

Despliegue de una aplicación en *cluster* con Node Express

Tabla de Contenido

1. Introducción	1
1.1. Un vistazo rápido a los clusters	2
2. Usando los clusters	3
2.1. Primero sin clúster	3
2.2. ¡Ahora con más clúster!	5
3. Métricas de rendimiento	8
4. Uso de PM2 para administrar un clúster de Node.js	11
5. Cuestiones	15
6. Referencias	17
7. Evaluación	17

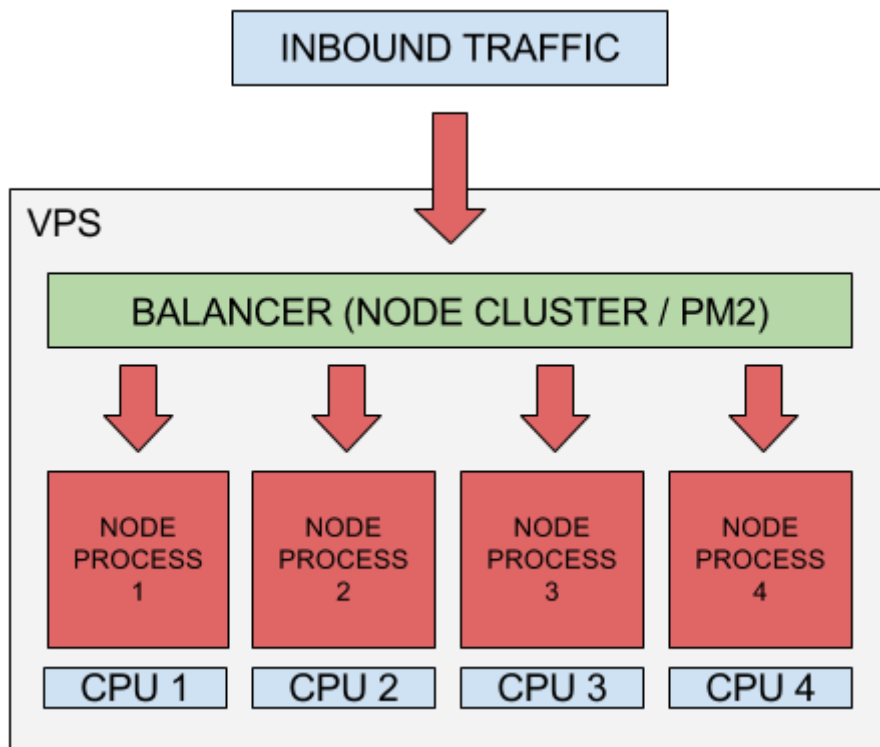


Esta práctica está realizada en [ubuntu/lunar64](#)

1. Introducción

Cuando se construye una aplicación de producción, normalmente se busca la forma de optimizar su rendimiento llegando a una solución de compromiso. En esta práctica echaremos un vistazo a un enfoque que puede ofrecer una victoria rápida cuando se trata de mejorar la manera en que las aplicaciones *Node.js* manejan la carga de trabajo.

Una instancia de *Node.js* se ejecuta en un solo hilo, lo que significa que en un sistema multinúcleo (como la mayoría de los ordenadores de hoy en día), no todos los núcleos serán utilizados por la aplicación. Para aprovechar los otros núcleos disponibles, podemos lanzar un cluster de procesos *Node.js* y distribuir la carga entre ellos.



Al tener varios hilos para manejar las peticiones, mejora el rendimiento (peticiones por segundo) del servidor, ya que varios clientes pueden ser atendidos simultáneamente. Veremos cómo crear procesos hijos con el módulo de cluster de *Node.js* para, más tarde, ver cómo gestionar el cluster con el gestor de procesos PM2.

1.1. Un vistazo rápido a los clusters

El [módulo de clúster de Node.js](#) permite la creación de procesos secundarios (*workers*) que se ejecutan simultáneamente y comparten el mismo puerto de servidor. Cada hijo generado tiene su propio ciclo de eventos y memoria. Los procesos secundarios utilizan IPC (Inter-Process Communication) o comunicación entre procesos, para comunicarse con el proceso principal de *Node.js*.

Tener múltiples procesos para manejar las solicitudes entrantes significa que se pueden procesar varias solicitudes simultáneamente y si hay una operación de bloqueo/ejecución prolongada en un *worker*, los otros *workers* pueden continuar administrando otras solicitudes entrantes; la aplicación no se detendrá hasta que finalice la operación de bloqueo.

La ejecución de varios *workers* también permite actualizar la aplicación en producción con poco o ningún tiempo de inactividad. Se pueden realizar cambios en la aplicación y reiniciar los *workers* uno por uno, esperando que un proceso secundario se genere por completo antes de reiniciar otro. De esta manera, siempre habrá *workers* ejecutándose mientras se produce la actualización.

Las conexiones entrantes se distribuyen entre los procesos secundarios de dos maneras:

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

2. Usando los clusters

2.1. Primero sin clúster

Para ver las ventajas que ofrece la agrupación en clústeres, comenzaremos con una aplicación de prueba en *Node.js* que no usa clústeres y la compararemos con una que sí los usa, se trata de la siguiente:

```
const express = require("express");
const app = express();
const port = 3000;
const limit = 5000000000;

app.get("/", (req, res) => {
  res.send("Hello World!");
});

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

  if (n > limit) n = limit;

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

  res.send(`Final count is ${count}`);
});

app.listen(port, () => {
  console.log(`App listening on port ${port}`);
});
```

Se trata de una aplicación un tanto *prefabricada* en el sentido de que es algo que jamás encontraríamos en el mundo real. No obstante, nos servirá para ilustrar nuestro propósito.

