Tracing Micro-services and Modular IDEs:
Performance evaluation in asynchronous requests context

Hervé KABAMBA   Michel Dagenais

June 11, 2021

Polytechnique Montréal

Laboratoire **DORSAL**

Agenda

- Introduction
- Objective
- Methodology
- Current Results
- Ongoing work

Introduction

- **THEIA: The framework is developed in Typescript**

  – Frontend runs on the browser

  – Backend runs on Node.js

- **An interaction of the frontend with the backend:**
  - Is mainly a communication through a websocket connection channel that carries json encoded messages containing the data.
  - Vscode libraries are used by the backend to listen to the socket to retrieve the data
  - The data is mainly the service that must be invoked remotely, sometimes with arguments

Introduction

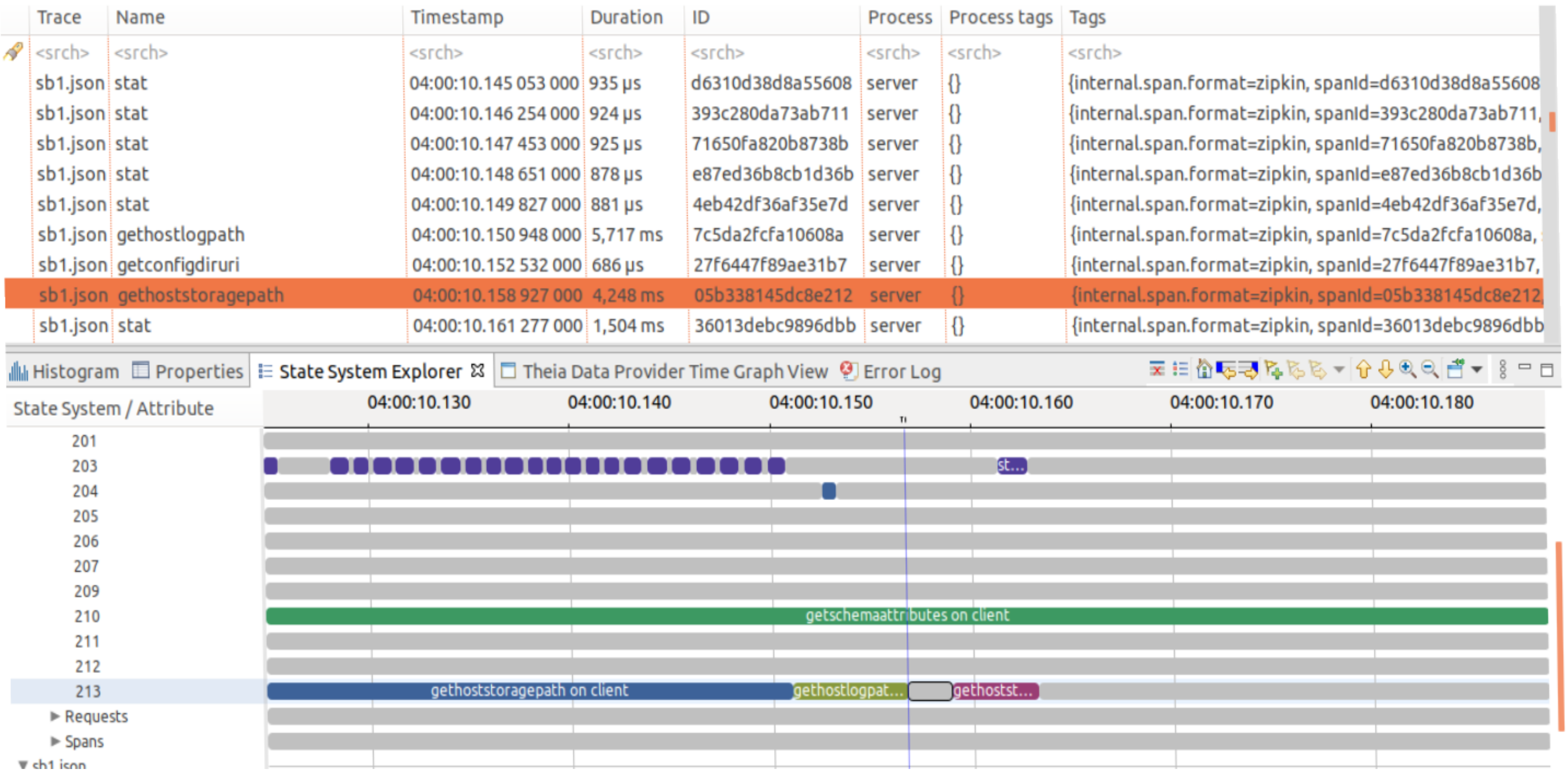
- **Backend:**
  - Operations are executed by invocation on the backend, and the returned results are sent back to the frontend
  - The backend is therefore mainly responsible for low level operations with the OS.
  - Node.js is single threaded and uses an event loop to handle asynchronous operations.
- **Intuitively:**
  - Evaluating the performance of applications running on Node.js brings complexity
  - High level tracing of distributed operations can only expose their latency in a global point of view.

