Analisis de resultados – Mediciones de performance del servidor

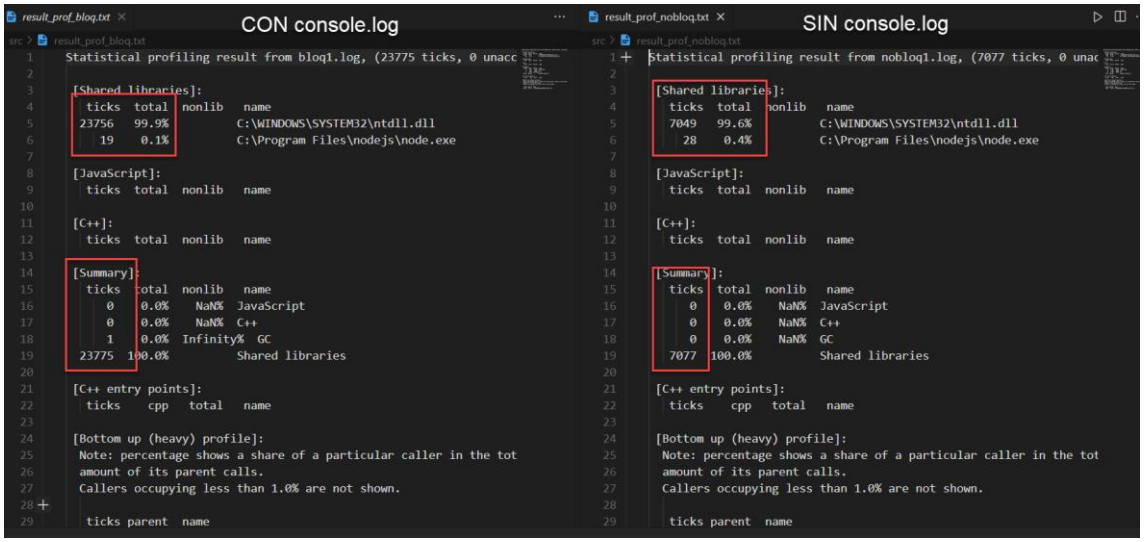
Verificar sobre la ruta /info con y sin compresion la diferencia de bytes

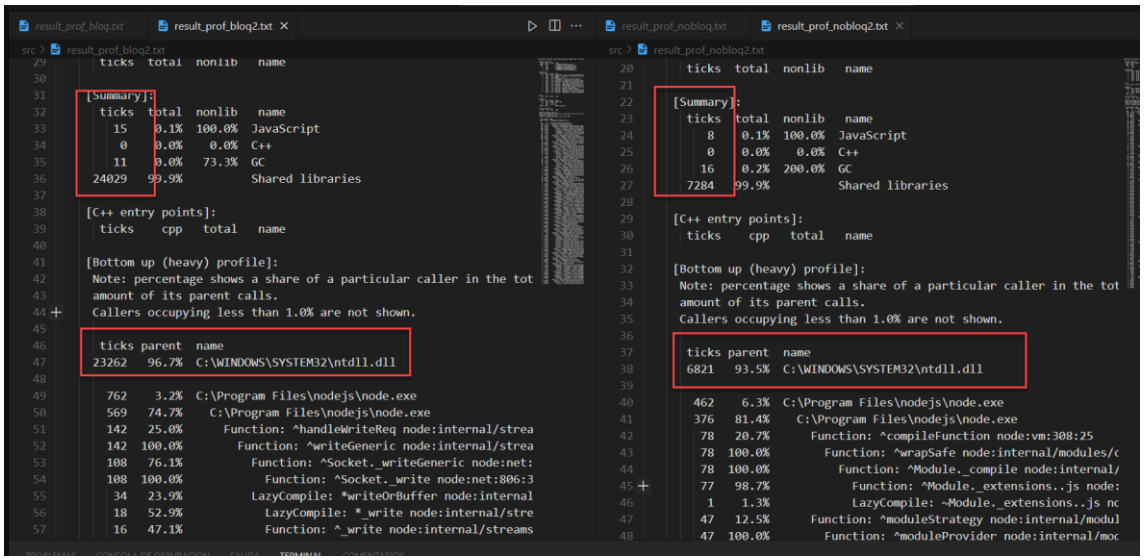
/info sin GZIP => 763B

/info con GZIP => 787B

Creo que como es tan poca info, no llega a comprimir y al agregarle headers de compresión termina siendo mas costoso.

