Do Not Thrash the Node.js Event Loop

@matteocollina | Co-Founder & CTO
Matteo Collina
CO-FOUNDER & CTO, Platformatic

8 years as a consultant focused on Node.js

Created Fastify and Pino

Node.js Technical Steering Committee member

Board member OpenJS Foundation

Subscribe to my newsletter at https://nodeland.dev.
17 Billions Downloads / Year

Downloads per Day

Date

Downloads


73.7M
Node.js is event loop based

I/O Event Queue

User JavaScript and C++ executes here

Kernel and Operating System
Node.js is event loop based

The Node.js process accepts new I/O

No asynchronous I/O happening
The “normal” flow of HTTP requests in Node.js

CAN YOU SPOT THE PROBLEM?

The Processing time of composing one request is composed by:

2 Synchronous Processing + 1 Asynchronous wait.

HTTP requests coming in → Synchronous Processing → Database querying → Synchronous Processing → HTTP responses going out
Do you like math?

Response time = 2 SP + 1 AS

Example:
- 10 ms of synchronous processing time
- 10 ms of I/O wait
Total response time: 30 ms.

Total number of request serviceable in 1 second by 1 CPU:

1000 ms / (10 ms * 2) = 50

(The processing time does not count)
Denial of Service Attack ahead.
What happens if they all arrive at the same time?
3 requests arrives at the same time.

What's the total response time of the last one?
Do you like math?

Response time = **2 SP + 1 AS**
3 requests arrives at the same time

Example:
- **10 ms** of synchronous processing time
- **10 ms** of I/O wait

Total response time of 1st request: **30 ms**.
Total response time of 2nd request: **50 ms**.
Total response time of 3rd request: **70 ms**.

Response Time x = **SPx *2 + ASx + (SPx-1 *2)**
Do you like math?

In our example, total number of request serviceable in 1 second by 1 CPU:

\[ 1000 \text{ ms} / (10 \text{ ms} \times 2) = 50 \]

(The processing time does not count)

What happens if we got more than that number?
Response Time (sync: 20, async: 10, rps: 50)
Isn’t scaling on CPU usage enough?
You can have > 100% CPU utilization and still have capacity left.
The actual event loop

- Incoming: Connections, data, etc.
- poll
- check
- close callbacks
- Idle, prepare
- Pending callbacks
- timers
Event loop delay

https://github.com/mcollina/loopbench/blob/master/loopbench.js

```javascript
function now () {
    return process.hrtime.bigint() / 1000000
}

setInterval(checkEventLoopDelay, 1000).unref()
let last = now()

function checkLoopDelay () {
    const toCheck = now()
    const delay = Number(toCheck - last - BigInt(1000))
    last = toCheck

    const overLimit = result.delay > 1000

    if (overLimit) {
        console.log('Event Loop delay over 1s')
    }
}
```
The event loop delay measures the effects after the problem already happened. It’s good at mitigating incidents but not at preventing them.
If you didn’t check out Node.js in the last few years, I have some news...
...Node.js is multithreaded!
A linearized model for the Event Loop

Source: https://nodesource.com/blog/event-loop-utilization-nodejs/
Event Loop Utilization

It's the cumulative duration of time the event loop has been both idle and active as a high resolution milliseconds timer. We can use it to know if there is “spare” capacity in the event loop!

```javascript
const { eventLoopUtilization } = require('perf_hooks').performance;
let lastELU = eventLoopUtilization();

setInterval(() => {
  // Store the current ELU so it can be assigned later.
  const tmpELU = eventLoopUtilization();
  // Calculate the diff between the current and last before sending.
  someExternalCollector(eventLoopUtilization(tmpELU, lastELU));
  // Assign over the last value to report the next interval.
  lastELU = tmpELU;
}, 100);
```
import fastify from 'fastify'
import underPressure from '@fastify/under-pressure'

const app = fastify()

app.register(underPressure, {
  maxEventLoopDelay: 1000,
  maxHeapUsedBytes: 1000000000,
  maxHeapUnreapedBytes: 1000000000,
  maxEventLoopUtilization: 0.98
})

app.get('/', (req, reply) => {
  // synchronous + asynchronous compute
  return ...
})

await app.listen({ port: 3000 })
Example using @fastify/under-pressure

HTTP requests coming in

Synchronous Processing

Database

HTTP responses going out

Event Loop Utilization is at 0.98, start dropping requests return a HTTP status code 503

Event Loop Utilization is at 0.98
Demo Time!

May the demo gods be with me
import { Piscina } from 'piscina';

const piscina = new Piscina({
  // The URL must be a file:// URL
  filename: new URL('./worker.mjs', import.meta.url).href
});

const result = await piscina.run({ a: 4, b: 6 });
console.log(result); // Prints 10
Deduplicating asynchronous calls

HTTP requests coming in

Synchronous Processing

The call to the database is deduplicated using async-cache-dedupe

Database

HTTP responses going out
```javascript
import { createCache } from 'async-cache-dedupe'

const cache = createCache({
  ttl: 5, // seconds
  stale: 5, // number of seconds to return data after ttl has expired
  storage: { type: 'memory' },
});

cache.define('fetchSomething', async (k) => {
  console.log('query', k)
  // query 42
  // query 24

  return { k }
});

const p1 = cache.fetchSomething(42)
const p2 = cache.fetchSomething(24)
const p3 = cache.fetchSomething(42)

const res = await Promise.all([p1, p2, p3])

console.log(res)
// [ 
//   { k: 42 },
//   { k: 24 },
//   { k: 42 }
// ]
```
The future of backend development. Now.
Enter Platformatic

We are helping developers get rid of the undifferentiated heavy lifting of building Node.js applications.
Thanks

Try our open-source tools at platformatic.dev

@platformatic @matteocollina