Low Overhead Transparent Microservices Tracing in Event Based Nodejs

Progress Report Meeting

Hervé KABAMBA

PhD Candidate

Supervisor: Michel Dagenais

June 01, 2023

Polytechnique Montréal

Département de Génie Informatique et Génie Logiciel

Agenda

1. Introduction
2. Our Approach
3. Some results
4. Conclusion

Introduction

Tracing microservices

- Observability is achieved to understand the behavior and the performance of microservices

- Telemetry data is obtained to monitor and identify problems in the system

- A lot of monitoring and tracing tools are available and achieve such requirements

- In this case, distributed tracers are used

Introduction(2)

Transparent Tracing?

- Developers should focus on the development of new features and the deployment of new components

- Instrumentation can sometimes be complex and time consuming

- Compromises are sometimes done between the need of observability and the modification of the application behavior brought by tracing

- Above all, the resulting overhead must be addressed carefully

Introduction(3)

Transparent microservices tracing in Nodejs

- Nodejs is a single-threaded environment orchestrating execution through an event-loop

- Low level socket communication can be captured to monitor the microservices interactions

- The available tools achieving transparency in microservices systems use such approach

Introduction (4)

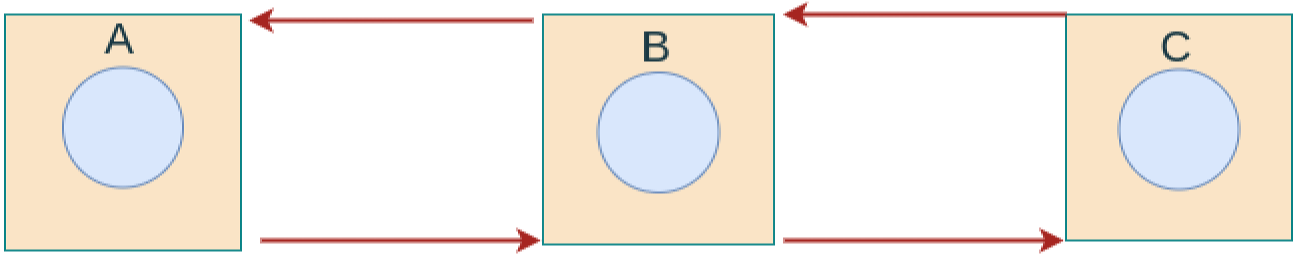
Problem:

Capturing interactions is simple through low level socket communication

However, transparently correlating requests and replies in asynchronous systems is a real challenge

Existing techniques in the literature are based on context horizontal propagation using distributed tracers. No transparency

Those that address transparency, intercept messages trough proxies, but can’t address correlation without injecting at the proxy level metadata and context information.



Introduction(5)

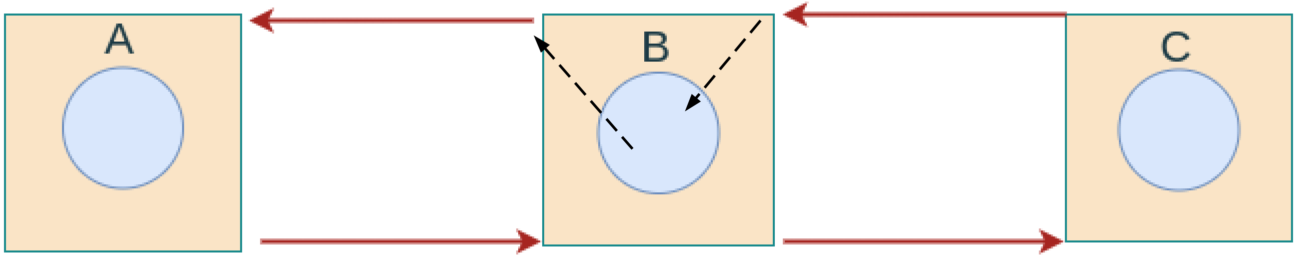
Problem (2):

Nodejs is a single-threaded system, everything is externally seen as a black box.