1) **Perfilamiento del servidor**
   Puede observarse en multiples métricas que la cantidad de ciclos de procesamiento (Ticks) aumenta considerablemente al agregar un console.log, ya que esta ejecución es bloqueante, reduciendo considerablemente la performance, lo que nos indica que es una mala practica, y en caso de ser necesario debe considerarse alguna alternativa mas performante como alguna libraría de logs.

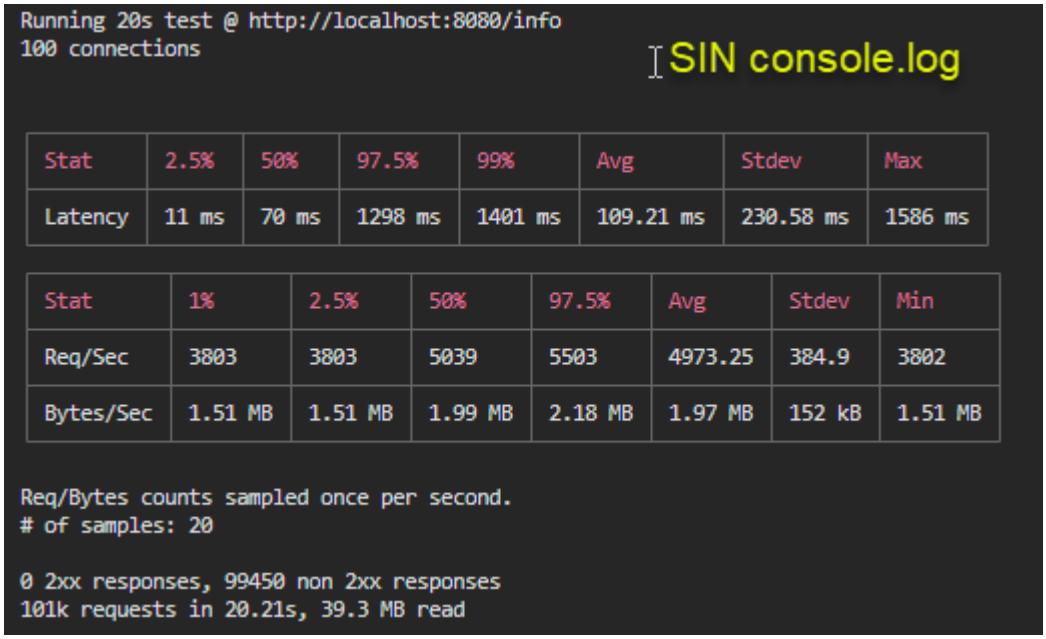




```
result_prof_bloq.txt   result_prof_bloq2.txt ×
src > result_prof_bloq2.txt
    ticks  total  nonlib   name

  [Summary]:
    ticks  total  nonlib   name
       15    0.1%  100.0%  JavaScript
        0    0.0%    0.0%  C++
       11    0.0%   73.3%  GC
    24029   99.9%          Shared libraries

  [C++ entry points]:
    ticks    cpp   total   name

  [Bottom up (heavy) profile]:
  Note: percentage shows a share of a particular caller in the tot
  amount of its parent calls.
  Callers occupying less than 1.0% are not shown.

    ticks parent  name
    23262   96.7%  C:\WINDOWS\SYSTEM32\ntdll.dll

      762    3.2%  C:\Program Files\nodejs\node.exe
      569   74.7%    C:\Program Files\nodejs\node.exe
      142   25.0%      Function: ^handleWriteReq node:internal/strea
      142  100.0%        Function: ^writeGeneric node:internal/strea
      108   76.1%          Function: ^Socket._writeGeneric node:net:
      108  100.0%            Function: ^Socket._write node:net:806:3
       34   23.9%          LazyCompile: *writeOrBuffer node:internal
       18   52.9%            LazyCompile: *_write node:internal/stre
       16   47.1%            Function: ^_write node:internal/streams
```

```
result_prof_nobloq.txt   result_prof_nobloq2.txt ×
src > result_prof_nobloq2.txt
    ticks  total  nonlib   name

  [Summary]:
    ticks  total  nonlib   name
        8    0.1%  100.0%  JavaScript
        0    0.0%    0.0%  C++
       16    0.2%  200.0%  GC
     7284   99.9%          Shared libraries

  [C++ entry points]:
    ticks    cpp   total   name

  [Bottom up (heavy) profile]:
  Note: percentage shows a share of a particular caller in the tot
  amount of its parent calls.
  Callers occupying less than 1.0% are not shown.

    ticks parent  name
    6821   93.5%  C:\WINDOWS\SYSTEM32\ntdll.dll

      462    6.3%  C:\Program Files\nodejs\node.exe
      376   81.4%    C:\Program Files\nodejs\node.exe
       78   20.7%      Function: ^compileFunction node:vm:308:25
       78  100.0%        Function: ^wrapSafe node:internal/modules/c
       78  100.0%          Function: ^Module._compile node:internal/
       77   98.7%            Function: ^Module._extensions..js node:
        1    1.3%            LazyCompile: ~Module._extensions..js nc
       47   12.5%      Function: ^moduleStrategy node:internal/modul
       47  100.0%        Function: ^moduleProvider node:internal/moc
```

