ROOKOUT

Distributed tracing

- Jaeger 101 -

About me

- Worked at **eBay**
- Worked at **Forter** as a backend engineer
- Joined **Rookout** as a first developer and production engineer
- @itielshwartz on both Github and Twitter
- Personal blog at: https://etlsh.com

Agenda

Intro

1. State of mind for this Meetup (super important!)
2. What is distributed tracing and why do I need it?
3. What is OpenTracing?
4. What is Jaeger?

Zero to hero using Jaeger

1. “Hello world” example
2. Jaeger terminology
3. Full-blown distributed app

Wrap-up

1. Demo wrap-up
2. Jaeger architecture
3. OpenTracing’s secret ability

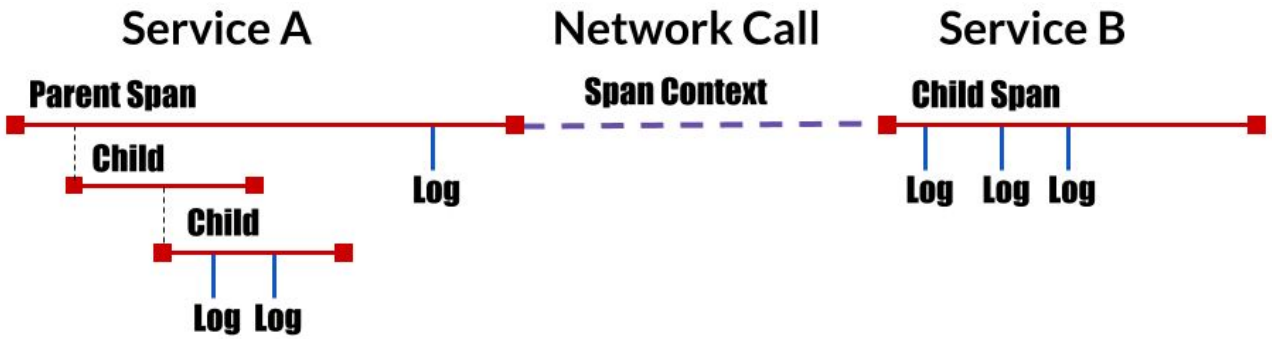
Before we begin (State of mind)

- The system will fail
- Your code is not perfect
- Other people's code is even less perfect
- Practice new tools in the daytime -- don't start using them when in crisis mode
- The system will fail
- Each minute you spend adding logs and metrics can reduce your Mean Time to Resolve (MTTR)
- Keep in mind that the developer who's going to get paged isn't the one that wrote the code
- Try to be nice to him - he is going to need it
- The system will fail

As you can probably see, I (tried to) emphasize the fact that your system is going to fail. This **DOESN'T** mean that I think you write bad code - only that we usually have much more trust in our code/infra then we should :)

What is distributed tracing?

With distributed tracing, we can track requests as they pass through multiple services, emitting timing and other metadata throughout, and this information can then be reassembled to provide a complete picture of the application's behavior at runtime - buoyant



Mental model of distributed tracing - Opentracing

Do I need distributed tracing?

As companies move from monolithic to multi-service architectures, existing techniques for debugging and profiling begin to break down.

Previously, troubleshooting could be accomplished by isolating a single instance of the monolith and reproducing the problem.

With microservices, this approach is no longer feasible, because no single service provides a complete picture of the performance or correctness of the application as a whole.

