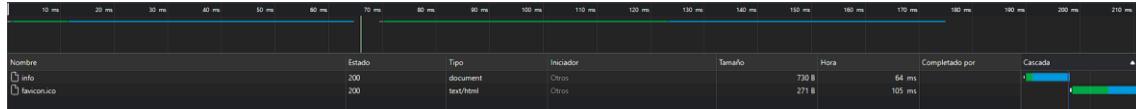


Solución:

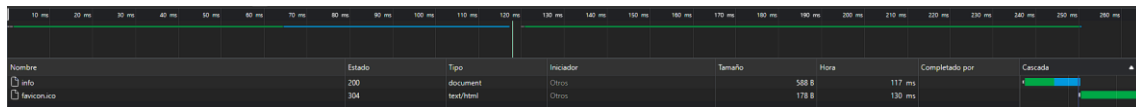
Verificar sobre la ruta “/info” con y sin compresión, la diferencia de cantidad de bytes devueltos en un caso y otro.

Sin compresión:



Nombre	Estado	Tipo	Iniciador	Tamaño	Hora	Completado por	Cascada
info	200	document	Otros	730 B	64 ms		
favicon.ico	200	text/html	Otros	271 B	105 ms		

Con compresión:



Nombre	Estado	Tipo	Iniciador	Tamaño	Hora	Completado por	Cascada
info	200	document	Otros	588 B	117 ms		
favicon.ico	304	text/html	Otros	178 B	130 ms		

El tamaño sin compresión es de 730B, mientras que el tamaño con compresión es de 588B. Hay un ahorro de 142B.

El perfilamiento del servidor, realizando el test con “--prof” de node.js. Analizar los resultados obtenidos luego de procesarlos con “--prof-process”.

Con console.log():

El test con “--prof” quedó:

```
All WUs finished. Total time: 19 seconds

-----
Summary report @ 18:34:27(-0300)
-----

http.codes.200: ..... 1000
http.request_rate: ..... 64/sec
http.requests: ..... 1000
http.response_time:
  min: ..... 2
  max: ..... 1977
  median: ..... 141.2
  p95: ..... 907
  p99: ..... 1085.9
http.responses: ..... 1000
vusers.completed: ..... 50
vusers.created: ..... 50
vusers.created_by_name.0: ..... 50
vusers.failed: ..... 0
vusers.session_length:
  min: ..... 15545.6
  max: ..... 18008.9
  median: ..... 16819.2
  p95: ..... 17505.6
  p99: ..... 17859.2
```

Respondiendo 64 request por segundo. Teniendo una media de 141.2 requests.

Mientras que la prueba con “--prof-process” quedó:

[Summary]:

ticks	total	nonlib	name
18	0.4%	100.0%	JavaScript
0	0.0%	0.0%	C++
14	0.3%	77.8%	GC
4570	99.6%		Shared libraries

```
ticks parent name
4241 92.4% C:\WINDOWS\SYSTEM32\ntdll.dll

328 7.1% C:\Program Files\nodejs\node.exe
226 68.9% C:\Program Files\nodejs\node.exe
77 34.1% Function: ^handleWriteReq node:internal/stream_base_commons:45:24
54 70.1% LazyCompile: *writeOrBuffer node:internal/streams/writable:367:23
39 72.2% LazyCompile: *Writable.write node:internal/streams/writable:335:36
35 89.7% Function: ^value node:internal/console/constructor:271:20
4 10.3% Function: ^ondata node:internal/streams/readable:752:18
13 24.1% LazyCompile: *_write node:internal/streams/writable:285:16
13 100.0% Function: ^Writable.write node:internal/streams/writable:335:36
2 3.7% Function: ^_write node:internal/streams/writable:285:16
2 100.0% Function: ^Writable.write node:internal/streams/writable:335:36
23 29.9% Function: ^writeGeneric node:internal/stream_base_commons:147:22
23 100.0% Function: ^Socket._writeGeneric node:net:848:42
23 100.0% Function: ^Socket._write node:net:885:35
20 8.8% Function: ^compileFunction node:vm:316:25
20 100.0% Function: ^wrapSafe node:internal/modules/cjs/loader:1040:18
20 100.0% Function: ^Module._compile node:internal/modules/cjs/loader:1080:37
19 95.0% Function: ^Module._extensions..js node:internal/modules/cjs/loader:1135:37
1 5.0% LazyCompile: ^Module._extensions..js node:internal/modules/cjs/loader:1135:37
17 7.5% Function: ^stat node:internal/modules/cjs/loader:151:14
9 52.9% Function: ^Module._findPath node:internal/modules/cjs/loader:505:28
```

Teniendo un total de 4570 ticks. Donde la mayoría lo hace “ntdll.dll”.

