DESPLIEGUE DE APLICACIONES WEB

DESPLIEGUE DE UNA APLICACIÓN "CLUSTERIZADA" CON NODE EXPRESS

AUTOR: Alberto Martínez Pérez

CURSO: 2º CFGS Desarrollo de Aplicaciones Web (DAW)

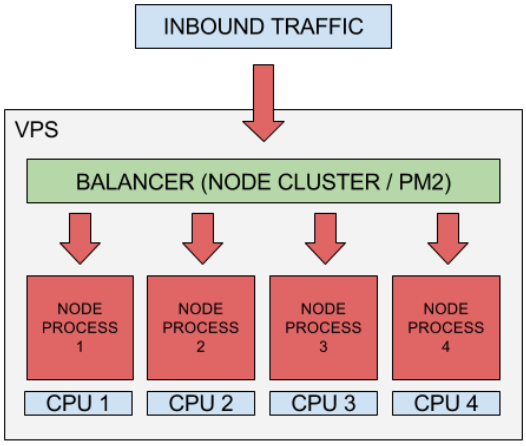
MÓDULO: Despliegue de Aplicaciones Web

ÍNDICE

INTRODUCCIÓN

Cuando se construye una aplicación de producción, lo que vamos a buscar es optimizar al máximo su rendimiento llegando a una solución de compromiso.

Node.JS se ejecuta en un único hilo del procesador, esto quiere decir que, en un procesador multinúcleo, sólo se utilizará uno de los núcleos del procesador. Para aprovechar el resto de los núcleos una solución que podemos aplicar es la utilización de un clúster de procesos Node.JS distribuyendo la carga por cada uno de ellos.



Al realizar esto conseguiremos que el rendimiento del servidor se mejore (en la forma de manejo de peticiones por unidad de tiempo) y conseguiremos que varios clientes sean atendidos al mismo tiempo.

En esta práctica veremos cómo crear un clúster de procesos Node.JS para mejorar el rendimiento de nuestro servidor y, además, veremos cómo gestionar el clúster utilizando un gestor de procesos (en este caso usaremos PM2).

A. ¿QUÉ ES EL MÓDULO CLÚSTER DE NODE.JS?

Este módulo nos va a permitir la creación de procesos secundarios (*workers*) que se ejecutan simultáneamente y comparten el mismo puerto de servidor. Hay que tener en cuenta que cada hijo tendrá su propio ciclo de eventos y su propia memoria asignada y que se comunicarán con el proceso principal (*master*) con una comunicación IPC.

Hay que pensar en el clúster como en un balanceador de carga de un proxy inverso, de forma que la carga de trabajo se repartirá entre los diferentes procesos hijos y, en el caso de que una operación bloquee o genere una ejecución prolongada de uno de estos procesos, el resto puedan encargarse de administrar las solicitudes entrantes. Es decir, la aplicación continuará funcionando y no cesará de ejecutarse durante la operación de bloqueo.

Además, el utilizar un clúster nos va a permitir realizar actualizaciones de la aplicación sin generar inactividad en la página, por ejemplo, podemos realizar cambios y reiniciar los *workers* uno a uno, esperando que un proceso secundario se genere por completo antes de reiniciar otro. De esta manera, siempre habrá procesos ejecutándose y respondiendo a operaciones mientras se produce la actualización.

Las conexiones entrantes se van a dividir entre los *workers* de dos maneras:

1. El *master* escucha las conexiones en un puerto y las distribuye entre los *workers* de forma rotatoria. Es el enfoque por defecto en todas las plataformas, excepto Windows.
2. El *master* crea un socket de escucha y lo envía a los *workers* interesados que luego podrán aceptar conexiones entrantes directamente.

B. REQUISITOS PARA LA PRÁCTICA

Vamos a trabajar con dos máquinas virtuales, en nuestro caso hemos decidido una máquina virtual con el sistema operativo Debian 12 y otra máquina virtual con el sistema operativo Ubuntu 22.04.

La primera de estas máquinas funcionará como servidor y debe tener instalados los paquetes Node.JS y NPM y la segunda máquina trabajará como cliente, pero lo que haremos será conectarnos a la máquina servidora a través de una conexión SSH.

Debido a esto ambas máquinas deben encontrarse en un entorno visible, en nuestro caso hemos decidido utilizar un tipo de red "Adaptador puente".

CREANDO LA APLICACIÓN SIN CLÚSTER

Lo primero que vamos a hacer es establecer la conexión SSH para simular un trabajo de forma remota en un servidor central, para ello debemos conectarnos de forma remota a la IP del servidor:

```
albertom-servidor@debian-deaw:~$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc nc
t qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host noprefixroute
       valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,DYNAMIC,UP,LOWER
te UP group default qlen 1000
    link/ether 08:00:27:2c:aa:0e brd ff:ff:ff:ff
    inet 192.168.1.78/24 brd 192.168.1.255 scope
       valid_lft 86281sec preferred_lft 86281sec
    inet6 fe80::a00:27ff:fe2c:aa0e/64 scope link
       valid_lft forever preferred_lft forever
```

Utilizando para ello el siguiente comando:

ssh usuario_servidor@ip_servidor

```
albertom-cliente@albertoMartinezPerez:~$ ssh albertom-servidor@192.168.1.78
The authenticity of host '192.168.1.78 (192.168.1.78)' can't be established.
ED25519 key fingerprint is SHA256:HBkCj5DHj/PySOvcT1ljlSZWx1pQprVoQclftgWwvUw.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '192.168.1.78' (ED25519) to the list of known hosts.
albertom-servidor@192.168.1.78's password:
Linux debian-deaw 6.1.0-13-amd64 #1 SMP PREEMPT_DYNAMIC Debian 6.1.55-1 (2023-09-29) x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
albertom-servidor@debian-deaw:~$
```

Una vez dentro del servidor vamos a crear un nuevo directorio para el proyecto, en nuestro caso lo llamaremos proyectoCluster:

mkdir /home/albertom-servidor/proyectoCluster

```
albertom-servidor@debian-deaw:~$ mkdir /home/albertom-servidor/proyectoCluster
albertom-servidor@debian-deaw:~$ ls -la
total 128
drwx------ 17 albertom-servidor albertom-servidor 4096 dic 15 16:33 .
drwxr-xr-x  3 root              root              4096 nov  1 11:15 ..
-rw-------  1 albertom-servidor albertom-servidor  494 dic 10 07:56 .bash_history
-rw-r--r--  1 albertom-servidor albertom-servidor  220 nov  1 11:15 .bash_logout
-rw-r--r--  1 albertom-servidor albertom-servidor 3526 nov  1 11:15 .bashrc
drwx------ 11 albertom-servidor albertom-servidor 4096 dic 10 07:53 .cache
drwxr-xr-x 12 albertom-servidor albertom-servidor 4096 nov  1 11:30 .config
drwxr-xr-x  2 albertom-servidor albertom-servidor 4096 nov  1 11:19 Descargas
drwxr-xr-x  2 albertom-servidor albertom-servidor 4096 nov  1 11:19 Documentos
drwxr-xr-x  2 albertom-servidor albertom-servidor 4096 nov  1 11:19 Escritorio
-rw-r--r--  1 albertom-servidor albertom-servidor 5290 nov  1 11:15 .face
lrwxrwxrwx  1 albertom-servidor albertom-servidor    5 nov  1 11:15 .face.icon -> .face
drwxr-xr-x  2 albertom-servidor albertom-servidor 4096 dic 10 07:53 .fontconfig
drwx------  2 albertom-servidor albertom-servidor 4096 nov  1 11:27 .gnupg
drwxr-xr-x  2 albertom-servidor albertom-servidor 4096 nov  1 11:19 Imágenes
drwx------  4 albertom-servidor albertom-servidor 4096 nov  1 11:19 .local
drwxr-xr-x  2 albertom-servidor albertom-servidor 4096 nov  1 11:19 Música
drwxr-xr-x  2 albertom-servidor albertom-servidor 4096 nov  1 11:19 Plantillas
-rw-r--r--  1 albertom-servidor albertom-servidor  807 nov  1 11:15 .profile
drwxr-xr-x  2 albertom-servidor albertom-servidor 4096 dic 15 16:33 proyectoCluster
drwxr-xr-x  2 albertom-servidor albertom-servidor 4096 nov  1 11:19 Público
```