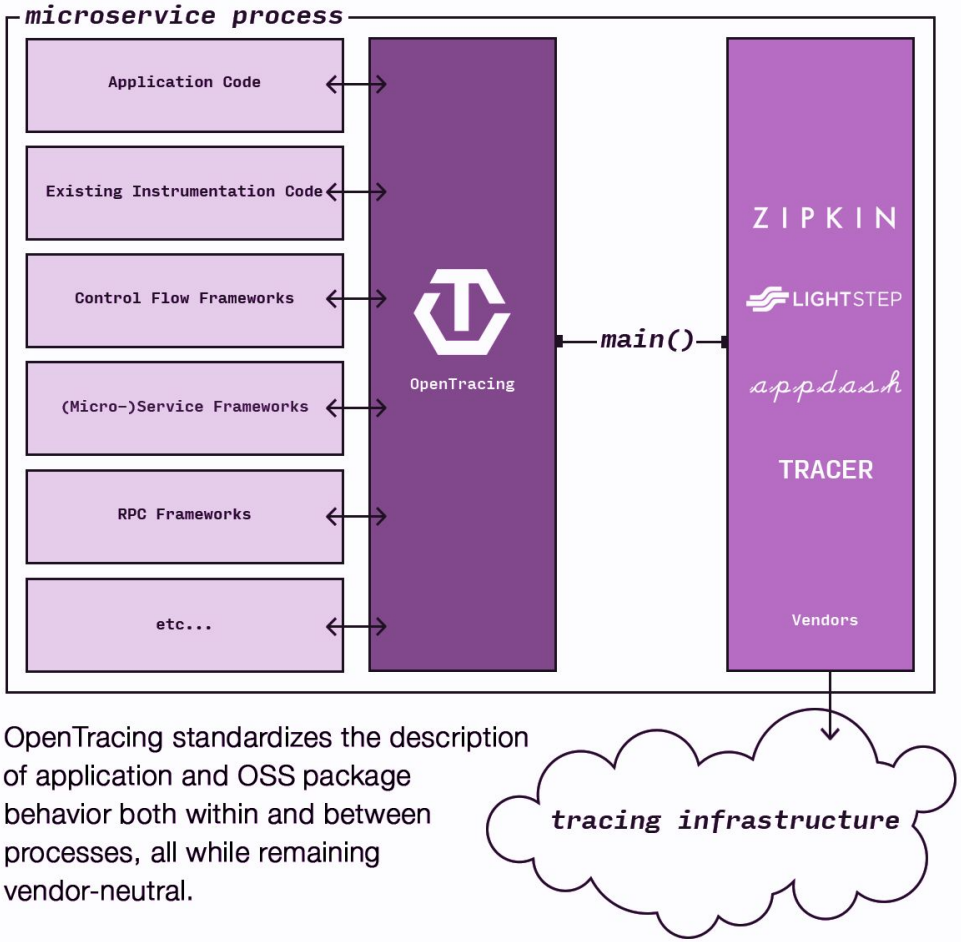
We need new tools to help us manage the real complexity of operating distributed systems at scale. - buoyant

What is OpenTracing?

The problem is that distributed tracing has long harbored a dirty secret: the necessary source code instrumentation has been complex, fragile, and difficult to maintain.

This is the problem that OpenTracing resolves.
Through standard, consistent APIs in many languages (Java, Javascript, Go, Python, C#, others), the OpenTracing project gives developers clean, declarative, testable, and vendor-neutral instrumentation.

OpenTracing has focused on standards for explicit software instrumentation.



OpenTracing standardizes the description of application and OSS package behavior both within and between processes, all while remaining vendor-neutral.

What is Jaeger?

Jaeger, inspired by Dapper and OpenZipkin, is a distributed tracing system released as open source by Uber Technologies.

It can be used for monitoring microservices-based distributed systems:

- Distributed context propagation
- Distributed transaction monitoring
- Root cause analysis
- Service dependency analysis
- Performance / latency optimization

Getting started - The Monolith

https://github.com/itielshwartz/jaeger-hello-world/tree/step-1-the-monolith

Getting started - Monolith going wild

https://github.com/itielshwartz/jaeger-hello-world/tree/step-2-the-monolith-going-wild