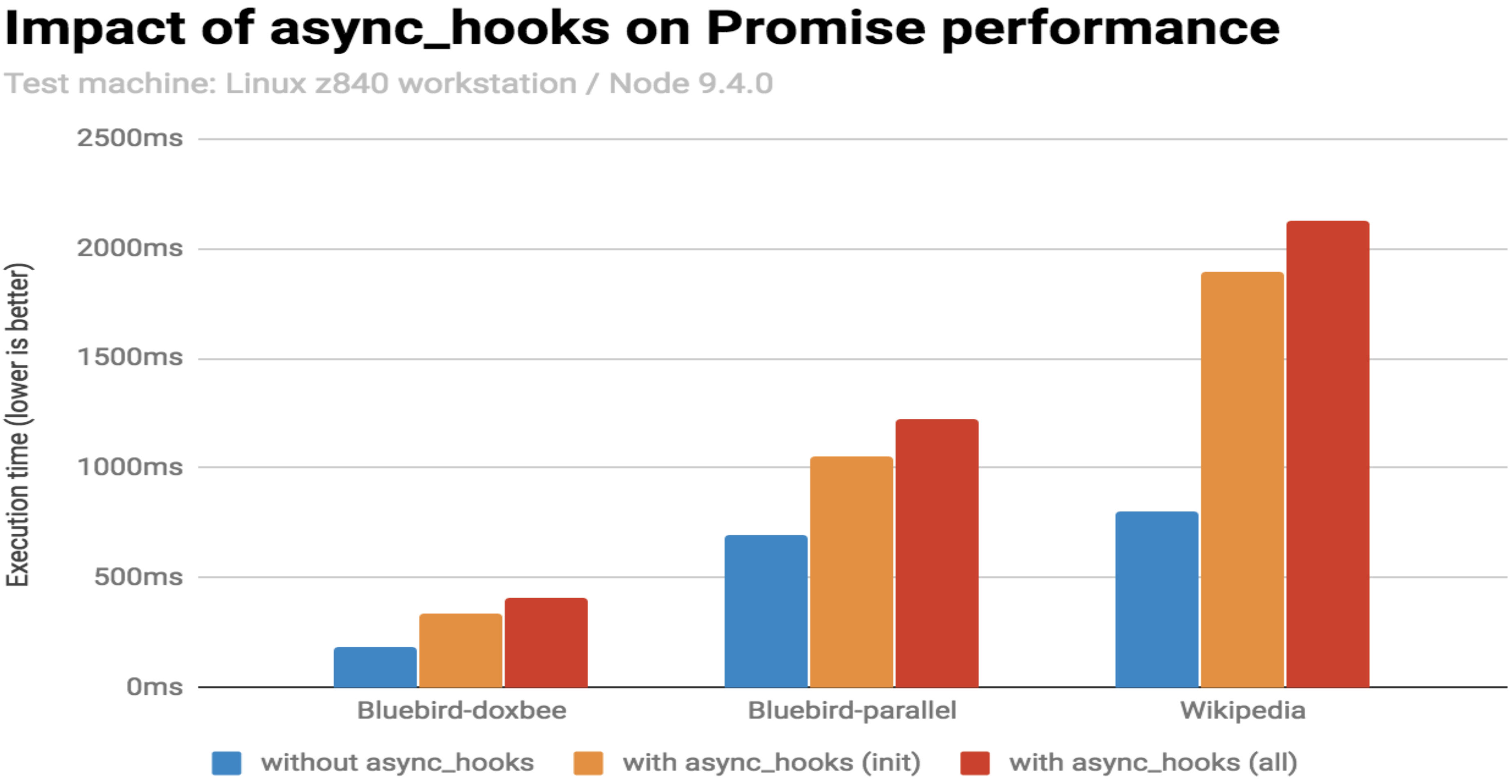
Internally, to track the life-cycle of registered callbacks, it uses AsyncHooks to ensure internal context propagation throughout objects life-cycle.

Asynchooks API is exposed at the Javascript land for context handling.

However, enabling Asynchooks brings a very large overhead overhead, especially for promises obejcts, that need to cross barrier from Javascript to C++ and back.



Introduction(6)



Source: https://github.com/bmeurer/async-hooks-performance-impact

Introduction(7)

Putting every together:

- Achieving transparency in such environment must be addressed differently

- Correlating requests is challenging in Nodejs. The only way to do it is to use Asynchooks, but with no transparency and a compromise on the overhead induced.