Una vez hecho esto nos movemos dentro del directorio e inicializamos un proyecto haciendo uso de npm:

npm init

```
albertom-servidor@debian-deaw:~/proyectoCluster$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
```

Esto iniciará un formulario donde se nos pedirán una serie de parámetros:

```
package name: (proyectocluster)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
```

- **Package name**: Es el nombre que tendrá nuestro paquete o proyecto.
- **Version**: Presenta la versión actual. Por lo general se utiliza un formato X.Y.Z, de forma que incrementar X indica cambios mayores, Y indica nuevas funcionalidades y Z indica correcciones de errores o parches menores.
- **Description**: Una breve descripción del proyecto.
- **Entry point**: Especifica el archivo principal que será ejecutado cuando el paquete se importe.
- **Test command**: Este campo generalmente incluye los comandos para ejecutar pruebas automatizadas en el proyecto.
- **Git repository**: Aquí se puede especificar una URL de un repositorio donde se vaya a alojar nuestro proyecto, por ejemplo, una URL de GitHub.
- **Keywords**: Palabras clave que describen nuestro proyecto.
- **Author**: El autor o autores del proyecto.
- **License**: Indica bajo que licencia se distribuirá el proyecto. La licencia ISC es una licencia de software libre que permite a los usuarios hacer casi todo lo que deseen con el código, excepto reclamar que ellos lo crearon.

Podemos o bien elegir un nuevo valor o dejarlo por defecto (el que aparece entre paréntesis).

Una vez completado el formulario se nos creará un archivo package.json

```
albertom-servidor@debian-deaw:~/proyectoCluster$ ls -la
total 12
drwxr-xr-x  2 albertom-servidor albertom-servidor 4096 dic 15 16:40 .
drwx------ 18 albertom-servidor albertom-servidor 4096 dic 15 16:39 ..
-rw-r--r--  1 albertom-servidor albertom-servidor  211 dic 15 16:40 package.json
```

El cual tiene esta estructura:

```
{
  "name": "proyectocluster",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Una vez hecho esto vamos a instalar Express en el proyecto, para ello usaremos el comando:

npm install express

```
albertom-servidor@debian-deaw:~/proyectoCluster$ npm install express

added 62 packages, and audited 63 packages in 4s

11 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

Esto nos habrá creado el archivo package-lock.json y el directorio node_modules:

```
albertom-servidor@debian-deaw:~/proyectoCluster$ ls -la
total 40
drwxr-xr-x  3 albertom-servidor albertom-servidor  4096 dic 15 16:50 .
drwx------ 18 albertom-servidor albertom-servidor  4096 dic 15 16:39 ..
drwxr-xr-x 64 albertom-servidor albertom-servidor  4096 dic 15 16:50 node_modules
-rw-r--r--  1 albertom-servidor albertom-servidor   261 dic 15 16:50 package.json
-rw-r--r--  1 albertom-servidor albertom-servidor 24294 dic 15 16:50 package-lock.json
```

Ahora vamos a crear un archivo JavaScript donde programar nuestro servidor:

sudo nano ./no-cluster.js

```
const express = require("express");
const app = express();
const port = 3000;

app.get("/", (req, res) => {
 res.send("¡Hola Mundo!");
});

app.get("/api/:n", function (req, res) {
 let n = parseInt(req.params.n);
 let count = 0;

 if (n > 5000000000) n = 5000000000;

 for (let i = 0; i <= n; i++) {
 count += i;
 }

 res.send(`El resultado final es ${count}`);
});

app.listen(port, () => {
 console.log(`Aplicación escuchando por el puerto ${port}`);
});
```

Se trata de una aplicación sencilla para comprobar el funcionamiento de la carga de una web.

Contiene dos rutas (cada una de las funciones app.get()) siendo la primera una ruta raíz (/) que devuelve un "¡Hola Mundo!" como respuesta:

```
app.get("/", (req, res) => {
 res.send("¡Hola Mundo!");
});
```

Por su parte, la segunda ruta responde a la dirección /api/n donde n es un valor límite. Lo que se va a hacer en esta ruta es ejecutar una suma por iteración a través de un bucle for, de forma que el resultado final se acumula en una variable de nombre count que se mostrará como resultado final.

Para evitar que se genere una operación demasiado elevada para nuestro equipo, se limita el número de iteraciones a 5.000.000.000, de forma que cualquier parámetro superior a este valor, se fija automáticamente al valor límite.

```
app.get("/api/:n", function (req, res) {
 let n = parseInt(req.params.n);
 let count = 0;

 if (n > 5000000000) n = 5000000000;

 for (let i = 0; i <= n; i++) {
 count += i;
 }

 res.send(`El resultado final es ${count}`);
});
```

Por último tenemos una función app.listen() que nos sirve para establecer el puerto de escucha (el puerto 3000) y un mensaje que aparecerá en la consola del servidor:

```
app.listen(port, () => {
 console.log(`Aplicación escuchando por el puerto ${port}`);
});
```

Para lanzar el proceso utilizamos el comando:

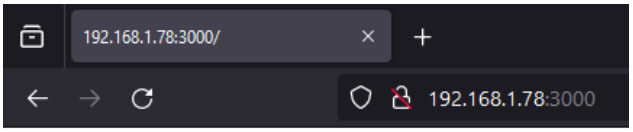
node ./no-cluster.js

```
albertom-servidor@debian-deaw:~/proyectoCluster$ node ./no-cluster.js
Aplicación escuchando por el puerto 3000
```

A. ACCESO A NUESTRA APLICACIÓN

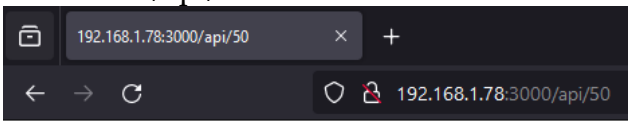
Para acceder a la aplicación debemos hacerlo a través de la IP de la máquina virtual servidora y el puerto 3000.

- Accediendo a la URL raíz:



¡Hola Mundo!

