Published in The JS runtimes

Mayank Choubey Follow
Feb 12 · 4 min read · Listen

Node.js vs Springboot: Hello world performance comparison

Continuing my quest of performance comparisons, this article is about Node.js and Springboot. Spring boot is a light-weight server technology built over Java that's mostly used for web services. Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run". Spring boot is quite popular in the java world.

Who do you think is faster for a simple hello world case? I guess the answer could be Springboot because it's compiled into byte code while Node.js is interpreted. If you're interested in finding out the correct answer, read on. You might be surprised!

Setup

The test is executed on MacBook P [illegible] of RAM. The test is executed using the well-known HTTP tester: *Bomb*[illegible] is executed for 50, 100, and 200 concurrent connections.

The versions are:

- Node.js v19.6.0
- Springboot 3.0.2 with Java17

The code is:

Node.js

The Node.js code has a minimal router that makes it equivalent to the functionality provided by Spring's native server.

```
import http from "node:http";

http.createServer((req, resp) => {
  try {
    if (req.method !== "GET") {
      return resp.writeHead(405).end();
    }
    if (req.url !== "/") {
      return resp.writeHead(404).end();
    }
    resp.writeHead(200, {
      "content-type": "text/plain",
    });
    resp.end("Hello world");
  } catch (e) {
    resp.writeHead(500).end();
  }
}).listen(3000);
```

Spring

```
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.http.ResponseEntity;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.RestController;
```

```
@SpringBootApplication
@RestController
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }

    @GetMapping("/")
    public String handleRequest() {
        return "Hello World!";
    }
}
```

A note about Springboot app: For Spring Boot, I've only shown the main application file. This is the only file which has been written by me. By default, this springboot app starts with 60 threads.

A total of ***5M (5 million)*** requests are executed for each concurrency level.

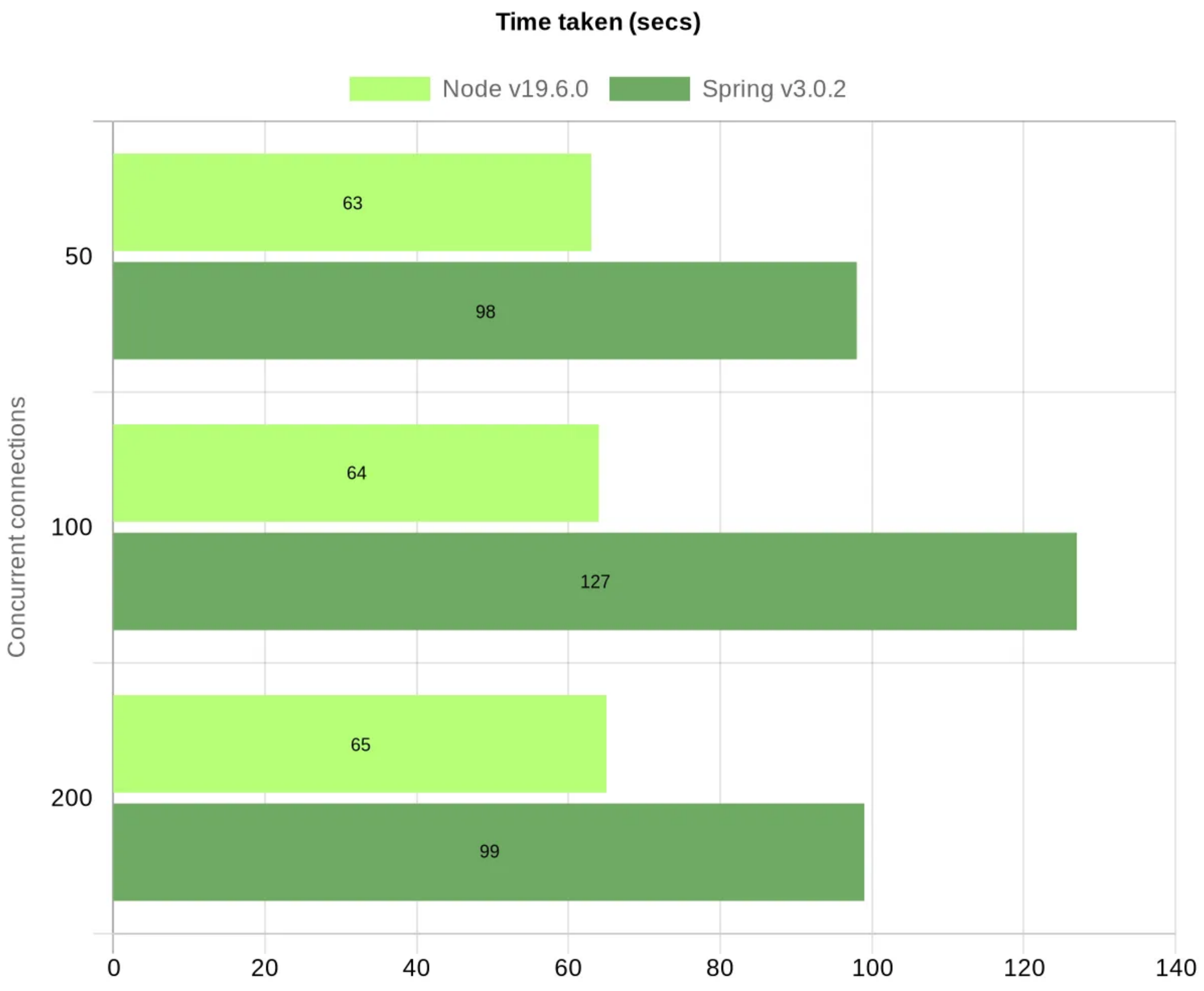
The following measurements are taken:

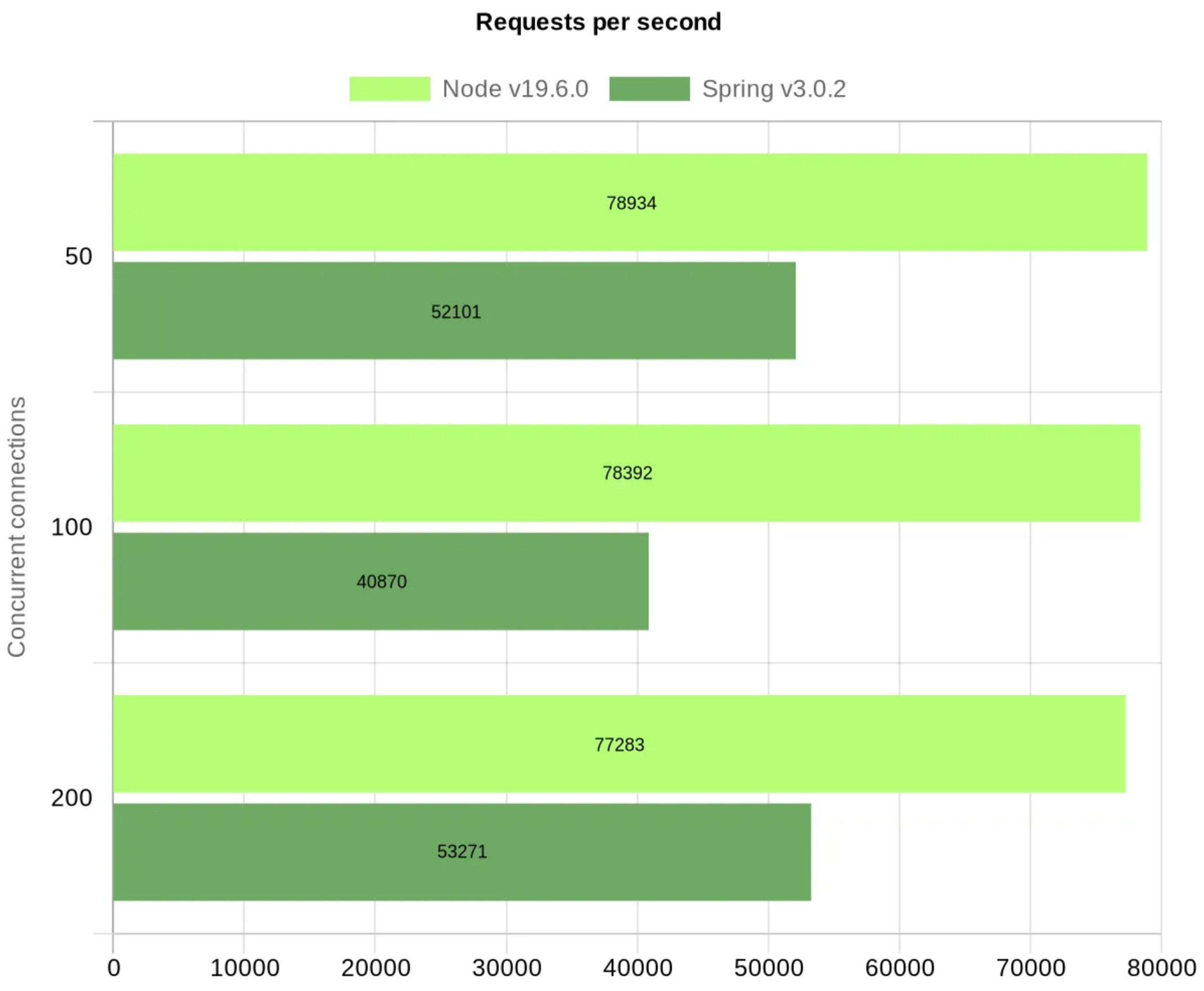
- Time taken
- Requests per second
- ***Latencies***: Mean, median, q25, q75, q90, maximum (in milliseconds)
- ***System usage***: Average CPU and memory usage