Introduction

- **Definition of the problem**
  - Most operations and interactions from the frontend and the backend in Theia are executed by Node.js in asynchronous ways.
  - Asynchronous callbacks such as Promises, SetTimeout, etc. in Javascript are used to handle the operations.
  - Internally, there is a lot of activities responsible for delivering the final result that increase the complexity of the performance analysis in such context.

Introduction

High level trace information view



Trace	Name	Timestamp	Duration	ID	Process	Process tags	Tags
<srch>	<srch>	<srch>	<srch>	<srch>	<srch>	<srch>	<srch>
sb1.json	stat	04:00:10.145 053 000	935 µs	d6310d38d8a55608	server	{}	{internal.span.format=zipkin, spanId=d6310d38d8a55608
sb1.json	stat	04:00:10.146 254 000	924 µs	393c280da73ab711	server	{}	{internal.span.format=zipkin, spanId=393c280da73ab711,
sb1.json	stat	04:00:10.147 453 000	925 µs	71650fa820b8738b	server	{}	{internal.span.format=zipkin, spanId=71650fa820b8738b,
sb1.json	stat	04:00:10.148 651 000	878 µs	e87ed36b8cb1d36b	server	{}	{internal.span.format=zipkin, spanId=e87ed36b8cb1d36b
sb1.json	stat	04:00:10.149 827 000	881 µs	4eb42df36af35e7d	server	{}	{internal.span.format=zipkin, spanId=4eb42df36af35e7d,
sb1.json	gethostlogpath	04:00:10.150 948 000	5,717 ms	7c5da2fcfa10608a	server	{}	{internal.span.format=zipkin, spanId=7c5da2fcfa10608a,
sb1.json	getconfigdiruri	04:00:10.152 532 000	686 µs	27f6447f89ae31b7	server	{}	{internal.span.format=zipkin, spanId=27f6447f89ae31b7,
sb1.json	gethoststoragepath	04:00:10.158 927 000	4,248 ms	05b338145dc8e212	server	{}	{internal.span.format=zipkin, spanId=05b338145dc8e212,
sb1.json	stat	04:00:10.161 277 000	1,504 ms	36013debc9896dbb	server	{}	{internal.span.format=zipkin, spanId=36013debc9896dbb

Fig1: high level trace

Introduction

- **Definition of the problem**
  - In the previous figure, the higher view of the latency of each operation executed in Theia is generally exposed in the way that distributed tracers works and do not tell much except latency.
  - Although the latency represents the real time the operation took to complete, pinpointing the source of the problem is something else in Node.js.
  - Tracing the application with distributed tracers, when such a problem happens most of the time, cannot help pinpointing the operation responsible of the fault propagation.
  - A non optimized code, or an operation can slow down or block the internal event-loop and consequently delay all pending operations in the stack.

Objective

- **Asynchronous operations**
  - Tracking asynchronous resources and their respective callbacks in Node.js is very important
  - Asynchronous operations go through phases in the event-loop and, at each phase, their respective callback are executed.
  - The way the event-loop will behave is a function of the operations that it has enqueued.
  - In this context, a real performance analysis of Node.js applications must involve collecting lower internal information on its functioning for correlation with the higher level ones.
  - Such approach should result in accurate tools for performance analysis in the Node.js environment.