Esta aplicación contiene dos rutas, una ruta raíz `/` que devuelve la cadena `Hello World!` y otra ruta `/api/n` donde se toma `n` como parámetro y va realizando una operación de suma (el bucle `for`) cuyo resultado acumula en la variable `count` que se muestra al final.

Si a este parámetro `n`, le damos un valor muy alto, nos permitirá simular operaciones intensivas y de ejecución prolongada en el servidor. Le damos como valor límite `5000000000` para evitar una operación demasiado costosa para nuestro ordenador.

Ahora seguiremos los siguientes pasos:

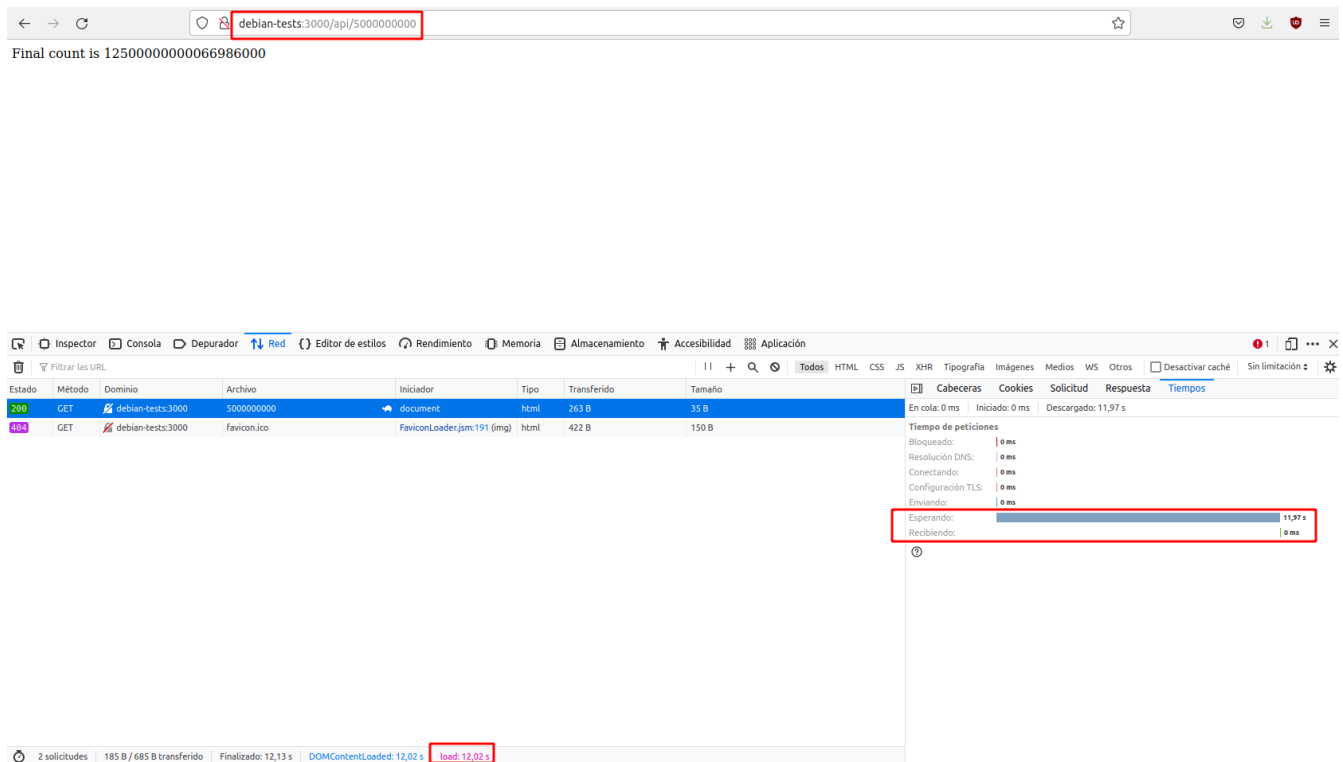
1. Debéis conectaros al servidor Debian mediante SSH.

2. Debéis crear un directorio para el proyecto de esta aplicación
3. DENTRO del directorio ejecutaréis 2 comandos:
 - `npm init` para crear automáticamente la estructura de carpetas y el archivo `package.json` (Con ir dándole a <ENTER> a todas las preguntas, os basta)
 - `npm install express` para instalar `express` para este proyecto
4. Tras esto, DENTRO del directorio, ya podéis iniciar la aplicación con: `node nombre_aplicacion.js`