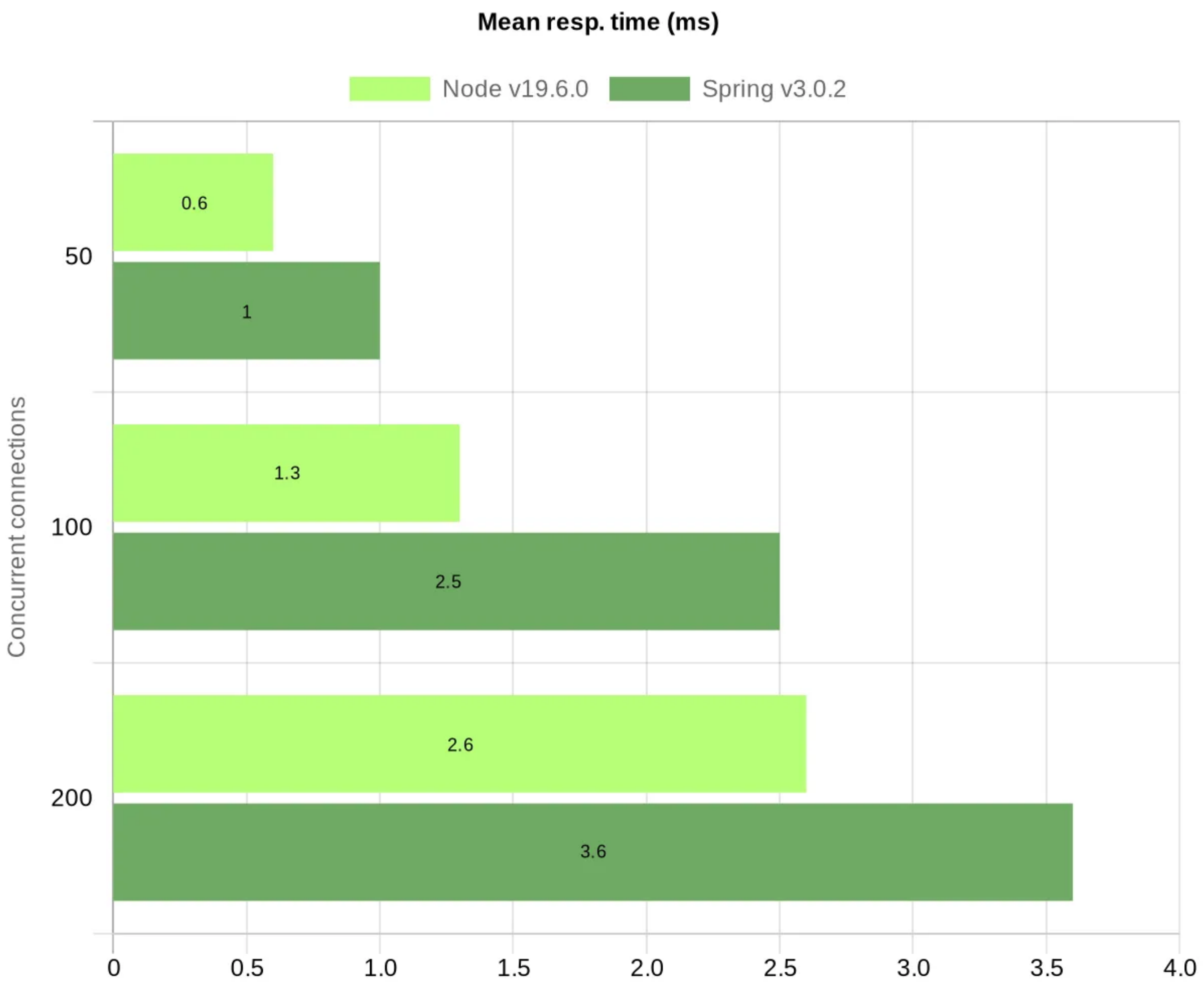
Results

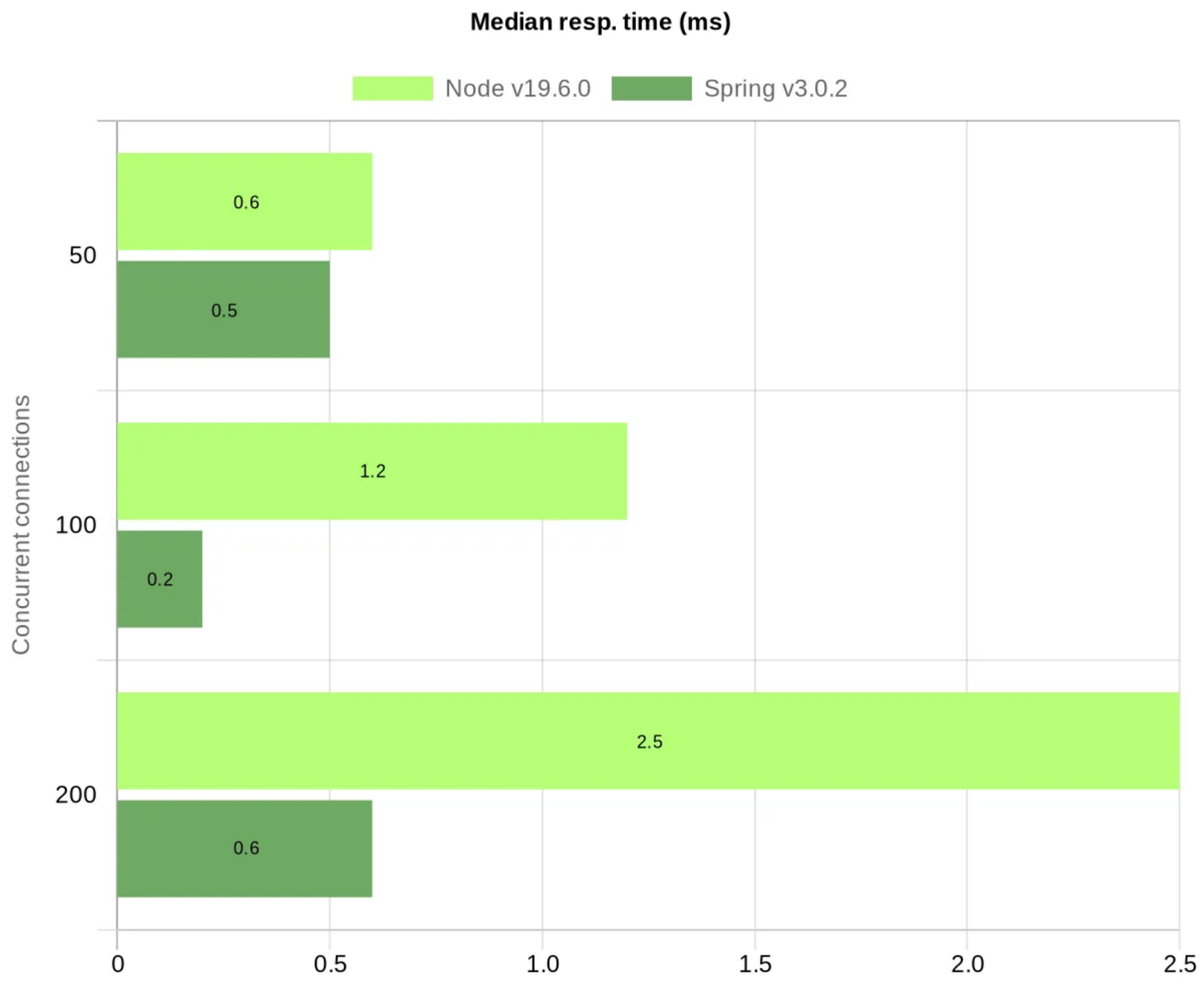
Here are the charts representing results for each type of measurement:

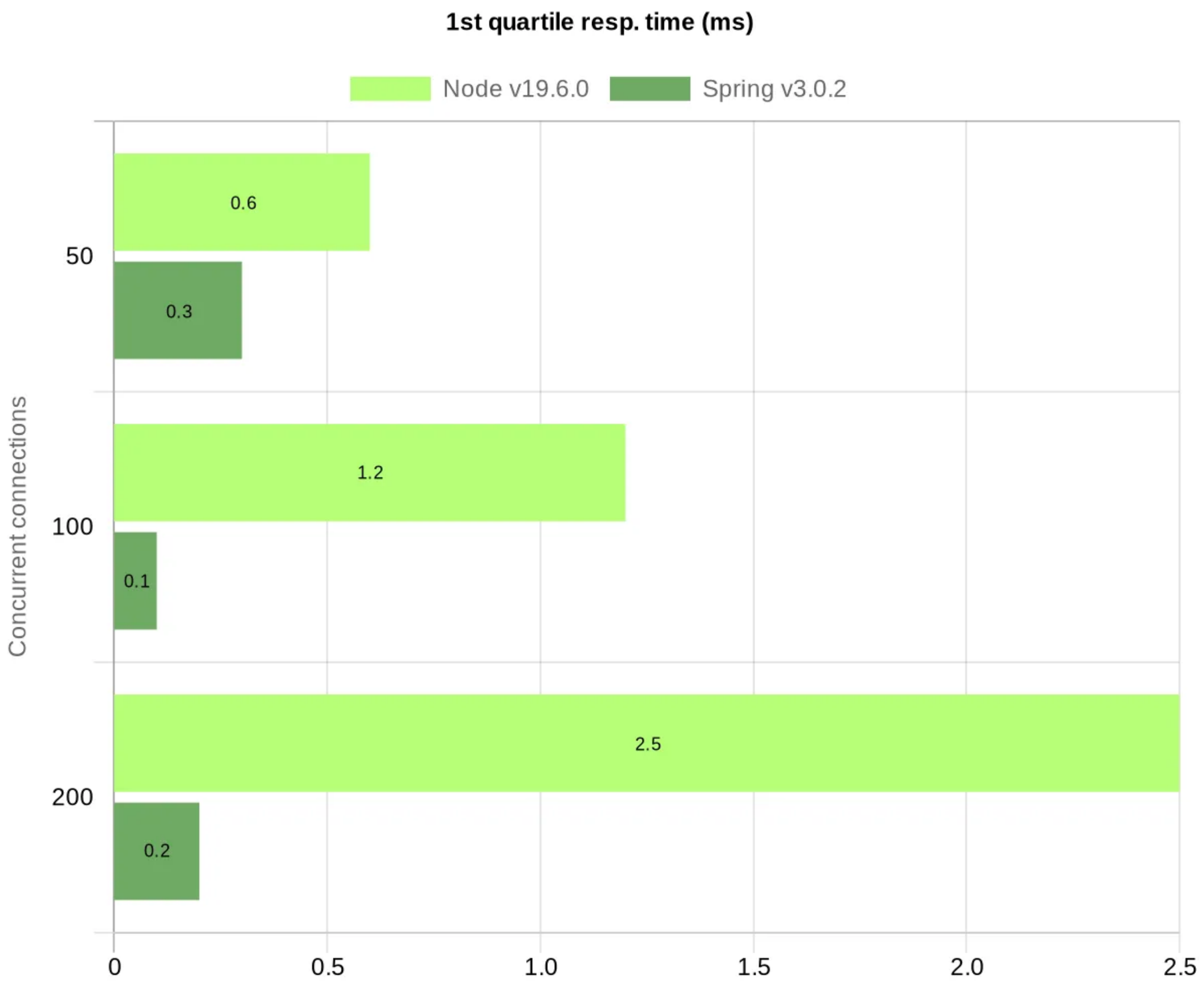
(Note that all latencies are in milliseconds)

