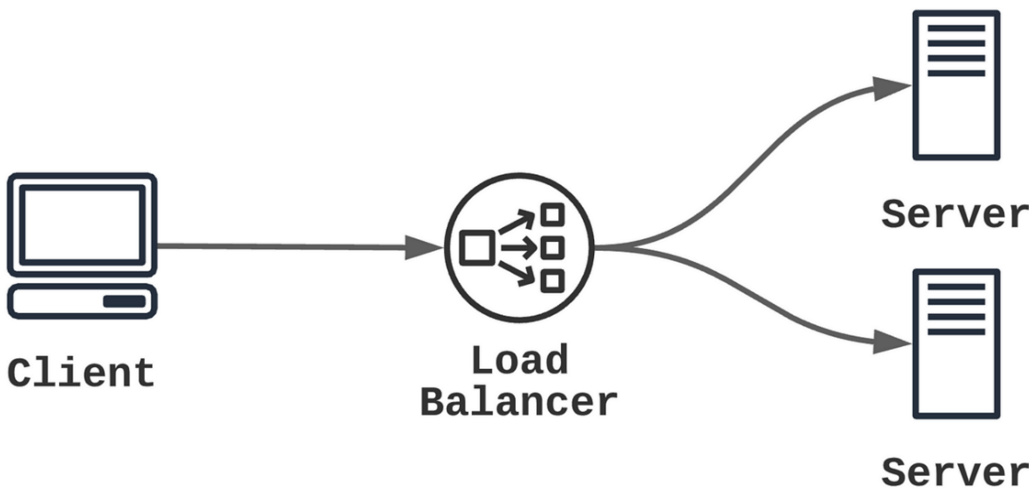
Objetivos específicos.

Implementar un sistema de servidores HTTP utilizando Node.js y Express, configurando múltiples servidores en diferentes puertos.

Diseñar una función de servicio de archivos (serveFile) que permita manejar dinámicamente las solicitudes de archivos HTML.

Balanceador de Carga Nivel de código fuente.

El balanceo de carga es una técnica utilizada en la administración de sistemas distribuidos y redes para distribuir el tráfico entrante de una aplicación o servicio entre múltiples servidores o recursos. Su principal objetivo es optimizar el uso de recursos, maximizar la capacidad de respuesta y garantizar la alta disponibilidad de los servicios. Según Cloudflare (2023), el balanceo de carga asegura que el tráfico de red se distribuya de manera eficiente para evitar la sobrecarga de un único servidor, asegurando así la estabilidad del sistema.



Con el crecimiento de las aplicaciones web y móviles, es fundamental mantener sistemas capaces de manejar grandes volúmenes de tráfico. Según un artículo de Nginx (2021), el balanceo de carga ayuda a garantizar la disponibilidad continua y la capacidad de respuesta del sistema, distribuyendo el tráfico de manera

equitativa entre varios servidores. Esto se puede implementar a través de configuraciones en servidores como Nginx, donde se definen grupos de servidores backend para gestionar las solicitudes entrantes.

En un entorno moderno, los usuarios esperan un servicio ininterrumpido, y el balanceo de carga se convierte en un componente esencial para cumplir con estas expectativas. No solo mejora la disponibilidad, sino que también ayuda a minimizar la latencia y a maximizar el rendimiento del sistema. Esto es crucial en servicios como comercio electrónico, plataformas de redes sociales y servicios de streaming, donde la eficiencia y la rapidez son primordiales.

La implementación adecuada del balanceo de carga a nivel de código fuente, como se ejemplifica en configuraciones en Node JS permite gestionar el tráfico de manera efectiva, asegurando que las solicitudes sean distribuidas entre los servidores disponibles, lo que contribuye a un mejor rendimiento y a una experiencia de usuario más satisfactoria.

Tecnologías a utilizadas.

Visual Studio Code: Visual Studio Code (VS Code) es un editor de código fuente desarrollado por Microsoft, diseñado para ser ligero, rápido y altamente personalizable. A diferencia de otros editores de código, como los IDE completos (Entornos de Desarrollo Integrados), VS Code se enfoca en ofrecer una experiencia ágil para la escritura y edición de código, sin sacrificar características avanzadas.

Node.js : Node.js es un entorno de ejecución de JavaScript del lado del servidor, construido sobre el motor V8 de Google Chrome. Permite a los desarrolladores ejecutar código JavaScript en el servidor, lo que facilita la creación de aplicaciones web escalables y de alto rendimiento.

Node.js viene con un gestor de paquetes robusto que permite instalar y gestionar dependencias fácilmente, facilitando la integración de bibliotecas y módulos externos

Express: Express es un framework minimalista para Node.js que simplifica el desarrollo de aplicaciones web y APIs. Proporciona una serie de herramientas y

características que hacen que la creación de servidores HTTP sea más sencilla y eficiente.

Middleware: Express permite el uso de middleware para gestionar solicitudes y respuestas, lo que facilita la implementación de funcionalidades como la autenticación, el manejo de errores y la manipulación de datos.

4. fs (File System): El módulo fs de Node.js permite interactuar con el sistema de archivos. Se utiliza para leer y escribir archivos, lo que es esencial para servir contenido estático como archivos HTML.

RELACIÓN DEL TEMA CON EL PROYECTO

En el contexto del desarrollo de aplicaciones móviles y web, el uso de un balanceador de carga es fundamental para garantizar que la aplicación pueda manejar un gran número de usuarios simultáneamente sin sacrificar el rendimiento o la disponibilidad. Este proyecto contempla el desarrollo de una aplicación escalable, que debe ser capaz de adaptarse a un tráfico de usuarios creciente. Por ello, la implementación de un sistema de balanceo de carga garantizará que las solicitudes se distribuyan de manera uniforme entre los diferentes servidores, reduciendo así los tiempos de respuesta y evitando fallos por sobrecarga. Además, al integrar tecnologías modernas como los servicios de balanceo en la nube, se permitirá que la infraestructura del proyecto crezca de manera flexible y que los recursos se ajusten dinámicamente a las demandas del sistema. De este modo, se asegurará una experiencia de usuario óptima y se optimizarán los recursos disponibles para el proyecto.

Implementa un sistema de balanceo de carga utilizando Node.js y Express.

Servidores HTTP.

const http = require("http");: Importa el módulo HTTP de Node.js, que proporciona la funcionalidad para crear servidores web.

const fs = require("fs");: Importa el módulo File System de Node.js, que permite leer y escribir archivos.

const path = require("path");: Importa el módulo Path de Node.js, que proporciona utilidades para trabajar con rutas de archivos.

```
1  const http = require("http");
2  const fs = require("fs");
3  const path = require("path");
```

Configuración de puertos:

const port1 = 3000;, const port2 = 3001;, const port3 = 3002;: Se definen tres constantes que representan los puertos en los que se ejecutarán los tres servidores HTTP.

```
5  const port1 = 3000;
6  const port2 = 3001;
7  const port3 = 3002;
```

Función serveFile:

Esta función se encarga de leer un archivo HTML y enviarlo como respuesta a una solicitud HTTP.

const serveFile = (filePath, res) => { ... }: Define una función flecha que toma dos parámetros: filePath (la ruta del archivo a servir) y res (el objeto de respuesta HTTP).

fs.readFile(filePath, (err, data) => { ... });: Utiliza el módulo File System para leer el contenido del archivo especificado por filePath.

if (err) { ... } else { ... }: Maneja los posibles errores de lectura del archivo. Si hay un error, envía una respuesta 404 Not Found. Si la lectura es exitosa, envía una respuesta 200 OK con el contenido del archivo.

```
Tabnine | Edit | Explain
const serveFile = (filePath, res) => {
    fs.readFile(filePath, (err, data) => {
        if (err) {
            res.writeHead(404, { 'Content-Type': 'text/plain' });
            res.end('404 Not Found');
        } else {
            res.writeHead(200, { 'Content-Type': 'text/html' });
            res.end(data);
        }
    });
};
```

Creación de servidores:

const server1 = http.createServer((req, res) => { ... });, const server2 = http.createServer((req, res) => { ... });, const server3 = http.createServer((req, res) => { ... });: Crea tres servidores HTTP independientes utilizando el módulo HTTP.

Cada servidor está configurado para servir un archivo HTML diferente utilizando la función serveFile.

```
const server1 = http.createServer((req, res) => {
    serveFile(path.join(__dirname, 'index1.html'), res);
});

const server2 = http.createServer((req, res) => {
    serveFile(path.join(__dirname, 'index2.html'), res);
});

const server3 = http.createServer((req, res) => {
    serveFile(path.join(__dirname, 'index3.html'), res);
});
```

Inicio de servidores:

server1.listen(port1, () => { ... });, server2.listen(port2, () => { ... });, server3.listen(port3, () => { ... });: Inicia la ejecución de cada uno de los tres servidores HTTP, escuchando en sus respectivos puertos.

```
Tabnine | Edit | Test | Explain | Document | Ask
server1.listen(port1, () => {
    console.log(`Servidor 1 arrancado en el puerto ${port1}`);
});

Tabnine | Edit | Test | Explain | Document | Ask
server2.listen(port2, () => {
    console.log(`Servidor 2 arrancado en el puerto ${port2}`);
});

Tabnine | Edit | Test | Explain | Document | Ask
server3.listen(port3, () => {
    console.log(`Servidor 3 arrancado en el puerto ${port3}`);
});
```

Cuando cada servidor se inicia correctamente, se imprime un mensaje de consola indicando el puerto en el que está escuchando.

```
PROBLEMAS   SALIDA   CONSOLA DE DEPURACIÓN   TERMINAL   PUERTOS   SEARCH ERROR

PS C:\Users\MINEDUCYT\Desktop\servidores1\servidorweb> node server.js
Servidor 1 arrancado en el puerto 3000
Servidor 2 arrancado en el puerto 3001
Servidor 3 arrancado en el puerto 3002
```

Balanceador de Carga

Importación de módulos:

Antes importar los módulos debes instalar los paquetes tanto de request como de express para que no muestren error.

```
INAL   PUERTOS   SEARCH ERROR   COMENTARIOS

PS C:\Users\MINEDUCYT\Desktop\servidores1\servidorweb
> npm install request
```

```
TERMINAL   PUERTOS   SEARCH ERROR   COMENTARIOS

PS C:\Users\MINEDUCYT\Desktop\servidores1\servidorweb
> npm npm install express
```

const express = require('express');: Importa el framework Express, que facilita la creación de aplicaciones web y APIs.

const request = require('request');: Importa el módulo Request, que se utiliza para hacer solicitudes HTTP.

```
1   const express = require('express');
2   const request = require('request');
3
4   const app = express();
```

Configuración de la aplicación Express:

const app = express();: Crea una instancia de la aplicación Express.

Definición de los servidores:

const servers = ['http://localhost:3000', 'http://localhost:3001', 'http://localhost:3002'];: Define un array con las URLs de los tres servidores HTTP.

let cur = 0;: Inicializa una variable cur que se utilizará para llevar un seguimiento del servidor actual al que se redirigen las solicitudes.

```
5
6   const servers = [
7       'http://localhost:3000', // Servidor 1
8       'http://localhost:3001', // Servidor 2
9       'http://localhost:3002'  // Servidor 3
10  ];
11
12  let cur = 0; // Comienza desde el primer servidor
```

Función handler:

const handler = (req, res) => { ... }: Define una función flecha que se encargará de redirigir las solicitudes a los servidores.

console.log(Redirigiendo a: ${servers[cur]});: Imprime un mensaje de consola indicando a qué servidor se está redirigiendo la solicitud.

req.pipe(request({ url: servers[cur] + req.url })).pipe(res);: Utiliza el módulo Request para enviar la solicitud al servidor correspondiente y devolver la respuesta al cliente.

cur = (cur + 1) % servers.length;: Actualiza la variable cur para apuntar al siguiente servidor en la lista de forma cíclica.

```
    Tabnine | Edit | Explain
14  const handler = (req, res) => {
15      console.log(`Redirigiendo a: ${servers[cur]}`);
16      req.pipe(request({ url: servers[cur] + req.url })).pipe(res);
17      cur = (cur + 1) % servers.length; // Cambia al siguiente servidor
18  };
```

Enrutamiento de solicitudes:

app.get('*', handler);: Configura una ruta de tipo GET que utilizará la función handler para redirigir las solicitudes.

app.post('*', handler);: Configura una ruta de tipo POST que también utilizará la función handler.

```
19
20    app.get('*', handler);
21    app.post('*', handler);
22
```

