Distributed Tracing in Kubernetes

Piotr Gaczkowski
IT CONSULTANT

@doomhammerng doomhammer.info

Overview

Distributed applications

Passing context for traces

Analyzing traces with Jaeger

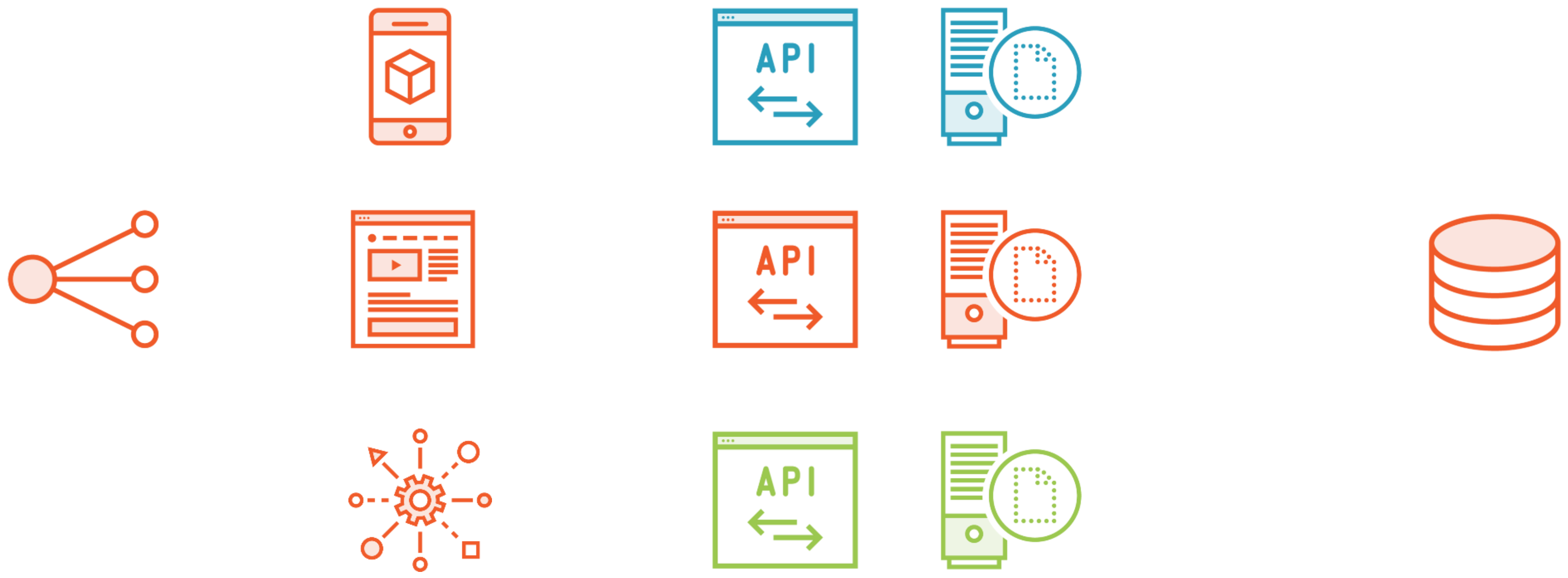
Distributed Applications

Even “Simple Web Applications” are distributed

Applications in Kubernetes are distributed by design

Microservices increase the distribution of the components

Distributed Applications



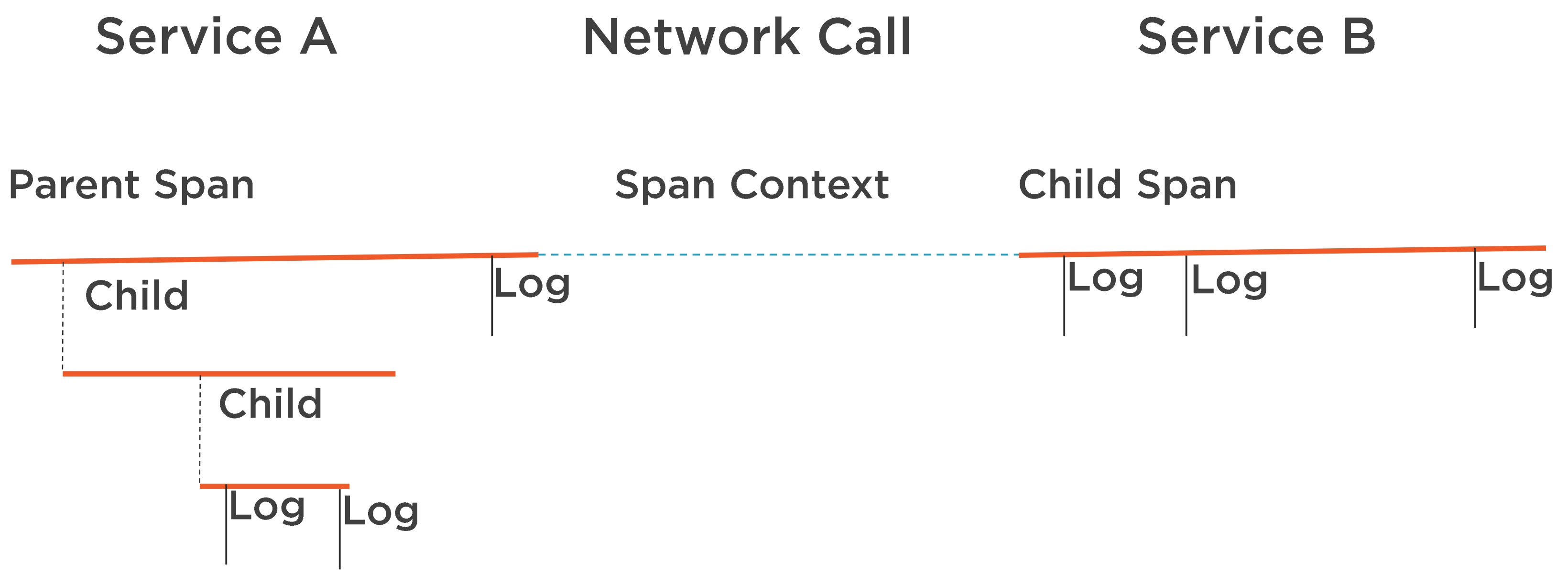
Distributed Tracing

Allows correlation of events from different parts of the system

Particularly fitting for microservices

Helps with debugging and optimizing the code

Distributed Tracing Model



Source: https://opentracing.io/docs/overview/

Trace

A visualization of the life of a request as it moves through a distributed system.

Span

Individual unit of work done in a distributed system

Corresponds to a named and timed operation

Refers to other spans so they can be assembled into one complete **Trace**

Tags, Logs, and Baggage

Tags are key:value pairs that enable user-defined annotation of spans in order to query, filter, and comprehend trace data

Logs are key:value pairs that are useful for capturing *timed* log messages and other debugging or informational output from the application itself

The **SpanContext** carries data across process boundaries

Baggage Items are key:value pairs that cross process boundaries

OpenTracing

A standard for distributed tracing proposed by Jaeger

Supports many tracers

Supports many languages

Supports plugins for existing solutions

Vendor-neutral API and instrumentation

OpenTracing: Supported Tracers

- CNCF Jaeger
- LightStep
- Instana
- Apache SkyWalking
- inspectIT
- stagemonitor
- Datadog
- Wavefront by VMware
- Elastic APM

OpenTracing: Supported Languages

- Go
- C++
- C#
- Java
- Javascript
- Objective-C
- PHP
- Python
- Ruby

Distributed Tracing Tools

Jaeger

OpenZipkin

Apache SkyWalking

Jaeger

Cloud Native Computing Foundation graduated project

Pluggable backend (Elasticsearch, Cassandra, Kafka)