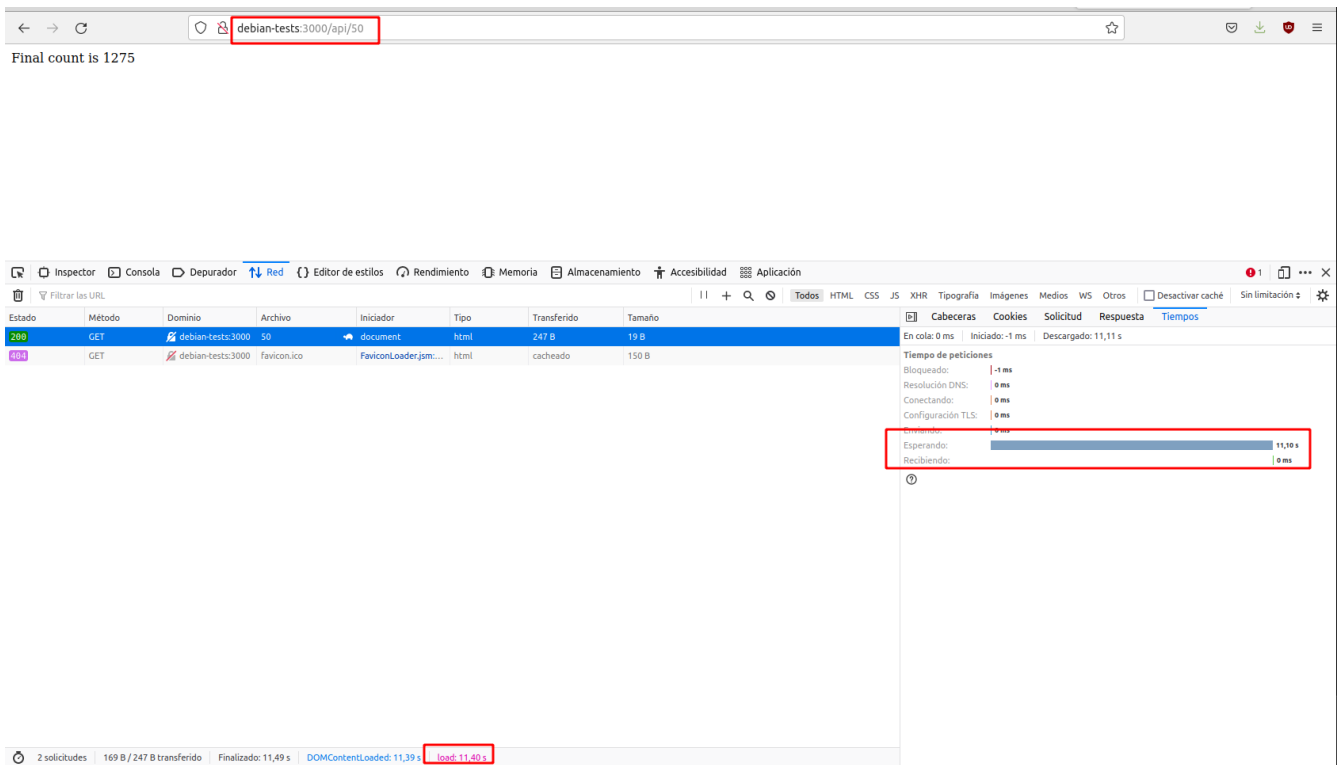
Para comprobarlo, podéis acceder a `http://IP-maq-virtual:3000` o a `http://IP-maq-virtual:3000/api/50` donde `IP-maq-virtual` es la IP del adaptador puente de vuestra Debian.

Utilizada un valor de `n` relativamente pequeño, como el `50` del ejemplo anterior y comprobaréis que se ejecutará rápidamente, devolviendo una respuesta casi inmediata.

Hagamos otra simple comprobación para valores de `n` más grandes. Desplegada e iniciada la aplicación, acceded a la ruta `http://IP-maq-virtual:3000/api/5000000000`.



Mientras esta solicitud que tarda unos segundos se está procesando, acceded en otra pestaña del navegador a `http://IP-maq-virtual:3000` o a `http://IP-maq-virtual:3000/api/n` siendo `n` el valor que le queráis dar.



Utilizando las *developer tools*, podemos ver el tiempo que tardan en procesarse las solicitudes:

1. La primera solicitud, al tener un valor de `n` grande, nos lleva unos cuantos segundos completarla.
2. La segunda solicitud, pese a tener un valor de `n` que ya habíamos comprobado que ofrecía una respuesta casi inmediata, también se demora unos segundos.

2.1.1. ¿Por qué ocurre esto?

Porque el único subproceso estará ocupado procesando la otra operación de ejecución prolongada. El único núcleo de la CPU tiene que completar la primera solicitud antes de que pueda encargarse de la otra.

2.2. ¡Ahora con más clúster!

Ahora usaremos el módulo de clúster en la aplicación para generar algunos procesos secundarios y ver cómo eso mejora las cosas.

A continuación se muestra la aplicación modificada:

```
const express = require("express");
const port = 3000;
const limit = 5000000000;
const cluster = require("cluster");
const totalCPUs = require("os").cpus().length;

if (cluster.isMaster) {
  console.log(`Number of CPUs is ${totalCPUs}`);
  console.log(`Master ${process.pid} is running`);
```

```

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

cluster.on("exit", (worker, code, signal) => {
  console.log(`worker ${worker.process.pid} died`);
  console.log("Let's fork another worker!");
  cluster.fork();
});
} else {
  const app = express();
  console.log(`Worker ${process.pid} started`);

  app.get("/", (req, res) => {
    res.send("Hello World!");
  });

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

    if (n > limit) n = limit;

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

    res.send(`Final count is ${count}`);
  });

  app.listen(port, () => {
    console.log(`App listening on port ${port}`);
  });
}

```

Esta aplicación hace lo mismo que antes pero esta vez estamos generando varios procesos secundarios que compartirán el puerto **3000** y que podrán manejar las solicitudes enviadas a este puerto. Los procesos de trabajo se generan utilizando el método **child_process.fork()**. El método devuelve un objeto **ChildProcess** que tiene un canal de comunicación incorporado que permite que los mensajes se transmitan entre el hijo y su padre.

