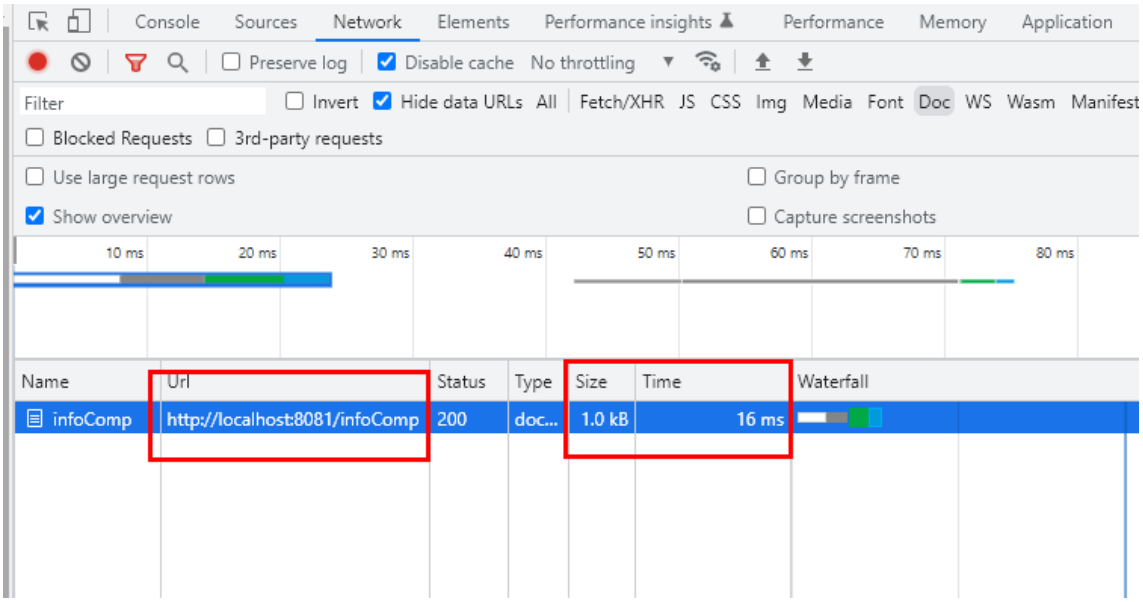
Desafio 16: Informe Análisis perfomance del Server

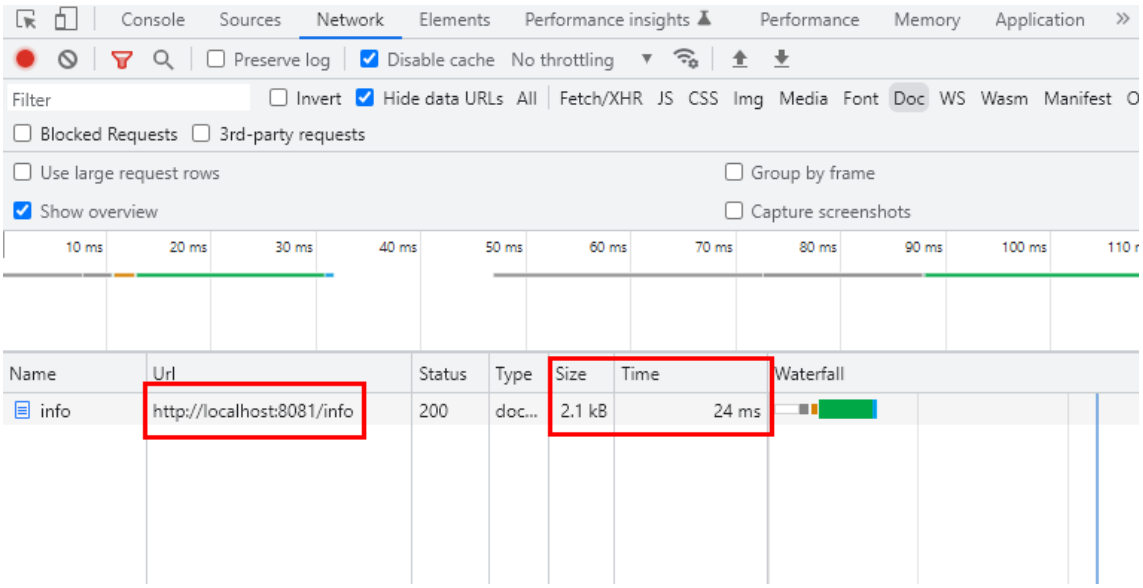
Luiggi Marquez

1. En la prueba de la ruta info con y sin console.log se puede ser que en la ruta comprimida con GZIP el tamaño del archivo se reduce a 1Kb, mientras que la ruta sin compresión pensa 2,2Kb. Al ser más liviana carga en menos tiempo

Ruta /infoComp (comprimida)



Ruta /info (sin compresión)

