

# Informe de resultados de pruebas

## Programación Backend

Comisión 32080

Entrega 16

Alumno: Camilo González

Tutor: Raúl Ahumada

Las pruebas se harán modificando la ruta original en server.js:



```
server.js M x
server.js > app.get("/info") callback
357 });
358
359 app.get("/info", (req, res) => {
360   logger.info(`ruta '${req.url}' metodo '${req.method}'`);
361   try {
362     res.render('info.pug')
363   } catch (error) {
364     console.log(error)
365   }
366 })
367
```

Agregando console.log de la información que se muestra en la ruta:



```
server.js x
server.js > app.get("/info") callback
357 });
358
359 app.get("/info", (req, res) => {
360   logger.info(`ruta '${req.url}' metodo '${req.method}'`);
361   try {
362     console.log(process.argv);
363     console.log(process.platform);
364     console.log(process.version);
365     console.log(process.memoryUsage.rss());
366     console.log(process.execPath);
367     console.log(process.pid);
368     console.log(process.cwd());
369     res.render('info.pug')
370   } catch (error) {
371     console.log(error)
372   }
373 })
374
```

## 1. Perfilamiento del servidor con --prof de node.js

### Artillery

El servidor se inicia con el comando:

```
node --prof server.js
```

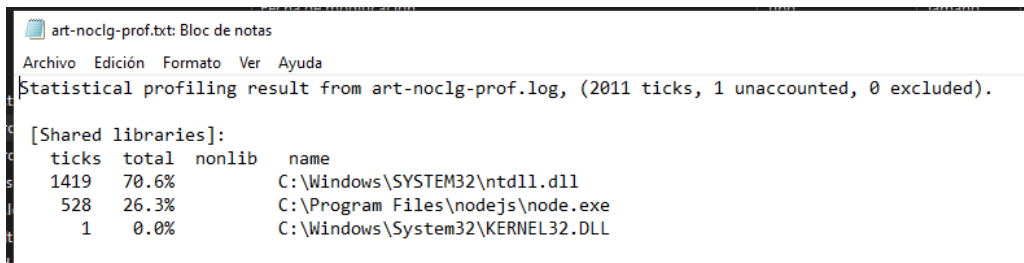
Artillery se inicia con el comando:

```
artillery quick --count 20 -n 50 "http://localhost:8080/info" > art-noclg-result.txt ó art-clg-result.txt
```

Luego de cada prueba se apaga el servidor y se procesan los logs generados por node.

### Logs procesados con --prof-process

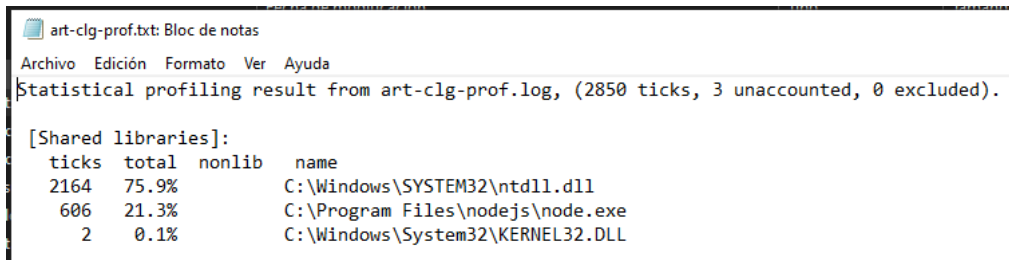
#### *Sin console.log*



```
art-noclg-prof.txt: Bloc de notas
Archivo Edición Formato Ver Ayuda
Statistical profiling result from art-noclg-prof.log, (2011 ticks, 1 unaccounted, 0 excluded).

[Shared libraries]:
ticks total nonlib name
1419 70.6% C:\Windows\SYSTEM32\ntdll.dll
528 26.3% C:\Program Files\nodejs\node.exe
1 0.0% C:\Windows\System32\KERNEL32.DLL
```

#### *Con console.log*



```
art-clg-prof.txt: Bloc de notas
Archivo Edición Formato Ver Ayuda
Statistical profiling result from art-clg-prof.log, (2850 ticks, 3 unaccounted, 0 excluded).

[Shared libraries]:
ticks total nonlib name
2164 75.9% C:\Windows\SYSTEM32\ntdll.dll
606 21.3% C:\Program Files\nodejs\node.exe
2 0.1% C:\Windows\System32\KERNEL32.DLL
```

El servidor que contiene console.log realiza mayor cantidad de ticks en los procesos.