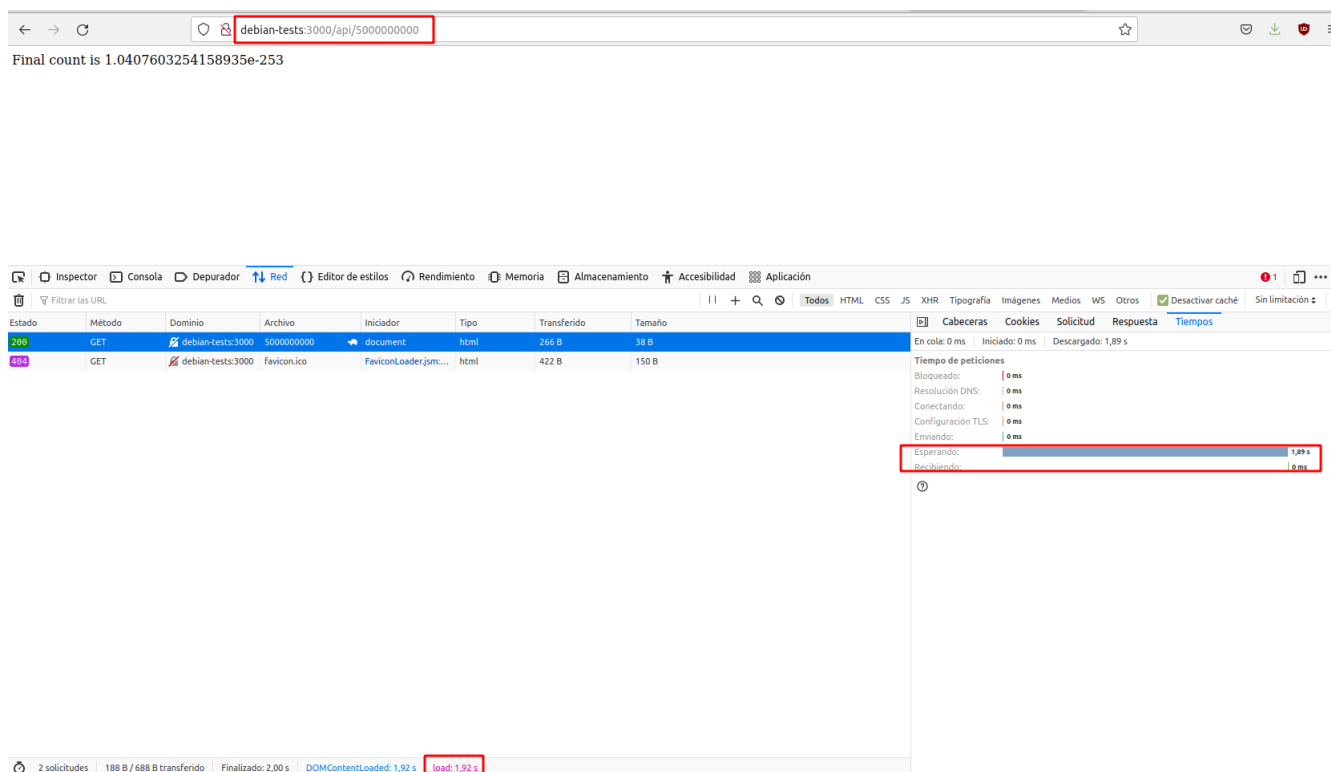
Creemos tantos procesos secundarios como núcleos de CPU hay en la máquina en la que se ejecuta la aplicación. Se recomienda no crear más *workers* que núcleos lógicos en la computadora, ya que esto puede causar una sobrecarga en términos de costos de programación. Esto sucede porque el sistema tendrá que programar todos los procesos creados para que se vayan ejecutando por turnos en los núcleos.

Los *workers* son creados y administrados por el proceso maestro. Cuando la aplicación se ejecuta

