How to Properly Blame Things for Causing Latency

An introduction to Distributed Tracing and Zipkin

@adrianfcole
works at Pivotal
works on Zipkin

Introduction

@adrianfcole

spring cloud at pivotal
focused on distributed tracing
helped open zipkin

Understanding Latency

Understanding Latency

`Unifying theory: Everything is based on events`

Logging - recording events

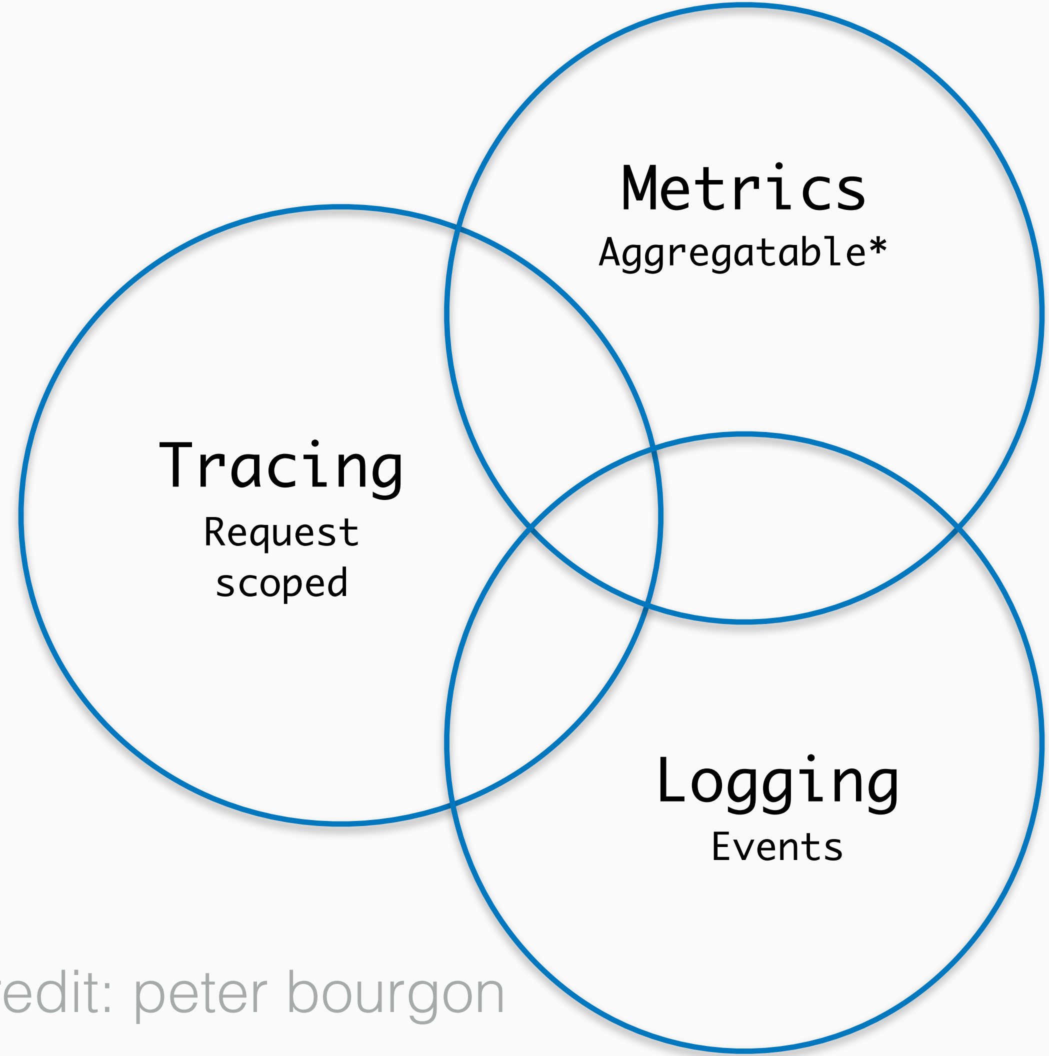
Metrics - data combined from measuring events