También se observa desde las estadísticas obtenidas con Artillery



```
Running 20s test @ http://localhost:8080/info
100 connections
```

I SIN console.log

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	11 ms	70 ms	1298 ms	1401 ms	109.21 ms	230.58 ms	1586 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	3803	3803	5039	5503	4973.25	384.9	3802
Bytes/Sec	1.51 MB	1.51 MB	1.99 MB	2.18 MB	1.97 MB	152 kB	1.51 MB

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

0 2xx responses, 99450 non 2xx responses
101k requests in 20.21s, 39.3 MB read
```

```
Running 20s test @ http://localhost:8080/info
100 connections                                  CON console.log
```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	11 ms	67 ms	1254 ms	1394 ms	108.33 ms	229.93 ms	1611 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	4107	4107	4975	5459	4950.61	325.59	4107
Bytes/Sec	1.63 MB	1.63 MB	1.97 MB	2.16 MB	1.96 MB	128 kB	1.63 MB

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

0 2xx responses, 99006 non 2xx responses
100k requests in 20.16s, 39.2 MB read
```

Se observa que los tiempos de respuesta de Requerimientos por segundo (Req / Seg) son mas eficientes sin utilizar console.log.

Aunque en este caso la diferencia es menos notable.

2) **Perfilamiento con Node –inspect**
   Tambien podemos apreciar la depreciación temporal que aplica el console.log, con la herramienta de inspección del navegador (Chrome)

app.js api.router.js info.router.js ×

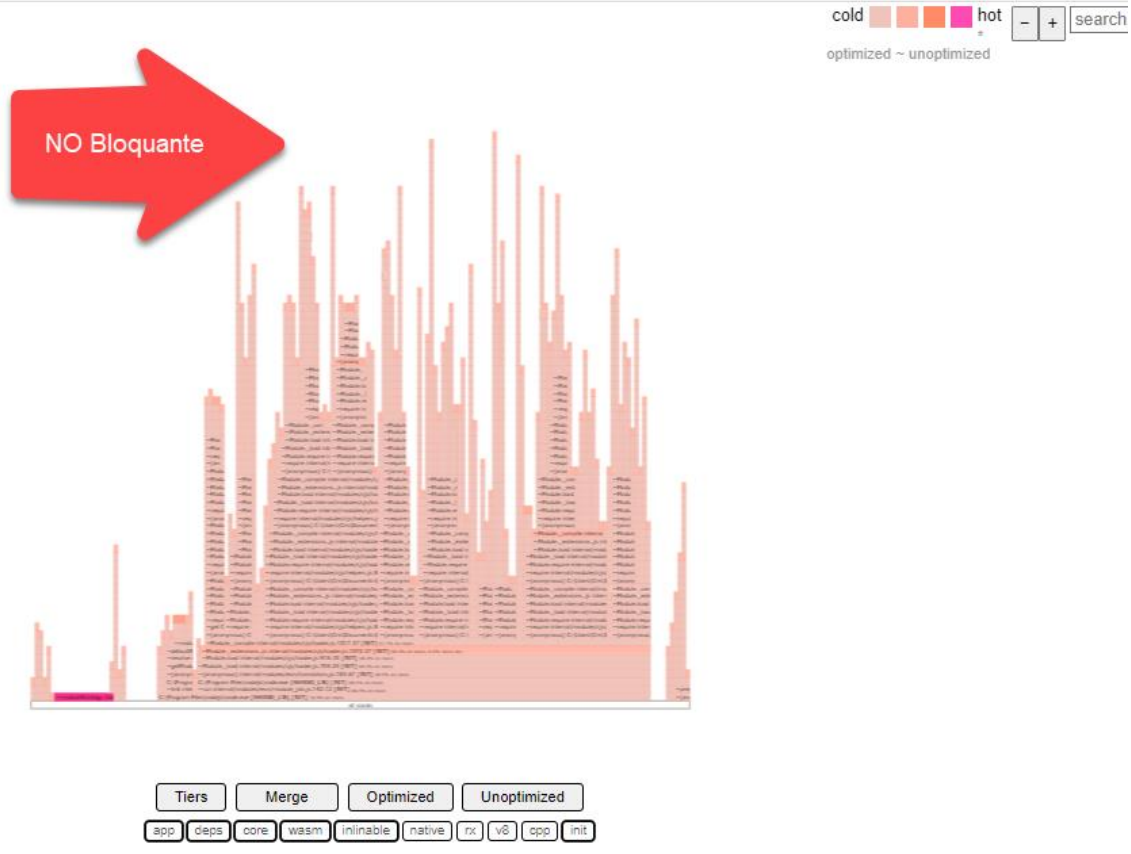
```
1   import express from 'express';
2   import { countCPUs } from '../app.js';
3
4   const { Router } = express;
5   const infoRouter = new Router();
6
7
8   infoRouter.get('/', (req, res) => {
19.7 ms     console.log({'Args': `${process.argv.join(", ")}`,
0.5 ms                   'SO': `${process.platform}`,
0.5 ms                   'Node Version': `${process.version}`,
15.2 ms                  'Mem rss': `${JSON.stringify(process.memoryUsage())}`,
0.7 ms                   'Path executing': `${process.argv[1]}`,
1.7 ms                   'PID': `${process.pid}`,
15.2 ms                  'Foldername Project': `${process.cwd().split('\\')[10]}`,
1.2 ms                   'CPUs': `${countCPUs}`})
31.2 ms     return res.send({
10.2 ms         'Args': `${process.argv.join(", ")}`,
0.7 ms          'SO': `${process.platform}`,
0.2 ms          'Node Version': `${process.version}`,
24.4 ms         'Mem rss': `${JSON.stringify(process.memoryUsage())}`,
1.2 ms          'Path executing': `${process.argv[1]}`,
1.1 ms          'PID': `${process.pid}`,
19.7 ms         'Foldername Project': `${process.cwd().split('\\')[10]}`,
2.4 ms          'CPUs': `${countCPUs}`
26      });
27  });
28
29  infoRouter.get('/*', (req, res) => {
30      res.redirect("/")
31  });
32
33  export default infoRouter;
```