Our Approach

Transparent tracing of Nodejs microservices

- We address transparency differently
- We deal with the V8 engine of Nodejs.
- We track the internal mechanisms of the V8 engine that handle asynchronous and context propagation
- LTTng tracepoints are then injected within them

Our Approach

Context reconstruction

- Instead of propagating context information that is costly as distributed tracers and other approaches do,

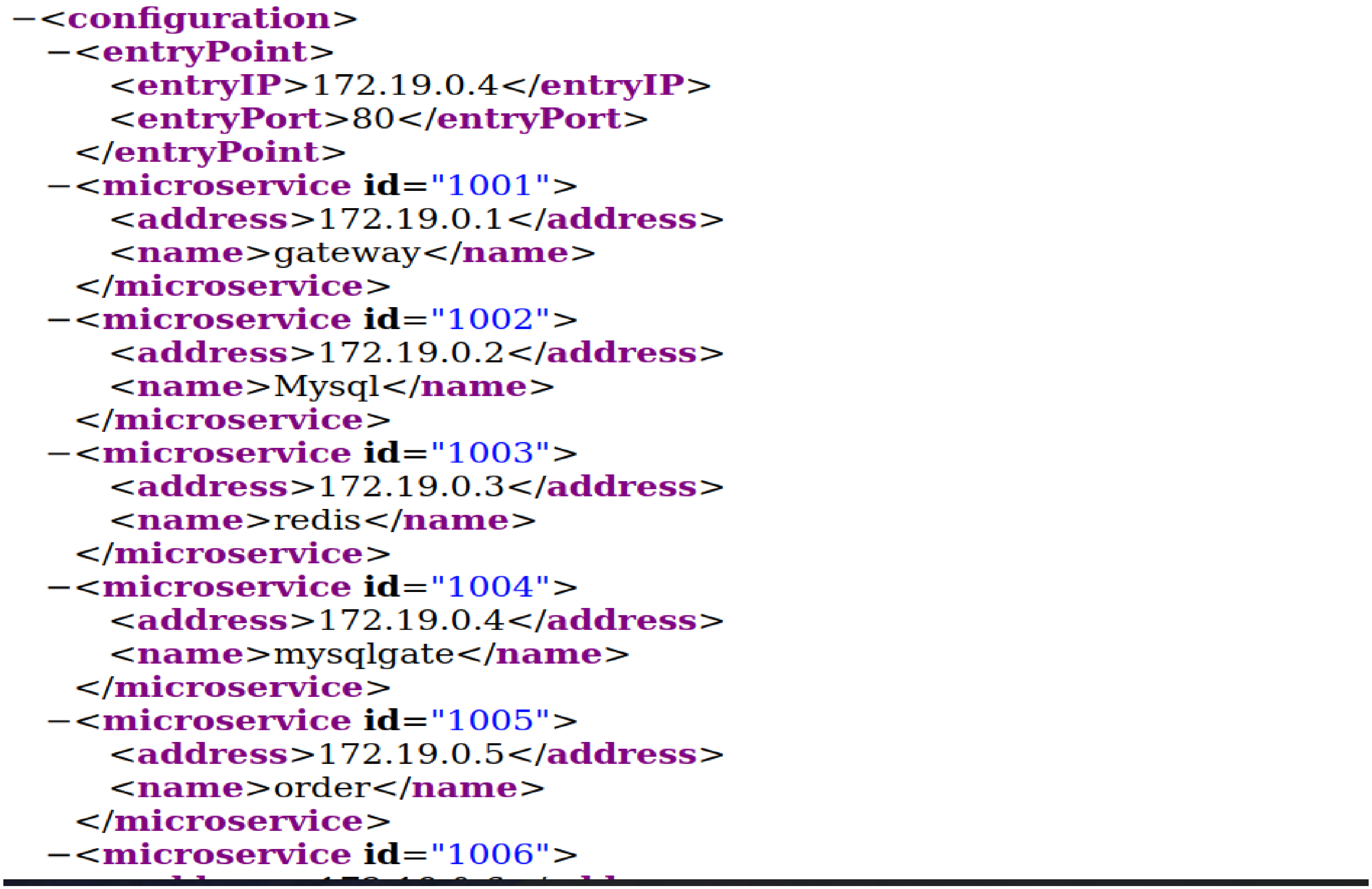
- We introduce the internal context reconstruction approach which achieve low overhead

- Therefore, from an experiment of **n** microservices traces, we reconstruct the context based on the tracking mechanisms of Nodejs

- A **6.8** % overhead is obtained outperforming existing transparency approaches for microservices tracing and context handling