Tracing - recording events with causal ordering

credit: coda hale

Different tools

Different focus



credit: peter bourgon

Let's use latency to compare a few tools

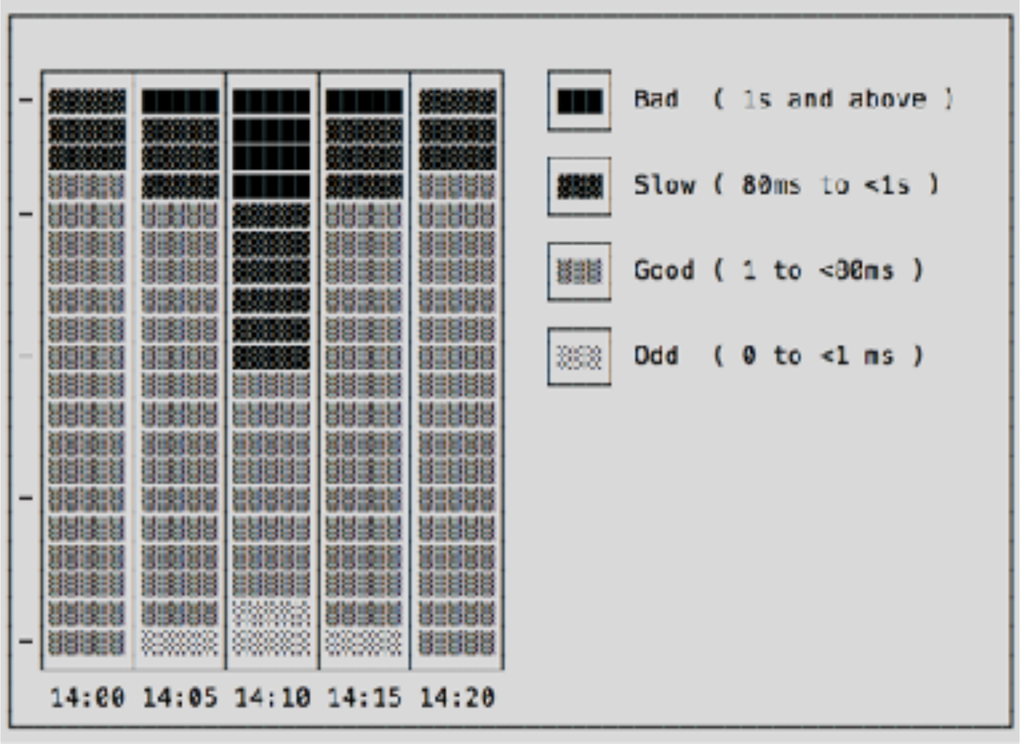
- Log - event (response time)
- Metric - value (response time)
- Trace - tree (response time)

Logs show response time

[20/Apr/2017:14:19:07 +0000] "GET / HTTP/1.1" 200 7918 "" "Mozilla/5.0 (X11; U; Linux i686; en-US; rv: 1.8.1.11) Gecko/20061201 Firefox/2.0.0.11 (Ubuntu-feisty)" **0/95491**

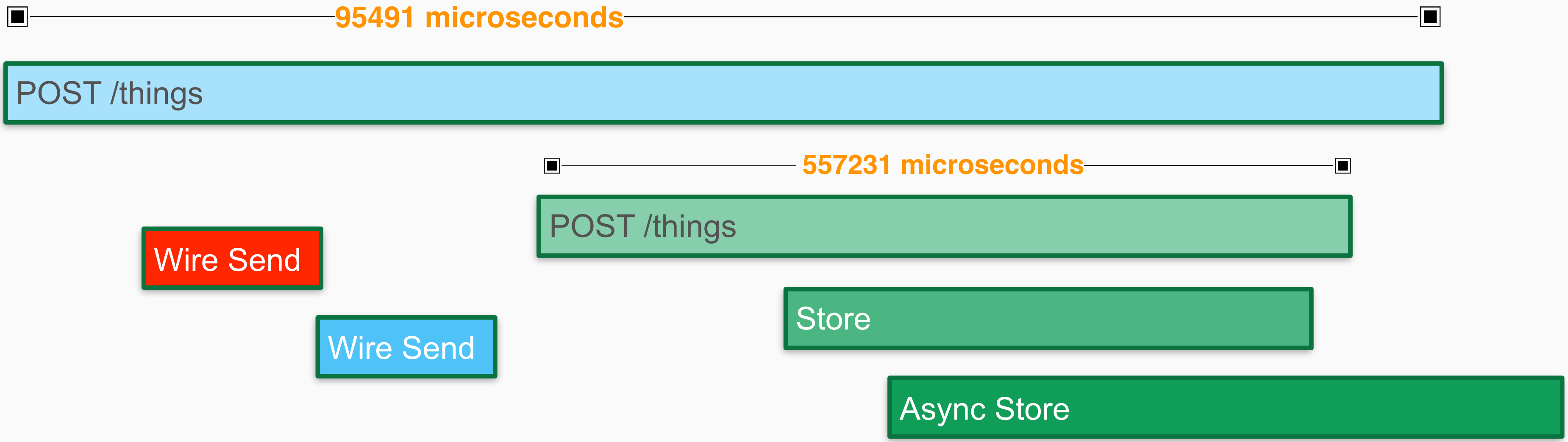
`Look! this request took 95 milliseconds!`

Metrics show response time



Is 95 milliseconds slow?
How fast were most requests at 14:19?

Traces show response time



What caused the request to take 95 milliseconds?

First thoughts….

Log - easy to “grep”, manually read

Metric - can identify trends

Trace - identify cause across services

`You can link together: For example add trace ID to logs`

Distributed Tracing

Distributed Tracing commoditizes knowledge

Distributed tracing systems collect end-to-end latency graphs (traces) in near real-time.

You can compare traces to understand why certain requests take longer than others.

Distributed Tracing Vocabulary

A **Span** is an individual operation that took place. A span contains **timestamped events** and **tags**.

A **Trace** is an end-to-end latency graph, composed of spans.

Tracers records spans and passes context required to connect them into a trace

Instrumentation uses a tracer to record a task such as an http request as a span

A Span is an individual operation

Operation

POST /things

wombats:10.2.3.47:8080

Events

Server Received a Request

Server Sent a Response

Tags

remote.ipv4	1.2.3.4
http.request-id	abcd-ffe
http.request.size	15 MiB
http.url	…&features=HD-uploads

Tracing is logging important events

POST /things

POST /things

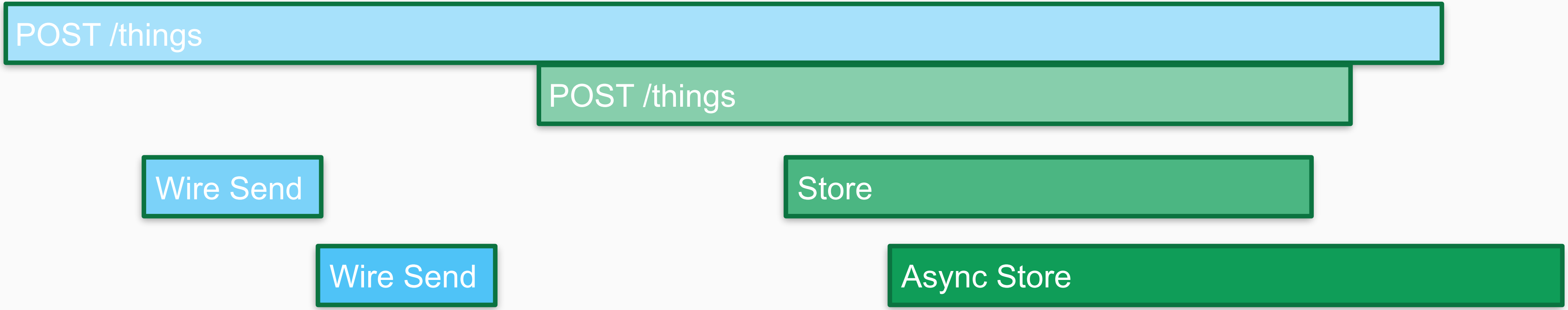
Wire Send

Store

Wire Send

Async Store

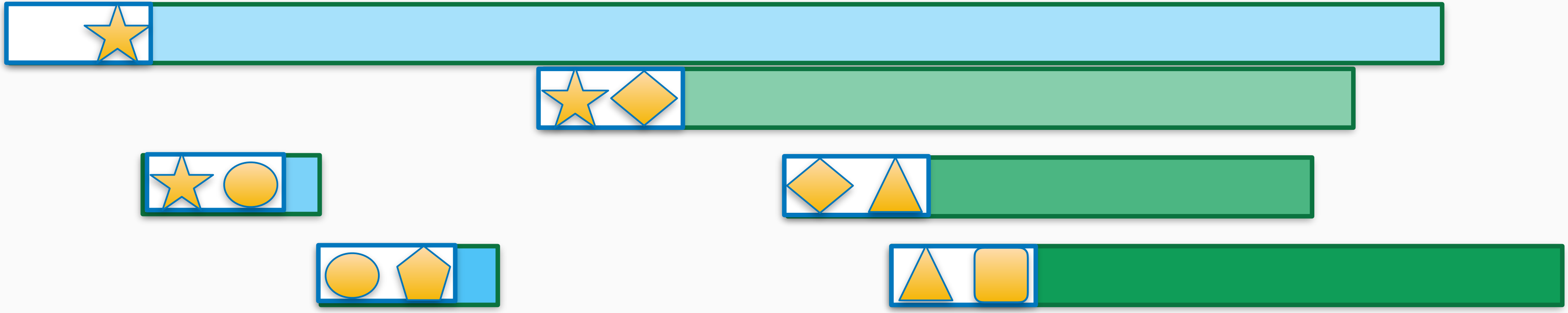
Tracers record time, duration and host



Tracers don’t decide what to record, instrumentation does.. we’ll get to that

Tracers send trace data out of process

Tracers propagate IDs in-band,
to tell the receiver there's a trace in progress



Completed spans are reported out-of-band,
to reduce overhead and allow for batching

Tracer == Instrumentation?

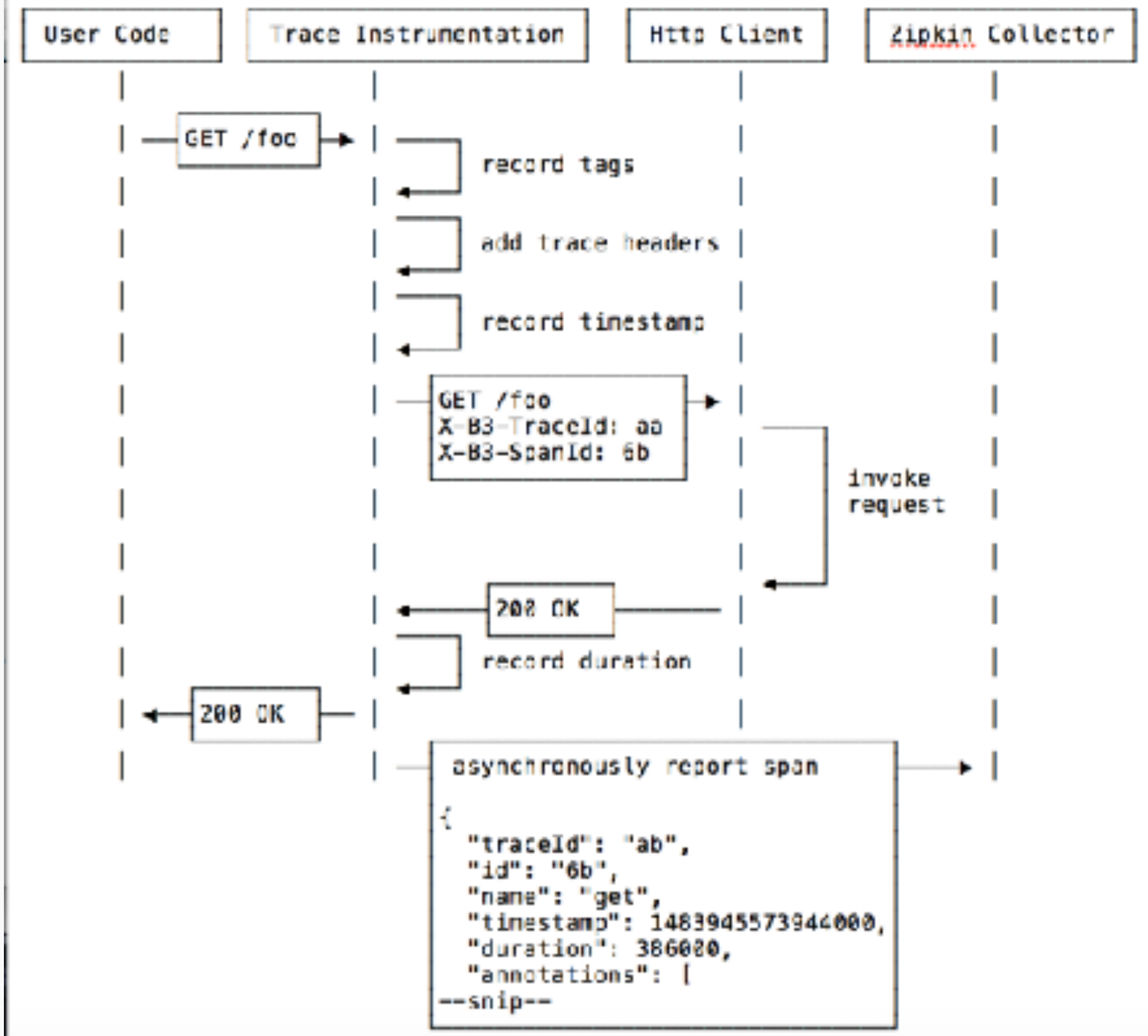
A tracer is a utility library, similar to metrics or logging libraries. It is a mechanism uses to trace an operation. Instrumentation is the what and how.

For example, instrumentation for ApacheHC and OkHttp record similar data with a tracer. How they do that is library specific.

Instrumentation is usually invisible to users

Instrumentation decides what to record

Instrumentation decides how to propagate state



Tracing affects your production requests

Tracing affects your production requests, causing size and latency overhead. Tracers are carefully written to not cause applications to crash. Instrumentation is carefully written to not slow or overload your requests.

- Tracers propagate structural data in-band, and the rest out-of-band

- Instrumentation has data and sampling policy to manage volume

- Often, layers such as HTTP have common instrumentation and/or models

Tracing Systems are Observability Tools

Tracing systems collect, process and present data reported by tracers.

- aggregate spans into trace trees
- provide query and visualization focused on latency
- have retention policy (usually days)

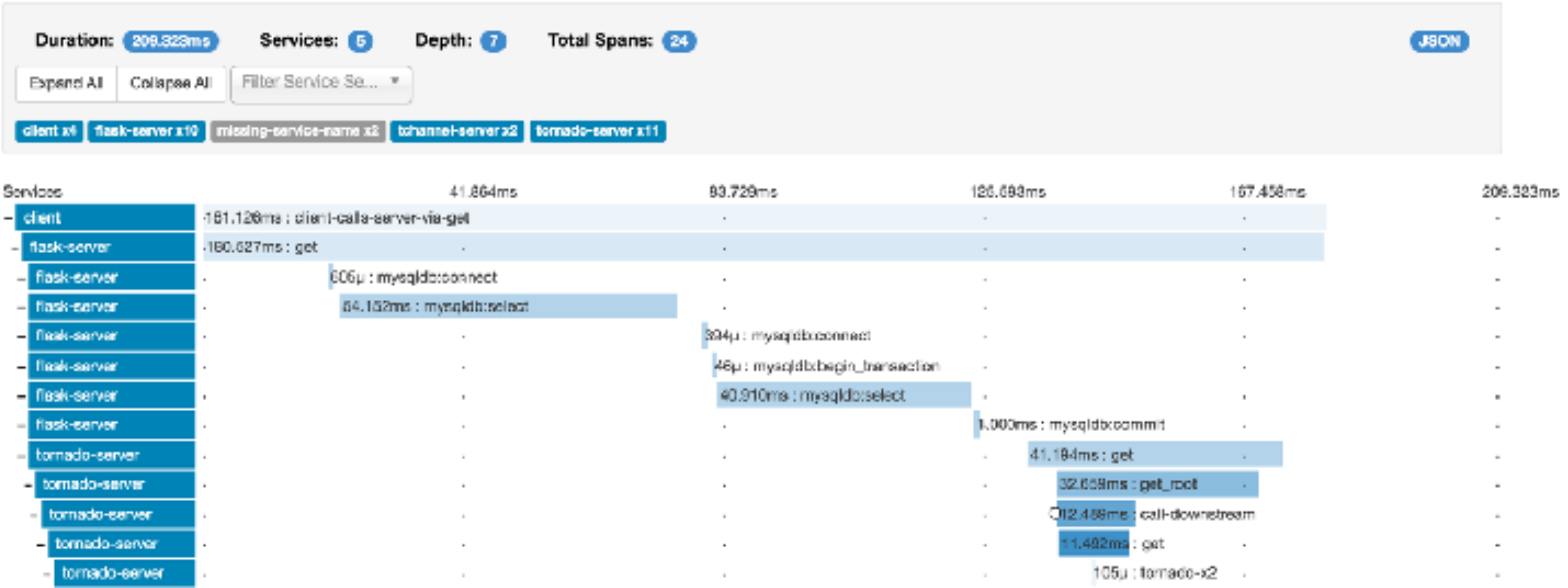
Protip: Tracing is not just for latency

Some wins unrelated to latency

- Understand your architecture
- Find who’s calling deprecated services
- Reduce time spent on triage

Zipkin

Zipkin is a distributed tracing system



Zipkin lives in GitHub

Zipkin was created by Twitter in 2012 based on the Google Dapper paper. In 2015, OpenZipkin became the primary fork.

OpenZipkin is an org on GitHub. It contains tracers, OpenApi spec, service components and docker images.

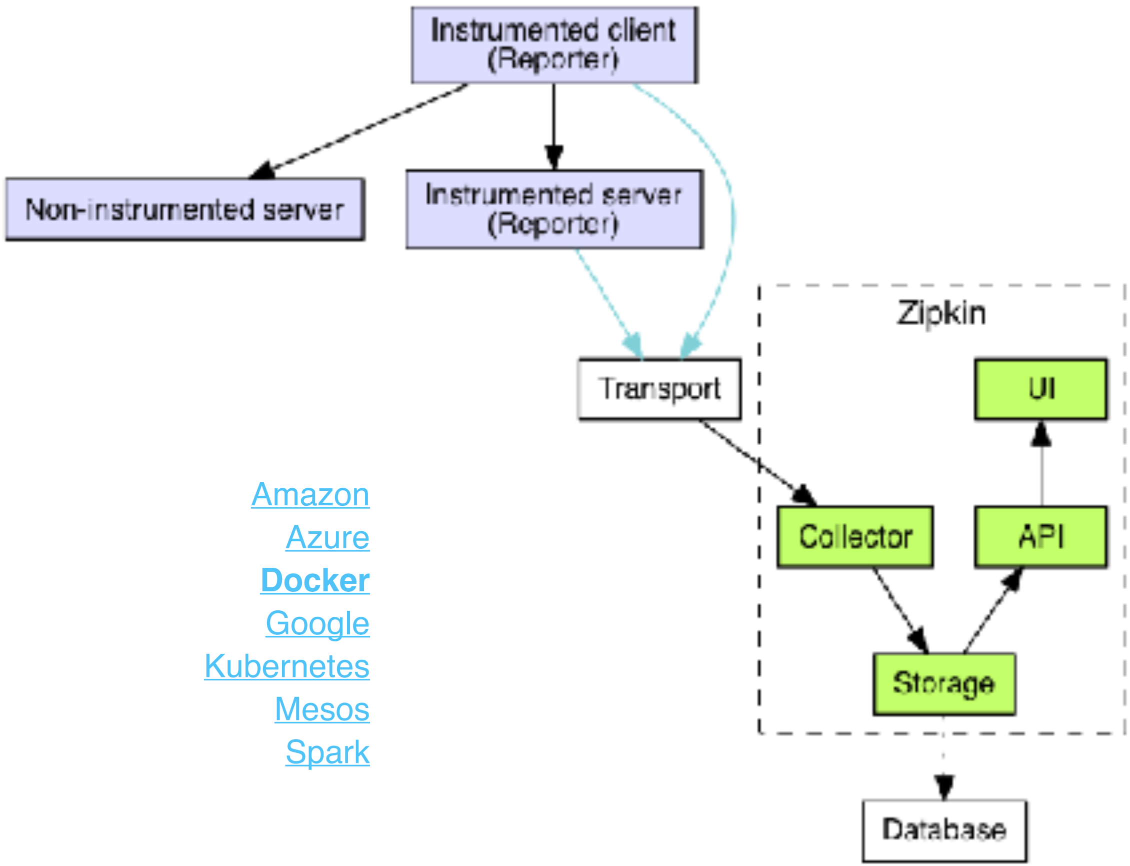
https://github.com/openzipkin

Zipkin Architecture

Tracers **report** spans HTTP or Kafka.

Servers **collect** spans, storing them in MySQL, Cassandra, or Elasticsearch.

Users **query** for traces via Zipkin's Web UI or Api.



Amazon
Azure
Docker
Google
Kubernetes
Mesos
Spark

Zipkin has starter architecture

Tracing is new for a lot of folks.

For many, the MySQL option is a good start, as it is familiar.