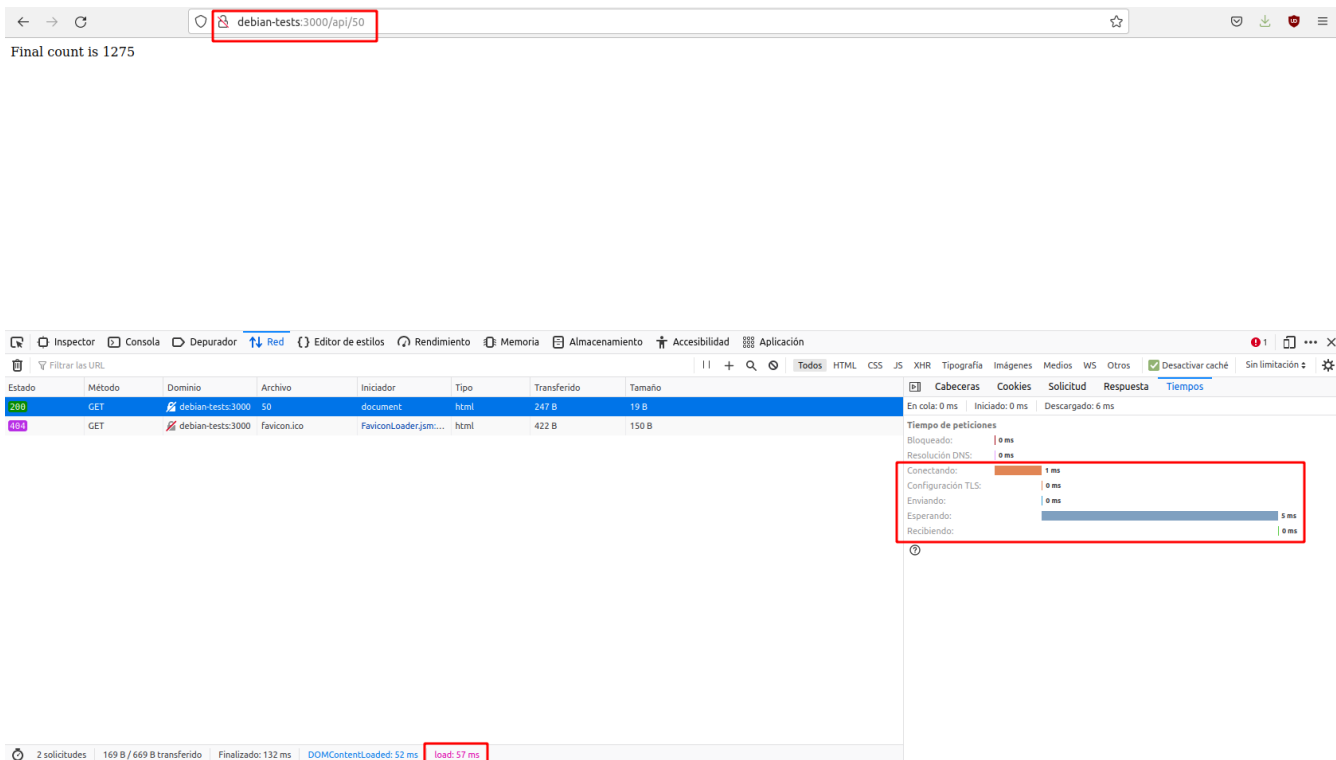
por primera vez, verificamos si es un proceso maestro con `isMaster`. Esto está determinado por la variable `process.env.NODE_UNIQUE_ID`. Si `process.env.NODE_UNIQUE_ID` tiene valor *undefined*, entonces `isMaster` será *true*.

Si el proceso es un maestro, llamamos a `cluster.fork()` para generar varios procesos. Registramos los ID de proceso maestro y *worker*. Cuando un proceso secundario muere, generamos uno nuevo para seguir utilizando los núcleos de CPU disponibles.

Ahora repetiremos el mismo experimento de antes, primero realizamos una solicitud al servidor con un valor alto `n`:



Y ejecutamos rápidamente otra solicitud en otra pestaña del navegador, midiendo los tiempos de procesamiento de ambas:



Comprobaremos que éstos se reducen drásticamente.



Con varios *workers* disponibles para aceptar solicitudes, se mejoran tanto la disponibilidad del servidor como el rendimiento.

Ejecutar una solicitud en una pestaña del navegador y ejecutar rápidamente otra en una segunda pestaña sirve para mostrarnos la mejora que ofrece la agrupación en clústeres para nuestro ejemplo de una forma más o menos rápida, pero es un método un tanto *chapucero* y no es una forma adecuada o confiable de determinar las mejoras de rendimiento.

En el siguiente apartado echaremos un vistazo a algunos puntos de referencia que demostrarán mejor cuánto ha mejorado la agrupación en clústeres nuestra aplicación.

3. Métricas de rendimiento

Realizaremos una prueba de carga en nuestras dos aplicaciones para ver cómo cada una maneja una gran cantidad de conexiones entrantes. Usaremos el paquete `loadtest` para esto.

El paquete `loadtest` nos permite simular una gran cantidad de conexiones simultáneas a nuestra API para que podamos medir su rendimiento.

Para usar `loadtest`, primero debemos instalarlo globalmente. Tras conectarnos por SSH al servidor Debian:

```
npm install -g loadtest
```

Luego ejecutamos la aplicación que queremos probar (`node nombre_aplicacion.js`). Comenzaremos probando la versión que no utiliza la agrupación en clústeres.

Mientras ejecutamos la aplicación, en otro terminal realizamos la siguiente prueba de carga:

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

El comando anterior enviará 1000 solicitudes a la URL dada, de las cuales 100 son concurrentes.

Ejecutar con 1000 solicitudes, 100 concurrentes el número 500000

```
Target URL:      http://localhost:3000/api/500000
Max requests:    1000
Concurrent clients: 200
Running on cores: 2 ①
Agent:          none
```

```
Completed requests: 1000
Total errors:       0
Total time:         1.717 s ②
Mean latency:       306.7 ms ③
Effective rps:      582 ④
```

Percentage of requests served within a certain time

50%	312 ms
90%	335 ms
95%	383 ms
99%	427 ms
100%	438 ms (longest request)

- ① Usando dos núcleos.
- ② Tiempo total de la prueba
- ③ Latencia media, el tiempo promedio que tarda en completar una solicitud
- ④ RPS (*Request per second*) o peticiones por segundo

Intentémoslo de nuevo, pero esta vez con más solicitudes 500000000 (y sin clústeres).

Ejecutar con 1000 solicitudes, 100 concurrentes el número 500000000

```
Target URL:      http://localhost:3000/api/500000000
Max requests:    1000
Concurrent clients: 200
Running on cores: 2
Agent:          none
```

```
Completed requests: 1000
Total errors:       0
Total time:         684.287 s ①
Mean latency:       123225.1 ms ②
Effective rps:      1 ③
```

Percentage of requests served within a certain time

50%	136441 ms
90%	137643 ms
95%	138096 ms
99%	138452 ms
100%	138459 ms (longest request)

- ① Tiempo total
- ② Latencia media
- ③ RPS

Vemos que las métricas arrojan resultados aún peores.