Our Approach

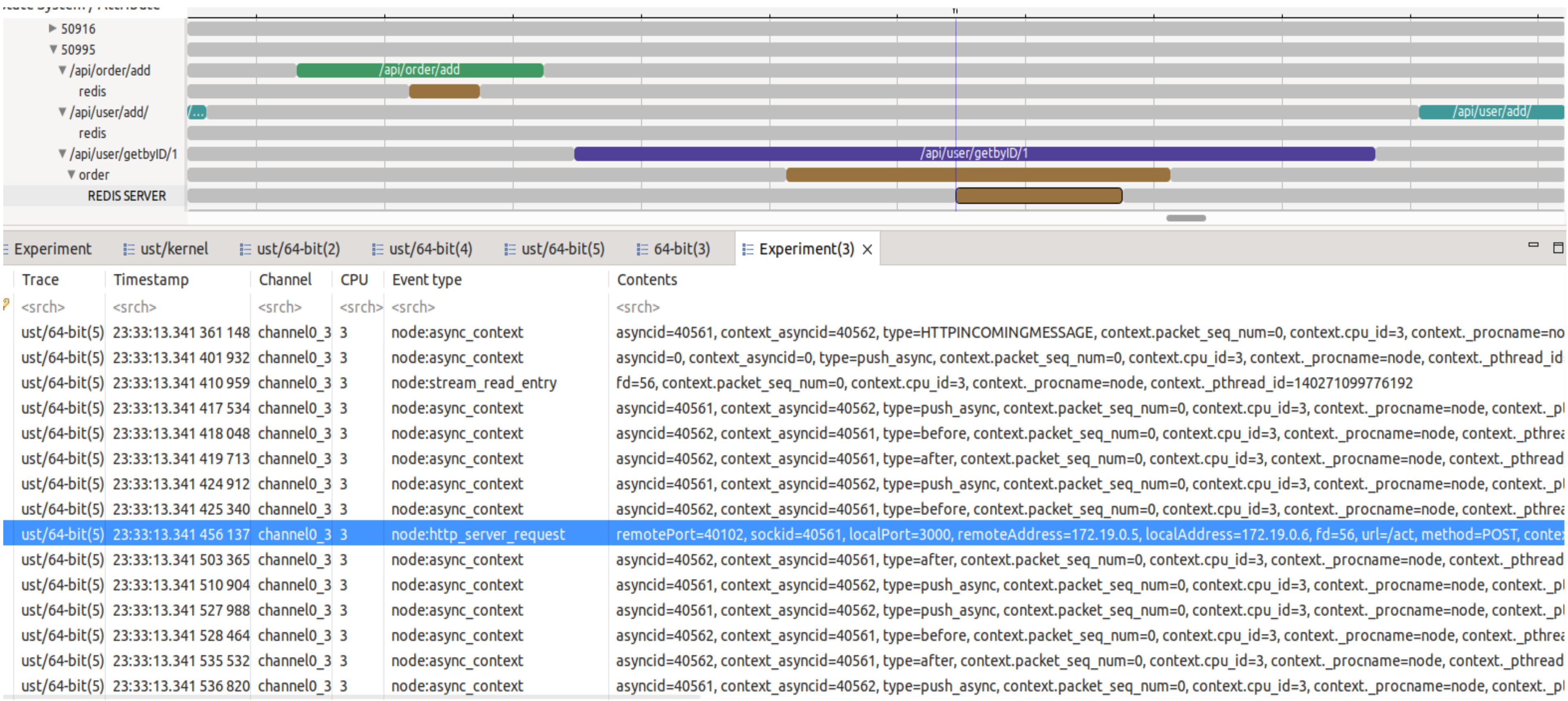
Example of a configuration file needed to run our analysis



```
–<configuration>
  –<entryPoint>
      <entryIP>172.19.0.4</entryIP>
      <entryPort>80</entryPort>
    </entryPoint>
  –<microservice id="1001">
      <address>172.19.0.1</address>
      <name>gateway</name>
    </microservice>
  –<microservice id="1002">
      <address>172.19.0.2</address>
      <name>Mysql</name>
    </microservice>
  –<microservice id="1003">
      <address>172.19.0.3</address>
      <name>redis</name>
    </microservice>
  –<microservice id="1004">
      <address>172.19.0.4</address>
      <name>mysqlgate</name>
    </microservice>
  –<microservice id="1005">
      <address>172.19.0.5</address>
      <name>order</name>
    </microservice>
  –<microservice id="1006">
```