Jaeger terminology - Span/ Trace

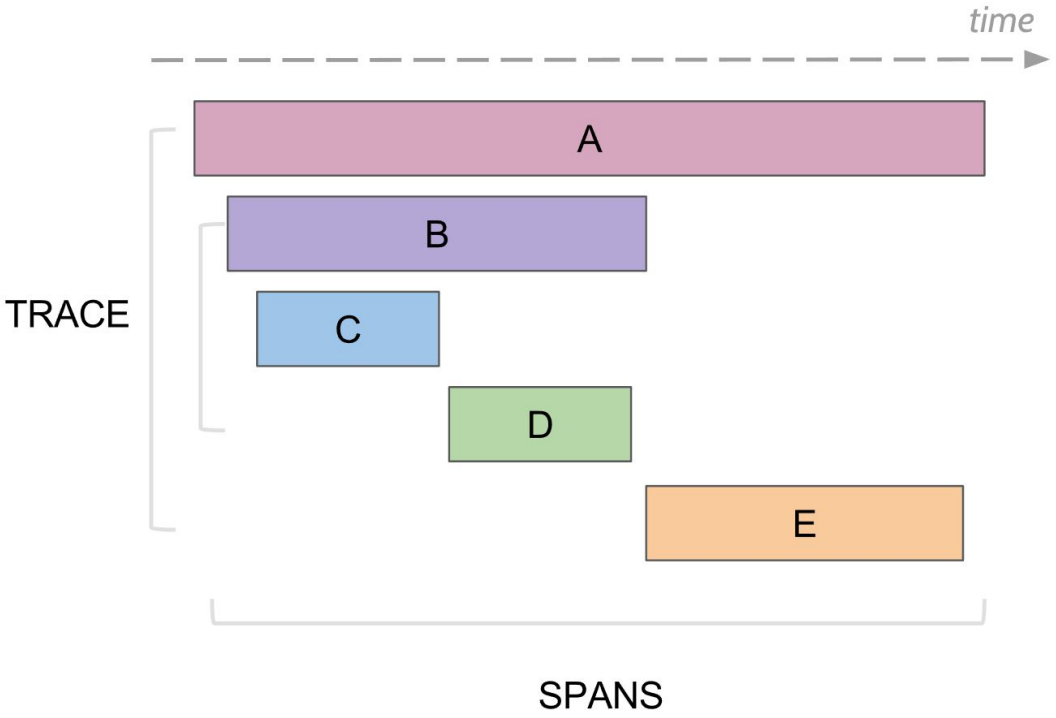
Span

A **span** represents a logical unit of work in Jaeger that has an operation name, the start time of the operation, and the duration. Spans may be nested and ordered to model causal relationships.

Trace

A **trace** is a data/execution path through the system, and can be thought of as a directed acyclic graph of spans.

Jaeger terminology - Span/ Trace



Getting started - Adding Jaeger

https://github.com/itielshwartz/jaeger-hello-world/tree/step-3-adding-jaeger

Config Jaeger part II - Multiple spans

https://github.com/itielshwartz/jaeger-hello-world/tree/step-4-multiple-spans

Jaeger architecture - Tag/Log

The recommended solution is to annotate spans with tags or logs.

Tag:

A *tag* is a key-value pair that provides certain metadata about the span.

Log:

A *log* is similar to a regular log statement, it contains a timestamp and some data, but it is associated with span from which it was logged.

When and why?

When should we use tags vs. logs? Tags are meant to describe attributes of the span that apply to the whole duration of the span. For example, if a span represents an HTTP request, then the URL of the request should be recorded as a tag because it does not make sense to think of the URL as something that's only relevant at different points in time on the span. On the other hand, if the server responded with a redirect URL, logging it would make more sense since there is a clear timestamp associated with such event. The OpenTracing Specification provides guidelines called Semantic Conventions for recommended tags and log fields.

https://github.com/yurishkuro/opentracing-tutorial/tree/master/python/lesson01#annotate-the-trace-with-tags-and-logs

