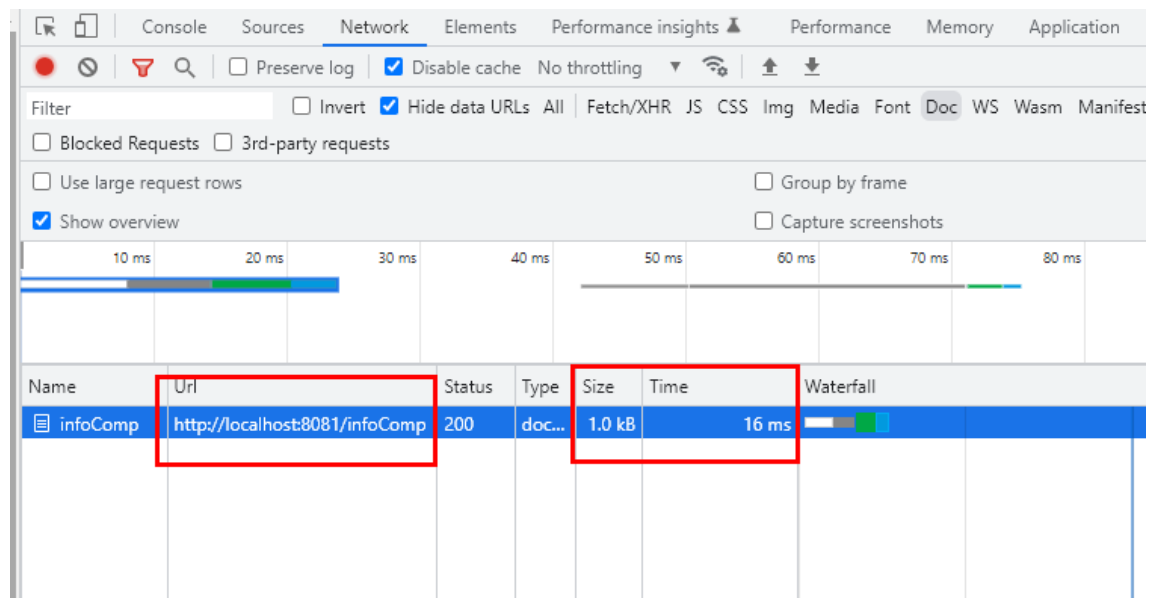


Desafio 16: Informe Análisis perfomance del Server

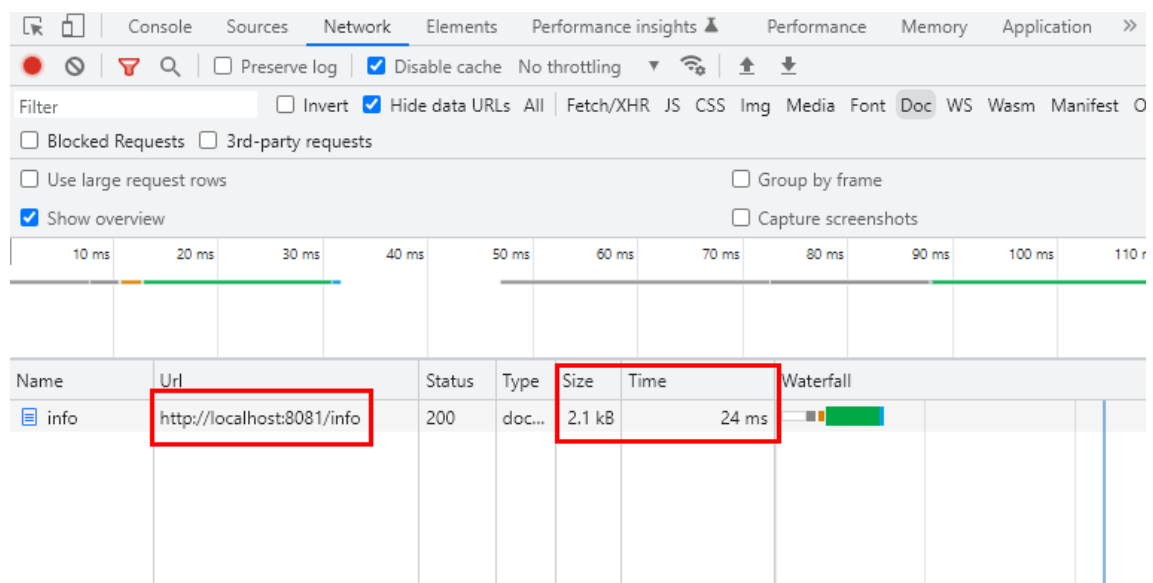
Luigi Marquez

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 el profiling se usó 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()

The screenshot displays the Chrome DevTools interface for Node.js. The top panel shows the Profiler tab with a 'Heavy (Bottom Up)' view. The 'Profiles' list on the left includes 'CPU PROFILES' and 'Profile 1'. The main table lists various functions and their execution times. The 'Self Time' column for the '(idle)' function is highlighted with a red circle, showing 64616.6 ms. The bottom panel shows the Console tab with a search bar and a list of messages. The messages include log entries for request received, subprocesos, and server status, along with a warning from DevTools about missing samples.