Our approach

Example the state system view capturing microservices interactions

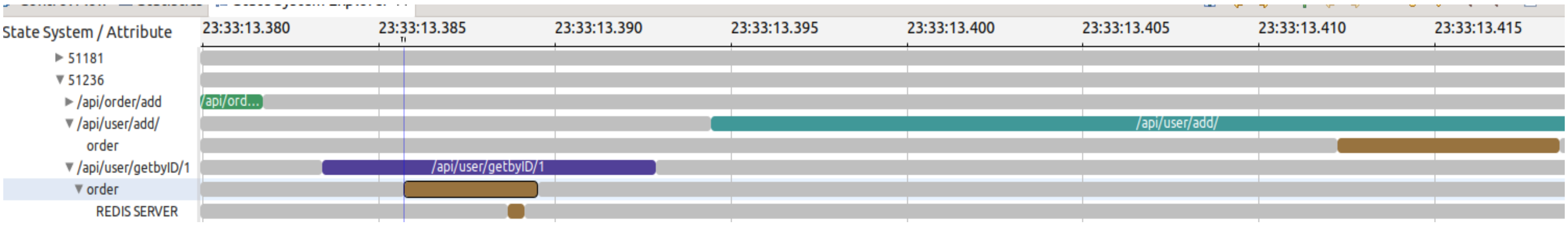


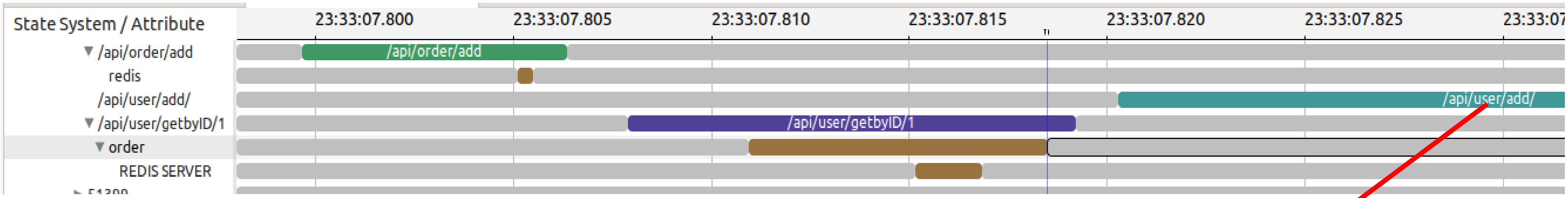
Trace	Timestamp	Channel	CPU	Event type	Contents
<srch>	<srch>	<srch>	<srch>	<srch>	<srch>
ust/64-bit(5)	23:33:13.341 361 148	channel0_3	3	node:async_context	asyncid=40561, context_asyncid=40562, type=HTTPINCOMINGMESSAGE, context.packet_seq_num=0, context.cpu_id=3, context._procname=no
ust/64-bit(5)	23:33:13.341 401 932	channel0_3	3	node:async_context	asyncid=0, context_asyncid=0, type=push_async, context.packet_seq_num=0, context.cpu_id=3, context._procname=node, context._pthread_id
ust/64-bit(5)	23:33:13.341 410 959	channel0_3	3	node:stream_read_entry	fd=56, context.packet_seq_num=0, context.cpu_id=3, context._procname=node, context._pthread_id=140271099776192
ust/64-bit(5)	23:33:13.341 417 534	channel0_3	3	node:async_context	asyncid=40561, context_asyncid=40562, type=push_async, context.packet_seq_num=0, context.cpu_id=3, context._procname=node, context._p
ust/64-bit(5)	23:33:13.341 418 048	channel0_3	3	node:async_context	asyncid=40562, context_asyncid=40561, type=before, context.packet_seq_num=0, context.cpu_id=3, context._procname=node, context._pthrea
ust/64-bit(5)	23:33:13.341 419 713	channel0_3	3	node:async_context	asyncid=40562, context_asyncid=40561, type=after, context.packet_seq_num=0, context.cpu_id=3, context._procname=node, context._pthread
ust/64-bit(5)	23:33:13.341 424 912	channel0_3	3	node:async_context	asyncid=40561, context_asyncid=40562, type=push_async, context.packet_seq_num=0, context.cpu_id=3, context._procname=node, context._p
ust/64-bit(5)	23:33:13.341 425 340	channel0_3	3	node:async_context	asyncid=40562, context_asyncid=40561, type=before, context.packet_seq_num=0, context.cpu_id=3, context._procname=node, context._pthrea
ust/64-bit(5)	23:33:13.341 456 137	channel0_3	3	node:http_server_request	remotePort=40102, sockid=40561, localPort=3000, remoteAddress=172.19.0.5, localAddress=172.19.0.6, fd=56, url=/act, method=POST, conte
ust/64-bit(5)	23:33:13.341 503 365	channel0_3	3	node:async_context	asyncid=40562, context_asyncid=40561, type=after, context.packet_seq_num=0, context.cpu_id=3, context._procname=node, context._pthread
ust/64-bit(5)	23:33:13.341 510 904	channel0_3	3	node:async_context	asyncid=40561, context_asyncid=40562, type=push_async, context.packet_seq_num=0, context.cpu_id=3, context._procname=node, context._p
ust/64-bit(5)	23:33:13.341 527 988	channel0_3	3	node:async_context	asyncid=40561, context_asyncid=40562, type=push_async, context.packet_seq_num=0, context.cpu_id=3, context._procname=node, context._p
ust/64-bit(5)	23:33:13.341 528 464	channel0_3	3	node:async_context	asyncid=40562, context_asyncid=40561, type=before, context.packet_seq_num=0, context.cpu_id=3, context._procname=node, context._pthrea
ust/64-bit(5)	23:33:13.341 535 532	channel0_3	3	node:async_context	asyncid=40562, context_asyncid=40561, type=after, context.packet_seq_num=0, context.cpu_id=3, context._procname=node, context._pthread
ust/64-bit(5)	23:33:13.341 536 820	channel0_3	3	node:async_context	asyncid=40561, context_asyncid=40562, type=push_async, context.packet_seq_num=0, context.cpu_id=3, context._procname=node, context._p