Event-loop at Glance

Abstraction of the Event-loop

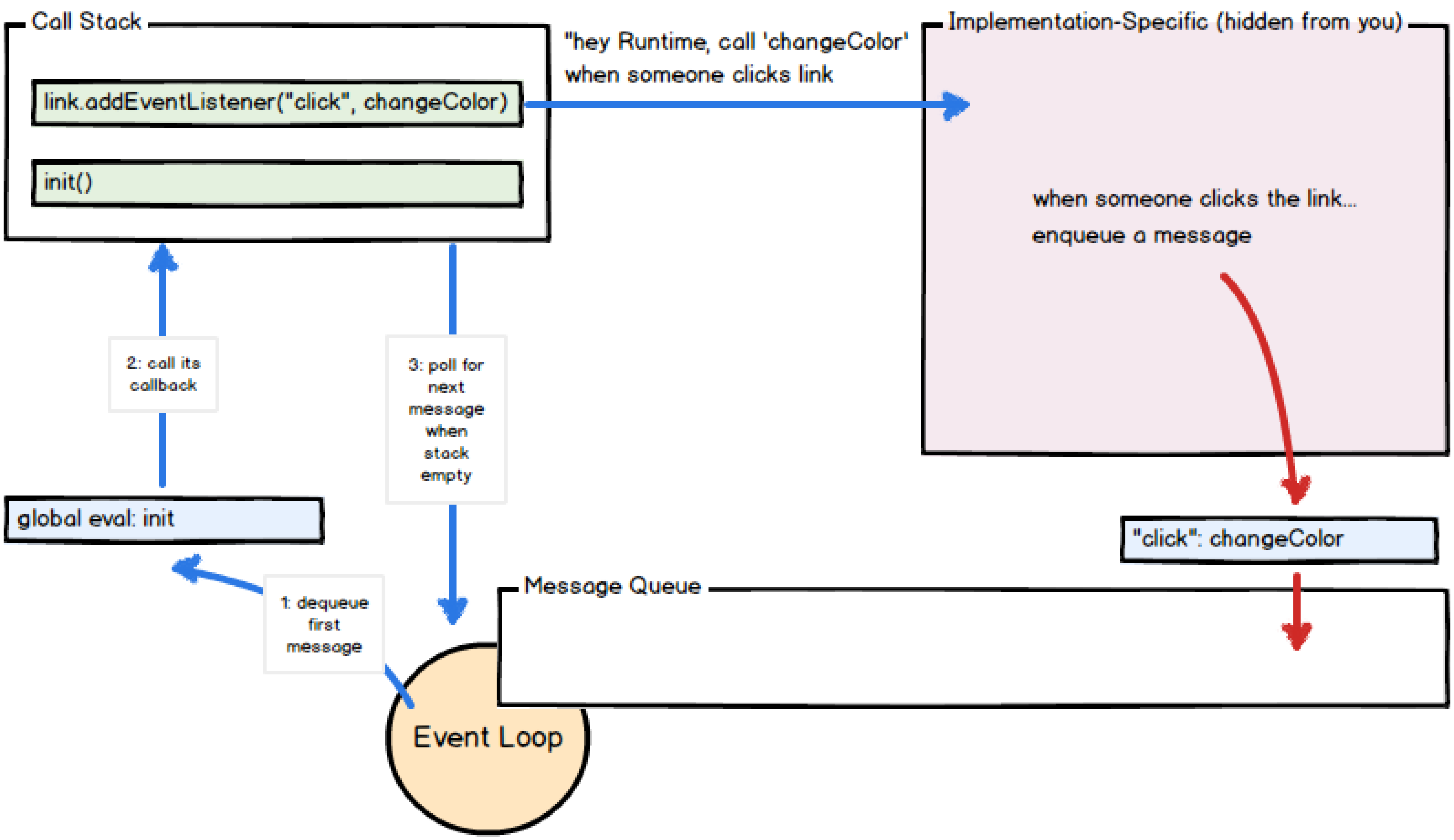


Fig 2: event-loop abstraction[2]