Sin console.log():

El test con “--prof” quedó:

All VUs finished. Total time: 19 seconds

Summary report @ 18:33:12(-0300)

```
http.codes.200: ..... 1000
http.request_rate: ..... 55/sec
http.requests: ..... 1000
http.response_time:
  min: ..... 1
  max: ..... 1936
  median: ..... 206.5
  p95: ..... 907
  p99: ..... 1107.9
http.responses: ..... 1000
vusers.completed: ..... 50
vusers.created: ..... 50
vusers.created_by_name.0: ..... 50
vusers.failed: ..... 0
vusers.session_length:
  min: ..... 15808.2
  max: ..... 17802.4
  median: ..... 16819.2
  p95: ..... 17505.6
  p99: ..... 17859.2
```

Respondiendo 55 request por segundo. Teniendo una media de 206.5 requests.

Mientras que la prueba con "--prof-process" quedó:

```
[Summary]:
```

ticks	total	nonlib	name
7	0.2%	100.0%	JavaScript
0	0.0%	0.0%	C++
13	0.3%	185.7%	GC
3954	99.8%		Shared libraries

ticks	parent	name
3732	94.2%	C:\WINDOWS\SYSTEM32\ntdll.dll
222	5.6%	C:\Program Files\nodejs\node.exe
140	63.1%	C:\Program Files\nodejs\node.exe
33	23.6%	Function: ^compileFunction node:vm:316:25
32	97.0%	Function: ^wrapSafe node:internal/modules/cjs/loader:1040:18
32	100.0%	Function: ^Module._compile node:internal/modules/cjs/loader:1080:37
32	100.0%	Function: ^Module._extensions..js node:internal/modules/cjs/loader:1135:37
1	3.0%	LazyCompile: ~wrapSafe node:internal/modules/cjs/loader:1040:18
1	100.0%	LazyCompile: ~Module._compile node:internal/modules/cjs/loader:1080:37
1	100.0%	LazyCompile: ~Module._extensions..js node:internal/modules/cjs/loader:1135:37
17	12.1%	Function: ^moduleStrategy node:internal/modules/esm/translators:114:56
17	100.0%	Function: ^moduleProvider node:internal/modules/esm/loader:459:28
17	100.0%	C:\Program Files\nodejs\node.exe
9	6.4%	Function: ^stat node:internal/modules/cjs/loader:151:14
5	55.6%	Function: ^tryFile node:internal/modules/cjs/loader:395:17
5	100.0%	Function: ^tryExtensions node:internal/modules/cjs/loader:411:23
5	100.0%	Function: ^Module._findPath node:internal/modules/cjs/loader:505:28
4	44.4%	Function: ^Module._findPath node:internal/modules/cjs/loader:505:28
4	100.0%	Function: ^Module._resolveFilename node:internal/modules/cjs/loader:865:35
4	100.0%	Function: ^Module._load node:internal/modules/cjs/loader:771:24
6	4.3%	Function: ^compileForInternalLoader node:internal/bootstrap/loaders:316:27

Teniendo un total de 3954 ticks. Donde la mayoría también lo hace "ntdll.dll".

Con console.log() hace 616 ticks más.

Utilizando Autocannon emulando 100 conexiones con un tiempo de 20 segundos.

Con console.log():

```
PS D:\Cursos\BackEnd\Desafios\Desafio-15-16> autocannon -c 100 -d 20 "http://localhost:8080/info"
Running 20s test @ http://localhost:8080/info
100 connections
```

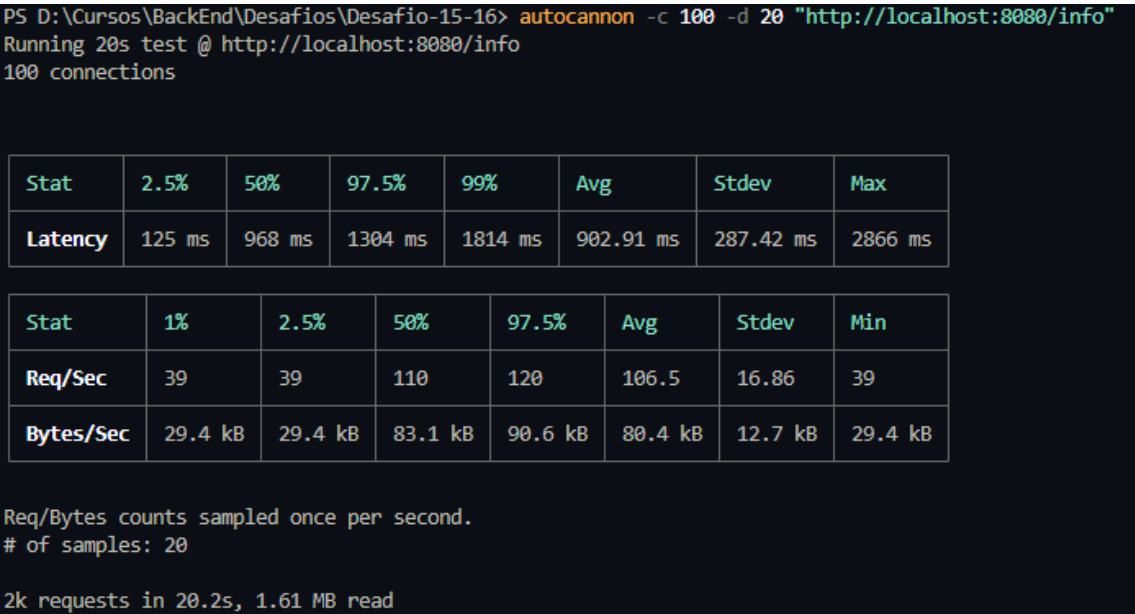
Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	144 ms	968 ms	1216 ms	1838 ms	900.11 ms	274.74 ms	2053 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	50	50	110	120	106.95	14.76	50
Bytes/Sec	37.7 kB	37.7 kB	82.9 kB	90.4 kB	80.6 kB	11.1 kB	37.7 kB