Native compatibility with OpenTracing

OpenZipkin

Longer in the market

Uses proprietary API or OpenTracing

Hosted by the Apache Foundation

Apache SkyWalking

Hosted by the Apache Foundation

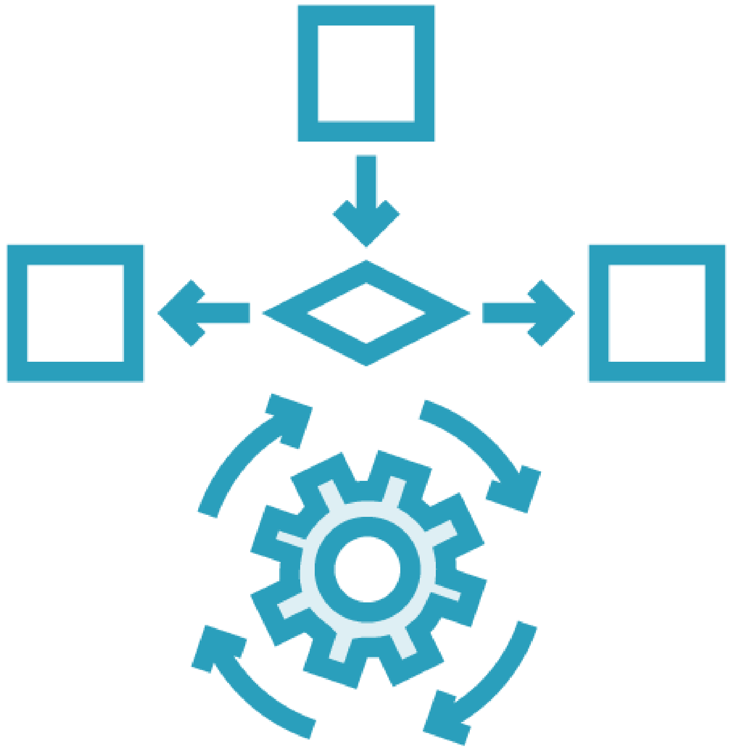
Native compatibility with OpenTracing

Analizes traces and metrics

Instrumenting an Application

- Check if the dependencies are already instrumented
- Start with areas of value
- Cover end-to-end instrumentation
- Look for the reuse potential

Passing Context in Process



`start_span` from OpenTracing creates a new span

The `child_of` parameter makes it a child span

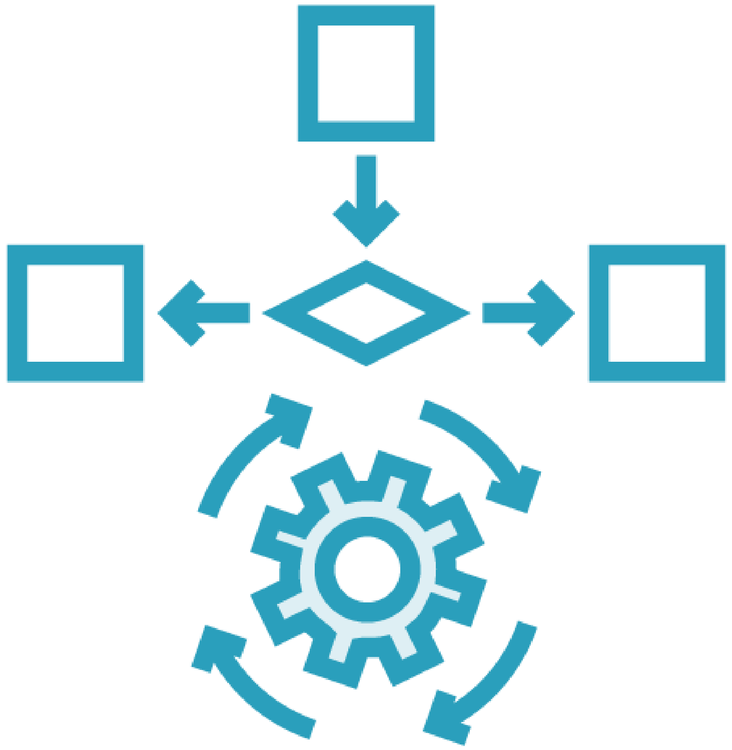
This requires passing a span object between function call

In Python, `start_active_span` automatically attaches the span to the parent

Passing Context in Process

```
def main(r=request):

    span_ctx = tracer.extract(Format.HTTP_HEADERS, request.headers)

    with tracer.start_active_span('processRun', finish_on_close=True,
child_of=span_ctx) as scope:

        request_id = str(uuid.uuid4())

        app.logger.debug("Received request %s to /", request_id)



        with tracer.start_active_span('validatePayload',
finish_on_close=True) as scope:

            content = r.json
```