Event-loop at Glance

Delegation of tasks to Workers

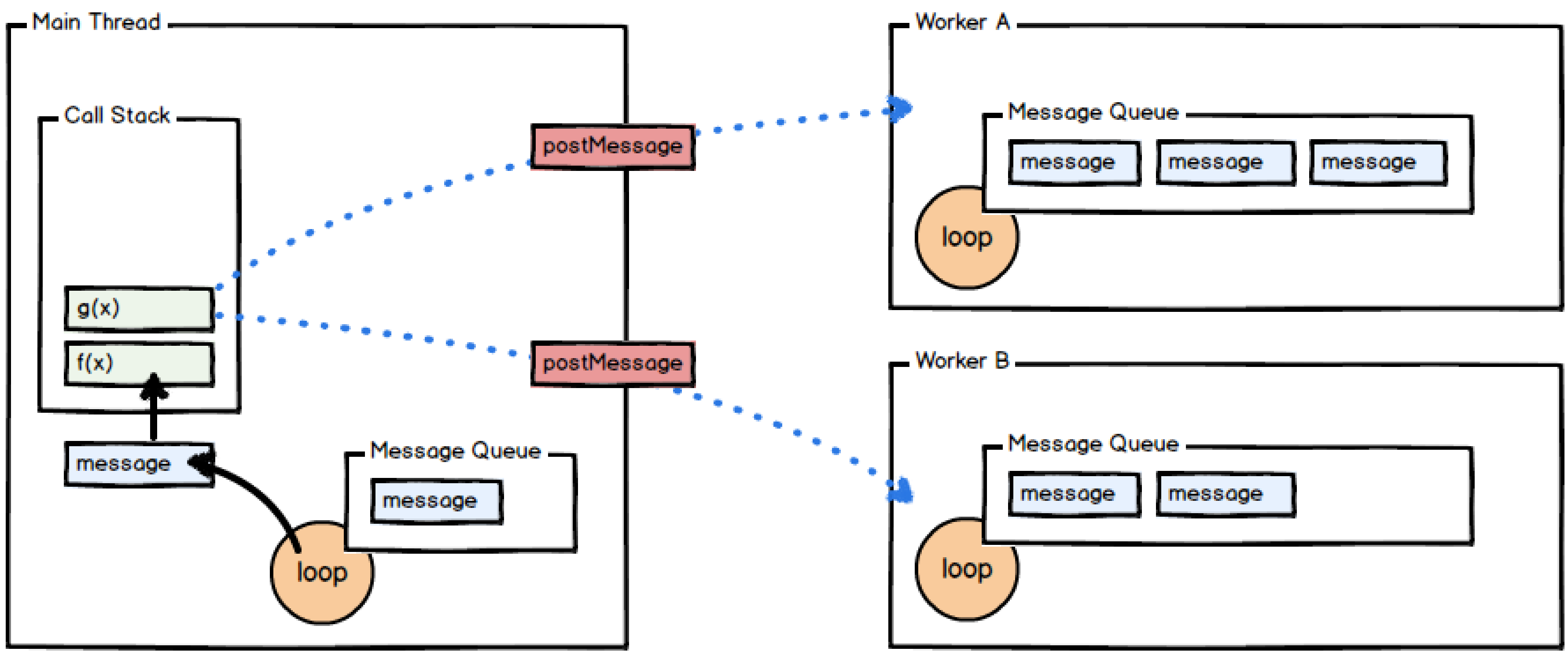
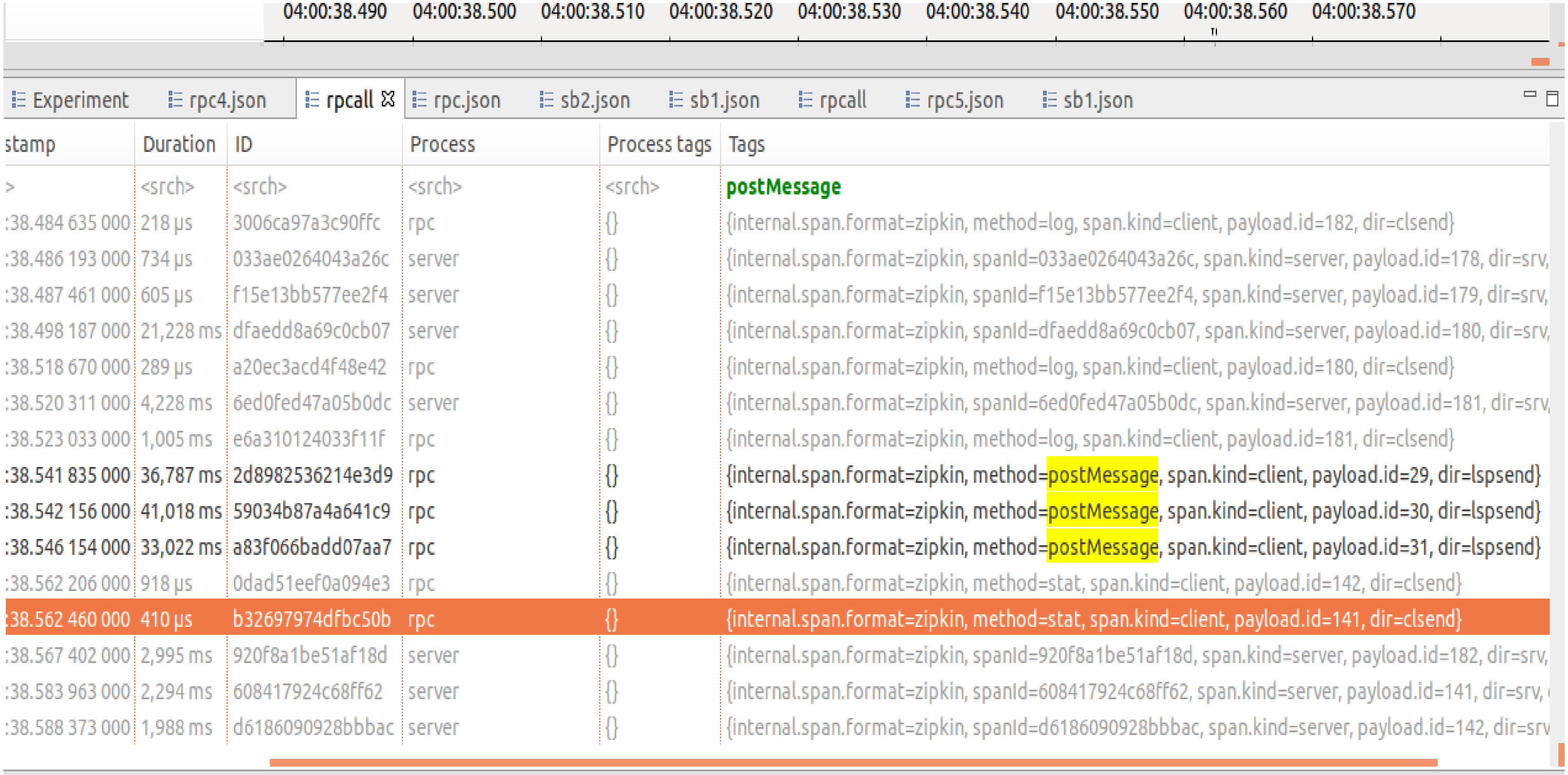