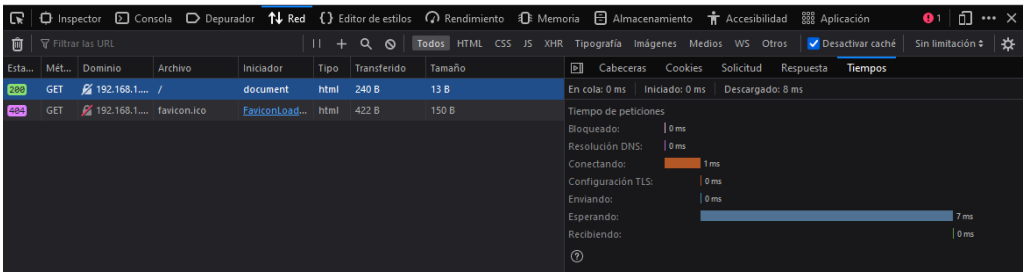
- Accediendo a la URL /api/n:



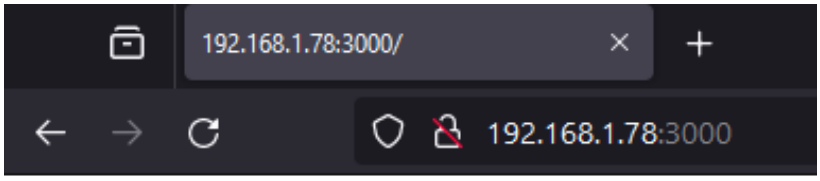
El resultado final es 1275

B. TIEMPOS DE RESPUESTA/ESPERA DEL SERVIDOR

Ahora vamos a comprobar el tiempo que tarda en realizar cada respuesta para ello debemos usar las herramientas de desarrollador del navegador y acceder al apartado "red". Aquí debemos hacer clic en el archivo del que queremos conocer sus datos y en el panel de la derecha elegir la opción "Tiempos".

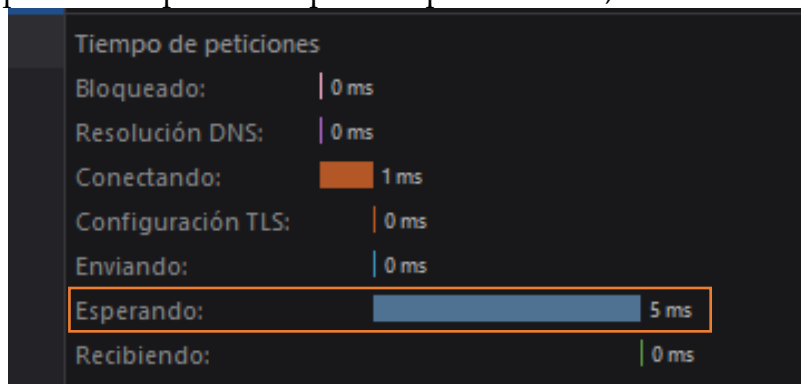


- Cuando accedemos a la URL raíz:

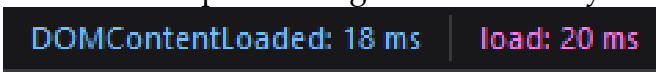


¡Hola Mundo!

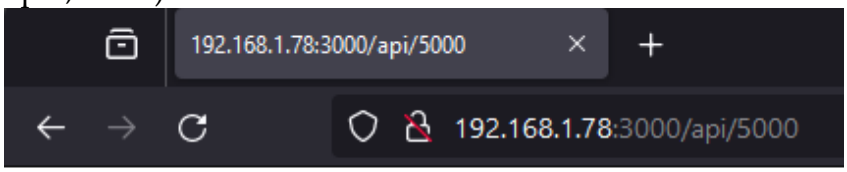
Comprobamos que el tiempo de espera es corto, de sólo 5ms:



Y que el tiempo de carga de la web también lo es, 18ms para cargar el DOM y 20ms en total para la carga de elementos y contenido:

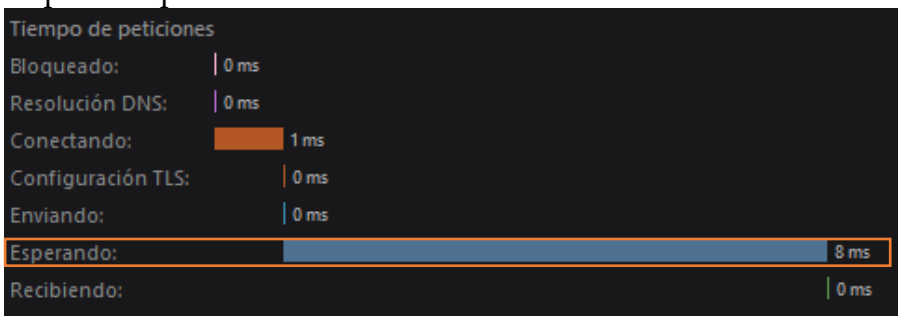


- Cuando accedemos a la URL /api/n con un valor bajo de n (por ejemplo, 5.000):

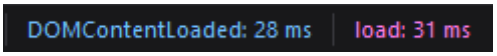


El resultado final es 12502500

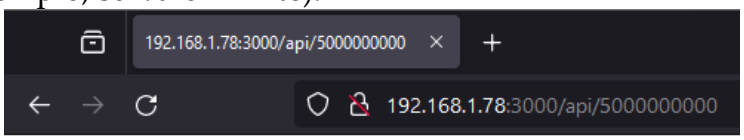
En este caso los valores siguen siendo bajos realmente con 8ms de tiempo de espera:



Y un tiempo de carga del DOM de 28ms y de carga del contenido de 31ms:



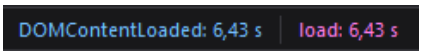
- Cuando accedemos a la URL /api/n con un número elevado de n (por ejemplo, su valor límite):



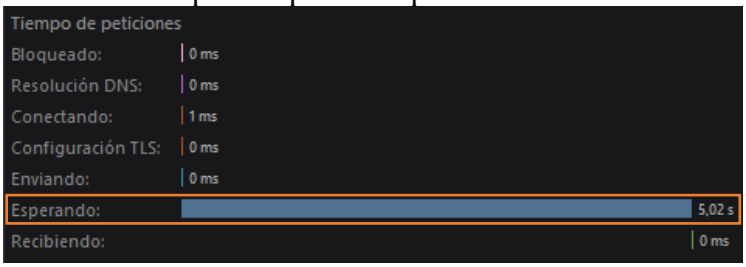
El resultado final es 12500000000066986000

En este caso el tiempo de espera es de 6.42s, muy superior a lo visto en los ejemplos anteriores:

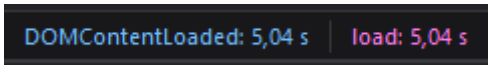
Lo mismo ocurre con la carga del DOM y del contenido, ambos tiempos de 6.43s:



- Cuando accedemos a la URL raíz, habiendo realizado primero una petición con número elevado de n iteraciones: En este caso vamos a ver como el tiempo de espera se eleva mucho (5.02s) en comparación al tiempo del primer apartado.



Lo mismo ocurre con los tiempos de carga del DOM y de su contenido, elevándose a 5.04s:



Esto se debe a que Node.JS trabaja con un único hilo de forma que hasta que no termina la primera operación (la solicitud con un número elevado de iteraciones), no se puede iniciar y ejecutar la siguiente operación, aunque sea mucho más rápida y requiera menos trabajo. Esta es una de las razones de que sea necesaria la creación de un clúster en nuestros servidores si vamos a utilizar Node.JS.

CREANDO NUESTRA APLICACIÓN HACIENDO USO DE CLÚSTERS

Ahora vamos a crear una aplicación similar, pero haciendo uso de un clúster, para ello paramos la ejecución del servidor (Ctrl + C) y creamos un nuevo fichero (o modificamos el anterior):

sudo nano ./cluster.js

```
const express = require("express");
const port = 3000;
const cluster = require("cluster");
const totalCPUs = require("os").cpus().length;

if (cluster.isMaster) {
 console.log(`El número de CPUs en total es: ${totalCPUs}`);
 console.log(`Master ${process.pid} está ejecutándose`);

 // Fork workers.
 for (let i = 0; i < totalCPUs; i++) {
 cluster.fork();
 }

 cluster.on("exit", (worker, code, signal) => {
 console.log(`worker ${worker.process.pid} murió`);
 console.log("¡Creemos otro worker!");
 cluster.fork();
 });
} else {
 const app = express();
 console.log(`Worker ${process.pid} iniciado`);

 app.get("/", (req, res) => {
 res.send("¡Hola Mundo!");
 });

 app.get("/api/:n", function (req, res) {
 let n = parseInt(req.params.n);
 let count = 0;

 if (n > 5000000000) n = 5000000000;

 for (let i = 0; i <= n; i++) {
 count += i;
 }

 res.send(`El resultado final es ${count}`);
 });

 app.listen(port, () => {
 console.log(`Aplicación escuchando por el puerto ${port}`);
 });
}
```

Es una aplicación muy similar a la anterior pero esta vez está preparada para generar varios procesos secundarios que van a compartir el puerto del servidor (el 3000), de forma que manejarán peticiones enviadas a este puerto.

Los procesos de trabajo se van a generar a través del método fork() del módulo cluster:

```
// Fork workers.
for (let i = 0; i < totalCPUs; i++) {
cluster.fork();
}
```

Este método devuelve un objeto de tipo ChildProcess que tiene un canal de comunicación incorporado para permitir transmisión entre el hijo (*worker*) y el padre (*master*).

De hecho con este método for, lo que vamos a hacer es crear tantos procesos secundarios como núcleos de CPU tenga la máquina (totalCPUs), valor que se consigue cargando el módulo OS y utilizando el método cpus() que devuelve un array de objetos que contienen información de cada núcleo de la CPU. De este array podemos extraer su propiedad length para conocer el número de núcleos:

```
const totalCPUs = require("os").cpus().length;
```

Utilizando esta forma de trabajo nos aseguraremos de no crear nunca más *workers* que núcleos de nuestra CPU, en el caso de que esto ocurriera podríamos sufrir problemas de sobrecarga.

Por tanto, como se puede deducir del código los *workers* son creados y administrados por el *master* y cuando la aplicación es ejecutada por primera vez se comprueba si el proceso es maestro (función isMaster())[1] y en el caso de que lo sea comienza la generación de *workers* con el bucle for.

Además, se registra la ID de proceso *master* y *worker* a través de impresiones por la consola.

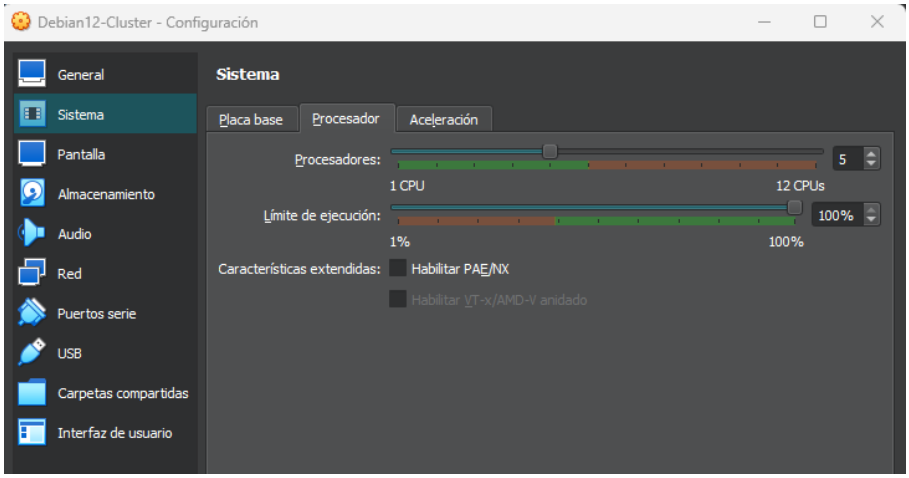
Por último, si un proceso secundario muere, se genera uno nuevo para seguir utilizando el número de núcleos de la CPU que se encuentran disponibles.

```
if (cluster.isMaster) {
 console.log(`El número de CPUs en total es: ${totalCPUs}`);
 console.log(`Master ${process.pid} está ejecutándose`);

 // Fork workers.
 for (let i = 0; i < totalCPUs; i++) {
 cluster.fork();
 }

 cluster.on("exit", (worker, code, signal) => {
 console.log(`worker ${worker.process.pid} murió`);
 console.log("¡Creemos otro worker!");
 cluster.fork();
 });
```

[1] Esto viene determinado por la variable process.env.NODE_UNIQUE_ID, si el valor de esta no está definido (*undefined*) entonces el método isMaster() devolverá un valor booleano true.

A. AUMENTAR EL NÚMERO DE PROCESADORES DE NUESTRO SERVIDOR

Para comprobar de mejor manera el funcionamiento del cluster vamos a aumentar el número de procesadores de nuestra máquina servidora, para ello en las opciones de configuración nos dirigimos a la sección "Sistema" y a la pestaña "Procesador" y ahí aumentamos el número de procesadores, en este caso elegimos 5 procesadores:



Una vez hecho esto volvemos a iniciar nuestra máquina virtual.

B. COMPROBAR EL TIEMPO DE ESPERA/RESPUESTA DEL SERVIDOR CON CLÚSTER ACTIVADO

Lanzamos nuestra aplicación cluster.js con el comando correspondiente de Node.js:

node ./cluster.js

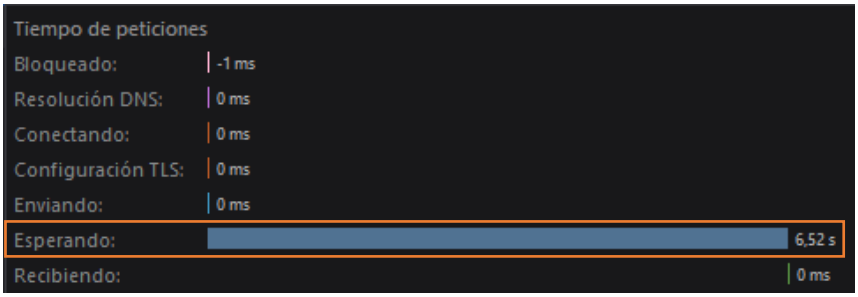
```
albertom-servidor@debian-deaw:~/proyectoCluster$ node ./cluster.js
```

Y podemos ver la primera diferencia con respecto a la prueba que hicimos sin clúster, ya que ahora en la terminal del servidor nos aparecen una serie de mensajes informativos sobre los procesos *master* y *workers* que se han creado:

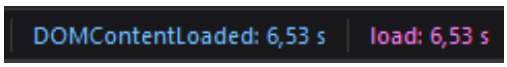
```
El número de CPUs en total es: 5
Master 3158 está ejecutándose
Worker 3168 iniciado
Aplicación escuchando por el puerto 3000
Worker 3169 iniciado
Aplicación escuchando por el puerto 3000
Worker 3166 iniciado
Aplicación escuchando por el puerto 3000
Worker 3167 iniciado
Aplicación escuchando por el puerto 3000
Worker 3165 iniciado
Aplicación escuchando por el puerto 3000
```

Ahora volvemos a nuestro navegador del equipo anfitrión y vamos a realizar la prueba de acceder a la URL /api/n con el valor límite de iteraciones y en otra pestaña accedemos a la URL raíz para comprobar cuánto tiempo tarda en realizarse toda la carga de ambas páginas.