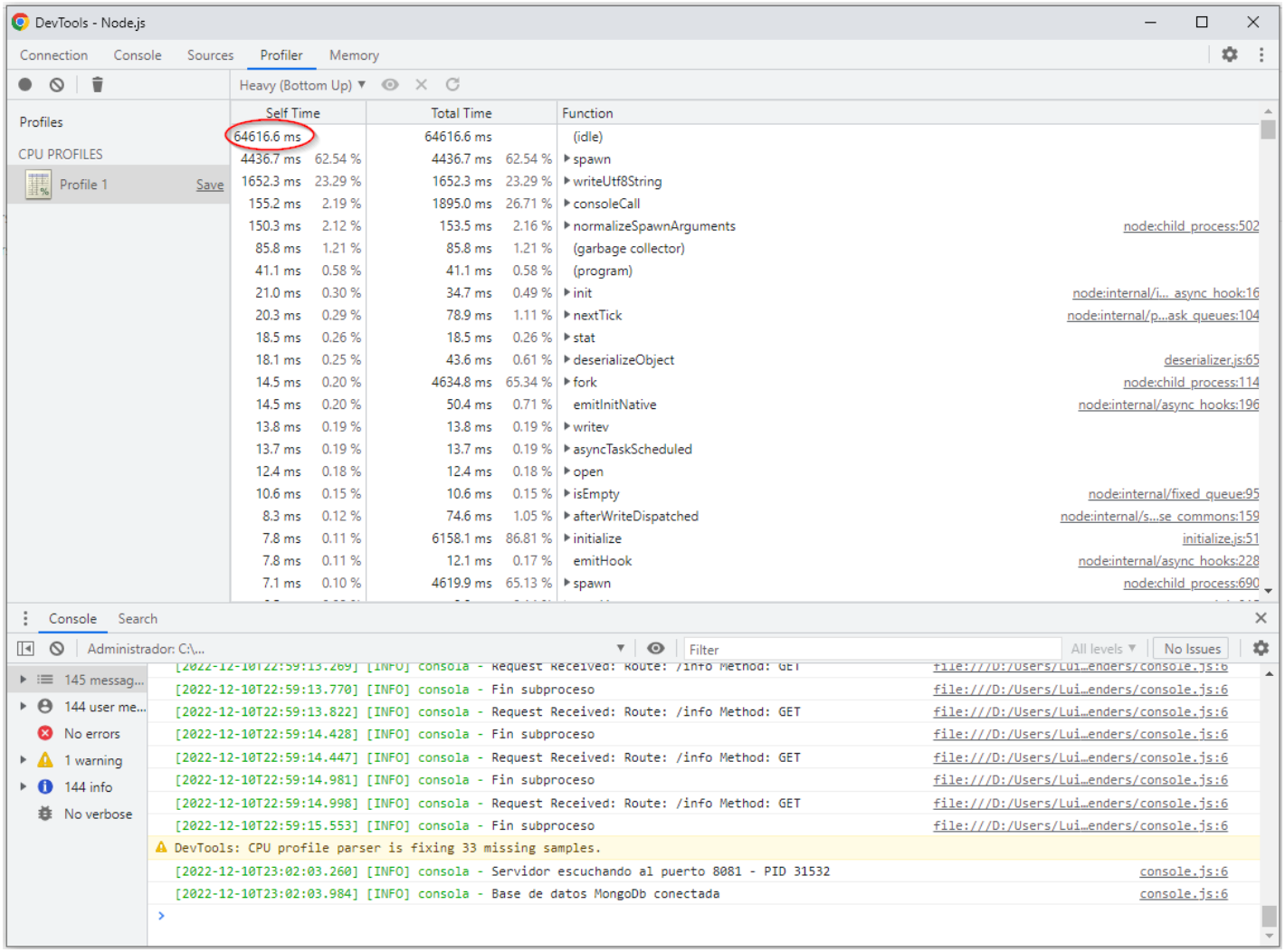


2. Para la prueba del profiling, se utilizó artillery con:

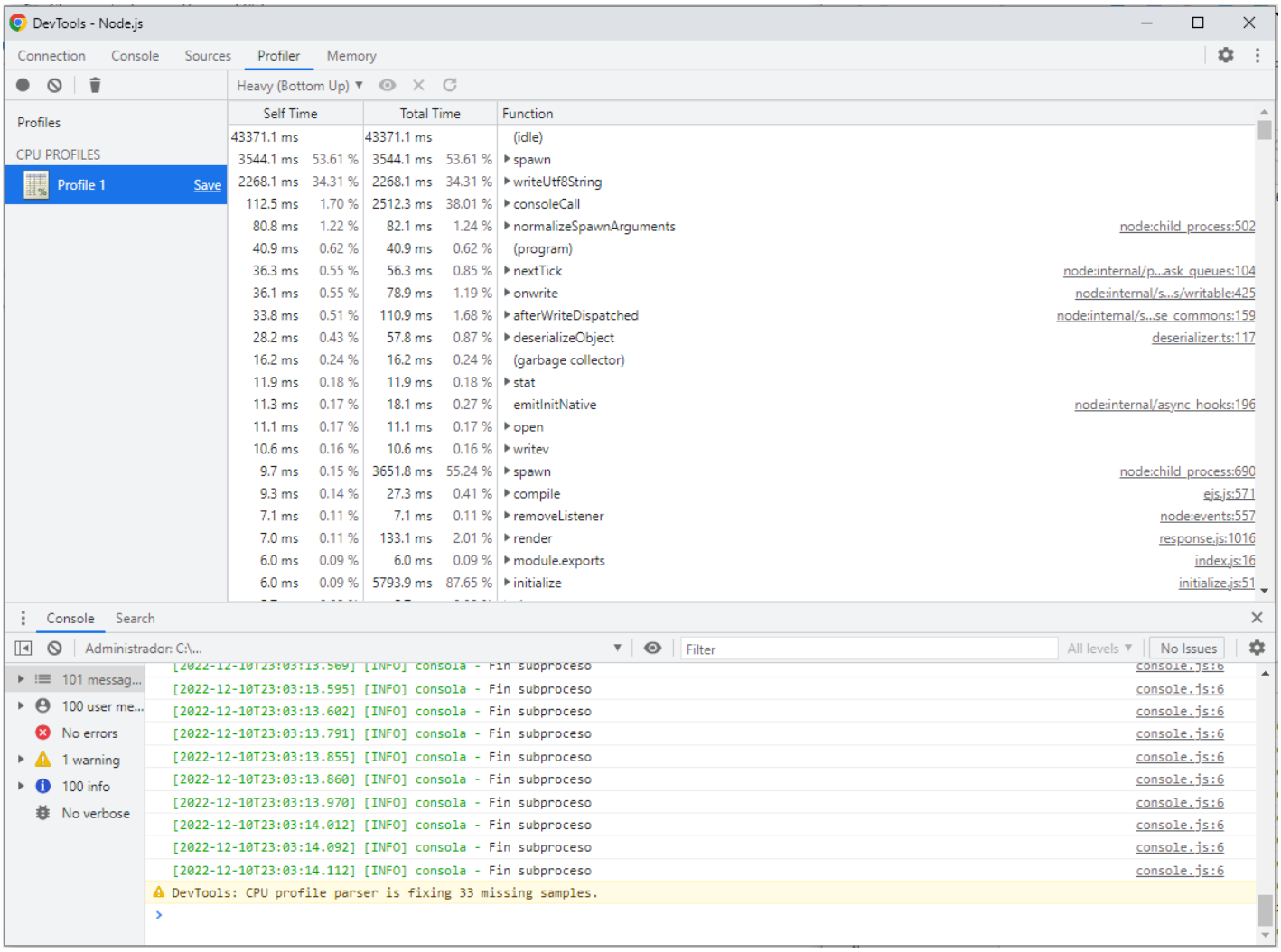
```
artillery quick --count 50 --num 20 'http://localhost:8081/info'
```

Es decir, 50 solicitudes con 20 peticiones GET; para le profiling se uso la herramienta de Chrome DevTools for Node en **chrome://inspect/** pestaña profiler, corriendo el server **como node --inspect server.js**: los resultados arrojados:

Ruta /Info con Console.log trabajando en FORK()



/info sin el console.log



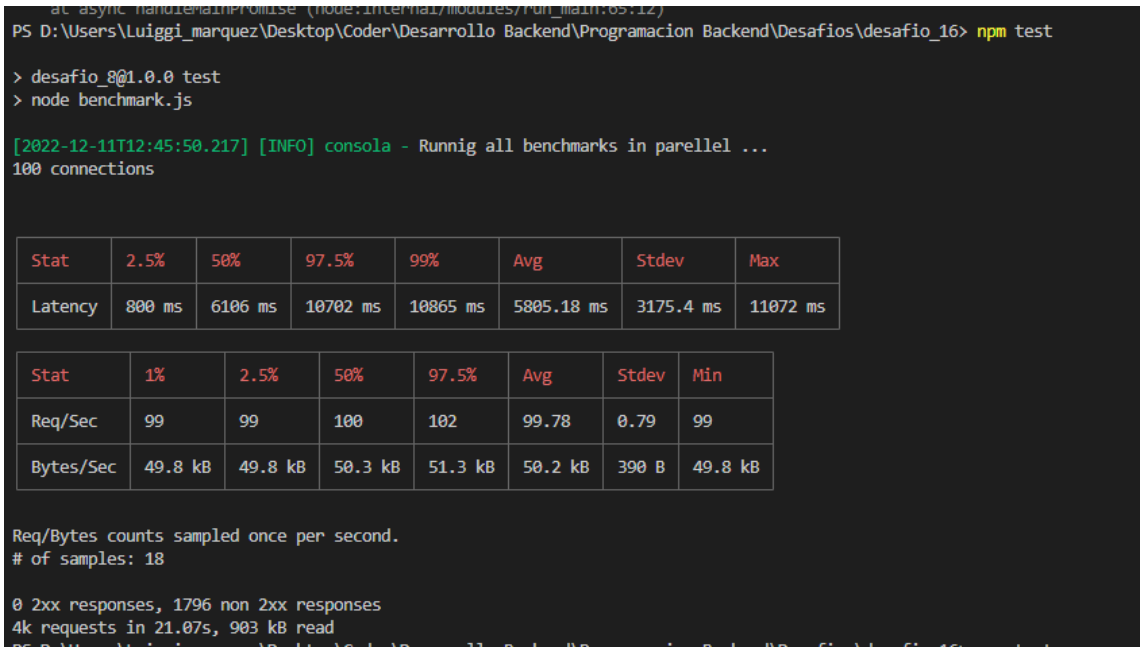

```
PS D:\Users\Luiggi_marquez\Desktop\Coder\Desarrollo Backend\Programacion Backend\Desafios\desafio_16> artillery quick --count 50 --num 20 'http://localhost:8081/info'
Phase started: unnamed (index: 0, duration: 1s) 22:58:49(-0300)

Phase completed: unnamed (index: 0, duration: 1s) 22:58:53(-0300)
```