Ahora detenemos nuestra aplicación sin clústers y ejecutamos la que sí los tiene (`node nombre_aplicacion_cluster.js`). Ejecutaremos exactamente las mismas pruebas con el objetivo de realizar una comparación:

Repetimos la prueba primera con cluster

```
Target URL:      http://localhost:3000/api/500000
Max requests:    1000
Concurrent clients: 200
Running on cores: 2
Agent:           none
```

```
Completed requests: 1000
Total errors:       0
Total time:         1.16 s ①
Mean latency:       208.1 ms ②
Effective rps:      862 ③
```

Percentage of requests served within a certain time

50%	197 ms
90%	277 ms
95%	355 ms
99%	420 ms
100%	429 ms (longest request)

- ① Tiempo total
- ② Latencia media
- ③ RPS

Repetimos la segunda prueba con cluster

```
Target URL:      http://localhost:3000/api/500000000
Max requests:    1000
Concurrent clients: 200
Running on cores: 2
Agent:           none
```

```
Completed requests: 1000
Total errors: 0
Total time: 175.594 s ①
Mean latency: 31537.1 ms ②
Effective rps: 6 ③
```

Percentage of requests served within a certain time

50%	34473 ms
90%	36082 ms
95%	36103 ms
99%	36135 ms
100%	36147 ms (longest request)

① Tiempo total

② Latencia media

③ RPS

Es obvio que los clústers permiten manejar una mayor cantidad de peticiones por segundo con una menor latencia.

4. Uso de PM2 para administrar un clúster de Node.js

En nuestra aplicación, hemos usado el módulo `cluster` de Node.js para crear y administrar manualmente los procesos.

Primero hemos determinado la cantidad de *workers* (usando la cantidad de núcleos de CPU como referencia), luego los hemos generado y, finalmente, escuchamos si hay *workers* muertos para poder generar nuevos.

En nuestra aplicación de ejemplo muy sencilla, tuvimos que escribir una cantidad considerable de código solo para administración la agrupación en clústeres. En una aplicación de producción es bastante probable que se deba escribir aún más código.

Existe una herramienta que nos puede ayudar a administrar todo esto un poco mejor: el administrador de procesos **PM2**. **PM2** es un administrador de procesos de producción para aplicaciones Node.js con un balanceador de carga incorporado.

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

PM2 facilita la parada, eliminación e inicio de procesos, además de disponer de algunas herramientas de monitorización que pueden ayudarnos a monitorizar y ajustar el rendimiento de su aplicación.

Para usar **PM2**, primero instalamos globalmente en nuestra Debian:

```
npm install pm2 -g
```

Vamos a utilizarlo con nuestra primera aplicación, la que no estaba *clusterizada* en el código. Para ello ejecutaremos el siguiente comando:

```
pm2 start nombre_aplicacion_sin_cluster.js -i 0
```

Donde:

-i

le indicará a **PM2** que inicie la aplicación en **cluster_mode** (a diferencia de **fork_mode**).

Si se establece a 0, **PM2** generará automáticamente tantos *workers* como núcleos de CPU haya. Y así, nuestra aplicación se ejecuta en modo de clúster, sin necesidad de cambios de código.



Tarea

Ejecuta y documenta con capturas de pantallas, las mismas pruebas que antes pero utilizando PM2 y comprueba si se obtienen los mismos resultados.

Por detrás, **PM2** también utiliza el módulo `cluster` de Node.js, así como otras herramientas que facilitan la gestión de procesos.

En el Terminal, obtendremos una tabla (parecida, ya que se ha adaptado la salida al documento) que muestra algunos detalles de los procesos generados:

Ejemplo 1. Salida del comando `pm2 start cluster-demo.js -i 0`

[illegible]

Runtime Edition

PM2 is a Production Process Manager for Node.js applications with a built-in Load Balancer.

Start and Daemonize any application:
\$ pm2 start app.js

Load Balance 4 instances of api.js:

```
$ pm2 start api.js -i 4
```

Monitor in production:

```
$ pm2 monitor
```

Make pm2 auto-boot at server restart:

```
$ pm2 startup
```

To go further checkout:

<http://pm2.io/>

```
[PM2] Spawning PM2 daemon with pm2_home=/home/fraya/.pm2
```

```
[PM2] PM2 Successfully daemonized
```

```
[PM2] Starting ... [línea cortada por formato]
```

```
[PM2] Done.
```

id	name	mode		status	cpu	memory
0	cluster	cluster	1	launching	0%	0b
1	cluster	cluster	0	online	0%	85.9mb
2	cluster	cluster	0	online	0%	68.3mb
3	cluster	cluster	0	online	0%	59.0mb

Podemos detener la aplicación con el siguiente comando:

```
pm2 stop cluster-demo.js
```

La aplicación se desconectará y la salida por terminal mostrará todos los procesos con un estado **stopped**.

Salida del comando anterior

```
[PM2] Applying action stopProcessId on app [cluster.js](ids: [ 0, 1, 2, 3 ])
```

```
[PM2] [cluster](0) ✓
```

```
[PM2] [cluster](1) ✓
```

```
[PM2] [cluster](3) ✓
```

```
[PM2] [cluster](2) ✓
```

id	name	mode		status	cpu	memory
0	cluster	cluster	87	stopped	0%	0b
1	cluster	cluster	87	stopped	0%	0b
2	cluster	cluster	88	stopped	0%	0b

En vez de tener que pasar siempre las configuraciones cuando ejecuta la aplicación con `pm2 start app.js -i 0`, podríamos facilitarnos la tarea y guardarlas en un archivo de configuración separado, llamado `Ecosystem`.