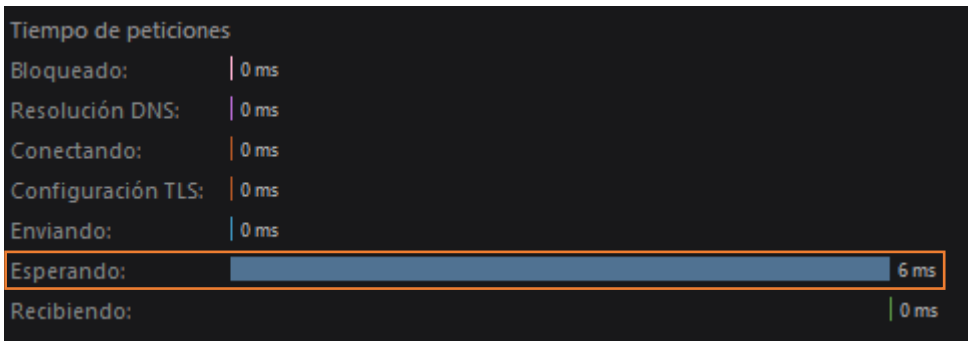
El acceso a la web /api/n con valor elevado sigue siendo lenta por el proceso de cálculo que se debe realizar, teniendo en este caso un tiempo de espera de 6.52s:



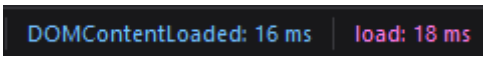
Y unos tiempos de carga del DOM y del contenido de 6.53s:



Pero sin embargo el acceso en la pestaña de la URL raíz ha sido mucho más rápido que en la prueba sin clúster con un tiempo de espera de sólo 6ms:



Y una carga de la web en 16ms para el DOM y 18ms para el contenido:



Esto es debido a que una solicitud ha sido atendida por un proceso secundario y otra por otro. Obviamente la tarea larga seguirá tardando en ejecutarse, pero esto no bloqueará el trabajo del servidor y las siguientes solicitudes que sean recibidas por este podrán ser atendidas por otros *workers* generando respuestas mucho más rápidas.

METRICAS DE RENDIMIENTO

Ejecutar solicitudes en diferentes pestañas del navegador es una forma sencilla y rápida de ver el funcionamiento del clúster, pero no es un método confiable, la mejor manera para ello es utiliza el paquete loadtest de NPM que nos permite simular una gran cantidad de conexiones simultáneas a nuestra aplicación para poder medir su rendimiento.

Para instalar este paquete de forma global utilizamos el siguiente comando:

sudo npm install -g loadtest

```
albertom-servidor@debian-deaw:~/proyectoCluster$ sudo npm install -g loadtest

added 30 packages in 6s

1 package is looking for funding
  run `npm fund` for details
```

Ahora vamos a probar las dos aplicaciones para ver los cambios.

A. PRUEBAS EN LA APLICACIÓN SIN CLÚSTER

Lo primero que debemos hacer es lanzar la aplicación:

node ./no-cluster.js

```
albertom-servidor@debian-deaw:~/proyectoCluster$ node ./no-cluster.js
Aplicación escuchando por el puerto 3000
```

Y ahora vamos a utilizar loadtest en otro terminal (debido a que el original estará bloqueado por el servidor), para ello usamos el siguiente comando:

loadtest URL -n número_solicitudes -c solicitudes_concurrentes

Por ejemplo:

loadtest http://localhost:3000/api/500000 -n 1000 -c 100

```
albertom-servidor@debian-deaw:~$ loadtest http://localhost:3000/api/500000 -n 1000 -c 100

Target URL:          http://localhost:3000/api/500000
Max requests:        1000
Concurrent clients:  300
Running on cores:    3
Agent:               none

Completed requests:  1000
Total errors:        0
Total time:          1.303 s
Mean latency:        324.8 ms
Effective rps:       767

Percentage of requests served within a certain time
  50%      351 ms
  90%      382 ms
  95%      384 ms
  99%      386 ms
 100%      387 ms (longest request)
```

Estos datos quieren decir que sobre la misma solicitud (a /api/500000) el servidor ha podido manejar 767 solicitudes por segundo con una latencia media de 324.8ms (tiempo que se tarda en completar una solicitud).

Ahora vamos a probar con un número mayor de iteraciones, por ejemplo, 5.000.000:

loadtest http://localhost:3000/api/5000000 -n 1000 -c 100

```
albertom-servidor@debian-deaw:~$ loadtest http://localhost:3000/api/5000000 -n 1000 -c 100
Requests: 300 (90%), requests per second: 60, mean latency: 1314.1 ms
Requests: 293 (88%), requests per second: 59, mean latency: 1521.9 ms
Requests: 231 (69%), requests per second: 46, mean latency: 1674.2 ms

Target URL:          http://localhost:3000/api/5000000
Max requests:        1000
Concurrent clients:  300
Running on cores:    3
Agent:               none

Completed requests:  1000
Total errors:        0
Total time:          6.055 s
Mean latency:        1500.3 ms
Effective rps:       165

Percentage of requests served within a certain time
  50%      1757 ms
  90%      1811 ms
  95%      1818 ms
  99%      1823 ms
 100%      1858 ms (longest request)
```

En este caso los resultados son aún peores pudiendo manejar sólo 165 solicitudes por segundo con una latencia media de 1500.3ms.

B. PRUEBAS EN LA APLICACIÓN CON CLÚSTER

Vamos a realizar las mismas pruebas en la aplicación con clúster:

```
albertom-servidor@debian-deaw:~/proyectoCluster$ node ./cluster.js
El número de CPUs en total es: 5
Master 3913 está ejecutándose
Worker 3924 iniciado
Worker 3921 iniciado
Worker 3920 iniciado
Worker 3923 iniciado
Aplicación escuchando por el puerto 3000
Aplicación escuchando por el puerto 3000
Aplicación escuchando por el puerto 3000
Aplicación escuchando por el puerto 3000
Worker 3922 iniciado
Aplicación escuchando por el puerto 3000
```

Primero realizamos la prueba con 500.000 iteraciones:

```
albertom-servidor@debian-deaw:~$ loadtest http://localhost:3000/api/500000 -n 1000 -c 100

Target URL:          http://localhost:3000/api/500000
Max requests:        1000
Concurrent clients:  300
Running on cores:    3
Agent:               none

Completed requests:  1000
Total errors:        0
Total time:          0.785 s
Mean latency:        198 ms
Effective rps:       1274

Percentage of requests served within a certain time
  50%      200 ms
  90%      236 ms
  95%      257 ms
  99%      303 ms
 100%      309 ms (longest request)
```

Como vemos con el clúster activado podemos manejar 1274 solicitudes efectivas por segundo con un tiempo de latencia medio de sólo 198ms, mejorando los datos de la misma prueba, pero sin clúster.

