Analisis de performance del servidor

Para el análisis se usó una ruta(“/info”) la cual tiene en uno de los dos casos un console.log el cual muestra el response antes de enviarla. Para este análisis se usaron las tecnologías Artillery, 0x, autocannon y las herramientas de node --prof e --inspect.

1. Análisis con --prof:

Se ejecutó el servidor y se procesó la información recogida

```
Statistical profiling result from .\isolate-0000014F584DA8B0-2032-v8.log, (623 ticks, 0 unaccounted, 0 excluded).

 [Shared libraries]:
   ticks  total  nonlib   name
    517   83.0%          C:\Windows\SYSTEM32\ntdll.dll
    102   16.4%          C:\Program Files\nodejs\node.exe

 [JavaScript]:
   ticks  total  nonlib   name
      2    0.3%   50.0%  LazyCompile: *resolve path.js:130:10
      1    0.2%   25.0%  LazyCompile: *hidden internal/errors.js:286:25
      1    0.2%   25.0%  LazyCompile: *dirname path.js:582:10

 [C++]:
   ticks  total  nonlib   name

 [Summary]:
   ticks  total  nonlib   name
      4    0.6%  100.0%  JavaScript
      0    0.0%    0.0%  C++
      2    0.3%   50.0%  GC
    619   99.4%          Shared libraries
```

Resultados sin console log

```
Statistical profiling result from .\isolate-0000020B88823EC0-4460-v8.log, (788 ticks, 0 unaccounted, 0 excluded).

 [Shared libraries]:
   ticks  total  nonlib   name
    683   86.7%          C:\Windows\SYSTEM32\ntdll.dll
     98   12.4%          C:\Program Files\nodejs\node.exe
      2    0.3%          C:\Windows\System32\KERNELBASE.dll

 [JavaScript]:
   ticks  total  nonlib   name
      3    0.4%   60.0%  LazyCompile: *resolve path.js:130:10
      2    0.3%   40.0%  LazyCompile: *dirname path.js:582:10

 [C++]:
   ticks  total  nonlib   name

 [Summary]:
   ticks  total  nonlib   name
      5    0.6%  100.0%  JavaScript
      0    0.0%    0.0%  C++
      4    0.5%   80.0%  GC
    783   99.4%          Shared libraries
```

Resultados con console log

Se puede observar que con el console log el servidor tiene más ticks

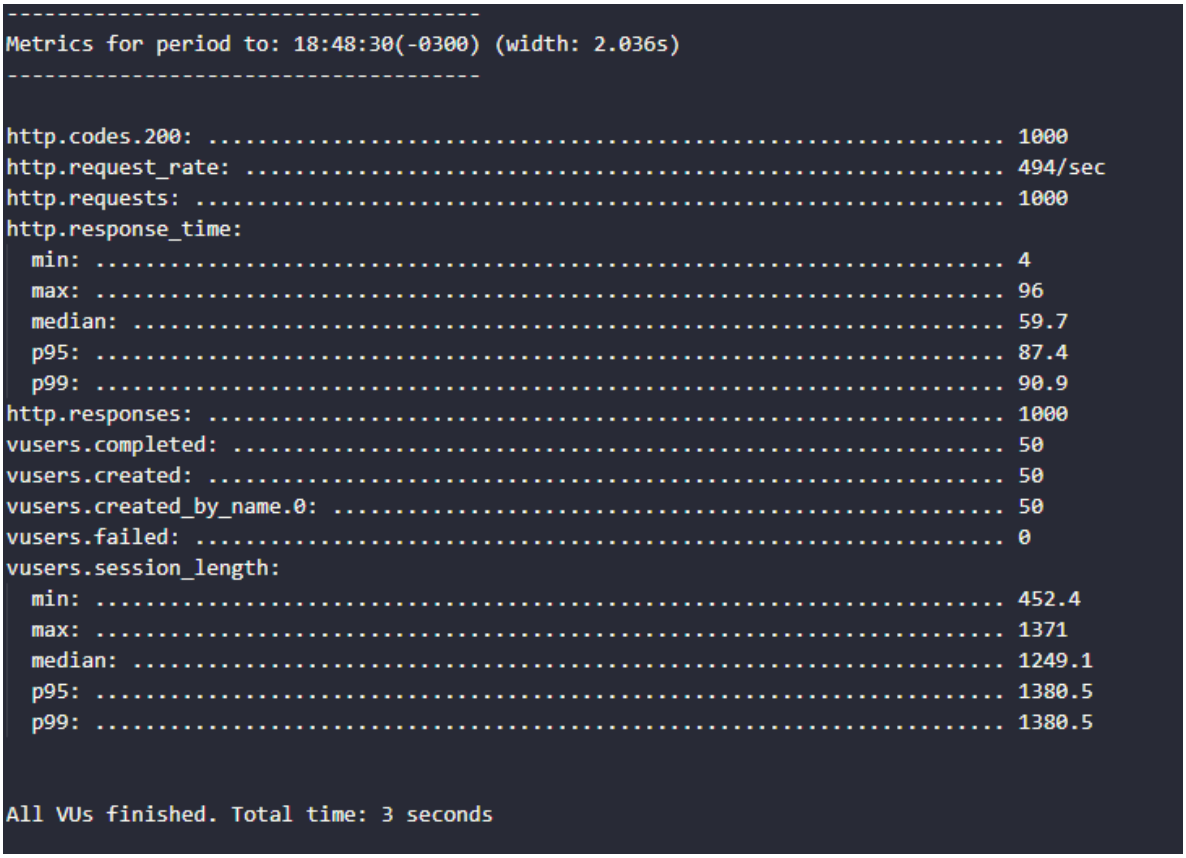
2. Artillery:

Se ejecutó emulando 50 conexiones concurrentes con 20 request por cada una

```
http.codes.200: ..................................................................... 1000
http.request_rate: .................................................................. 890/sec
http.requests: ...................................................................... 1000
http.response_time:
  min: .............................................................................. 1
  max: .............................................................................. 23
  median: ........................................................................... 13.9
  p95: .............................................................................. 19.1
  p99: .............................................................................. 21.1
http.responses: ..................................................................... 1000
vusers.completed: ................................................................... 50
vusers.created: ..................................................................... 50
vusers.created_by_name.0: ........................................................... 50
vusers.failed: ...................................................................... 0
vusers.session_length:
  min: .............................................................................. 153.4
  max: .............................................................................. 337.6
  median: ........................................................................... 308
  p95: .............................................................................. 333.7
  p99: .............................................................................. 333.7


All VUs finished. Total time: 2 seconds
```

Resultados sin console log



Resultados con console log

Se puede apreciar que la media con console log se incrementa en tres veces y la brecha entre la mínima y la máxima es bastante grande

3. Autocannon:

Se configuró para realizar 100 conexiones concurrentes en un tiempo de 20 segundos

```
PS C:\Users\Cristian\Documents\Cursos coder\Backend\Desafios\Desafio-clase32> node .\benchmark.js
Running all benchmarks in parallel ...
Running 20s test @ http://localhost:8080/info
100 connections
```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	59 ms	65 ms	91 ms	110 ms	67.36 ms	9.85 ms	191 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	900	900	1527	1585	1474.55	151.8	900
Bytes/Sec	2.44 MB	2.44 MB	4.15 MB	4.3 MB	4 MB	412 kB	2.44 MB

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

30k requests in 20.06s, 80 MB read
```

Resultados sin console log

```
PS C:\Users\Cristian\Documents\Cursos coder\Backend\Desafios\Desafio-clase32> node .\benchmark.js
Running all benchmarks in parallel ...
Running 20s test @ http://localhost:8080/info
100 connections
```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	115 ms	135 ms	175 ms	190 ms	137.56 ms	14.92 ms	233 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	500	500	731	789	725.35	60.42	500
Bytes/Sec	1.36 MB	1.36 MB	1.98 MB	2.14 MB	1.97 MB	164 kB	1.36 MB

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

15k requests in 20.13s, 39.4 MB read
```

Resultados con console log

Se aprecia como la latencia promedio es del doble con el console y la cantidad de request que se realizan es la mitad

4. Perfilamiento con --inspect:

Se usó el modo inspect de node para ver los procesos menos performantes en cada caso simulando conexiones con artillery.