Config Jaeger part III - Tags and Log

https://github.com/itielshwartz/jaeger-hello-world/tree/step-5-tags-and-logs

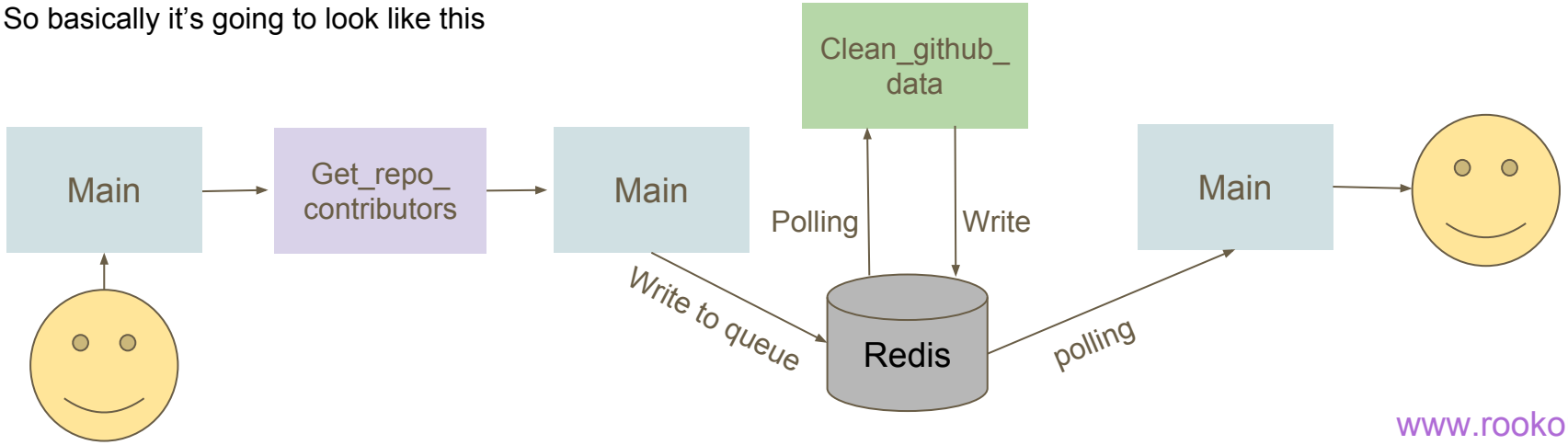
Going distributed

Until now we had single server (which kind of defies the purpose of distributed tracing).
Now let's split our monolith into small parts: We will still have a main server (customer facing) but now we'll split get_repo_contributors and clean_github_data into two different servicea.

Get_repo_contributors will be a flask server (same as our main)

Clean_github_data will consume data from redis (pushed to it by the master)

So basically it's going to look like this



Going distributed - Single span

https://github.com/itielshwartz/jaeger-hello-world/tree/step-6-distribute-single-span

Going distributed - Multiple span

https://github.com/itielshwartz/jaeger-hello-world/tree/step-7-distribute-multiple-spans

Demo wrap-up

We now have successfully transformed a monolith beast into a set of small microservices -- without losing visibility.

The nice thing about OpenTracing is that it allow us to move from Jaeger to datadog to other solutions (almost) without needing to rewrite our code.