Fig 3:Tasks delegation to workers[2]

Workers view

The communication event between the workers are captured in this view (postMessage method) of the trace.



04:00:38.490 04:00:38.500 04:00:38.510 04:00:38.520 04:00:38.530 04:00:38.540 04:00:38.550 04:00:38.560 04:00:38.570

Experiment | rpc4.json | rpcall | rpc.json | sb2.json | sb1.json | rpcall | rpc5.json | sb1.json

stamp	Duration	ID	Process	Process tags	Tags
>	<srch>	<srch>	<srch>	<srch>	postMessage
:38.484 635 000	218 µs	3006ca97a3c90ffc	rpc	{}	{internal.span.format=zipkin, method=log, span.kind=client, payload.id=182, dir=clsend}
:38.486 193 000	734 µs	033ae0264043a26c	server	{}	{internal.span.format=zipkin, spanId=033ae0264043a26c, span.kind=server, payload.id=178, dir=srv,
:38.487 461 000	605 µs	f15e13bb577ee2f4	server	{}	{internal.span.format=zipkin, spanId=f15e13bb577ee2f4, span.kind=server, payload.id=179, dir=srv,
:38.498 187 000	21,228 ms	dfaedd8a69c0cb07	server	{}	{internal.span.format=zipkin, spanId=dfaedd8a69c0cb07, span.kind=server, payload.id=180, dir=srv,
:38.518 670 000	289 µs	a20ec3acd4f48e42	rpc	{}	{internal.span.format=zipkin, method=log, span.kind=client, payload.id=180, dir=clsend}
:38.520 311 000	4,228 ms	6ed0fed47a05b0dc	server	{}	{internal.span.format=zipkin, spanId=6ed0fed47a05b0dc, span.kind=server, payload.id=181, dir=srv,
:38.523 033 000	1,005 ms	e6a310124033f11f	rpc	{}	{internal.span.format=zipkin, method=log, span.kind=client, payload.id=181, dir=clsend}
:38.541 835 000	36,787 ms	2d8982536214e3d9	rpc	{}	{internal.span.format=zipkin, method=postMessage, span.kind=client, payload.id=29, dir=lspsend}
:38.542 156 000	41,018 ms	59034b87a4a641c9	rpc	{}	{internal.span.format=zipkin, method=postMessage, span.kind=client, payload.id=30, dir=lspsend}
:38.546 154 000	33,022 ms	a83f066badd07aa7	rpc	{}	{internal.span.format=zipkin, method=postMessage, span.kind=client, payload.id=31, dir=lspsend}
:38.562 206 000	918 µs	0dad51eef0a094e3	rpc	{}	{internal.span.format=zipkin, method=stat, span.kind=client, payload.id=142, dir=clsend}
:38.562 460 000	410 µs	b32697974dfbc50b	rpc	{}	{internal.span.format=zipkin, method=stat, span.kind=client, payload.id=141, dir=clsend}
:38.567 402 000	2,995 ms	920f8a1be51af18d	server	{}	{internal.span.format=zipkin, spanId=920f8a1be51af18d, span.kind=server, payload.id=182, dir=srv,
:38.583 963 000	2,294 ms	608417924c68ff62	server	{}	{internal.span.format=zipkin, spanId=608417924c68ff62, span.kind=server, payload.id=141, dir=srv,
:38.588 373 000	1,988 ms	d6186090928bbbac	server	{}	{internal.span.format=zipkin, spanId=d6186090928bbbac, span.kind=server, payload.id=142, dir=srv