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

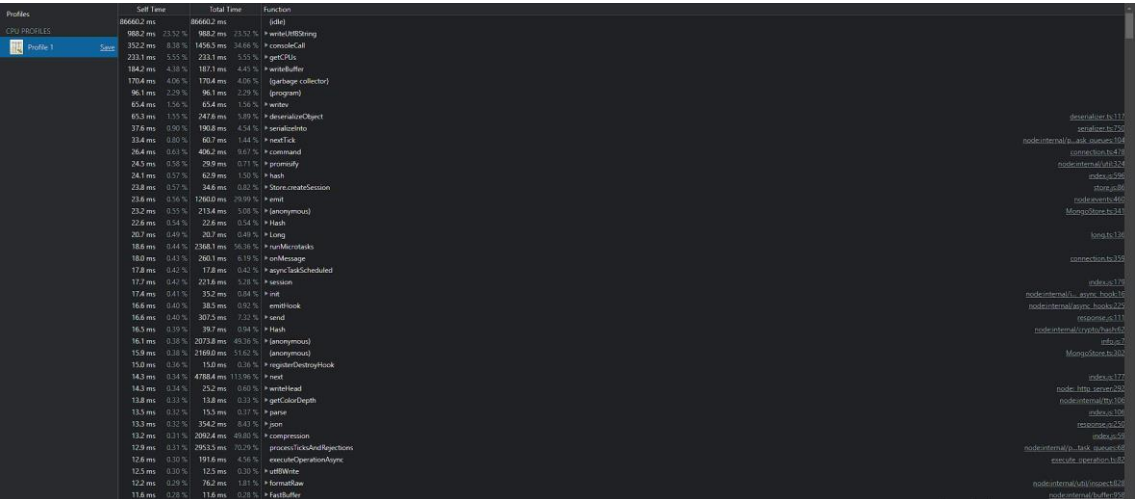
2k requests in 20.19s, 1.61 MB read

Sin console.log():



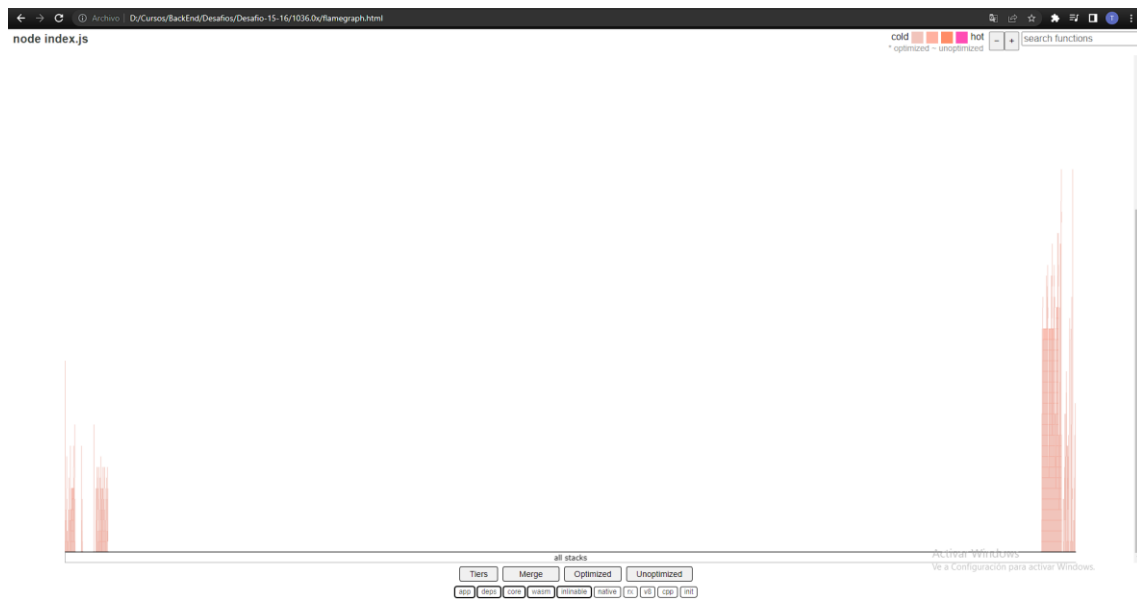
Podemos notar que sin console.log() tenemos menos latencia. Siendo en la máxima una diferencia de 813 ms.

El perfilamiento del servidor con el modo inspector de node.js –inspect queda así:



Con respecto al diagrama de flama con 0x.

Con console.log() el diagrama queda así:



Sin console.log() el diagrama queda así:

