NODEJS (EVENT LOOP MISCONCEPTIONS)

Node.js is an event-based platform. This means that everything that happens in Node is the reaction to an event. A transaction passing through Node traverses a cascade of callbacks.

Abstracted away from the developer, this is all handled by a library called libuv which provides a mechanism called an event loop.

This event loop is maybe the most misunderstood concept of the platform. When we approached the topic of event loop monitoring, we put a lot of effort into properly understanding what we are actually measuring.

In this article I will cover our learnings about how the event loop really works and how to monitor it properly.

Common misconceptions:

- The event loop runs in a separate thread in the user code.
- Everything that's asynchronous is handled by a threadpool.
- The event loop is something like a stack or queue.

Let's discuss those phases:

Timers

Everything that was scheduled via setTimeout() or setInterval() will be processed here.

IO Callbacks

Here most of the callbacks will be processed. As all userland code in Node.js is basically in callbacks (e.g a callback to an incoming http request triggers a cascade of callbacks), this is the userland code.

IO Polling

Polls for new events to be processed on the next run.

Set Immediate

Runs all callbacks registered via setImmediate().

Close

Here all on('close') event callbacks are processed.

Monitoring the Event Loop

We see that in fact everything that goes on in a Node applications runs through the event loop. This means that if we could get metrics out of it, they should give us valuable information about the overall health and performance of an application.

There is no API to fetch runtime metrics from the event loop and as such each monitoring tool provides their own metrics. Let's see what we came up with.

Tick Frequency

The number of ticks per time.

Tick Duration

The time one tick takes.

As our agent runs as a native module it was relatively easy for us to add probes to provide us this information.

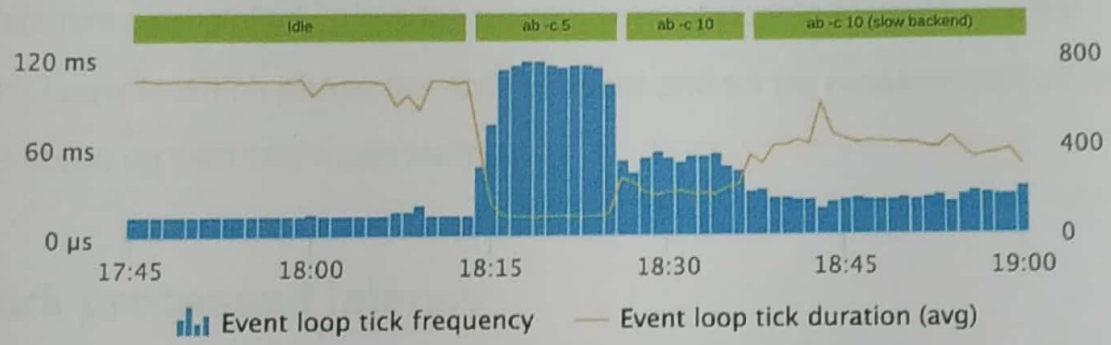
Tick frequency and tick duration metrics in action.

When we did our first tests under different loads, the results were surprising – let me show you an example:

In the following scenario I am calling an express.js application that does an outbound call to another http server.

There are four scenarios:

1. Idle
   There are no incoming requests
2. ab -c 5
   Using apache bench I created 5 concurrent requests at a time
3. ab -c 10
   10 concurrent at a time
4. ab -c 10 (slow backend)
   The http server that is called returns data after 1s to simulate a slow backend. This should cause something called back pressure as requests waiting for the backend to return pile up inside Node.



If we look at the resulting chart we can make an interesting observation:

Event loop duration and frequency are dynamically adapted

If the application is idle, which means that there are no pendings tasks (Timers, callbacks, etc), **it would not make sense to run through the**

Event Loop Latency

The event loop latency measures how long it additionally takes until a task scheduled with setTimeout(X) really gets processed.

A high event loop latency indicates an event loop busy with processing callbacks.

Utilize all CPUs

A Node.js application runs on a single thread. On multicore machines that means that the load isn't distributed over all cores. Using the cluster module that comes with Node it's easy to spawn a child process per CPU. Each child process maintains its own event loop and the master process transparently distributes the load between all childs.

Tune the Thread Pool

As mentioned, libuv will create a thread pool with the size of 4. The default size of the pool can be overridden by setting the environment variable UV_THREADPOOL_SIZE.

While this can solve load problems on I/O-bound applications I'd recommend excessive load testing as a larger thread pool might still exhaust the memory or the CPU.

Offload the work to Services

If Node.js spends too much time with CPU heavy operations, offloading work to services maybe even using another language that better suits a specific task might be a viable option.

Summary

Let's summarize what we've learned in this post:

- The event loop is what keeps a Node.js application running
- Its functionality is often misunderstood – it is a set of phases that are traversed continuously with specific tasks for each phase
- There are no out-of-the-box metrics provided by the event loop so the metrics collected are different between APM vendors
- The metrics clearly provide valuable insights about bottlenecks but deep understanding of the event loop and also the code that is running is key
- In the future Dynatrace will add event loop telemetry to its root cause detection to correlate event loop anomalies with problems