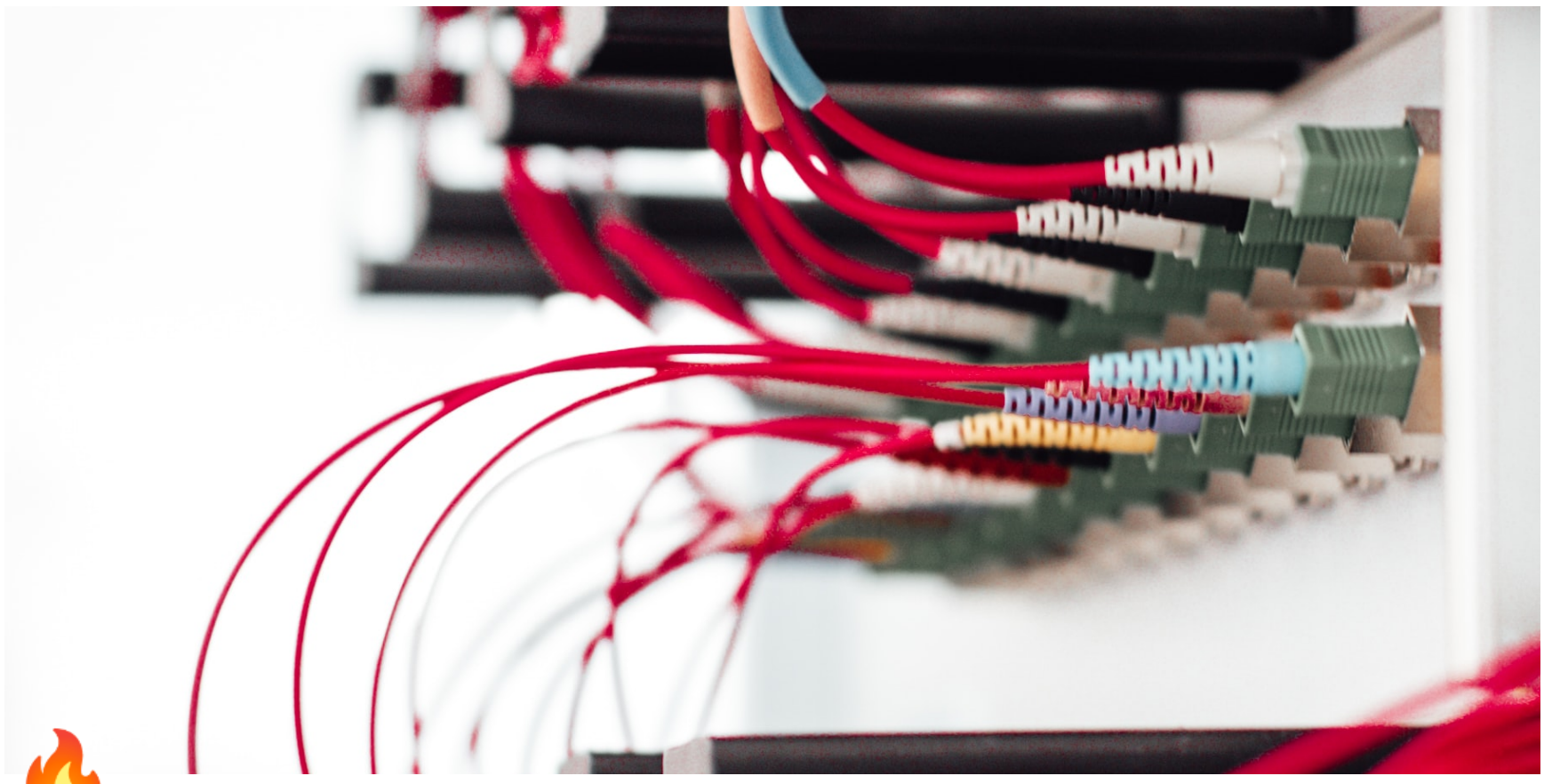
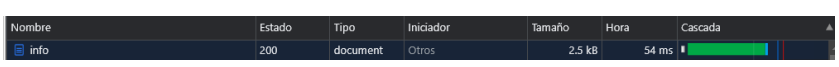