## Logs generados por artillery

### *Sin console.log*

```
art-noclg-result.txt: Bloc de notas
Archivo Edición Formato Ver Ayuda
Running scenarios...
Phase started: unnamed (index: 0, duration: 1s) 18:28:06(-0300)

Phase completed: unnamed (index: 0, duration: 1s) 18:28:07(-0300)

All VUs finished. Total time: 10 seconds

-----
Summary report @ 18:28:13(-0300)
-----

http.codes.200: ..... 1000
http.request_rate: ..... 77/sec
http.requests: ..... 1000
http.response_time:
  min: ..... 7
  max: ..... 329
  median: ..... 117.9
  p95: ..... 144
  p99: ..... 162.4
http.responses: ..... 1000
vusers.completed: ..... 20
vusers.created: ..... 20
vusers.created_by_name.0: ..... 20
vusers.failed: ..... 0
vusers.session_length:
  min: ..... 5662.3
  max: ..... 6209.3
  median: ..... 5944.6
  p95: ..... 6187.2
  p99: ..... 6187.2
```

### *Con console.log*

```
art-clg-result.txt: Bloc de notas
Archivo Edición Formato Ver Ayuda
Running scenarios...
Phase started: unnamed (index: 0, duration: 1s) 18:26:11(-0300)

Phase completed: unnamed (index: 0, duration: 1s) 18:26:12(-0300)

All VUs finished. Total time: 15 seconds

-----
Summary report @ 18:26:23(-0300)
-----

http.codes.200: ..... 1000
http.request_rate: ..... 35/sec
http.requests: ..... 1000
http.response_time:
  min: ..... 7
  max: ..... 355
  median: ..... 162.4
  p95: ..... 186.8
  p99: ..... 242.3
http.responses: ..... 1000
vusers.completed: ..... 20
vusers.created: ..... 20
vusers.created_by_name.0: ..... 20
vusers.failed: ..... 0
vusers.session_length:
  min: ..... 7814.8
  max: ..... 8370.2
  median: ..... 8024.5
  p95: ..... 8352
  p99: ..... 8352
```

El servidor con console log recibe menor cantidad de pedidos por segundo por el retraso que generan los comandos. Esto también genera un mayor tiempo de respuesta.

## Autocannon

Autocannon se inicia con el comando:

```
autocannon -c 100 -d 20 "http://localhost:8080/info"
```

### Capturas de consola

#### *Sin console.log*

```
PS C:\Users\milog\Documents\32080> 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	467 ms	555 ms	729 ms	916 ms	560 ms	83.36 ms	1139 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	91	91	181	189	176.4	20.55	91
Bytes/Sec	592 kB	592 kB	1.18 MB	1.23 MB	1.15 MB	134 kB	592 kB

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

4k requests in 20.07s, 23 MB read

#### *Con console.log*

```
PS C:\Users\milog\Documents\32080> 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	575 ms	742 ms	969 ms	1142 ms	750.69 ms	96.35 ms	1330 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	63	63	135	139	131.15	16.39	63
Bytes/Sec	410 kB	410 kB	879 kB	905 kB	853 kB	107 kB	410 kB

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

3k requests in 20.1s, 17.1 MB read

El servidor con console log tiene mayor latencia y menores pedidos por segundo, por lo tiene menor transferencia de datos por segundo.

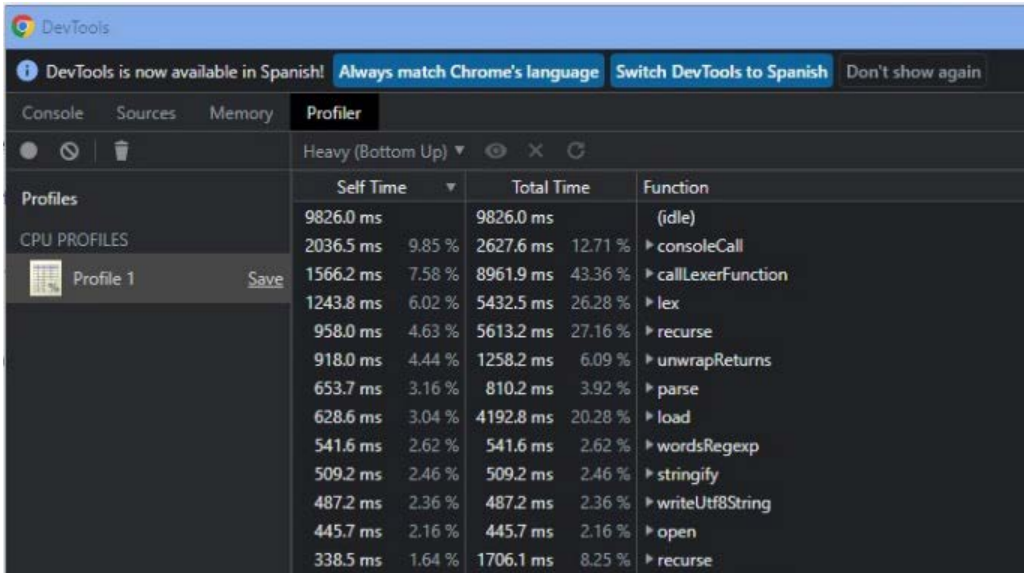
## 2. Perfilamiento del servidor con el modo inspector de node.js – inspect