Tiempo individual ▼		Tiempo total		Función	
19310.2 ms	93.84%	19310.2 ms	93.84%	(program)	
63.6 ms	0.31%	63.6 ms	0.31%	(garbage collector)	
51.2 ms	0.25%	51.2 ms	0.25%	▸ stat	
43.1 ms	0.21%	46.0 ms	0.22%	▸ next	parser.js:478
42.5 ms	0.21%	209.2 ms	1.02%	▸ SourceNode_walk	source-node.js:221
34.1 ms	0.17%	43.5 ms	0.21%	▸ SourceNode_add	source-node.js:172
32.8 ms	0.16%	32.8 ms	0.16%	▸ quotedString	code-gen.js:118
28.5 ms	0.14%	87.9 ms	0.43%	▸ createFunctionConte:	javascript-compiler.js:216
26.9 ms	0.13%	96.2 ms	0.47%	▸ parse	parser.js:269
20.7 ms	0.10%	20.7 ms	0.10%	▸ writev	
18.3 ms	0.09%	59.7 ms	0.29%	▸ setupHelperArgs	javascript-compiler.js:1104
17.4 ms	0.08%	97.5 ms	0.47%	▸ wrap	code-gen.js:101
15.1 ms	0.07%	19.5 ms	0.09%	▸ resolve	path.js:130
13.5 ms	0.07%	285.6 ms	1.39%	▸ compile	javascript-compiler.js:73
13.2 ms	0.06%	1128.2 ms	5.48%	▸ step	express-handlebars.js:31
12.0 ms	0.06%	226.5 ms	1.10%	▸ session	index.js:179
11.1 ms	0.05%	239.9 ms	1.17%	▸ cookieParser	index.js:44
10.9 ms	0.05%	852.4 ms	4.14%	▸ next	index.js:176
10.7 ms	0.05%	10.7 ms	0.05%	▸ extend	utils.js:28
9.2 ms	0.04%	29.4 ms	0.14%	▸ hash	index.js:596
8.7 ms	0.04%	15.0 ms	0.07%	▸ writeHead	http_server.js:250
8.6 ms	0.04%	35.6 ms	0.17%	▸ replaceStack	javascript-compiler.js:892
8.4 ms	0.04%	170.9 ms	0.83%	▸ send	response.js:107
8.4 ms	0.04%	9.1 ms	0.04%	▸ nextTick	internal/proces... queues.js:101
8.2 ms	0.04%	8.2 ms	0.04%	▸ FSReqCallback	
8.1 ms	0.04%	8.8 ms	0.04%	▸ template	runtime.js:52

Resultados sin console log

Tiempo individual		Tiempo total		Función	
12602.3 ms	84.08%	12602.3 ms	84.08%	(program)	
568.5 ms	3.79%	568.5 ms	3.79%	▸ writeUtf8String	
316.5 ms	2.11%	960.3 ms	6.41%	▾ consoleCall	
316.5 ms	2.11%	960.3 ms	6.41%	▸ (anónimas)	server.js:460
89.8 ms	0.60%	89.8 ms	0.60%	(garbage collector)	
54.1 ms	0.36%	266.0 ms	1.77%	▸ SourceNode_walk	source-node.js:221
51.2 ms	0.34%	51.2 ms	0.34%	▸ stat	
47.1 ms	0.31%	49.6 ms	0.33%	▸ next	parser.js:478
39.1 ms	0.26%	39.1 ms	0.26%	▸ quotedString	code-gen.js:118
37.9 ms	0.25%	47.1 ms	0.31%	▸ SourceNode_add	source-node.js:172
29.3 ms	0.20%	100.6 ms	0.67%	▸ parse	parser.js:269
27.0 ms	0.18%	99.0 ms	0.66%	▸ createFunctionContext	javascript-compiler.js:216
22.3 ms	0.15%	22.3 ms	0.15%	▸ writev	
21.0 ms	0.14%	112.3 ms	0.75%	▸ wrap	code-gen.js:101
17.1 ms	0.11%	309.9 ms	2.07%	▸ compile	javascript-compiler.js:73
16.6 ms	0.11%	59.8 ms	0.40%	▸ setupHelperArgs	javascript-compiler.js:1104
15.1 ms	0.10%	1245.3 ms	8.31%	▸ step	express-handlebars.js:31
14.2 ms	0.09%	1239.4 ms	8.27%	▸ cookieParser	index.js:44
13.6 ms	0.09%	16.8 ms	0.11%	▸ resolve	path.js:130
12.8 ms	0.09%	1221.8 ms	8.15%	▸ session	index.js:179
12.7 ms	0.08%	12.7 ms	0.08%	▸ extend	utils.js:28
12.1 ms	0.08%	3856.6 ms	25.73%	▸ next	index.js:176
11.6 ms	0.08%	12.3 ms	0.08%	▸ nextTick	internal/proces... queues.js:101
9.6 ms	0.06%	17.4 ms	0.12%	▸ writeHead	http_server.js:250
9.6 ms	0.06%	27.1 ms	0.18%	▸ setopts	common.js:48
9.4 ms	0.06%	183.6 ms	1.22%	▸ send	response.js:107