De estos resultados vemos que en con la ruta con el console.log tarda 64616 ms en completarse contra 43371 ms de la prueba sin el console.log; el console.log retrasa considerablemente la realización del ciclo de la aplicación, esto por ser de naturaleza síncrono y comportarse como elemento bloqueante en al ejecución del programa

3. Prueba realizada con autocannon para ruta info con y sin console.log, se realizo con 100 conexiones en un intervalo de 20 segundos. El código de la prueba está en el archivo berchmark.js en el root del proyecto:

Autocannon con ruta info no bloqueante:



```
    at async handleMainPromise (node:internal/modules/run_main:65:12)
PS D:\Users\Luiggi_marquez\Desktop\Coder\Desarrollo Backend\Programacion Backend\Desafios\desafio_16> npm test

> desafio_8@1.0.0 test
> node benchmark.js

[2022-12-11T12:45:50.217] [INFO] consola - Runnig all benchmarks in parellel ...
100 connections


| Stat | 2.5% | 50% | 97.5% | 99% | Avg | Stdev | Max |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Latency | 800 ms | 6106 ms | 10702 ms | 10865 ms | 5805.18 ms | 3175.4 ms | 11072 ms |

| Stat | 1% | 2.5% | 50% | 97.5% | Avg | Stdev | Min |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Req/Sec | 99 | 99 | 100 | 102 | 99.78 | 0.79 | 99 |
| Bytes/Sec | 49.8 kB | 49.8 kB | 50.3 kB | 51.3 kB | 50.2 kB | 390 B | 49.8 kB |

Req/Bytes counts sampled once per second.
# of samples: 18

0 2xx responses, 1796 non 2xx responses
4k requests in 21.07s, 903 kB read
```

Perfilamiento con ruta /info no bloqueante:

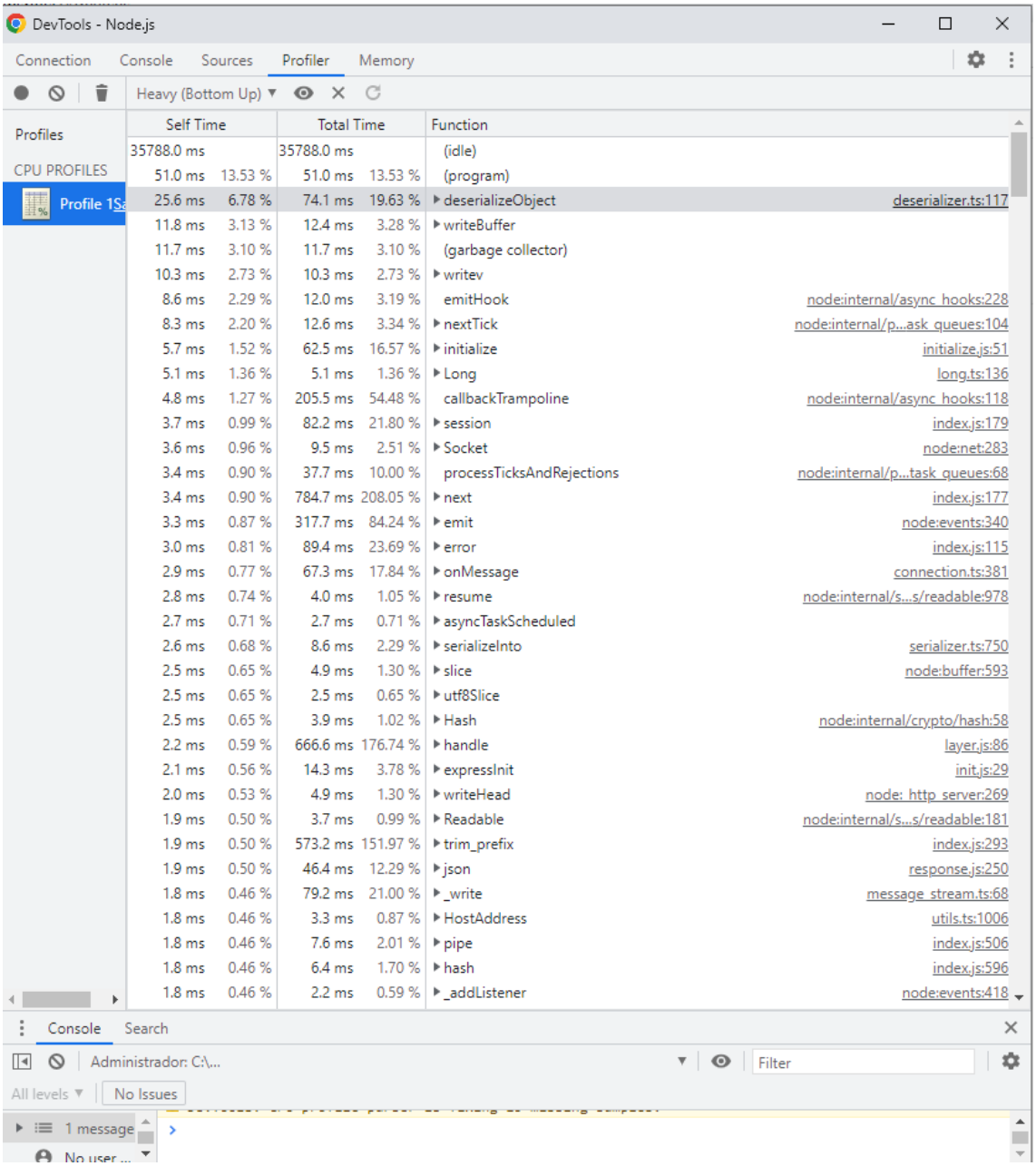
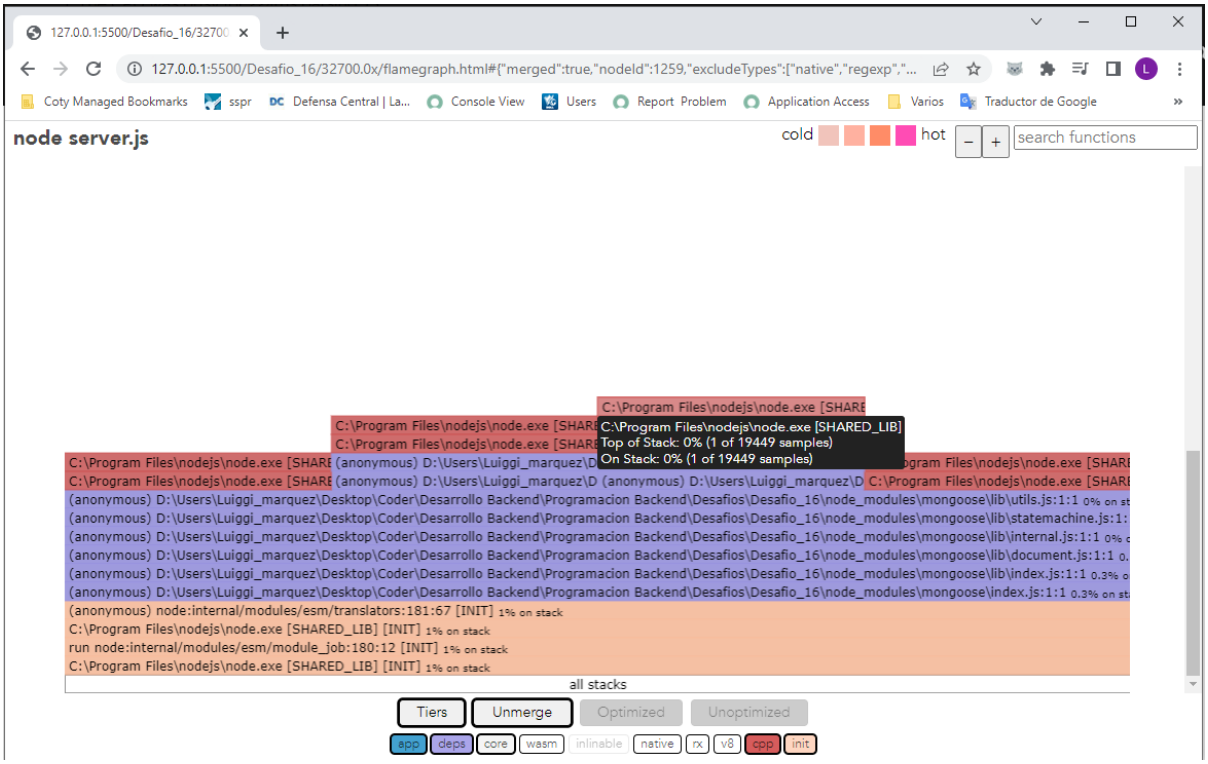


Grafico 0x con ruta /info no bloqueante



Autocannon con ruta /info bloqueante:

```
PS D:\Users\Luiggi_marquez\Desktop\Coder\Desarrollo Backend\Programacion Backend\Desafios\desafio_16> npm test

> desafio_8@1.0.0 test
> node benchmark.js

[2022-12-11T18:17:46.723] [INFO] consola - Runnig all benchmarks in parellel ...
Running 20s test @ http:localhost:8081/info
100 connections


| Stat | 2.5% | 50% | 97.5% | 99% | Avg | Stdev | Max |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Latency | 1045 ms | 6505 ms | 9821 ms | 9843 ms | 5936.1 ms | 2732.81 ms | 9890 ms |

| Stat | 1% | 2.5% | 50% | 97.5% | Avg | Stdev | Min |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Req/Sec | 99 | 99 | 99 | 101 | 99.72 | 0.8 | 99 |
| Bytes/Sec | 49.8 kB | 49.8 kB | 49.8 kB | 50.8 kB | 50.2 kB | 393 B | 49.8 kB |

Req/Bytes counts sampled once per second.
# of samples: 14

0 2xx responses, 1396 non 2xx responses
3k requests in 21.54s, 702 kB read
```