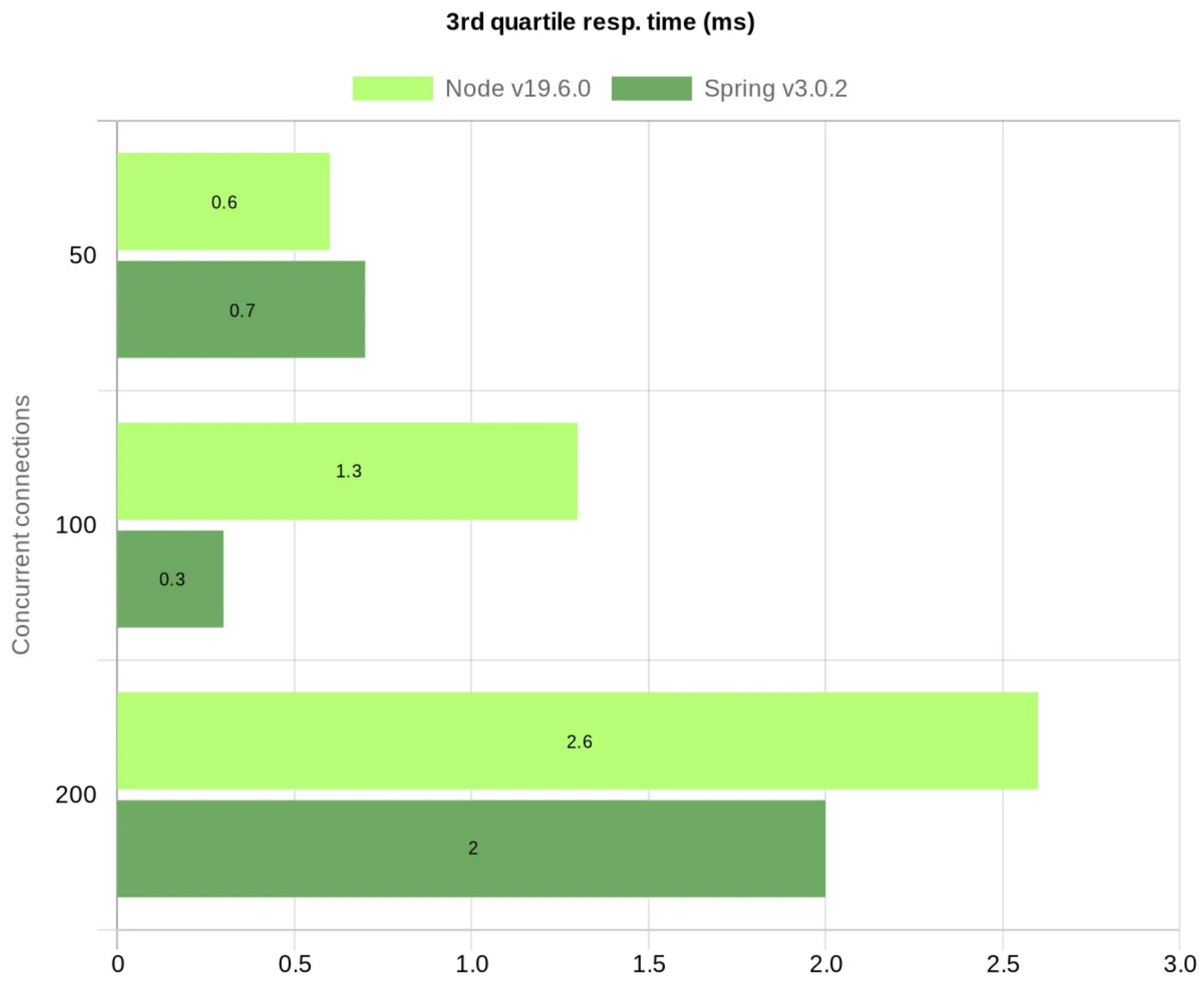

Time taken (secs)
Node v19.6.0
Spring v3.0.2
Concurrent connections
50
100
200
63
98
64
127
65
99
0
20
40
60
80
100
120
140