Fig 4: Intercepting workers communication

Event-loop at Glance

Event-loop phases

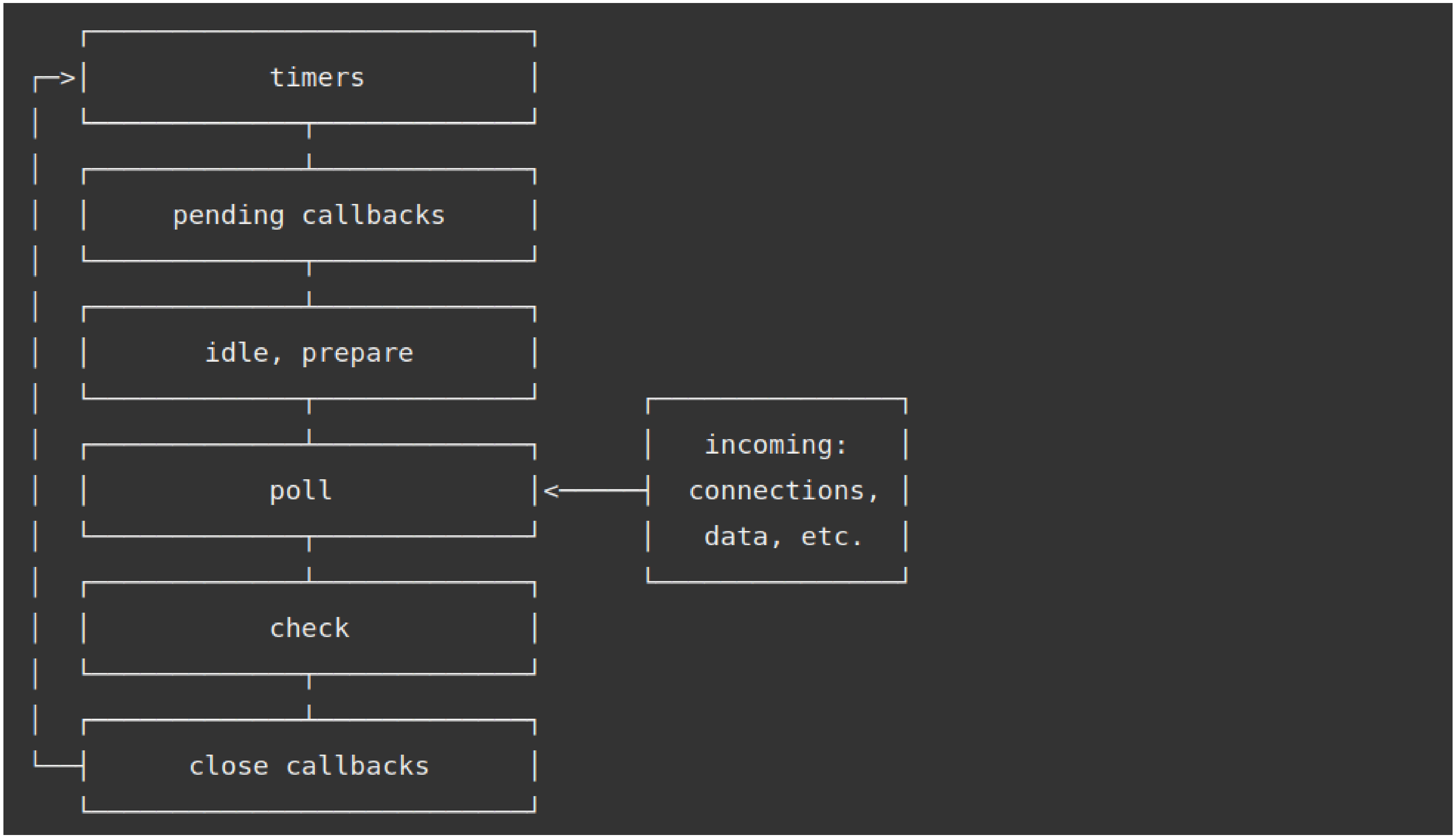


Fig 5: event-loop phase[5]

Methodology

- Tracing Node.js internals should give more insights on application bottlenecks
- Internal queues are sometimes responsible for high latency propagation at higher level operations
- A common problem with asynchronous operations is that they may enqueue other operations as a tree of operations.
- In such scenario, it results in the event-loop stucking in the same phase until it completes all operations.
- This results in increasing the latency of other pending operations in queues of other phases.

Methodology

- **Our Approach**

  – From high level information, track down the different operations at lower layers and reconstruct a vertical sequence of the request

  – 3 layers are considered: The application layer, the Node.js layer and the kernel layer

  – In a vertical request sequencing, latency at each layer can be identified

- **Problem**

  How to vertically inject the context of the trace to reconstruct the sequences of the execution?