Si ahora realizamos la prueba con 5.000.000 de iteraciones obtenemos estos resultados:

```
albertom-servidor@debian-deaw:~$ loadtest http://localhost:3000/api/5000000 -n 1000 -c 100

Target URL:          http://localhost:3000/api/5000000
Max requests:        1000
Concurrent clients:  300
Running on cores:    3
Agent:               none

Completed requests:  1000
Total errors:        0
Total time:          1.821 s
Mean latency:        453.1 ms
Effective rps:       549

Percentage of requests served within a certain time
  50%      504 ms
  90%      555 ms
  95%      565 ms
  99%      580 ms
 100%      587 ms (longest request)
```

En este caso también se mejoran los resultados de la prueba sin clúster al poder atender a 549 solicitudes efectivas por segundo con un tiempo de latencia de 453.1ms.

USO DE PM2 PARA ADMINISTRAR UN CLÚSTER DE NODE.JS

Process Manager for Node

En esta práctica se ha utilizado el módulo "cluster" de Node.JS para crear y administrar los procesos a través de ciertas funcionalidades en nuestro código JS que hemos tenido que programar de forma manual, pero en la práctica real esto no es eficiente ya que habría que escribir aún más código y lo que se suele utilizar son administradores de procesos que integran un balanceador de carga, uno de estos administradores es PM2.

Cuando se ha configurado PM2 de forma correcta, ejecuta automáticamente la aplicación modo clúster, generando *workers* y se encarga de generar nuevos *workers* cuando uno de ellos muera.

Además de esto va a facilitar las tareas de inicio, parada y eliminación de procesos, así como de simplificar la monitorización ya que incluye una herramienta que ayuda en esta tarea y permite ajustar el rendimiento de su aplicación.

Lo primero que debemos hacer es instalar PM2 de forma global:

sudo npm install pm2 -g

```
albertom-servidor@debian-deaw:~$ sudo npm install pm2 -g
npm WARN deprecated uuid@3.4.0: Please upgrade  to versio
ath.random() in certain circumstances, which is known to
/math-random for details.

added 157 packages in 16s

13 packages are looking for funding
  run `npm fund` for details
```

Para lanzar PM2 se utiliza el siguiente comando:

pm2 start nombre_aplicacion -i 0

Siendo la opción -i el indicador para PM2 de que inicie la aplicación en cluster_mode (en lugar de fork_mode) y el valor 0 para que genere tantos *workers* como núcleos de CPU se encuentren disponibles.

A. UTILIZANDO PM2 CON NUESTRA APLICACIÓN SIN CLÚSTER

Estando dentro del directorio de nuestra aplicación, vamos a lanzar PM2 para que gestione nuestra aplicación sin clúster definido en código:

pm2 start ./no-cluster.js -i 0

```
albertom-servidor@debian-deaw:~/proyectoCluster$ pm2 start ./no-cluster.js -i 0

                          -------------

__/\\\\\\\\\\\\\\\_____/\\\\\\\\\\\\\\\_____/\\\\\\\\\\\\\\\_____
 _\/\\\///////////\\\_\/\\\\\\\\\\\\\\\_____/\\\///////////\\\___
  _\/\\\_______\/\\\_\/\\\/////////\\\_____/\\\_\/\\\_____\/\\\__
   _\/\\\\\\\\\\\\\/__\/\\\_______\/\\\_____/\\\_\/\\\_____\/\\\___
    _\/\\\/////////____\/\\\_______\/\\\_____/\\\_\/\\\_____\/\\\____
     _\/\\\_____________\/\\\_______\/\\\_____/\\\_\/\\\_____\/\\\_____
      _\/\\\_____________\/\\\_______\/\\\_____/\\\_\/\\\_____\/\\\______
       _\/\\\_____________\/\\\_____/\\\\\\\\\\\\\\\_\/\\\_____\/\\\_____
        _\/\\\_____________\/\\\_____/\\\\\\\\\\\\\\\_\/\\\_____\/\\\_____
         _\///______________\///_____/\\\\\\\\\\\\\\\_\///_____\/\\\_____
                                     _\///////////////__\///////////////__
```

Esto nos genera los siguientes *workers:*

```
[PM2] Spawning PM2 daemon with pm2_home=/home/albertom-servidor/.pm2
[PM2] PM2 Successfully daemonized
[PM2] Starting /home/albertom-servidor/proyectoCluster/no-cluster.js in cluster_mode (0 instance)
[PM2] Done.
```

id	name	mode	↺	status	cpu	memory
0	no-cluster	cluster	0	online	0%	69.8mb
1	no-cluster	cluster	0	online	0%	68.1mb
2	no-cluster	cluster	0	online	0%	71.0mb
3	no-cluster	cluster	0	online	0%	64.9mb
4	no-cluster	cluster	0	online	0%	47.6mb

Tantos como núcleos habíamos configurado en nuestro servidor.

Si ahora realizamos las mismas pruebas con loadtest que hicimos en el apartado anterior para comprobar el balanceo, obtendremos los siguientes resultados:

- En la prueba realizada sobre URL/api/500000 con 1000 solicitudes siendo 100 concurrentes, obtenemos estos resultados:

```
albertom-servidor@debian-deaw:~/proyectoCluster$ loadtest http://localhost:3000/api/500000 -n 1000 -c 100

Target URL:          http://localhost:3000/api/500000
Max requests:        1000
Concurrent clients:  300
Running on cores:    3
Agent:               none

Completed requests:  1000
Total errors:        0
Total time:          0.882 s
Mean latency:        206.2 ms
Effective rps:       1134

Percentage of requests served within a certain time
  50%      193 ms
  90%      274 ms
  95%      339 ms
  99%      402 ms
 100%      408 ms (longest request)
```

El número de respuestas efectivas por segundo es de 1134 con una latencia media de 206.2ms, resultados que son mejores que en la prueba realizada sobre la aplicación clusterizada.

- En la prueba realizada sobre URL/api/5000000 con 1000 solicitudes siendo 100 concurrentes, obtenemos estos resultados:

```
albertom-servidor@debian-deaw:~/proyectoCluster$ loadtest http://localhost:3000/api/5000000 -n 1000 -c 100

Target URL:          http://localhost:3000/api/5000000
Max requests:        1000
Concurrent clients:  300
Running on cores:    3
Agent:               none

Completed requests:  1000
Total errors:        0
Total time:          1.904 s
Mean latency:        487.1 ms
Effective rps:       525

Percentage of requests served within a certain time
  50%      532 ms
  90%      601 ms
  95%      607 ms
  99%      626 ms
 100%      635 ms (longest request)
```

En este caso obtenemos 525 respuestas efectivos por segundo con una latencia media 487.1ms. Estos resultados son un poco mejores que los obtenidos en la prueba sobre la aplicación con el clúster codificado en el fichero.

B. DETENER LA EJECUCIÓN DE UNA APLICACIÓN USANDO PM2

Para detener la aplicación se utiliza este comando:

pm2 stop nombre_aplicación

En este caso:

pm2 stop ./no-cluster.js

```
albertom-servidor@debian-deaw:~/proyectoCluster$ pm2 stop ./no-cluster.js
[PM2] Applying action stopProcessId on app [./no-cluster.js](ids: [ 0, 1, 2, 3, 4 ])
[PM2] [no-cluster](0) ✓
[PM2] [no-cluster](1) ✓
[PM2] [no-cluster](2) ✓
[PM2] [no-cluster](3) ✓
[PM2] [no-cluster](4) ✓
```

id	name	mode	↺	status	cpu	memory
0	no-cluster	cluster	0	stopped	0%	0b
1	no-cluster	cluster	0	stopped	0%	0b
2	no-cluster	cluster	0	stopped	0%	0b
3	no-cluster	cluster	0	stopped	0%	0b
4	no-cluster	cluster	0	stopped	0%	0b

C. CONFIGURANDO EL FICHERO ECOSYSTEM

De forma alternativa a cómo hemos lanzado la aplicación en el apartado anterior, podemos facilitarnos la tarea y generar el archivo de configuración propio de PM2 y que recibe el nombre de Ecosystem.