Este archivo también nos permite establecer configuraciones específicas para diferentes aplicaciones.

Crearemos el archivo `Ecosystem` con el siguiente comando:

```
pm2 ecosystem
```

Que generará un archivo llamado `ecosystem.config.js`. Para el caso concreto de nuestra aplicación, necesitamos modificarlo como se muestra a continuación:

```
module.exports = {
  apps: [{
    name: "nombre_aplicacion",
    script: "nombre_aplicacion_sin_cluster.js",
    instances: 0,
    exec_mode: "cluster",
  },
],
};
```

Al configurar `exec_mode` con el valor `cluster`, le indica a `PM2` que balancee la carga entre cada instancia. `instances` está configurado a `0` como antes, lo que generará tantos *workers* como núcleos de CPU.

La opción `-i` o `instances` se puede establecer con los siguientes valores:

`0` o `max`

(en desuso) para ``repartir" la aplicación entre todas las CPU

`-1`

para ``repartir" la aplicación en todas las CPU - 1

`número`

para difundir la aplicación a través de un número concreto de CPU

Ahora podemos ejecutar la aplicación con:

```
pm2 start ecosystem.config.js
```

La aplicación se ejecutará en modo clúster, exactamente como antes.

Podremos iniciar, reiniciar, recargar, detener y eliminar una aplicación con los siguientes comandos, respectivamente:

```
$ pm2 start nombre_aplicacion
$ pm2 restart nombre_aplicacion
$ pm2 reload nombre_aplicacion
$ pm2 stop nombre_aplicacion
$ pm2 delete nombre_aplicacion

# Cuando usemos el archivo Ecosystem:

$ pm2 [start|restart|reload|stop|delete] ecosystem.config.js
```

El comando **restart** elimina y reinicia inmediatamente los procesos, mientras que el comando **reload** logra un tiempo de inactividad de 0 segundos donde los *workers* se reinician uno por uno, esperando que aparezca un nuevo *worker* antes de matar al anterior.

También puede verificar el estado, los registros y las métricas de las aplicaciones en ejecución.

Tarea

Investiga los siguientes comandos y explica que salida por terminal nos ofrecen y para qué se utilizan:



1. **pm2 ls**
2. **pm2 logs**
3. **pm2 monit**



Documenta la realización de toda esta práctica adecuadamente, con las explicaciones y justificaciones necesarias, las respuestas a las preguntas planteadas y las capturas de pantalla pertinentes.

5. Cuestiones

Fijáos en las siguientes imágenes:

```

raul-debian@debian-tests:~/Documentos/cluster_demo$ loadtest http://localhost:3000/api/50 -n 1000 -c 100
[Sat Jul 23 2022 13:20:18 GMT+0200 (hora de verano de Europa central)] INFO Requests: 0 (0%), requests per second: 0, mean latency: 0 ms
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO Target URL: http://localhost:3000/api/50
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO Max requests: 1000
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO Concurrency level: 100
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO Agent: none
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO Completed requests: 1000
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO Total errors: 0
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO Total time: 1.1908691169999999 s
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO Requests per second: 840
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO Mean latency: 109.4 ms
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO Percentage of the requests served within a certain time
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO 50% 103 ms
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO 90% 154 ms
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO 95% 161 ms
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO 99% 183 ms
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO 100% 192 ms (longest request)
^C
raul-debian@debian-tests:~/Documentos/cluster_demo$ loadtest http://localhost:3000/api/5000 -n 1000 -c 100
[Sat Jul 23 2022 13:20:33 GMT+0200 (hora de verano de Europa central)] INFO Requests: 0 (0%), requests per second: 0, mean latency: 0 ms
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO Target URL: http://localhost:3000/api/5000
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO Max requests: 1000
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO Concurrency level: 100
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO Agent: none
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO Completed requests: 1000
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO Total errors: 0
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO Total time: 1.088355608 s
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO Requests per second: 919
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO Mean latency: 97.6 ms
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO Percentage of the requests served within a certain time
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO 50% 88 ms
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO 90% 124 ms
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO 95% 130 ms
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO 99% 158 ms
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO 100% 162 ms (longest request)
raul-debian@debian-tests:~/Documentos/cluster_demo$ node cluster_demo.js
App listening on port 3000