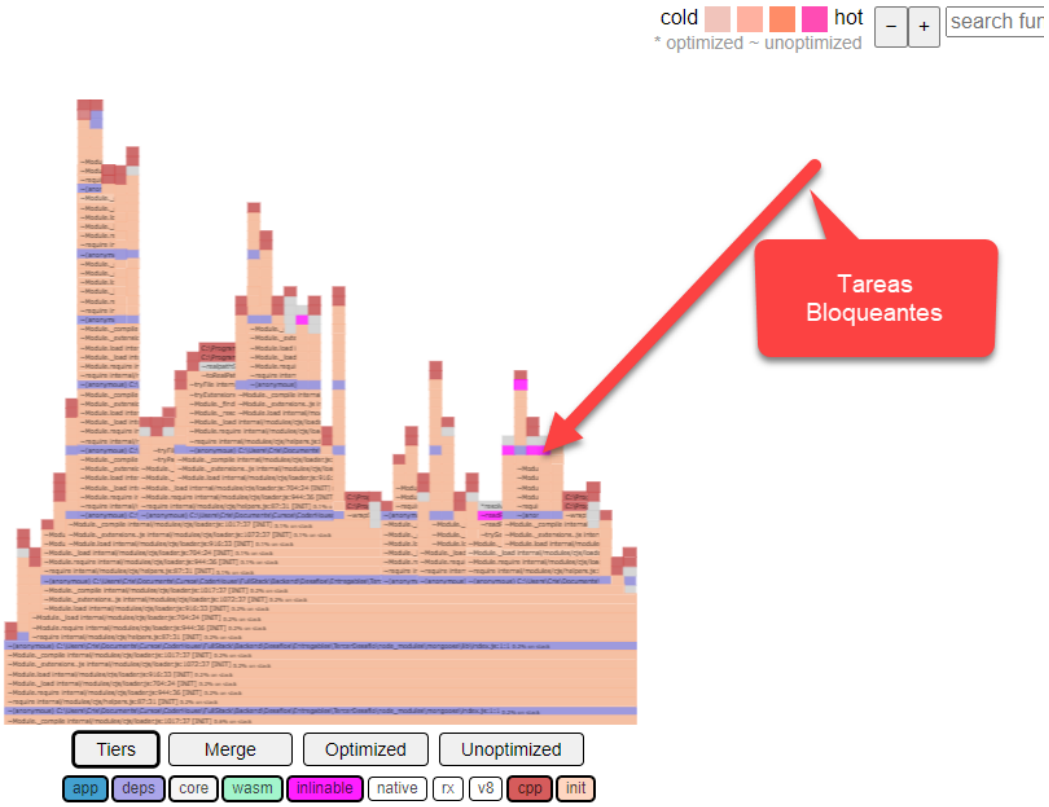
app.js | info.router.js × | api.router.js

```
1   import express from 'express';
2   import { countCPUs } from '../app.js';
3
4   const { Router } = express;
5   const infoRouter = new Router();
6
7
8   infoRouter.get('/', (req, res) => {
9       // console.log({'Args': `${process.argv.join(", ")}`,
10      //                 'SO': `${process.platform}`,
11      //                 'Node Version': `${process.version}`,
12      //                 'Mem rss': `${JSON.stringify(process.memoryUsage())}`,
13      //                 'Path executing': `${process.argv[1]}`,
14      //                 'PID': `${process.pid}`,
15      //                 'Foldername Project': `${process.cwd().split('\\')[10]}`,
16      //                 'CPUs': `${countCPUs}`})
17      return res.send({
18          'Args': `${process.argv.join(", ")}`,
19          'SO': `${process.platform}`,
20          'Node Version': `${process.version}`,
21          'Mem rss': `${JSON.stringify(process.memoryUsage())}`,
22          'Path executing': `${process.argv[1]}`,
23          'PID': `${process.pid}`,
24          'Foldername Project': `${process.cwd().split('\\')[10]}`,
25          'CPUs': `${countCPUs}`
26      });
27  });
28
29  infoRouter.get('/*', (req, res) => {
30      res.redirect("/")
31  });
32
33  export default infoRouter;
```

6.9 ms
6.5 ms
0.2 ms
0.2 ms
15.2 ms
0.2 ms
1.5 ms
13.1 ms
1.8 ms
0.2 ms

3) Diagrama de flama con 0x





Se observa que cuando hay tareas bloqueantes el grafico se ensanacha ya que las tareas bloquantes demoran mas tiempo, mientras que en el grafico “no bloqueante” se forman “torres” finitas dado que se resuelven rápidamente las tareas encoladas.
Si bien se forman “torres” altas nuevamente, debido a la cascada de procesamiento, se resuelven rápidamente representándose gráficamente como “columnas” muy angostas en lugar de “montañas anchas”.