Function	Self Time	Total Time
(idle)	64616.6 ms	64616.6 ms
spawn	4436.7 ms	4436.7 ms
writeUtf8String	1652.3 ms	1652.3 ms
consoleCall	155.2 ms	1895.0 ms
normalizeSpawnArguments	150.3 ms	153.5 ms
(garbage collector)	85.8 ms	85.8 ms
(program)	41.1 ms	41.1 ms
init	21.0 ms	34.7 ms
nextTick	20.3 ms	78.9 ms
stat	18.5 ms	18.5 ms
deserializeObject	18.1 ms	43.6 ms
fork	14.5 ms	4634.8 ms
emitInitNative	14.5 ms	50.4 ms
writev	13.8 ms	13.8 ms
asyncTaskScheduled	13.7 ms	13.7 ms
open	12.4 ms	12.4 ms
isEmpty	10.6 ms	10.6 ms
afterWriteDispatched	8.3 ms	74.6 ms
initialize	7.8 ms	6158.1 ms
emitHook	7.8 ms	12.1 ms
spawn	7.1 ms	4619.9 ms

Console messages:

- [2022-12-10T22:59:13.269] [INFO] consola - Request Received: Route: /info Method: GET
- [2022-12-10T22:59:13.770] [INFO] consola - Fin subproceso
- [2022-12-10T22:59:13.822] [INFO] consola - Request Received: Route: /info Method: GET
- [2022-12-10T22:59:14.428] [INFO] consola - Fin subproceso
- [2022-12-10T22:59:14.447] [INFO] consola - Request Received: Route: /info Method: GET
- [2022-12-10T22:59:14.981] [INFO] consola - Fin subproceso
- [2022-12-10T22:59:14.998] [INFO] consola - Request Received: Route: /info Method: GET
- [2022-12-10T22:59:15.553] [INFO] consola - Fin subproceso
- DevTools: CPU profile parser is fixing 33 missing samples.
- [2022-12-10T23:02:03.260] [INFO] consola - Servidor escuchando al puerto 8081 - PID 31532
- [2022-12-10T23:02:03.984] [INFO] consola - Base de datos MongoDB conectada

/info sin el console.log

The screenshot shows the DevTools Node.js Profiler interface. The 'Profiler' tab is active, displaying a table of CPU profiles. The table has columns for 'Self Time', 'Total Time', and 'Function'. The 'Profile 1' is selected, and its details are shown in the main pane. The 'Console' tab is also visible, showing a list of messages and a warning about missing samples.

Function	Self Time	Total Time
(idle)	43371.1 ms	43371.1 ms
spawn	3544.1 ms	3544.1 ms
writeUtf8String	2268.1 ms	2268.1 ms
consoleCall	112.5 ms	2512.3 ms
normalizeSpawnArguments	80.8 ms	82.1 ms
(program)	40.9 ms	40.9 ms
nextTick	36.3 ms	56.3 ms
onwrite	36.1 ms	78.9 ms
afterWriteDispatched	33.8 ms	110.9 ms
deserializeObject	28.2 ms	57.8 ms
(garbage collector)	16.2 ms	16.2 ms
stat	11.9 ms	11.9 ms
emitInitNative	11.3 ms	18.1 ms
open	11.1 ms	11.1 ms
writv	10.6 ms	10.6 ms
spawn	9.7 ms	3651.8 ms
compile	9.3 ms	27.3 ms
removeListener	7.1 ms	7.1 ms
render	7.0 ms	133.1 ms
module.exports	6.0 ms	6.0 ms
initialize	6.0 ms	5793.9 ms

The console shows a list of messages and a warning: "DevTools: CPU profile parser is fixing 33 missing samples."