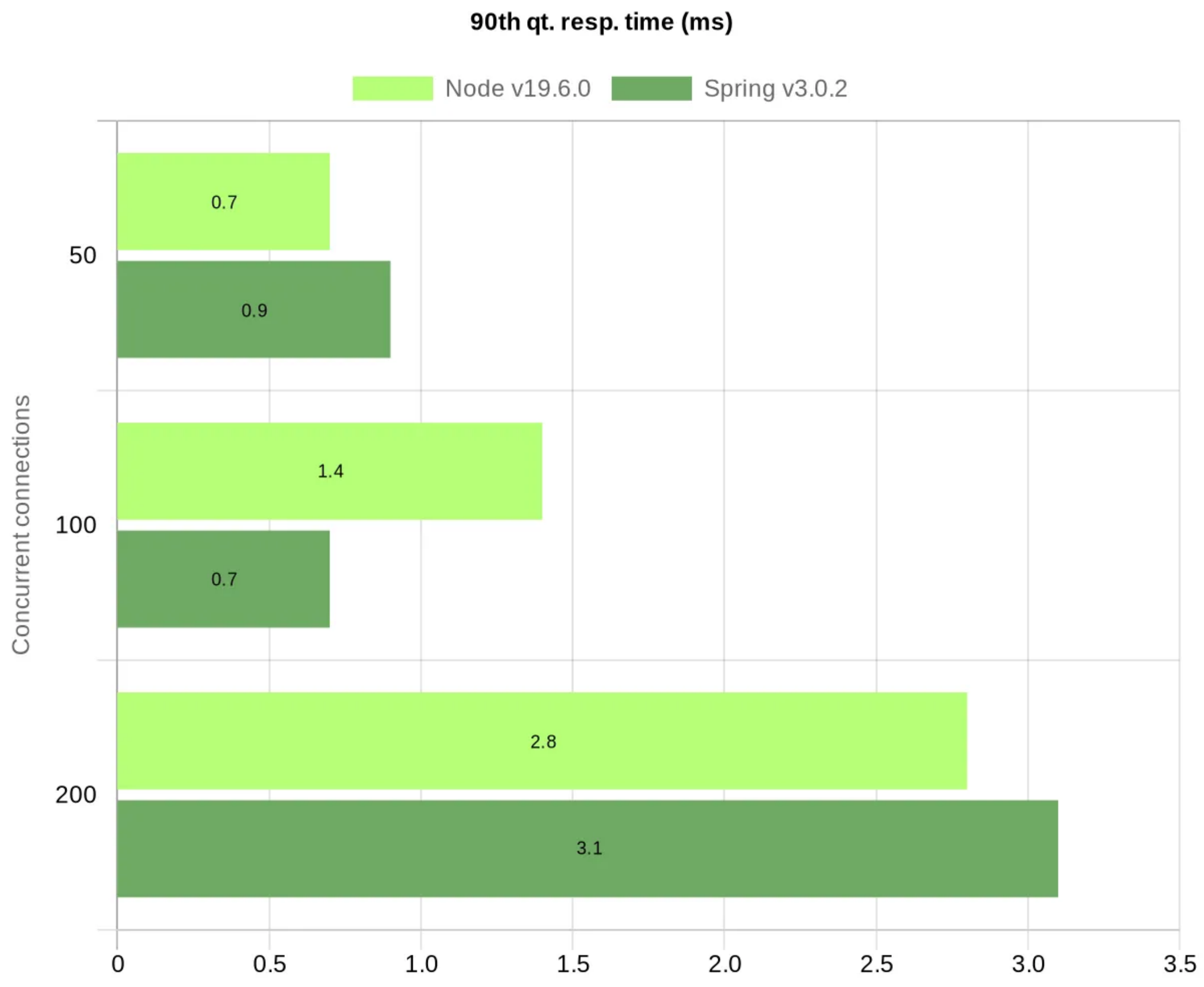

Requests per second
Node v19.6.0
Spring v3.0.2
Concurrent connections
50
78934
52101
100
78392
40870
200
77283
53271
0
10000
20000
30000
40000
50000
60000
70000
80000