Informe de performance

Modulo `compression`

Con `compresion`

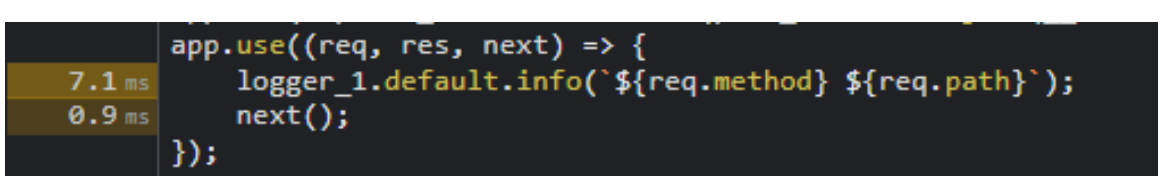
Sin `compresion`



Se aprecia que con el middleware de compresión se **reduce** el tamaño de la petición en un **62.12%** y el tiempo un **15%** (este ultimo es relativo ya que depende de otros factores como velocidad de internet, primera descarga, etc…)

Perfilamiento y test de carga

Al agregar el logger para todas las peticiones este generó código bloqueante para los dos casos generando retraso por cada respuesta



Muestra de logger en modo inspect

Artillery.io

node --prof

```
artillery quick --count 50 -n 20 "http://localhost:8080/info" > artillery.txt

artillery quick --count 50 -n 20 "http://localhost:8080/info" > artillery-log.txt
```

```
[Summary]:
   ticks  total  nonlib   name
   2154   15.2%   57.5%  JavaScript
   1558   11.0%   41.6%  C++
    700    4.9%   18.7%  GC
  10466   73.6%          Shared libraries
     33    0.2%          Unaccounted
```

```
[Summary]:
   ticks  total  nonlib   name
   2042   14.5%   55.8%  JavaScript
   1570   11.2%   42.9%  C++
    576    4.1%   15.7%  GC
  10418   74.0%          Shared libraries
     46    0.3%          Unaccounted
```

Comparando cada componente se nota mayormente que en los ticks de JavaScript se destaca con 112 ticks de diferencia, por otro lado también se generan mas ticks en el Garbage Collector con una diferencia de 124 ticks

Autocannon

```
autocannon -c 100 -d 20  "http://localhost:8080/info"
```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	943 ms	1044 ms	1241 ms	1408 ms	1045.33 ms	98.28 ms	1608 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	48	48	96	103	93.3	11.25	48
Bytes/Sec	119 kB	119 kB	238 kB	256 kB	232 kB	27.9 kB	119 kB

Req/Bytes counts sampled once per second.
# of samples: 20

2k requests in 20.04s, 4.63 MB read

Imagen de muestra sin console.log

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	933 ms	977 ms	1168 ms	1340 ms	979.28 ms	93.39 ms	1525 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	51	51	102	106	99.5	11.45	51
Bytes/Sec	127 kB	127 kB	253 kB	263 kB	247 kB	28.4 kB	127 kB

Req/Bytes counts sampled once per second.
# of samples: 20

2k requests in 20.05s, 4.94 MB read

Imagen de muestra sin console.log

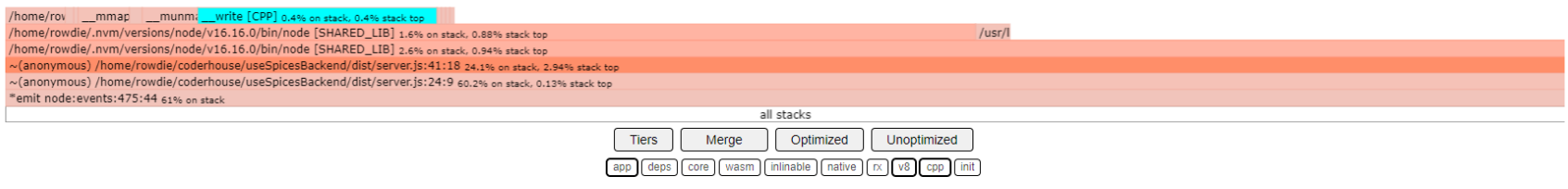
Se puede ver una diferencia notable en cada dato, notando que hay menos latencia y mas peticiones por segundo al no tener ese console.log