El servidor se inicia con el comando:

```
node --inspect server.js
```

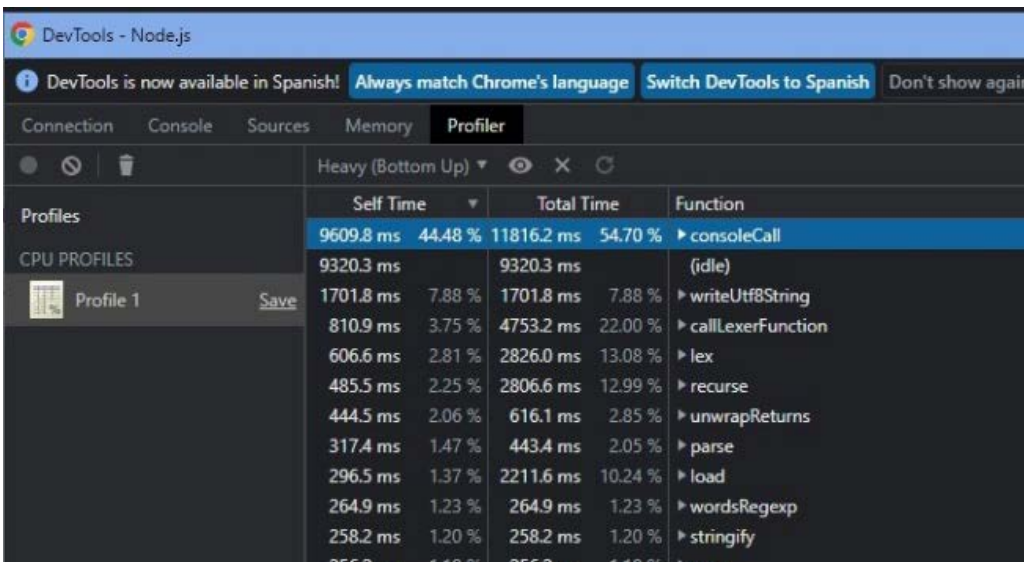
Capturas de consola

*sin console.log*



Profiles	Self Time	Total Time	Function
CPU PROFILES	9826.0 ms	9826.0 ms	(idle)
Profile 1	2036.5 ms 9.85 %	2627.6 ms 12.71 %	▸ consoleCall
	1566.2 ms 7.58 %	8961.9 ms 43.36 %	▸ callLexerFunction
	1243.8 ms 6.02 %	5432.5 ms 26.28 %	▸ lex
	958.0 ms 4.63 %	5613.2 ms 27.16 %	▸ recurse
	918.0 ms 4.44 %	1258.2 ms 6.09 %	▸ unwrapReturns
	653.7 ms 3.16 %	810.2 ms 3.92 %	▸ parse
	628.6 ms 3.04 %	4192.8 ms 20.28 %	▸ load
	541.6 ms 2.62 %	541.6 ms 2.62 %	▸ wordsRegexp
	509.2 ms 2.46 %	509.2 ms 2.46 %	▸ stringify
	487.2 ms 2.36 %	487.2 ms 2.36 %	▸ writeUtf8String
	445.7 ms 2.16 %	445.7 ms 2.16 %	▸ open
	338.5 ms 1.64 %	1706.1 ms 8.25 %	▸ recurse

*con console.log*



Profiles	Self Time	Total Time	Function
CPU PROFILES	9609.8 ms 44.48 %	11816.2 ms 54.70 %	▸ consoleCall
Profile 1	9320.3 ms	9320.3 ms	(idle)
	1701.8 ms 7.88 %	1701.8 ms 7.88 %	▸ writeUtf8String
	810.9 ms 3.75 %	4753.2 ms 22.00 %	▸ callLexerFunction
	606.6 ms 2.81 %	2826.0 ms 13.08 %	▸ lex
	485.5 ms 2.25 %	2806.6 ms 12.99 %	▸ recurse
	444.5 ms 2.06 %	616.1 ms 2.85 %	▸ unwrapReturns
	317.4 ms 1.47 %	443.4 ms 2.05 %	▸ parse
	296.5 ms 1.37 %	2211.6 ms 10.24 %	▸ load
	264.9 ms 1.23 %	264.9 ms 1.23 %	▸ wordsRegexp
	258.2 ms 1.20 %	258.2 ms 1.20 %	▸ stringify
	256.2 ms 1.10 %	256.2 ms 1.10 %	▸ open

En el servidor con console.log la función consoleCall cuadruplicó la cantidad de tiempo utilizado.