Our approach

Symptomatic execution of the system

We use the



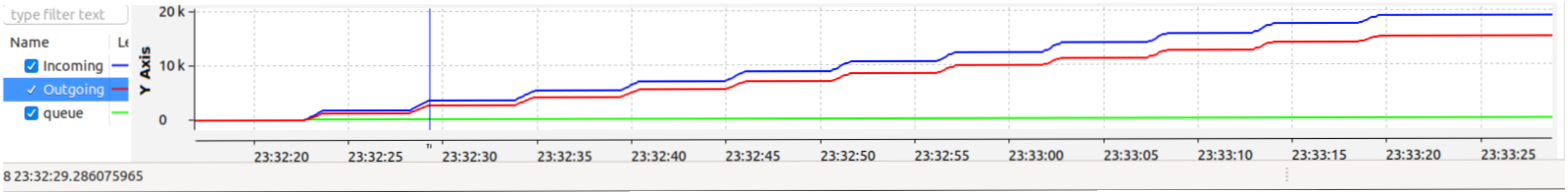


Faulty request. Returns an error, spans no other microservices

Our approach

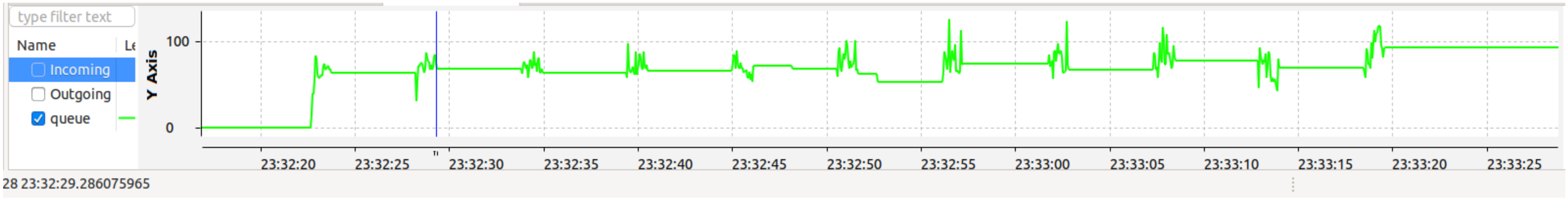
Some other available views

Incoming and outgoing request flow



8 23:32:29.286075965

Queued requests



28 23:32:29.286075965

Conclusion & Ongoing work

- Any Nodejs Restful microservice can be run transparently with our tool
- Docker images of Nodejs V16, V17, V18 are available with the instrumentation
- Very low overhead is achieved
- The analysis may be apply to any distributed Nodejs applications
- Low level correlation with kernel events
- Docker name-space metadata inclusion for event correlation and complex analysis

Thank you