Profiling con info bloqueante:

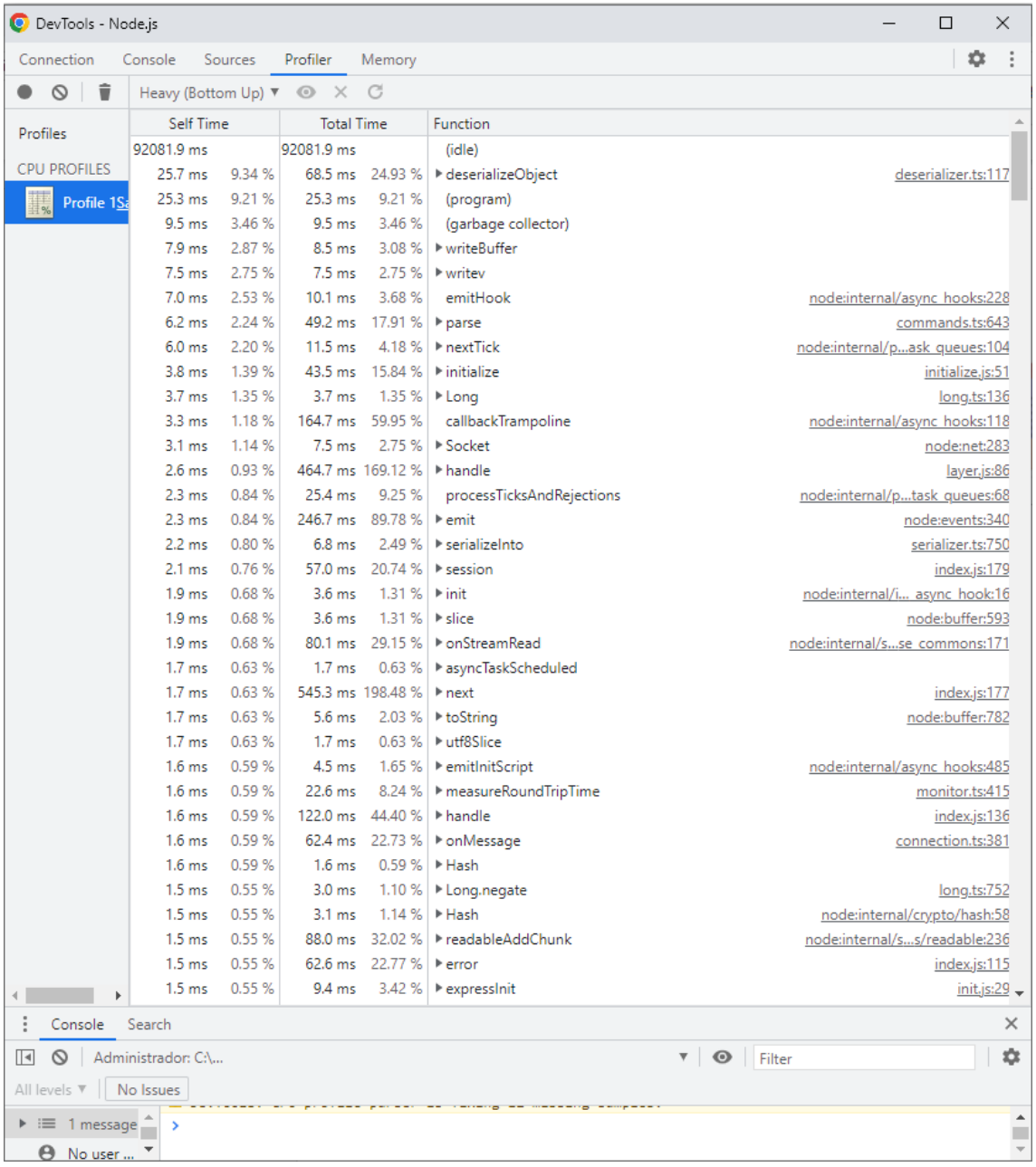
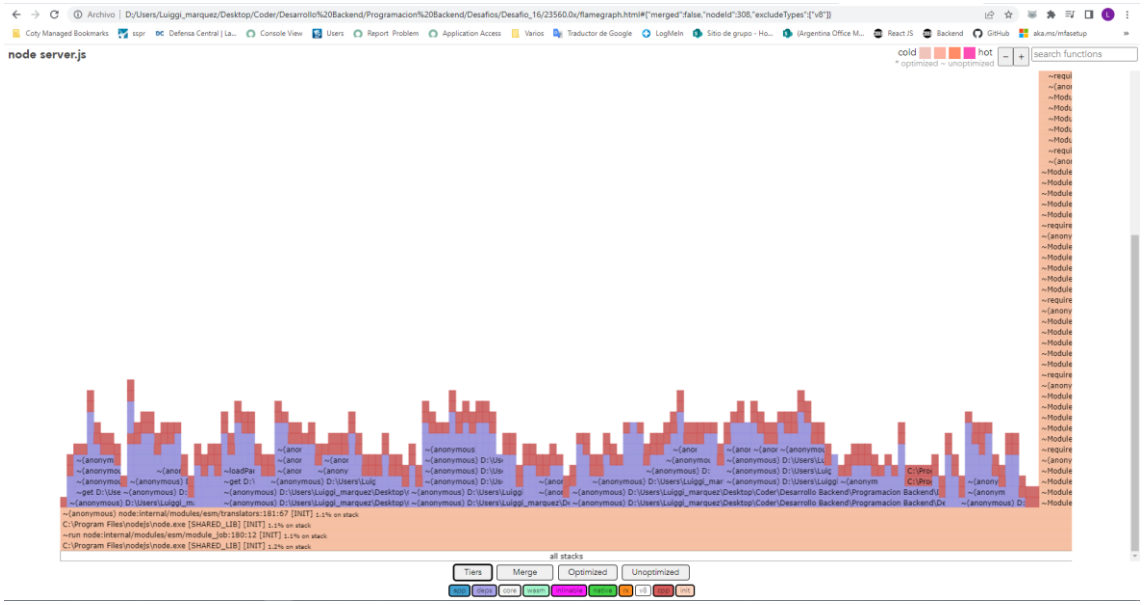
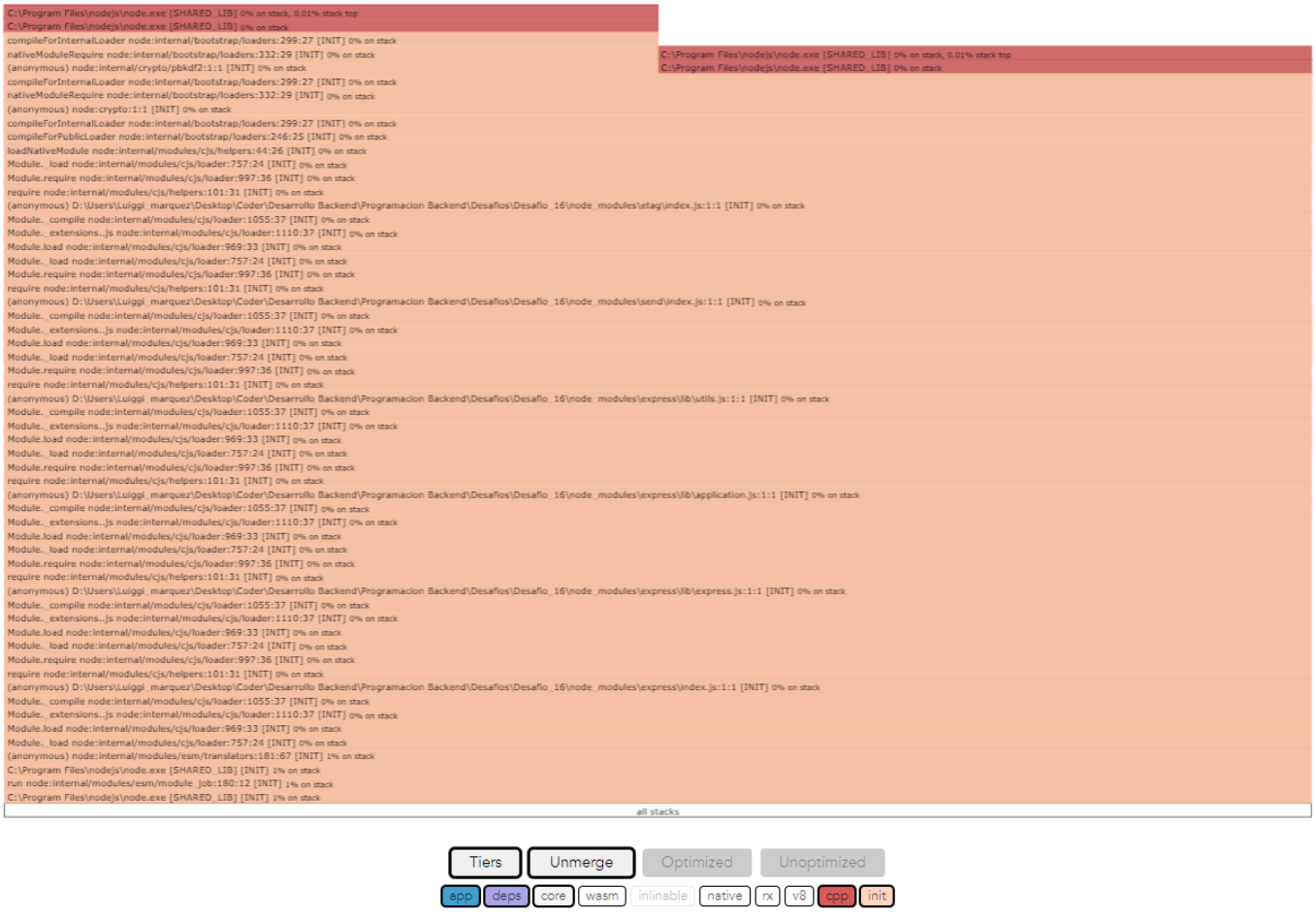