Resultados con console log

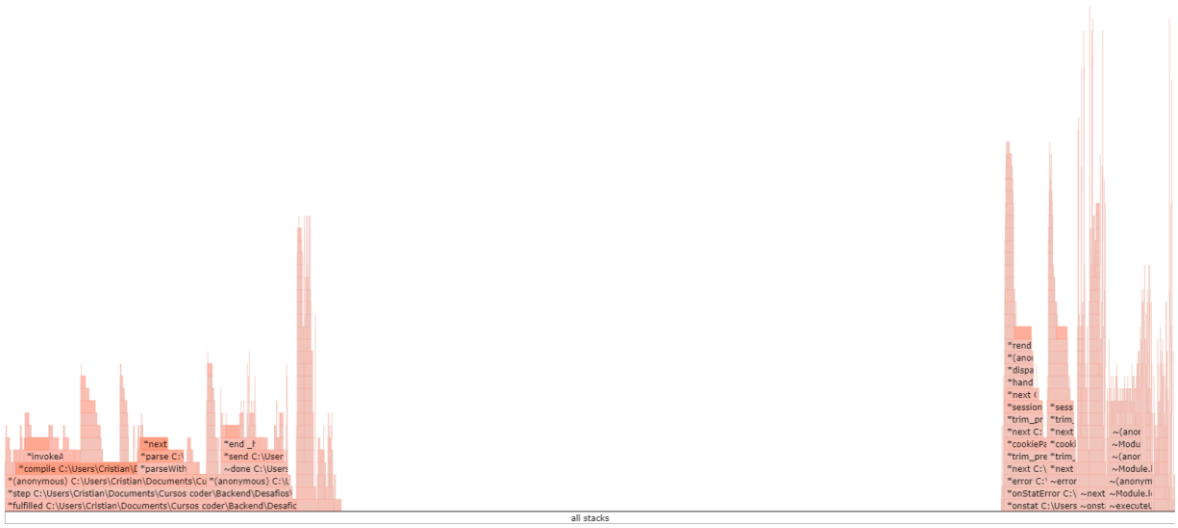
```
459
460                 app.get("/info", (req,res) =>{
461      0.5 ms         const datos = {
462      0.3 ms             argumento: process.argv.slice(2),
463      0.1 ms             plataforma: process.platform,
464                         version: process.version,
465      0.3 ms             memoria: process.memoryUsage().rss,
466      0.2 ms             id: process.pid,
467                         carpeta: process.cwd(),
468      0.3 ms             cpus: numCPUs
469                     }
470      4.2 ms         console.log(datos);
471      0.2 ms         res.render("info", datos)
472                 })
473
```

Tiempo de demora por ejecución

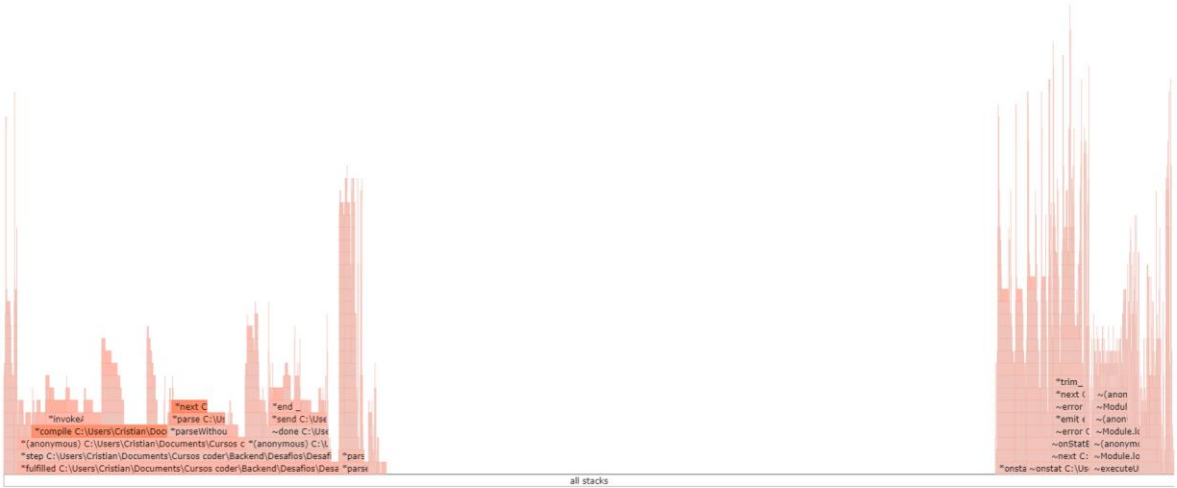
En el caso del código con console log se puede ver dos funciones que superan al resto, una para convertir en string y las llamadas de consola las cuales se pueden ver en la línea de código que tienen una alta demora de ejecución.

5. Diagrama de flama con 0x:

Se usó autocannon con las mismas configuraciones para realizar el grafico



Resultados sin console log



Resultados con console log

Se puede observar un incremento en la duración y el anidamiento en el caso con el console log, además de tener zonas más calientes que el caso sin el console log

Conclusión:

Se ve demostrado que el uso de herramientas de desarrollo en la etapa de producción no solo puede ser poco útiles, sino que también afectan considerablemente el funcionamiento de un servidor, sobre todo si el servidor no tiene mejoras de rendimiento como balanceo de carga o enrutamiento.