Dentro de este fichero podemos guardar los parámetros de nombre de la aplicación, instancias a crear, modo de ejecución, etc.

Para utilizarlo tenemos que ejecutar el siguiente comando dentro de nuestro directorio de trabajo:

pm2 ecosystem

```
albertom-servidor@debian-deaw:~/proyectoCluster$ pm2 ecosystem
File /home/albertom-servidor/proyectoCluster/ecosystem.config.js generated
```

Esto habrá generado un fichero ecosystem.config.js dentro de nuestro directorio:

```
albertom-servidor@debian-deaw:~/proyectoCluster$ ls -la
total 52
drwxr-xr-x  3 albertom-servidor albertom-servidor  4096 dic 15 20:34 .
drwx------ 19 albertom-servidor albertom-servidor  4096 dic 15 20:18 ..
-rw-r--r--  1 root              root               997 dic 15 18:29 cluster.js
-rw-r--r--  1 albertom-servidor albertom-servidor   489 dic 15 20:34 ecosystem.config.js
-rw-r--r--  1 root              root               460 dic 15 17:00 no-cluster.js
drwxr-xr-x 64 albertom-servidor albertom-servidor  4096 dic 15 16:50 node_modules
-rw-r--r--  1 albertom-servidor albertom-servidor   261 dic 15 16:50 package.json
-rw-r--r--  1 albertom-servidor albertom-servidor_24294 dic 15 16:50 package-lock.json
```

Este fichero tiene este aspecto de base:

```
module.exports = {
  apps : [{
    script: 'index.js',
    watch: '.'
  }, {
    script: './service-worker/',
    watch: ['./service-worker']
  }],

  deploy : {
    production : {
      user : 'SSH_USERNAME',
      host : 'SSH_HOSTMACHINE',
      ref  : 'origin/master',
      repo : 'GIT_REPOSITORY',
      path : 'DESTINATION_PATH',
      'pre-deploy-local': '',
      'post-deploy' : 'npm install && pm2 reload ecosystem.config.js --env production',
      'pre-setup': ''
    }
  }
};
```

Y lo debemos modificar de la forma siguiente para que lo podamos utilizar:

```
module.exports = {
  apps : [
  {
    name: "proyectoCluster",
    script: 'no-cluster.js',
    instances: 0,
    exec_mode: "cluster",
  },
  ],
};
```

Cuando configuramos exec_mode en cluster se le indica a PM2 que balancee la carga entre cada instancia y, al igual que hicimos para lanzar PM2 se coloca un 0 en instancies[2] para que genere tantos *workers* como núcleos de CPU.

[2] La opción -i o instances se pude establecer en los siguientes valores:

- 0 o max (actualmente en desuso): para generar un número de *workers* idéntico al número de CPUs.
- -1: para crear una cantidad de *workers* igual al número de CPUs - 1.
- Número exacto: para generar dicha cantidad de *workers*.

Ahora podremos lanzar la aplicación utilizando el siguiente comando:

pm2 start ecosystem.config.js

```
albertom-servidor@debian-deaw:~/proyectoCluster$ pm2 start ecosystem.config.js
[PM2][WARN] Applications proyectoCluster not running, starting...
[PM2] App [proyectoCluster] launched (5 instances)
```

id	name	mode	↺	status	cpu	memory
0	proyectoCluster	cluster	0	online	0%	69.0mb
1	proyectoCluster	cluster	0	online	0%	68.0mb
2	proyectoCluster	cluster	0	online	0%	68.9mb
3	proyectoCluster	cluster	0	online	0%	60.2mb
4	proyectoCluster	cluster	0	online	0%	48.2mb

De esta manera la aplicación se lanzará en modo clúster.

D. OTROS COMANDOS DE PM2

Además de los comandos start y stop. Tenemos los siguientes comandos:

- **pm2 restart [nombre_aplicación | ecosystem.config.js]**: Este comando reinicia una aplicación que ya está siendo administrada por PM2. Detiene la aplicación y la vuelve a iniciar inmediatamente. Es útil cuando se necesitan aplicar cambios o actualizaciones en la aplicación sin detener el servicio por completo.

pm2 restart ecosystem.config.js

```
albertom-servidor@debian-deaw:~/proyectoCluster$ pm2 restart ecosystem.config.js
[PM2] Applying action restartProcessId on app [proyectoCluster](ids: [ 0, 1, 2, 3, 4 ])
[PM2] [proyectoCluster](1) ✓
[PM2] [proyectoCluster](0) ✓
[PM2] [proyectoCluster](2) ✓
[PM2] [proyectoCluster](3) ✓
[PM2] [proyectoCluster](4) ✓
```

id	name	mode	↺	status	cpu	memory
0	proyectoCluster	cluster	1	online	0%	69.0mb
1	proyectoCluster	cluster	1	online	0%	68.5mb
2	proyectoCluster	cluster	1	online	0%	70.8mb
3	proyectoCluster	cluster	1	online	0%	68.6mb
4	proyectoCluster	cluster	1	online	0%	47.9mb

- **pm2 reload [nombre_aplicación | ecosystem.config.js]**: La instrucción reload reinicia una aplicación de manera similar a restart, pero con algunas diferencias en la forma en que se maneja el proceso. Por lo general, reload trata de reiniciar la aplicación sin interrumpir las conexiones activas, si es posible.

pm2 reload ecosystem.config.js

```
albertom-servidor@debian-deaw:~/proyectoCluster$ pm2 reload ecosystem.config.js
[PM2] Applying action reloadProcessId on app [proyectoCluster](ids: [ 0, 1, 2, 3, 4 ])
[PM2] [proyectoCluster](1) ✓
[PM2] [proyectoCluster](0) ✓
[PM2] [proyectoCluster](2) ✓
[PM2] [proyectoCluster](3) ✓
[PM2] [proyectoCluster](4) ✓
```

- **pm2 delete [nombre_aplicación | ecosystem.config.js]**: Elimina una aplicación de la lista de aplicaciones administradas por PM2. Esto detiene el proceso de la aplicación y elimina la entrada correspondiente de la configuración de PM2. Una vez eliminada,

no se podrá reiniciar la aplicación utilizando el nombre que se ha borrado a menos que se vuelva a agregar.

pm2 delete ecosystem.config.js

```
albertom-servidor@debian-deaw:~/proyectoCluster$ pm2 delete ecosystem.config.js
[PM2] [proyectoCluster](0) ✓
[PM2] [proyectoCluster](1) ✓
[PM2] [proyectoCluster](2) ✓
[PM2] [proyectoCluster](3) ✓
[PM2] [proyectoCluster](4) ✓
```

id	name	mode	↺	status	cpu	memory

- **pm2 ls**: Lista de todas las aplicaciones que están siendo administradas por PM2.