```
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
95.84 MB/sec, 43.34 req/sec, 49.8 kB/sec, 11.07s, 903 kB read

Perfilamiento con ruta /info no bloqueante:

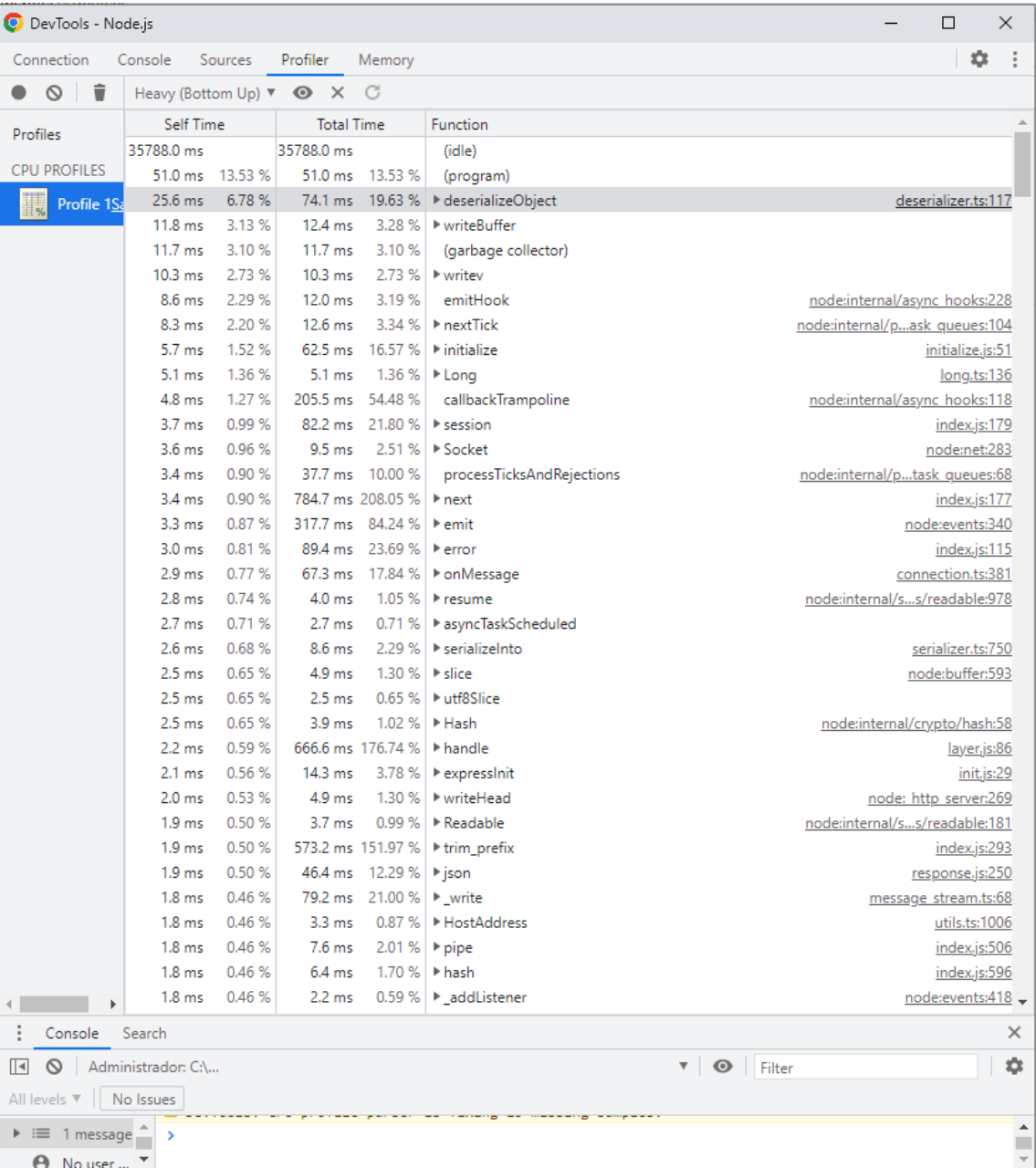
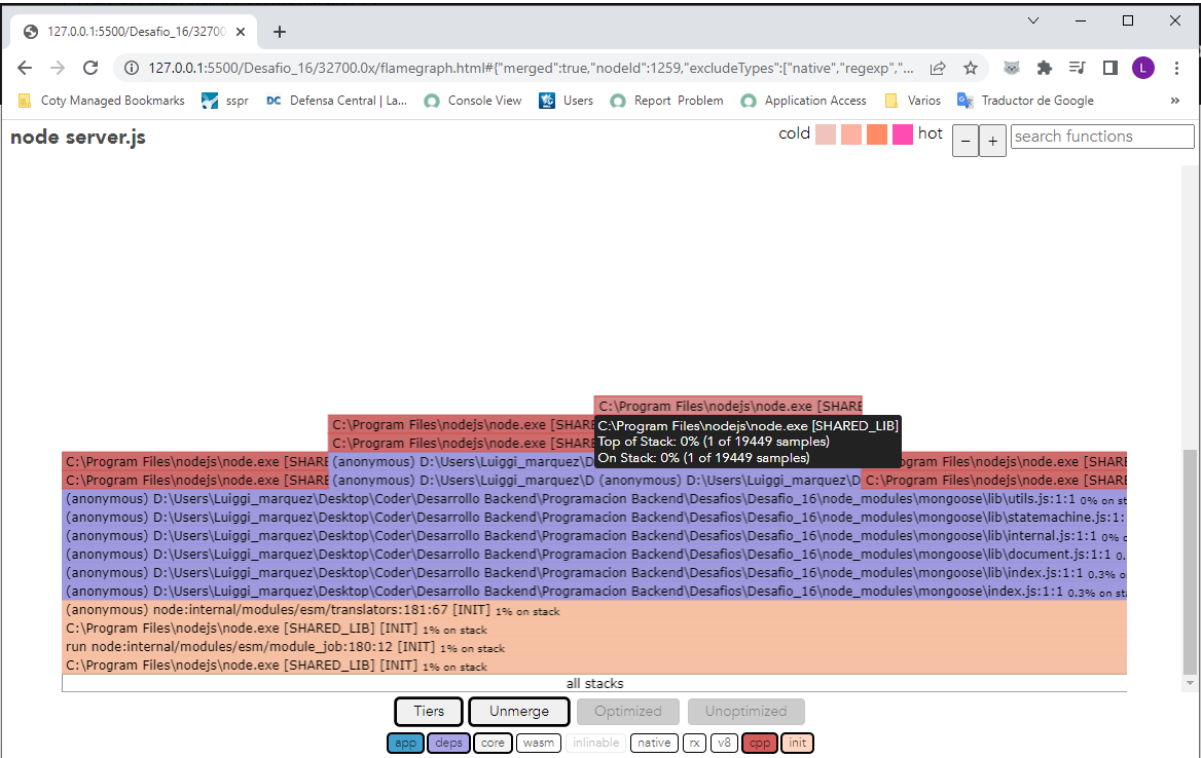