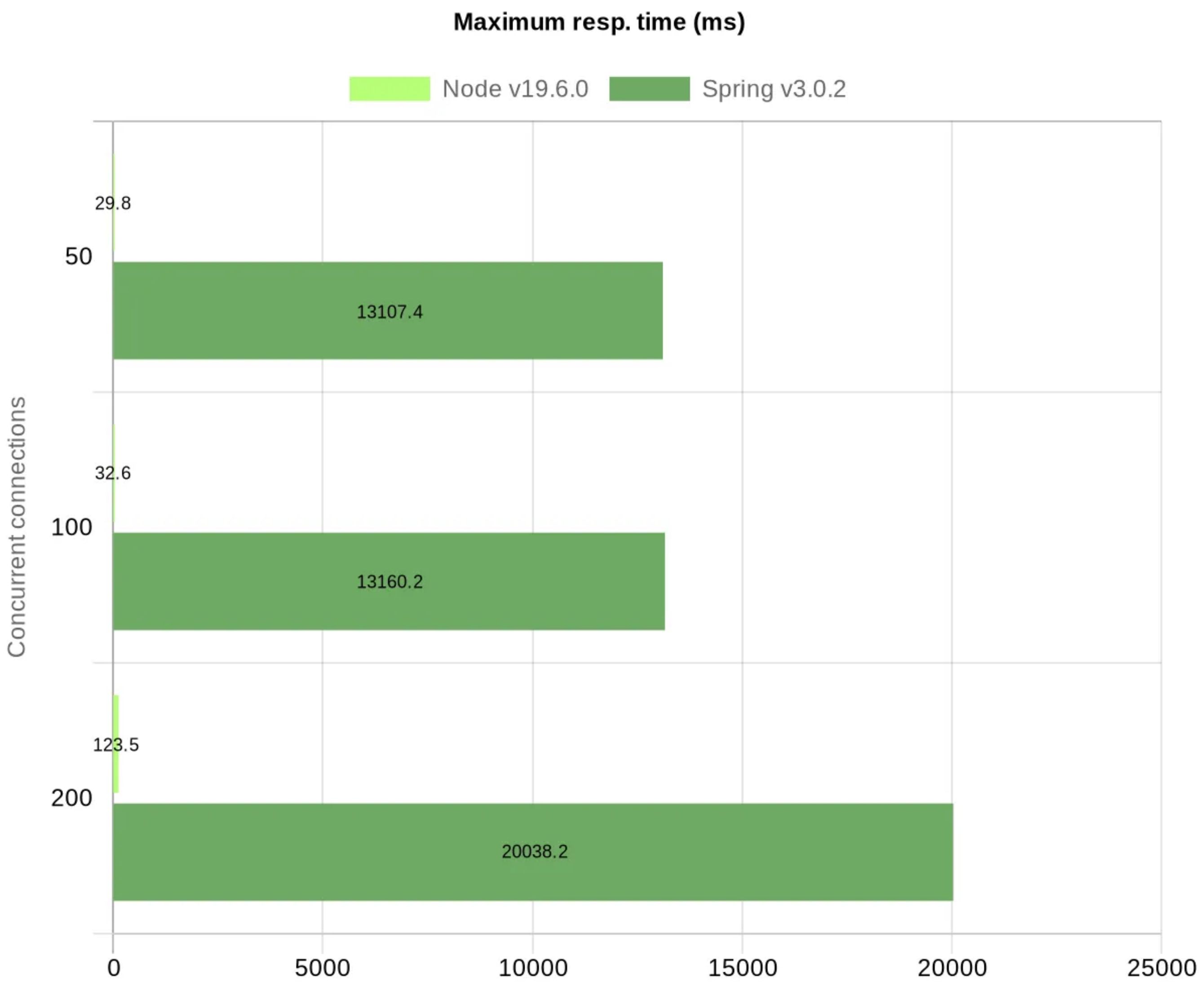

Mean resp. time (ms)
Node v19.6.0
Spring v3.0.2
Concurrent connections
50
100
200
0.6
1
1.3
2.5
2.6
3.6
0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

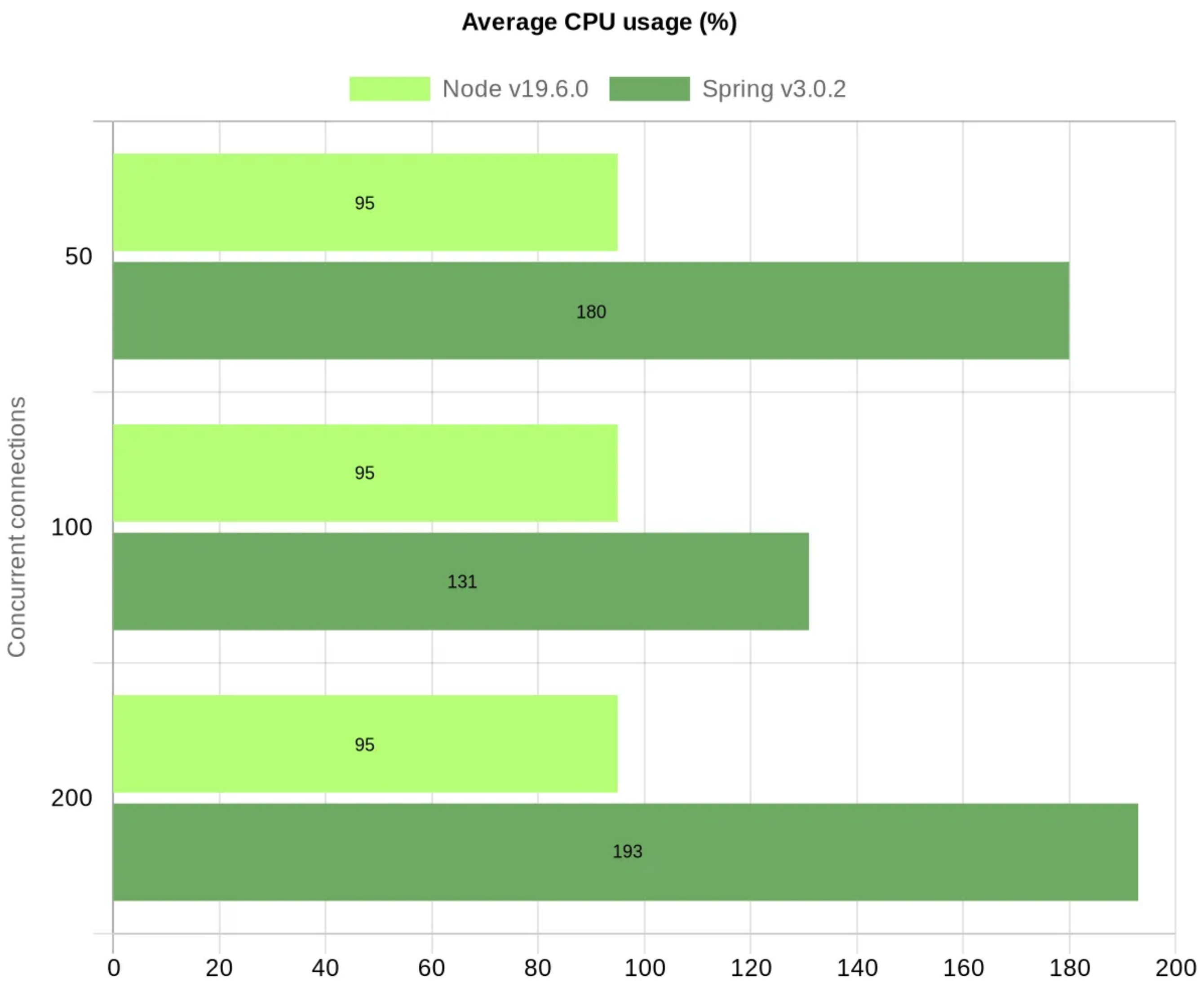

Median resp. time (ms)
Node v19.6.0
Spring v3.0.2
Concurrent connections
50
100
200
0.6
0.5
1.2
0.2
2.5
0.6
0
0.5
1.0
1.5
2.0
2.5