pm2 ls

```
albertom-servidor@debian-deaw:~/proyectoCluster$ pm2 ls
```

id	name	mode	↺	status	cpu	memory
0	proyectoCluster	cluster	0	online	0%	68.4mb
1	proyectoCluster	cluster	0	online	0%	67.7mb
2	proyectoCluster	cluster	0	online	0%	68.1mb
3	proyectoCluster	cluster	0	online	0%	67.5mb
4	proyectoCluster	cluster	0	online	0%	68.2mb

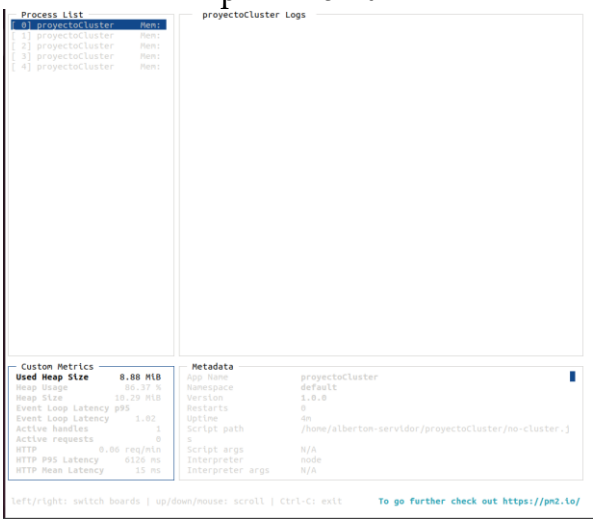
- **pm2 log**: Este comando muestra los registros (logs) en tiempo real de todas las aplicaciones administradas por PM2.

pm2 log

```
albertom-servidor@debian-deaw:~/proyectoCluster$ pm2 log
[TAILING] Tailing last 15 lines for [all] processes (change the value with --lines option)
/home/albertom-servidor/.pm2/pm2.log last 15 lines:
PM2        | 2023-12-15T20:52:52: PM2 log: Stopping app:proyectoCluster id:4
PM2        | 2023-12-15T20:52:52: PM2 log: pid=4950 msg=process killed
PM2        | 2023-12-15T20:52:52: PM2 log: App name:proyectoCluster id:4 disconnected
PM2        | 2023-12-15T20:52:52: PM2 log: App [proyectoCluster:4] exited with code [0] via sign
PM2        | 2023-12-15T20:52:52: PM2 log: pid=4968 msg=process killed
PM2        | 2023-12-15T20:53:42: PM2 log: App [proyectoCluster:0] starting in -cluster mode-
PM2        | 2023-12-15T20:53:42: PM2 log: App [proyectoCluster:0] online
PM2        | 2023-12-15T20:53:42: PM2 log: App [proyectoCluster:1] starting in -cluster mode-
PM2        | 2023-12-15T20:53:42: PM2 log: App [proyectoCluster:1] online
PM2        | 2023-12-15T20:53:42: PM2 log: App [proyectoCluster:2] starting in -cluster mode-
PM2        | 2023-12-15T20:53:42: PM2 log: App [proyectoCluster:2] online
PM2        | 2023-12-15T20:53:42: PM2 log: App [proyectoCluster:3] starting in -cluster mode-
PM2        | 2023-12-15T20:53:42: PM2 log: App [proyectoCluster:3] online
PM2        | 2023-12-15T20:53:42: PM2 log: App [proyectoCluster:4] starting in -cluster mode-
PM2        | 2023-12-15T20:53:42: PM2 log: App [proyectoCluster:4] online
```

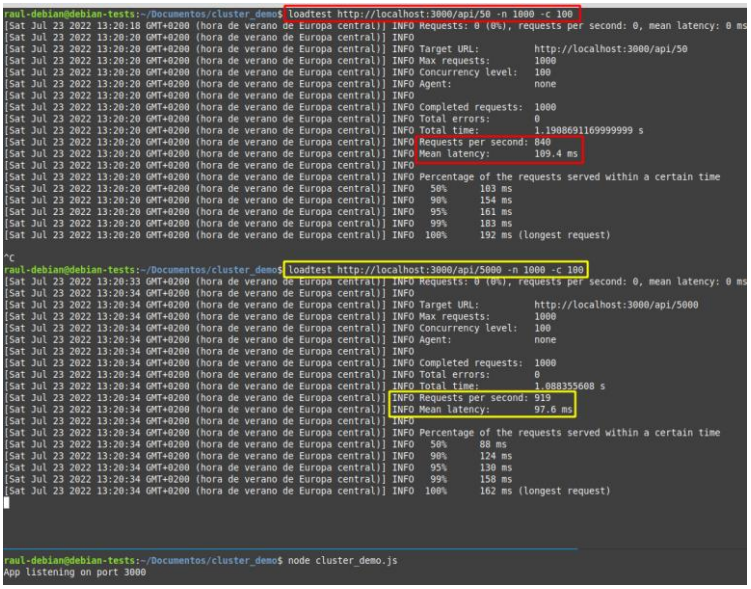
- **pm2 monit**: Inicia una interfaz de monitorización interactiva que muestra información en tiempo real sobre el uso de recursos (CPU, memoria, etc.) de las aplicaciones administradas por PM2. Proporciona una vista detallada y dinámica que facilita la supervisión de las aplicaciones en ejecución, lo que permite identificar posibles problemas de rendimiento o cuellos de botella.

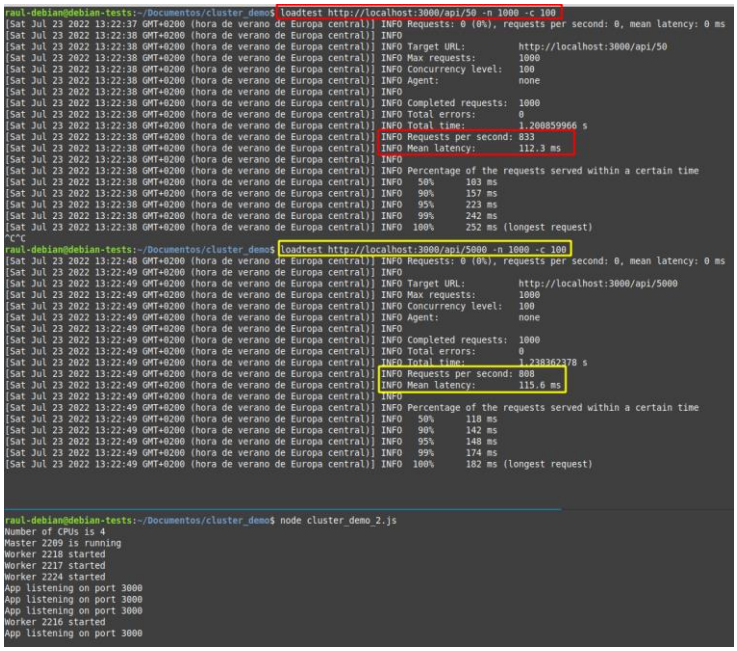
pm2 monit



CUESTIONES FINALES

Fijaos en las siguientes imágenes:





La primera imagen ilustra los resultados de unas pruebas de carga sobre la aplicación sin clúster y la segunda sobre la aplicación clusterizada.

¿Sabrías decir por qué en algunos casos concretos, como este, la aplicación sin clusterizar tiene mejores resultados?

Se obtienen mejores resultados debido a que la carga de trabajo en ambos casos es pequeña (50 iteraciones y 5000 iteraciones) y en este escenario el coste adicional de comunicación entre *workers* y entre *worker* y *master* puede tener un impacto negativo en comparación a una situación donde haya una instancia de la aplicación sin clúster.