0x

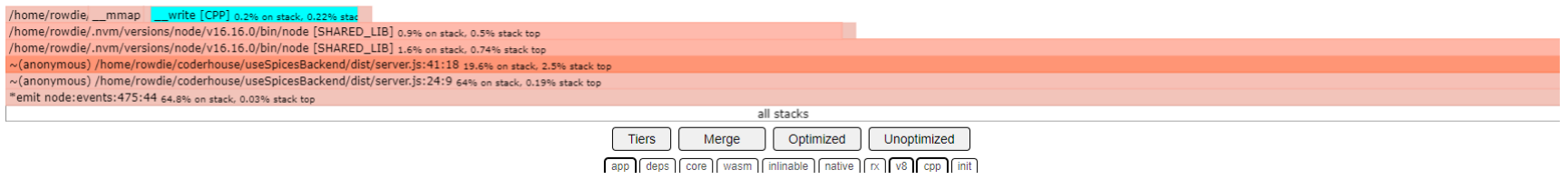
Con console.log los gráficos de flamas de observa que en las líneas 24 y 41 se genera mas ‘calor’ las cuales corresponden a el middleware del logger y la ruta “/info”

Grafico de flama considerando “app” y “inlinable”.

Como Node.js ocupa C++ por debajo, al filtrar por la función `__write` se ve mas claro cuanto retraso genera el console.log



y sin console.log en la ruta “info” la función `__write` esta un **0.2%** menos tiempo en el stack del event loop, donde ese porcentaje que resta es del logger que muestra las peticiones que llegan al servidor



node —inspect y Chrome Inspect

```
41          app.get('/info', (req, res) => {
42              // eslint-disable-next-line @typescript-eslint/no-var-requires
43    1.5 ms    const os = require('os');
44    0.2 ms    const systemInfo = {
45    0.7 ms        platform: process.platform,
46              version: process.version,
47    1.6 ms    cpus: os.cpus().length,
48    1.1 ms    rss: process.memoryUsage.rss(),
49              pid: process.pid,
50    0.5 ms    dirname: process.cwd(),
51    0.4 ms    execPath: process.execPath,
52              args: args,
53          };
54    7.8 ms    console.log(systemInfo);
55   22.9 ms    res.render('layouts/info', systemInfo);
56      });
```

Imagen de muestra con console.log

```
41          app.get('/info', (req, res) => {
42              // eslint-disable-next-line @typescript-eslint/no-var-requires
43    1.3 ms    const os = require('os');
44              const systemInfo = {
45    2.3 ms        platform: process.platform,
46    0.2 ms    version: process.version,
47    1.8 ms    cpus: os.cpus().length,
48    2.0 ms    rss: process.memoryUsage.rss(),
49    0.4 ms    pid: process.pid,
50    0.5 ms    dirname: process.cwd(),
51    0.5 ms    execPath: process.execPath,
52              args: args,
53          };
54          //console.log(systemInfo)
55   14.0 ms    res.render('layouts/info', systemInfo);
56      });
```

Con el modo de inspección de Google Chrome se puede ver a mas detalle que es lo que retrasa en la petición donde el console.log resalta con un tiempo de 7.8 segundos

Conclusión Final

Como es de notar, el uso de código sincrónico en las rutas denota una demora en la respuesta de una petición ocasionando más gastos de recursos innecesarios.

Por lo tanto el empleo de loggers en producción se debe usar en pocas ocasiones o de forma asincrónica para no afectar el flujo de la aplicación.