Inicio del balanceador de carga:

const server = app.listen(8080, () => { ... });: Inicia la ejecución del balanceador de carga, escuchando en el puerto 8080.

console.log('Balanceador de carga escuchando en el puerto 8080');: Imprime un mensaje de consola indicando que el balanceador de carga está en ejecución.

```
const server = app.listen(8080, () => {
    console.log('Balanceador de carga escuchando en el puerto 8080');
});
```

El sistema de balanceo de carga que distribuye las solicitudes HTTP (tanto GET como POST) entre tres servidores independientes, cada uno sirviendo un archivo HTML diferente. El balanceador de carga, construido con Express, recibe las solicitudes de los clientes y las redirige de manera cíclica a los tres servidores. Al momento de levantar los servidores ya se puede comprobar en el puerto 8080.

Diferencias entre balanceador de carga a nivel de código y balanceador de carga a nivel de red:

Un balanceador de carga a nivel de código fuente, como el ejemplo mostrado en el video de SantiLopezWeb, se implementa directamente en la aplicación o servidor web. Esto significa que la lógica de balanceo de carga se maneja dentro del código de la aplicación, sin necesidad de un componente de red dedicado.

Por otro lado, un balanceador de carga a nivel de red (balanceador proxy tradicional) es un dispositivo o servicio externo a la aplicación que se encarga de distribuir el tráfico entre los diferentes servidores o instancias de la aplicación. Este tipo de balanceador opera a nivel de red, interceptando y redirigiendo las solicitudes antes de que lleguen a la aplicación.

Las principales diferencias son:

Ubicación: El balanceador de carga a nivel de código está integrado en la aplicación, mientras que el balanceador de carga a nivel de red es un componente independiente.

Complejidad: El balanceador de carga a nivel de código suele ser más sencillo de implementar, ya que no requiere de un componente de red adicional. El balanceador de carga a nivel de red puede ser más complejo de configurar y mantener.

Escalabilidad: El balanceador de carga a nivel de red puede escalar mejor al manejar el tráfico a nivel de red, mientras que el balanceador de carga a nivel de código depende más de los recursos de la aplicación.

Transparencia: El balanceador de carga a nivel de red es más transparente para los clientes, ya que estos se conectan directamente al balanceador sin conocer la topología de la aplicación. El balanceador de carga a nivel de código puede ser más visible para los clientes.

Conclusión

El balanceo de carga es una herramienta esencial en la administración de sistemas distribuidos, ya que permite optimizar el uso de recursos, mejorar la capacidad de respuesta de los servicios y garantizar su alta disponibilidad. Al distribuir de manera eficiente el tráfico de red, se evita la sobrecarga de servidores individuales, asegurando la estabilidad y fiabilidad del sistema, especialmente en entornos con alta demanda. Su implementación es clave para mantener un rendimiento óptimo y una experiencia de usuario satisfactoria en aplicaciones y servicios modernos.

El balanceo de carga asegura que los usuarios sean dirigidos al servidor más cercano o menos saturado dentro de un conjunto de centros de datos, optimizando el tiempo de respuesta y reduciendo los costos de red.

Bibliografías.

- 🧢 D. E. C. (2023, July 17). *¿Qué es un balanceador de carga? - Diego Esteban C 🧢*. Medium. https://medium.com/@diego.coder/qu%C3%A9-es-un-balanceador-de-carga-6ca76a4b123e

- Acibeiro, M. (2022, February 7). Qué es Node.js y para qué sirve. *Blog de LucusHost*. https://www.lucushost.com/blog/que-es-node-js/

- *Express 5.X - API reference*. (n.d.). Expressjs.com. Retrieved November 23, 2024, from https://expressjs.com/en/5x/api.html

- Academind. (2020, May 6). Node.js Load Balancing with Node.js https://youtu.be/xMOkvoR3VCI?si=gcFx14RHaN1kdH-4

- SantiLopezWeb. (2021, May 6). Balanceador de carga qué es y para qué sirve | Ejemplo de balanceador de carga https://youtu.be/qPb4U5xJA70?si=jlyf0yq350fEF0Gw

- *Descripción general del balanceador de cargas de red del proxy externo*. (n.d.). Google Cloud. Retrieved November 23, 2024, from https://cloud.google.com/load-balancing/docs/tcp?hl=es-419