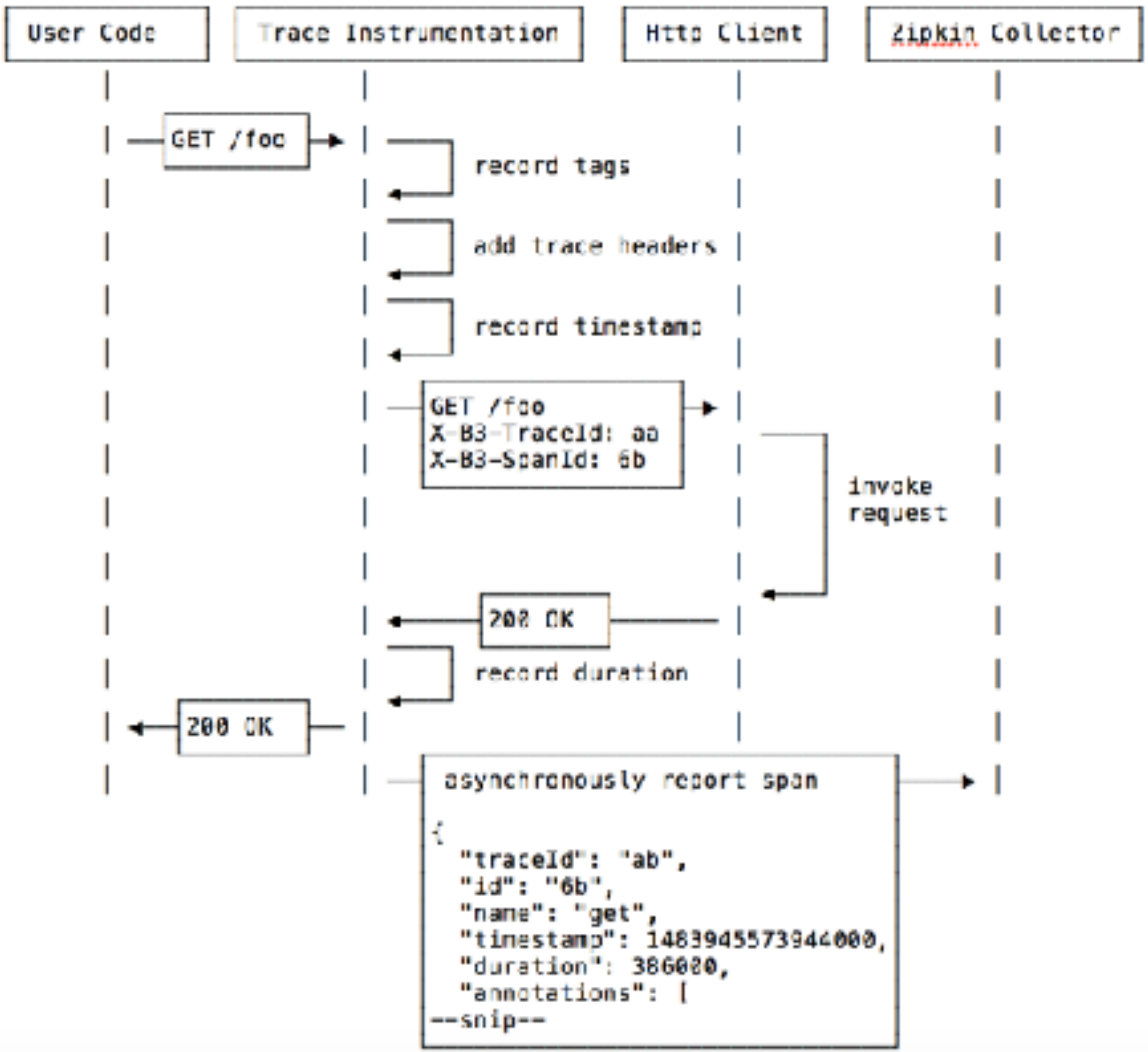
```
services:
  storage:
    image: openzipkin/zipkin-mysql
    container_name: mysql
    ports:
      - 3306:3306
  server:
    image: openzipkin/zipkin
    environment:
      - STORAGE_TYPE=mysql
      - MYSQL_HOST=mysql
    ports:
      - 9411:9411
    depends_on:
      - storage
```

Zipkin can be as simple as a single file

```
$ curl -SL 'https://search.maven.org/remote_content?g=io.zipkin.java&a=zipkin-server&v=LATEST&c=exec' > zipkin.jar
$ SELF_TRACING_ENABLED=true java -jar zipkin.jar
                                    ********
                                  **        **
                                 *            *
                                **            **
                                **            **
                                 **          **
                                  **        **
                                    ********
                                      ****
                                      ****
        ****                          ****
     ******                           ****                                 ***
  ****************************************************************************
    *******                           ****                                 ***
        ****                          ****
                                       **
                                       **


             *****      **     *****     ** **       **     **   **
               **       **     **  *     ***         **     **** **
              **        **     *****     ****        **     **  ***
             ******     **     **        **  **      **     **   **

:: Powered by Spring Boot ::         (v1.5.4.RELEASE)

2016-08-01 18:50:07.098  INFO 8526 --- [           main] zipkin.server.Zi
example/zipkin.jar started by acole in /Users/acole/oss/sleuth-webmvc-exa
-snip-
```

```
$ curl -s localhost:9411/api/v2/services|jq .
[
  "gateway"
]
```

How data gets to Zipkin —>



Looks easy right?

Brave: the most popular Zipkin Java tracer

- **Brave** - OpenZipkin's java library and instrumentation
  - Layers under projects like Ratpack, Dropwizard, Play
- **Spring Cloud Sleuth** - automatic tracing for Spring Boot
  - Includes many common spring integrations
  - Starting in version 2, Sleuth is a layer over Brave!