### 3. Diagrama de flama con 0x

El servidor se inicia con el comando:

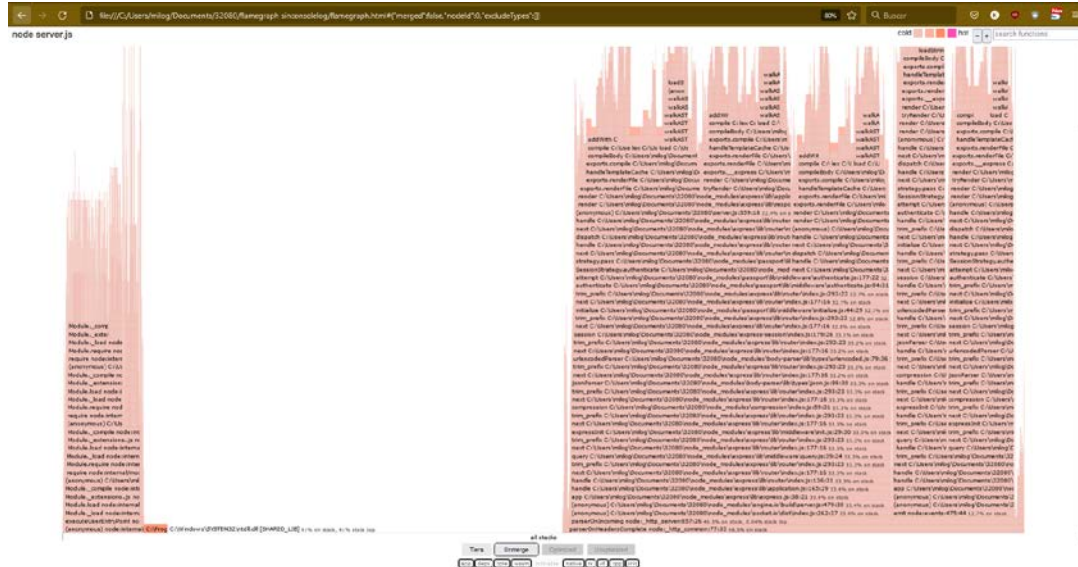
```
0x server.js
```

Utilizamos autocannon con el comando:

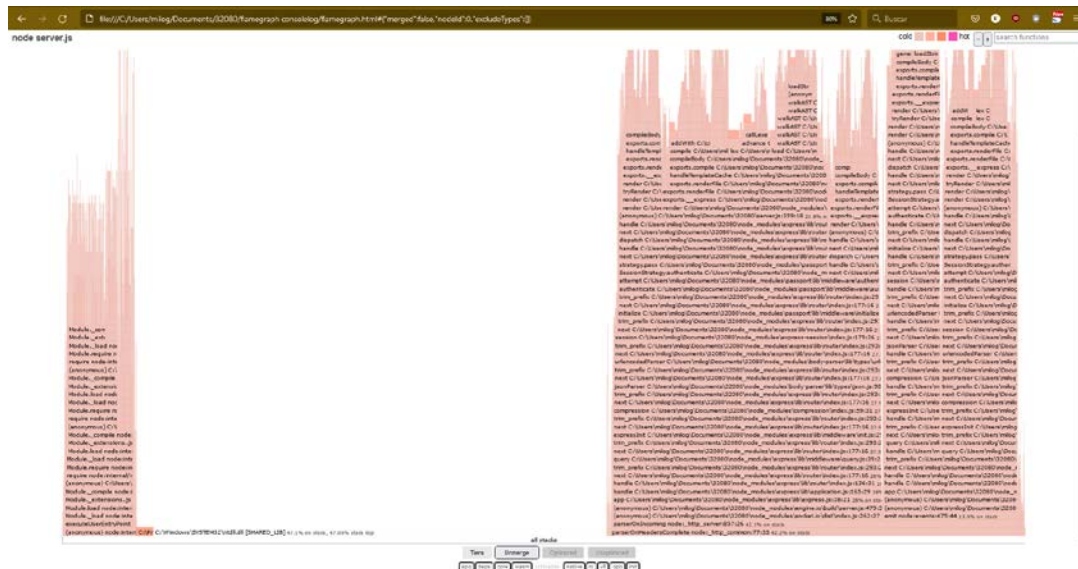
```
autocannon -c 100 -d 20 "http://localhost:8080/info"
```

Capturas de diagramas generados por 0x

sin console.log



con console.log



En el servidor con console.log hubo un aumento de un 6% en el tiempo del proceso node.

Todas las capturas se encuentran en la carpeta public/images

Todos los logs están en la carpeta principal

Las 2 carpetas generadas por 0x están en carpeta principal.