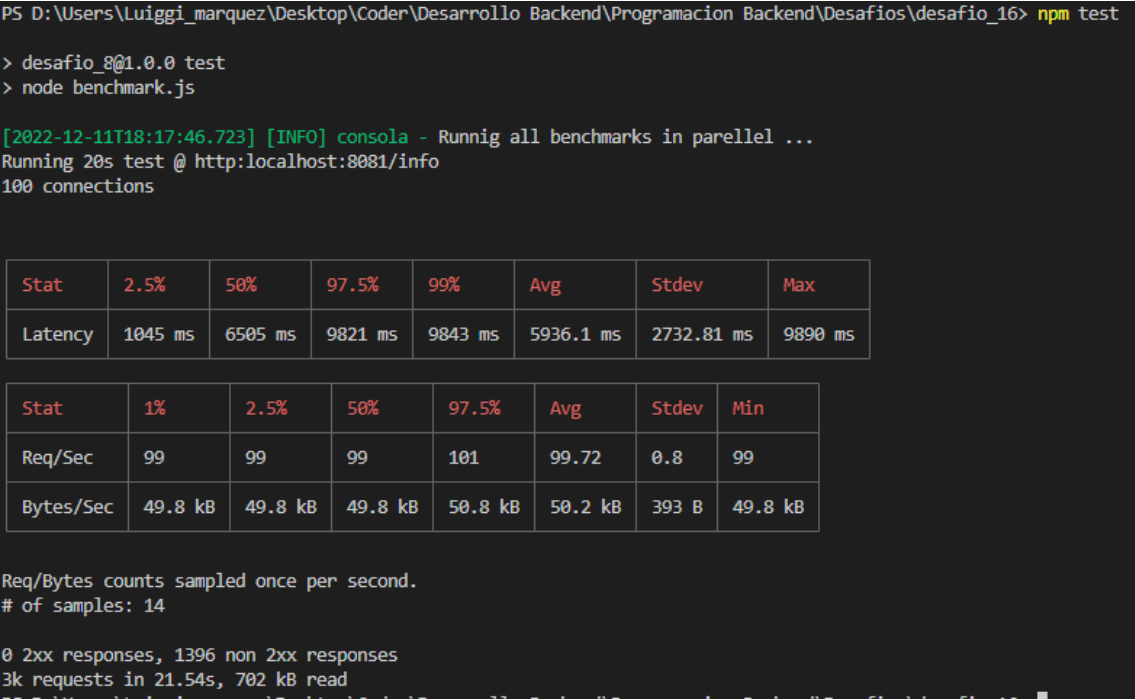


Grafico 0x con ruta /info no bloqueante



Autocannon con ruta /info bloqueante:



Profiling con info bloqueante:

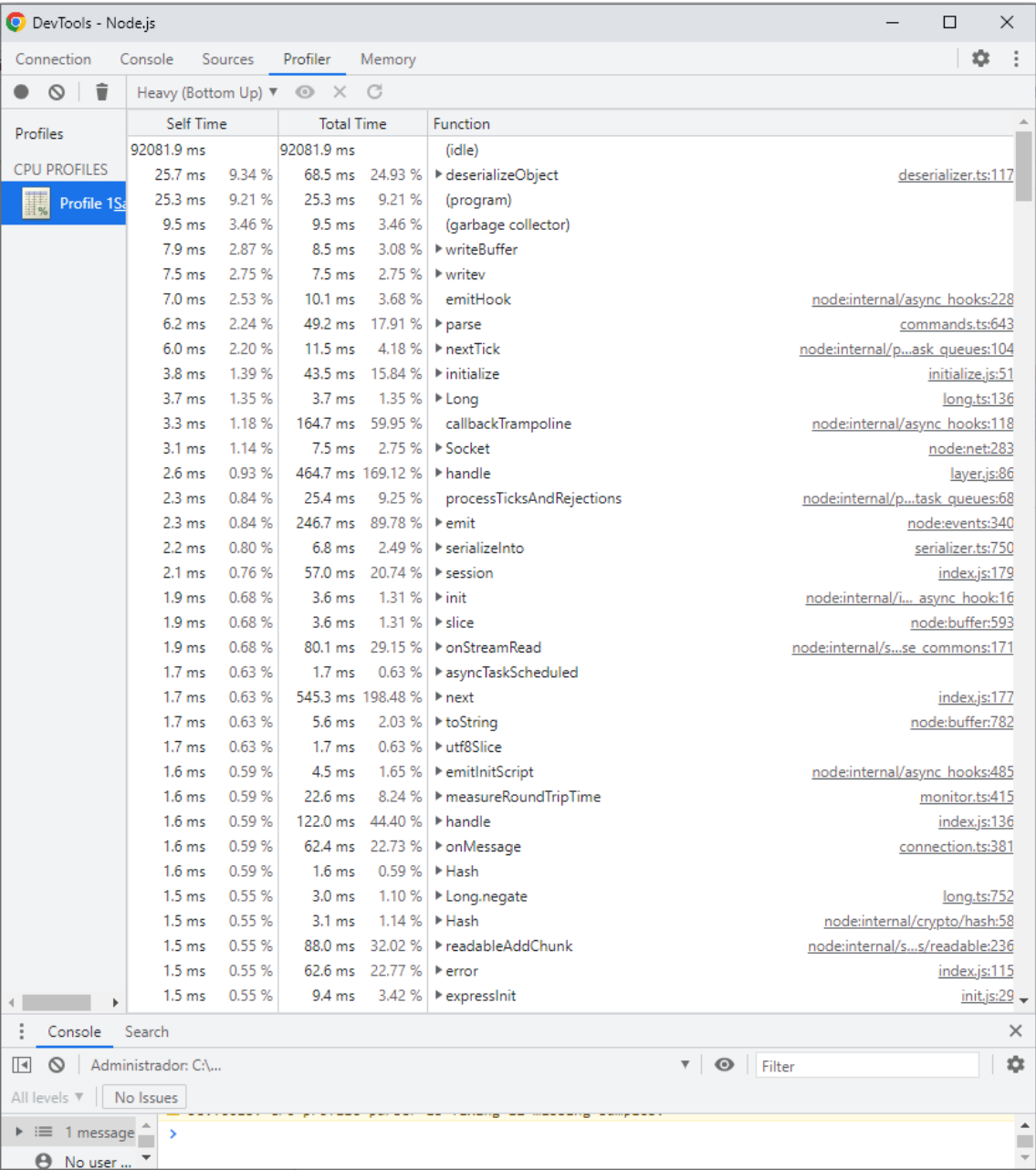
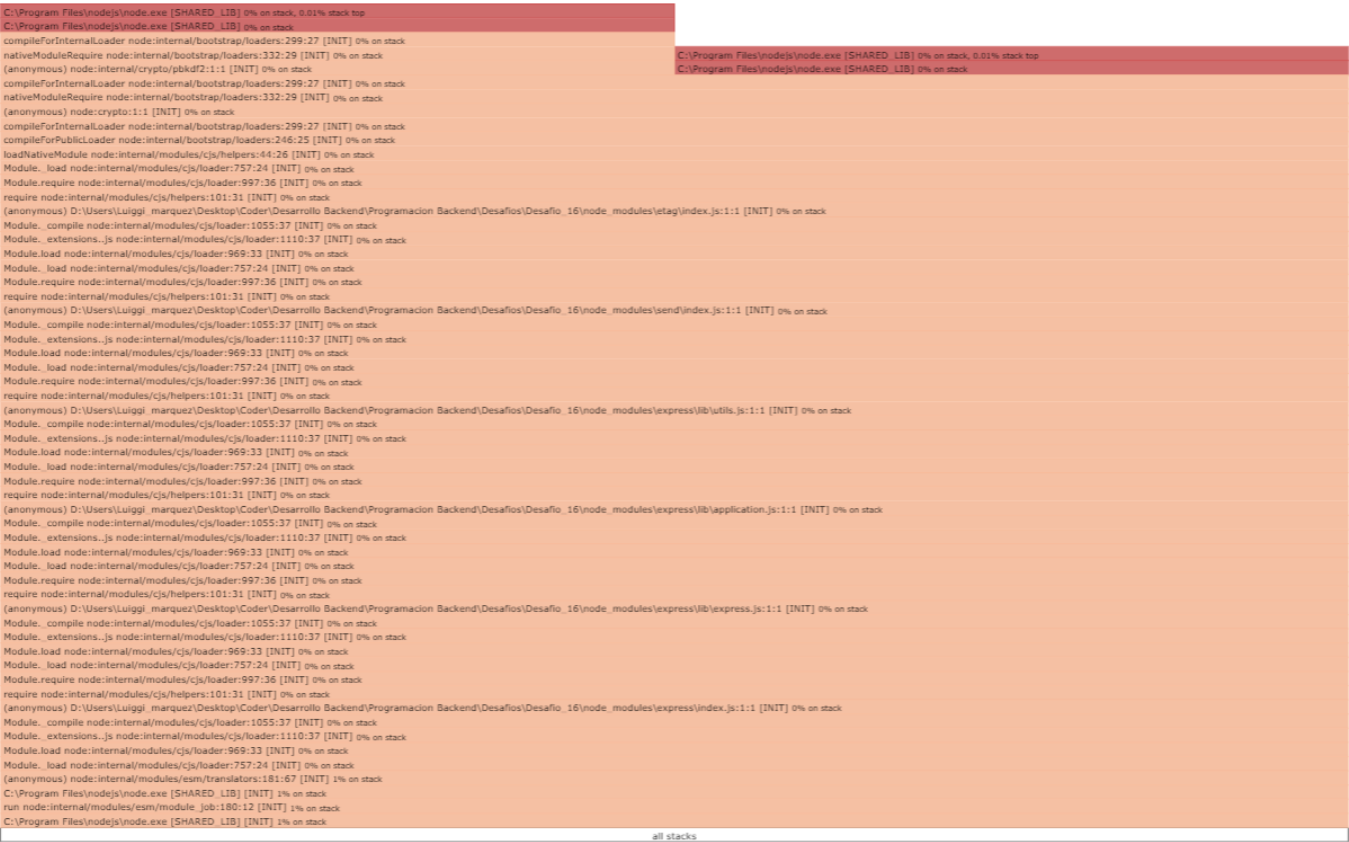
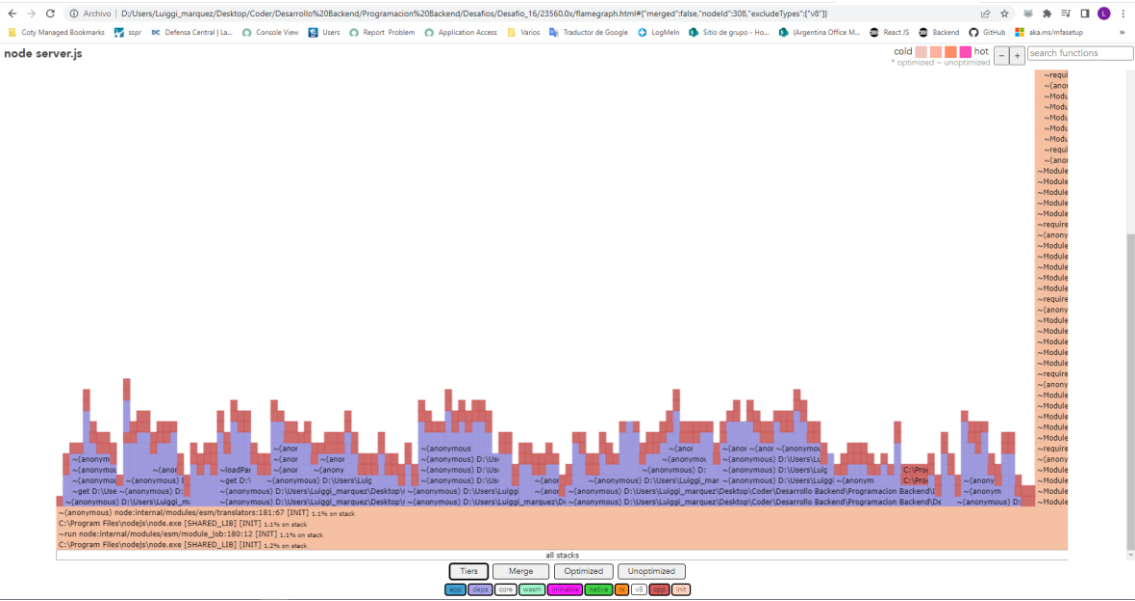