`c, c#, erlang, javascript, go, php, python, ruby, too`

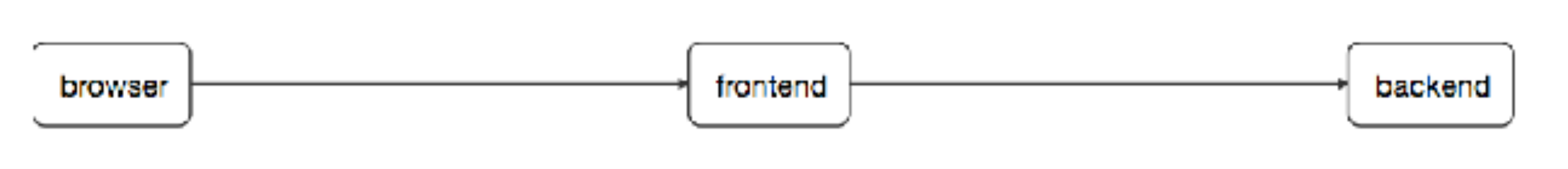
Some notable open source tracing libraries

- **OpenCensus** - Observability SDK (metrics, tracing, tags)
  - Most notably, gRPC’s tracing library
  - Includes exporters in Zipkin format and B3 propagation format
- **OpenTracing** - trace instrumentation library api definitions
  - Bridge to Zipkin tracers available in Java, Go and PHP
- **SkyWalking** - APM with a java agent developed in China
  - Work in progress to send trace data to zipkin

Demo

Distributed Tracing across multiple apps

A web browser calls a service that calls another.



Zipkin will show how long the whole operation took, as well how much time was spent in each service.

openzipkin/zipkin-js

spring-cloud-sleuth

zipkin-js JavaScript

JavaScript referenced in index.html fetches an api request. The fetch function is traced via a Zipkin wrapper.

openzipkin/zipkin-js-example

Spring Cloud Sleuth Java

Api requests are served by Spring Boot applications. Tracing of these are automatically performed by Spring Cloud Sleuth.

openzipkin/sleuth-webmvc-example

Propagation

Under the covers, tracing code can be tricky

Timing correctly

Trace state

Error callbacks

Version woes

```
// This is real code, but only one callback of Apache HC

Span span = handler.nextSpan(req);
CloseableHttpResponse resp = null;
Throwable error = null;
try (SpanInScope ws = tracer.withSpanInScope(span)) {
  return resp = protocolExec.execute(route, req, ctx, exec);
} catch (IOException | HttpException | RuntimeException | Error e) {
  error = e;
  throw e;
} finally {
  handler.handleReceive(resp, error, span);
}
```

Instrumentation

Instrumentation record behavior of a request or a message. Instrumentation is applied use of Tracer libraries.

They extract trace context from incoming messages, pass it through the process, allocating child spans for intermediate operations. Finally, they inject trace context onto outgoing messages so the process can repeat on the other side.

Propagation

Instrumentation encode request-scoped state required for tracing to work. Services that use a compatible context format can understand their position in a trace.

Regardless of libraries used, tracing can interop via propagation. Look at B3 and trace-context for example.

Propagation is the hardest part

- **In process** - place state in scope and always remove
- **Across processes** - inject state into message and out on the other side
- **Among other contexts** - you may not be the only one

In process propagation

- **Scoping api** - ensures state is visible to downstream code and always cleaned up. ex try/finally
- **Instrumentation** - carries state to where it can be scoped
  - **Async** - you may have to stash it between callbacks
  - **Queuing** - if backlog is possible, you may have to attach it to the message even in-process

Across process propagation

- **Headers** - usually you can encode state into a header
  - some proxies will drop it
  - some services/clones may manipulate it
- **Envelopes** - sometimes you have a custom message envelope
  - this implies coordination as it can make the message unreadable

Among other tracing implementations

- **In-process** - you may be able to join their context
  - you may be able to read their data (ex thread local storage)
  - you may be able to correlate with it
- **Across process** - you may be able to share a header
  - only works if your ID format can fit into theirs
  - otherwise you may have to push multiple headers

Wrapping Up

- introduction
- understanding latency
- distributed tracing
- zipkin
- demo
- **wrapping up**

Wrapping up

Start by sending traces directly to a zipkin server.

Grow into fanciness as you need it: sampling, streaming, etc

Remember you are not alone!

@zipkinproject

gitter.im/openzipkin/zipkin

Example Tracing Flow