Methodology

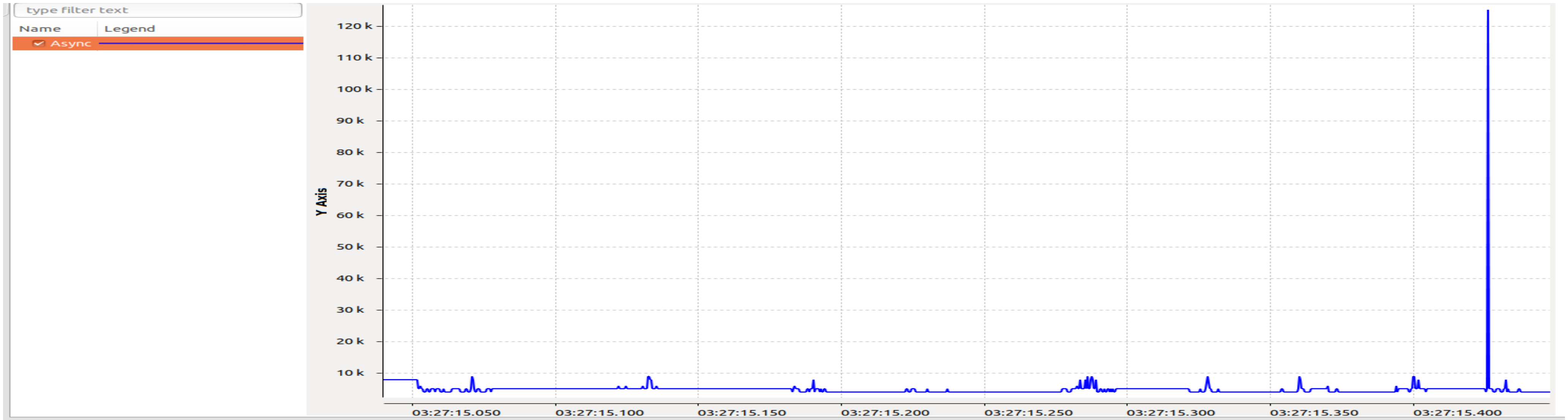
- **Libuv**
  - Node.js relies on this library to manage asynchronous operations
  - It clearly is a core Node.js IO operations library and is responsible for interacting with the OS
- **Instrumentation**
  - Libuv and Node.js internals are instrumented to obtain low level information at the intermediate layer
  - Application (Theia) is instrumented with Zipkin to obtain high level information
  - Kernel trace is collected and correlation algorithms are designed for vertical context injection and sequence reconstruction

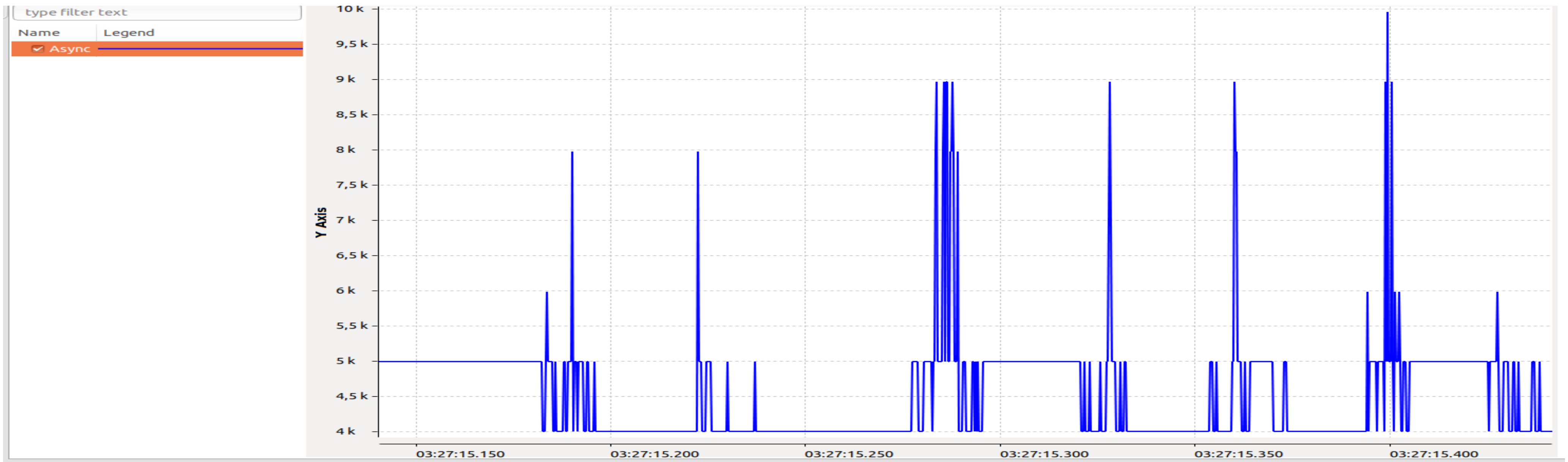
Current Results

- **Algorithms for heterogeneous traces correlation and vertical context injection**
- **Correlation between high level information and intermediate layer (Node.js), and intermediate layer with Kernel layer reconstruction of the execution sequence)**