Grafico 0x con ruta /info bloqueante



Tiers

Unmerge

Optimized

Unoptimized

app

deps

core

wasm

inlinable

native

rx

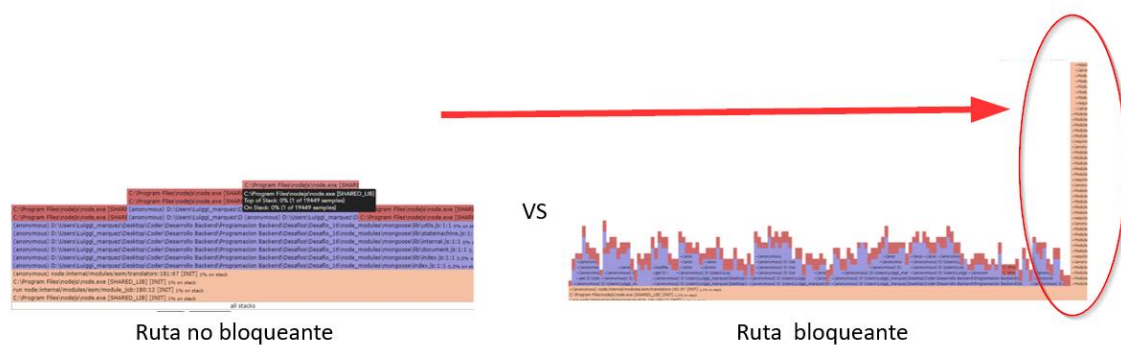
v8

cpp

init

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 muchísimo más tiempo en ejecutarse, ya que con el console.log tarda 35788ms y cuando solo tiene un console.log aumenta 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:



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 subprocessos 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
9

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 traduce 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 demás fueron sustituidos en el desafío 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 de 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.