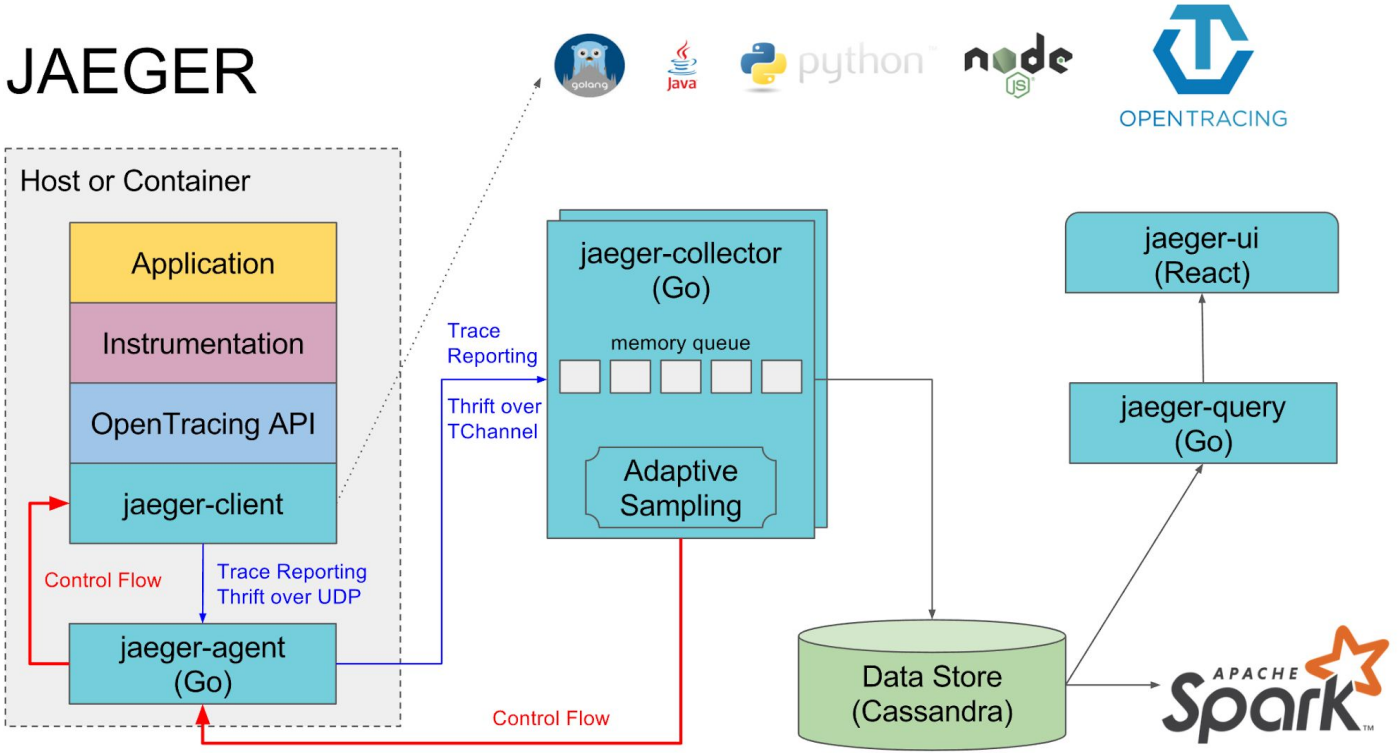
The other cool thing about it is that you **don't need to do everything I just did in this demo**!

There are official wrappers for most of the common frameworks. Those tools allow you you to integrate with OpenTracing and Jaeger without needing to think "How do I pass the headers inside the request?" or " How do I read the headers to start a new span?"

Examples:

- urllib2
- requests
- SQLAlchemy
- MySQLdb
- Tornado HTTP client
- redis
- Flask
- Django
- More

Jaeger Architecture



Jaeger architecture

Agent

The Jaeger **agent** is a network daemon that listens for spans sent over UDP, which it batches and sends to the collector. It is designed to be deployed to all hosts as an infrastructure component. The agent abstracts the routing and discovery of the collectors away from the client.

Collector

The Jaeger **collector** receives traces from Jaeger agents and runs them through a processing pipeline. Currently our pipeline validates traces, indexes them, performs any transformations, and finally stores them.

Jaeger's storage is a pluggable component which currently supports Cassandra and ElasticSearch.

Query

Query is a service that retrieves traces from storage and hosts a UI to display them.

OpenTracing's secret ability

Context propagation

With OpenTracing instrumentation in place, we can support general purpose *distributed context propagation* where we associate some metadata with the transaction and make that metadata available anywhere in the distributed call graph. In OpenTracing this metadata is called *baggage*, to highlight the fact that it is carried over in-band with all RPC requests, just like baggage. opentracing-tutorial

The client may use the Baggage to pass additional data to the server and any other downstream server it might call.

```
# client side
span.context.set_baggage_item('auth-token', '.....')

# server side (one or more levels down from the client)
token = span.context.get_baggage_item('auth-token')
```

Questions?