Current Results

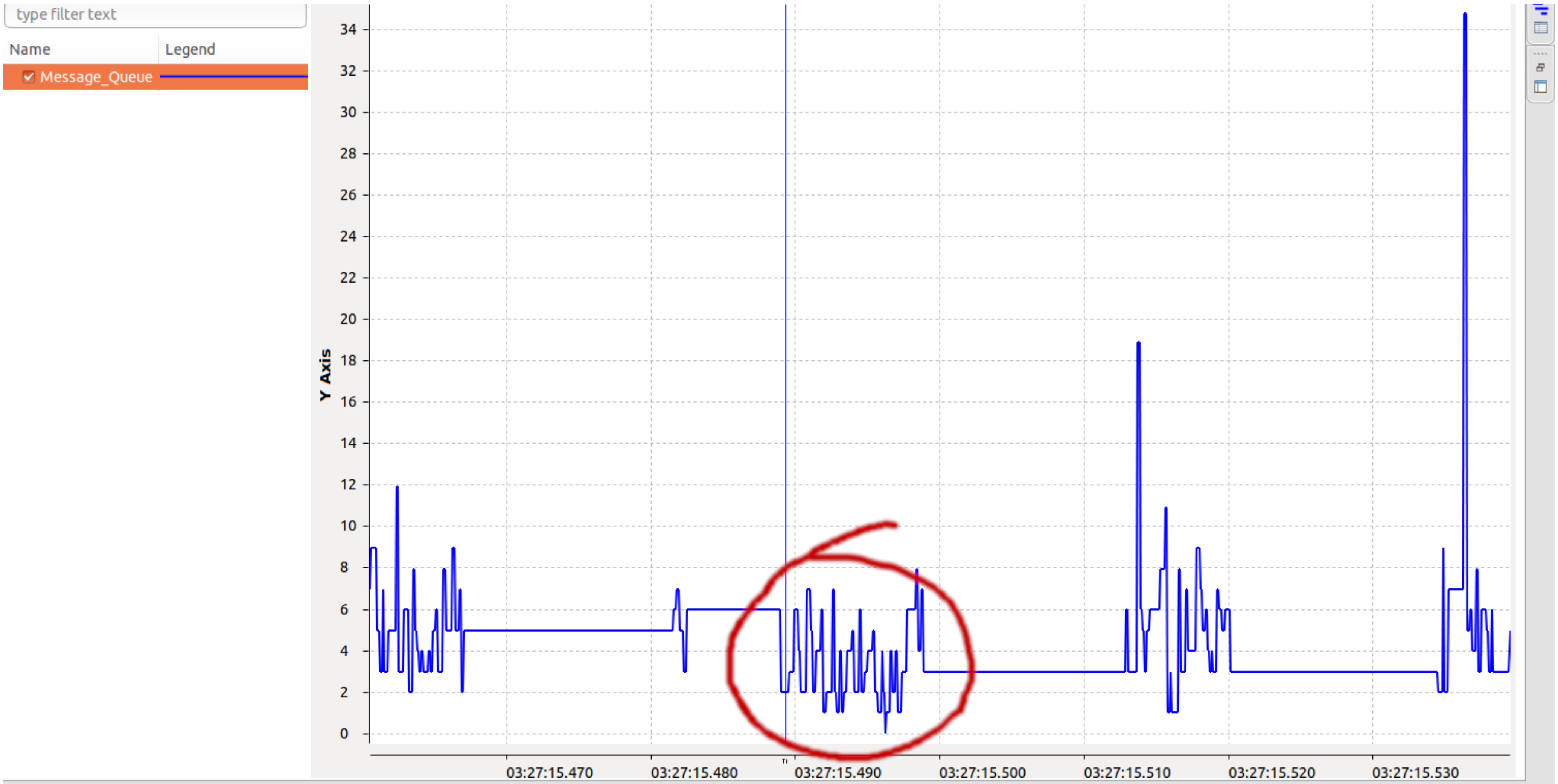
Async Operations times in the event-loop





Current Results

Number of operations enqueued / 100 ms



Ongoing Work

- **Intensive work is being done on the development of views tailored to the performance analysis of Theia, based on the preliminary results**

- **Work on identifying the critical path is also ongoing**

References

[1] Piero Borrelli,
https://blog.logrocket.com/a-complete-guide-to-the-node-js-event-loop/

[2] Erin Swenson-Healey,
https://blog.carbonfive.com/the-javascript-event-loop-explained/

[3] Aman Agrawal,
https://www.loginradius.com/blog/async/understanding-event-loop/

[4] Tania Rascia,
https://www.digitalocean.com/community/tutorials/understanding-the-event-loop-callbacks-promises-and-async-await-in-javascript

[5] https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/

Questions?

herve.kabamba-mbikayi@polymtl.ca