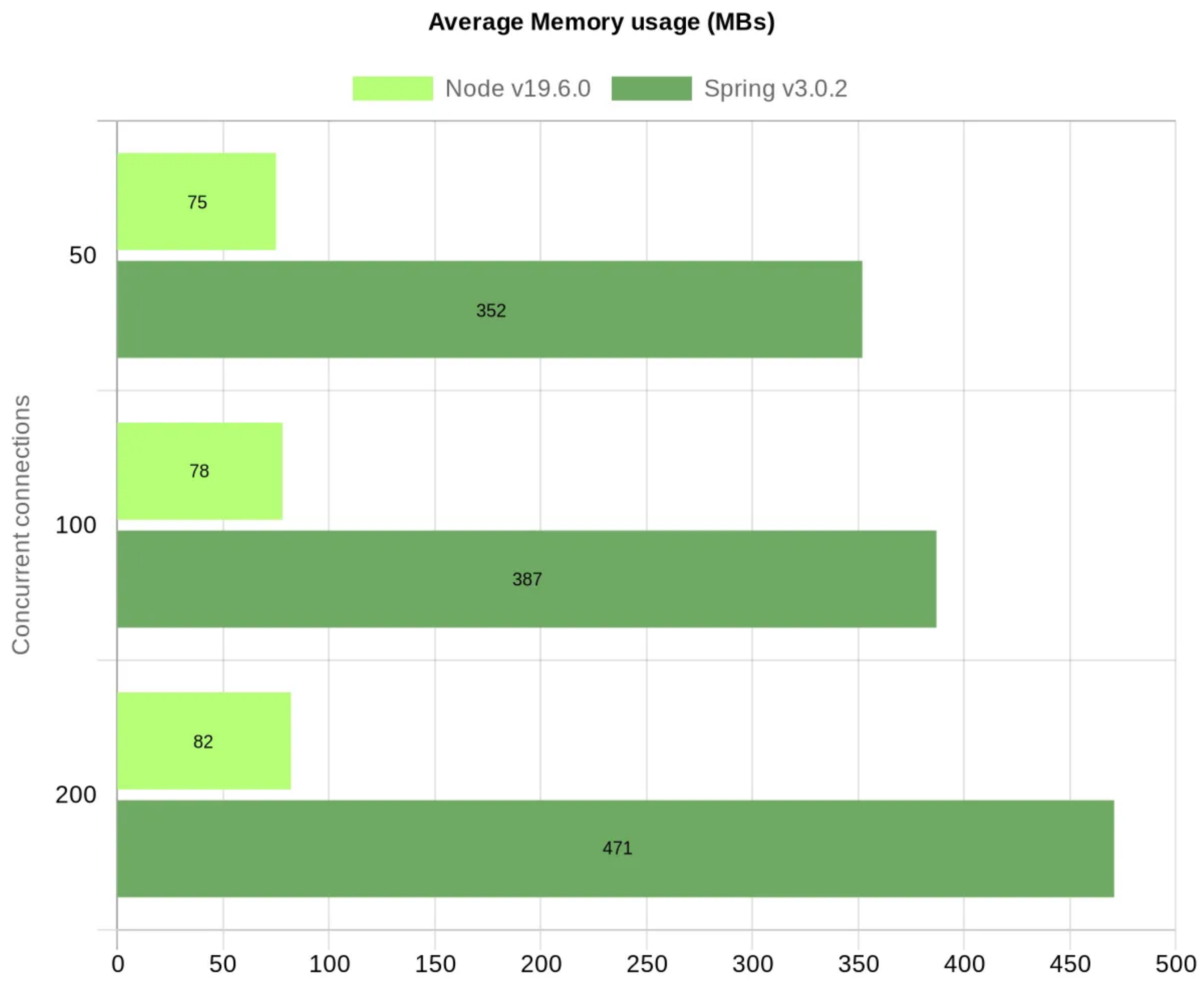

1st quartile resp. time (ms)
Node v19.6.0
Spring v3.0.2
Concurrent connections
50
100
200
0.6
0.3
1.2
0.1
2.5
0.2
0
0.5
1.0
1.5
2.0
2.5


3rd quartile resp. time (ms)
Node v19.6.0
Spring v3.0.2
Concurrent connections
50
100
200
0.6
0.7
1.3
0.3
2.6
2
0
0.5
1.0
1.5
2.0
2.5
3.0


90th qt. resp. time (ms)
Node v19.6.0
Spring v3.0.2
Concurrent connections
50
100
200
0.7
0.9
1.4
0.7
2.8
3.1
0
0.5
1.0
1.5
2.0
2.5
3.0
3.5


Maximum resp. time (ms)
Node v19.6.0
Spring v3.0.2
Concurrent connections
50
100
200
29.8
13107.4
32.6
13160.2
123.5
20038.2
0
5000
10000
15000
20000
25000


Average CPU usage (%)
Node v19.6.0
Spring v3.0.2
Concurrent connections
50
100
200
95
180
95
131
95
193
0
20
40
60
80
100
120
140
160
180
200



My analysis

First, Spring/Java is a very resource intensive application. For a simple hello world case with 200 concurrent connections, Spring uses ~190% CPU and 470M of memory. Comparatively, Node.js uses 95% CPU and 82M of memory.

Now for the latencies, spring's latencies are lower than Node.js till 3rd quartile. Spring fails with maximum latency readings which runs into seconds, while Node.js's maximum latency reaches 123ms. Even the median latency of Spring (0.6ms) is significantly better than Node.js's (2.5ms).

To conclude, Node.js offers more RPS using significantly less system resources.

For more stories like this, please follow the medium magazine: The JS runtimes.

Nodejs Spring