Passing Context for RPC Calls



OpenTracing allows injecting span context into an RPC call

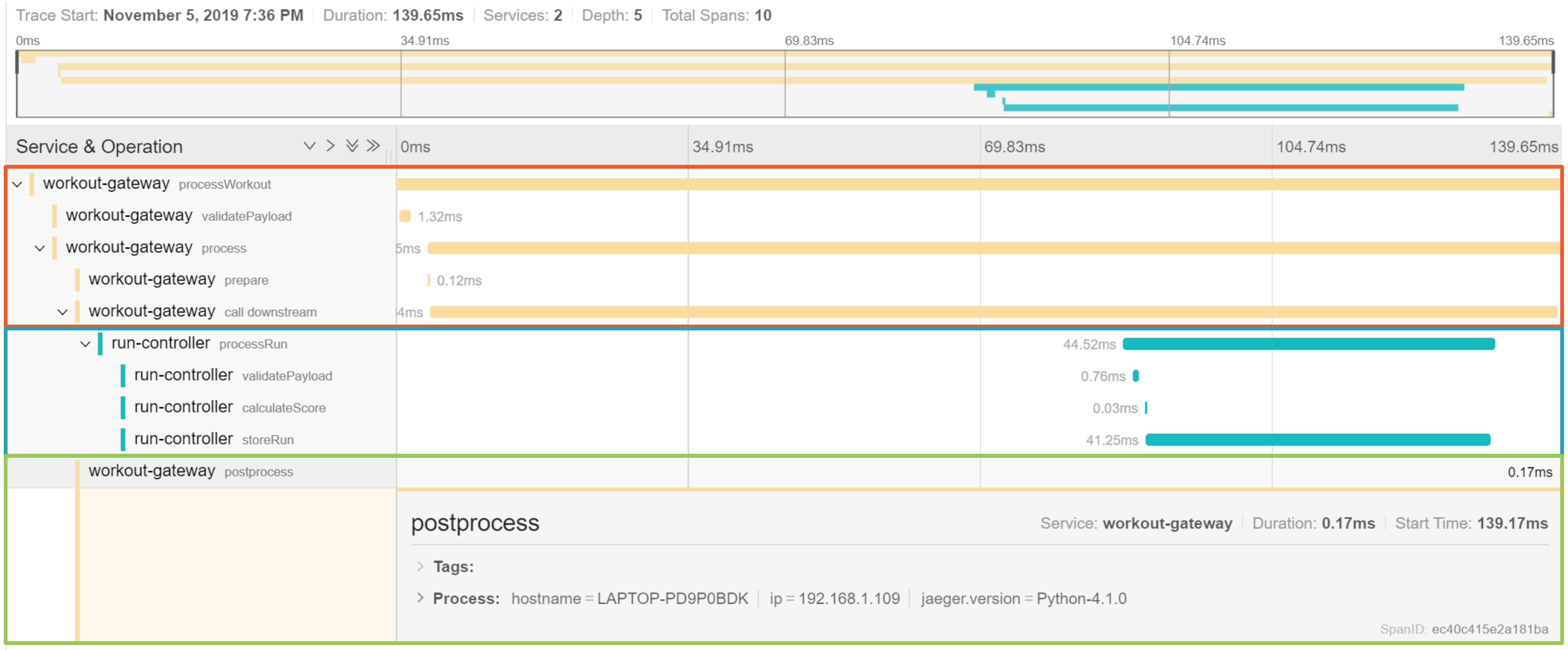
Different RPC protocols are supported including HTTP (REST) and gRPC

Injected context can be extracted on the remote end

Passing Context for RPC Calls

```
with tracer.start_active_span('call downstream', finish_on_close=True) as
scope:

    headers = {}

    tracer.inject(scope.span, Format.HTTP_HEADERS, headers)

    payload = {**workout.as_dict(), "workout_id": workout_id}


    r = requests.post(RUN_CONTROLLER_URL, json=payload, headers=headers)
```

Analyzing Traces with Jaeger



Trace Start: **November 5, 2019 7:36 PM** | Duration: **139.65ms** | Services: **2** | Depth: **5** | Total Spans: **10**

0ms 34.91ms 69.83ms 104.74ms 139.65ms

Service & Operation | 0ms | 34.91ms | 69.83ms | 104.74ms | 139.65ms

```
workout-gateway  processWorkout
  workout-gateway  validatePayload        1.32ms
  workout-gateway  process                5ms
    workout-gateway  prepare              0.12ms
    workout-gateway  call downstream      4ms
      run-controller  processRun          44.52ms
        run-controller  validatePayload   0.76ms
        run-controller  calculateScore    0.03ms
        run-controller  storeRun          41.25ms
    workout-gateway  postprocess          0.17ms
```

postprocess Service: **workout-gateway** | Duration: **0.17ms** | Start Time: **139.17ms**

> **Tags:**

> **Process:** hostname = LAPTOP-PD9P0BDK | ip = 192.168.1.109 | jaeger.version = Python-4.1.0

SpanID: ec40c415e2a181ba

Demo

Deploy Jaeger

Collect traces

Use Jaeger UI to identify performance problems

Summary

Kubernetes applications are distributed by default

Tracing makes it possible to understand the workflow of a distributed system

Precise timing is useful to investigate performance issues