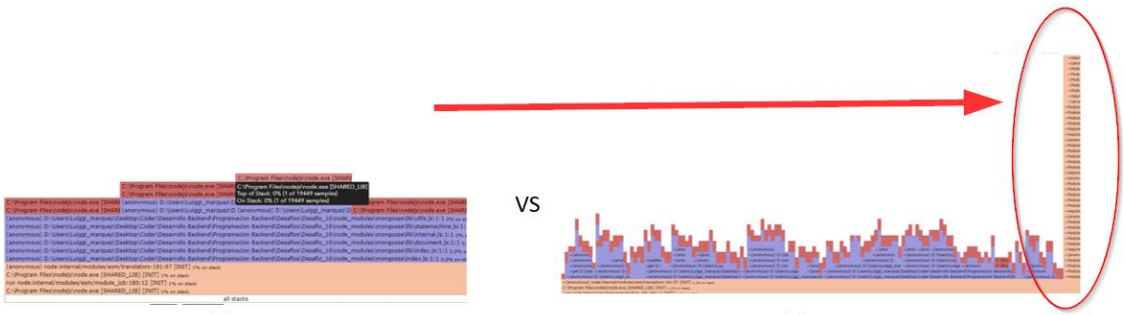
Grafico 0x con ruta /info bloqueante





En estos resultados obtenidos con test de carga realizados con autocannon, podemos analizar que en la ruta /info que actúa como bloqueante, la latencia de la prueba de 5800 ms contra 5900 ms de la prueba con la ruta no bloqueante. En el profiling podemos ver con mas detalle que tarda muchismo más tiempo en ejecutarse, ya que con el console.log tarda 35788ms y cuando solo tiene un console.log aumente a casi el triple, 92081ms.

Revisando los gráficos 0x, resalta este tema del tiempo de retraso generado por bloqueo, debido a que los valores superior corresponden al tiempo que tardan en completarse los ticks y en el caso del bloqueante se dispara y en comportamiento no es tan estable, varía mucho:



Ruta no bloqueante Ruta bloqueante

Recordando:

> *" los ticks son llamadas síncronas de funciones callbacks asociadas con eventos externos; Una vez que se vacía la cola y regresa la última función, un tick termina: se vuelve al principio (el siguiente tick) y se verifica los eventos que se agregaron a la cola desde otros subprocesos mientras se ejecutaba el código JavaScript"*

Así tenemos, en las pruebas del profiler realizadas con el autocannon se tiene:

Desafios > Desafio_16 > result_bloqueante.txt

```
1   Statistical profiling result from 32700.0x\bloqueante-v8.log, (19594 ticks, 0 unaccounted,
2
3    [Shared libraries]:
4      ticks  total  nonlib   name
5    19039   97.2%          C:\WINDOWS\SYSTEM32\ntdll.dll
6      535    2.7%          C:\Program Files\nodejs\node.exe
7        3    0.0%          C:\WINDOWS\System32\KERNELBASE.dll
8        3    0.0%          C:\WINDOWS\System32\KERNEL32.DLL
```

Desafios > Desafio_16 > result_no_bloqueante.txt

```
1   Statistical profiling result from 23560.0x\no_bloqueante-v8.log, (14241 ticks, 0 unaccounted
2
3    [Shared libraries]:
4      ticks  total  nonlib   name
5    13841   97.2%          C:\WINDOWS\SYSTEM32\ntdll.dll
6      388    2.7%          C:\Program Files\nodejs\node.exe
7        1    0.0%          C:\WINDOWS\System32\KERNEL32.DLL
8
```

Tenemos más cantidad de ticks que han sido completados en las pruebas hechas en la ruta bloqueante, con 19039 contra 13841 ticks en la no bloqueante. Esto se tranduce en que hay más tiempo de ejecución para la ruta que contiene el console.log, por lo que consume más recursos y tarda más en ejecutarse.

Se puede concluir que al incluir elementos síncronos en el desarrollo de la aplicación genera retrasos de funcionamientos importantes; si bien, las pruebas consistieron en solo un console.log (los demas fueron sustituidos en el desafio por la dependencia log4js) se observa una subida importante de milisegundos, hasta segundos en el tiempo de ejecución del programa. Si se considera que normalmente se colocan elementos bloqueantes sin control, en el caso de los anteriores desafíos muchos console.logs, llamada a funciones síncronas, y se le suma una concurrencia alta de usuarios en la práctica, el tiempo se ejecución pasa a ser preocupante porque podría elevarse a muchos segundos y lo importante es conservar recursos de procesamiento y ejecutar las funciones en el menor tiempo posible para darle a nuestro desarrollo funcionalidad, eficiencia y confort a los usuarios finales. Además que node.js no es multihilo, conviene combinar buenas practicas como usar clusters o trabajar con pm2 para hacer mejor uso del procesamiento, ya que los recursos de hardware no son infinitos y además muchas veces tienen costo adicional en las PaaS y evitar caer en uso de elementos que nos termine ralentizando nuestro desarrollo, como funciones síncronas, que termina afectándola experiencia del usuario.