```



```

raul-debian@debian-tests:~/Documentos/cluster_demo$ loadtest http://localhost:3000/api/50 -n 1000 -c 100
[Sat Jul 23 2022 13:22:37 GMT+0200 (hora de verano de Europa central)] INFO Requests: 0 (0%), requests per second: 0, mean latency: 0 ms
[Sat Jul 23 2022 13:22:38 GMT+0200 (hora de verano de Europa central)] INFO
[Sat Jul 23 2022 13:22:38 GMT+0200 (hora de verano de Europa central)] INFO Target URL: http://localhost:3000/api/50
[Sat Jul 23 2022 13:22:38 GMT+0200 (hora de verano de Europa central)] INFO Max requests: 1000
[Sat Jul 23 2022 13:22:38 GMT+0200 (hora de verano de Europa central)] INFO Concurrency level: 100
[Sat Jul 23 2022 13:22:38 GMT+0200 (hora de verano de Europa central)] INFO Agent: none
[Sat Jul 23 2022 13:22:38 GMT+0200 (hora de verano de Europa central)] INFO
[Sat Jul 23 2022 13:22:38 GMT+0200 (hora de verano de Europa central)] INFO Completed requests: 1000
[Sat Jul 23 2022 13:22:38 GMT+0200 (hora de verano de Europa central)] INFO Total errors: 0
[Sat Jul 23 2022 13:22:38 GMT+0200 (hora de verano de Europa central)] INFO Total time: 1.200859966 s
[Sat Jul 23 2022 13:22:38 GMT+0200 (hora de verano de Europa central)] INFO Requests per second: 833
[Sat Jul 23 2022 13:22:38 GMT+0200 (hora de verano de Europa central)] INFO Mean latency: 112.3 ms
[Sat Jul 23 2022 13:22:38 GMT+0200 (hora de verano de Europa central)] INFO
[Sat Jul 23 2022 13:22:38 GMT+0200 (hora de verano de Europa central)] INFO Percentage of the requests served within a certain time
[Sat Jul 23 2022 13:22:38 GMT+0200 (hora de verano de Europa central)] INFO 50% 103 ms
[Sat Jul 23 2022 13:22:38 GMT+0200 (hora de verano de Europa central)] INFO 90% 157 ms
[Sat Jul 23 2022 13:22:38 GMT+0200 (hora de verano de Europa central)] INFO 95% 223 ms
[Sat Jul 23 2022 13:22:38 GMT+0200 (hora de verano de Europa central)] INFO 99% 242 ms
[Sat Jul 23 2022 13:22:38 GMT+0200 (hora de verano de Europa central)] INFO 100% 252 ms (longest request)
^C^C
raul-debian@debian-tests:~/Documentos/cluster_demo$ loadtest http://localhost:3000/api/5000 -n 1000 -c 100
[Sat Jul 23 2022 13:22:48 GMT+0200 (hora de verano de Europa central)] INFO Requests: 0 (0%), requests per second: 0, mean latency: 0 ms
[Sat Jul 23 2022 13:22:49 GMT+0200 (hora de verano de Europa central)] INFO
[Sat Jul 23 2022 13:22:49 GMT+0200 (hora de verano de Europa central)] INFO Target URL: http://localhost:3000/api/5000
[Sat Jul 23 2022 13:22:49 GMT+0200 (hora de verano de Europa central)] INFO Max requests: 1000
[Sat Jul 23 2022 13:22:49 GMT+0200 (hora de verano de Europa central)] INFO Concurrency level: 100
[Sat Jul 23 2022 13:22:49 GMT+0200 (hora de verano de Europa central)] INFO Agent: none
[Sat Jul 23 2022 13:22:49 GMT+0200 (hora de verano de Europa central)] INFO
[Sat Jul 23 2022 13:22:49 GMT+0200 (hora de verano de Europa central)] INFO Completed requests: 1000
[Sat Jul 23 2022 13:22:49 GMT+0200 (hora de verano de Europa central)] INFO Total errors: 0
[Sat Jul 23 2022 13:22:49 GMT+0200 (hora de verano de Europa central)] INFO Total time: 1.238362378 s
[Sat Jul 23 2022 13:22:49 GMT+0200 (hora de verano de Europa central)] INFO Requests per second: 808
[Sat Jul 23 2022 13:22:49 GMT+0200 (hora de verano de Europa central)] INFO Mean latency: 115.6 ms
[Sat Jul 23 2022 13:22:49 GMT+0200 (hora de verano de Europa central)] INFO
[Sat Jul 23 2022 13:22:49 GMT+0200 (hora de verano de Europa central)] INFO Percentage of the requests served within a certain time
[Sat Jul 23 2022 13:22:49 GMT+0200 (hora de verano de Europa central)] INFO 50% 118 ms
[Sat Jul 23 2022 13:22:49 GMT+0200 (hora de verano de Europa central)] INFO 90% 142 ms
[Sat Jul 23 2022 13:22:49 GMT+0200 (hora de verano de Europa central)] INFO 95% 148 ms
[Sat Jul 23 2022 13:22:49 GMT+0200 (hora de verano de Europa central)] INFO 99% 174 ms
[Sat Jul 23 2022 13:22:49 GMT+0200 (hora de verano de Europa central)] INFO 100% 182 ms (longest request)

raul-debian@debian-tests:~/Documentos/cluster_demo$ node cluster_demo_2.js
Number of CPUs is 4
Master 2209 is running
Worker 2218 started
Worker 2217 started
Worker 2224 started
App listening on port 3000
App listening on port 3000
App listening on port 3000
Worker 2216 started
App listening on port 3000

```

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?

6. Referencias

1. [How to install ExpressJS on Debian 11?](#)
2. [Improving Node.js Application Performance With Clustering](#)

7. Evaluación

Criterio	Puntuación
Pruebas correctas y bien documentadas de despliegue de la aplicación sin cluster	2 puntos
Pruebas correctas y bien documentadas de despliegue de la aplicación con cluster	2 puntos

Criterio	Puntuación
Pruebas correctas y bien documentadas de todas las opciones con loadtest	2 puntos
Pruebas correctas y bien documentadas de todas las opciones con PM2	2 puntos
Respuestas correctas a las cuestiones	1 puntos
Se ha prestado especial atención al formato del documento, utilizando la plantilla actualizada y haciendo un correcto uso del lenguaje técnico	1 puntos