How to Properly Blame Things for Causing Latency

An introduction to Distributed Tracing and Zipkin

@adrianfcole
works at Pivotal
works on Zipkin

Introduction

introduction
understanding latency
distributed tracing
zipkin
demo
propagation
wrapping up

@adrianfcole

spring cloud at pivotal
focused on distributed tracing
helped open zipkin

Understanding Latency

introduction
understanding latency
distributed tracing
zipkin
demo
propagation
wrapping up

Understanding Latency

`Unifying theory: Everything is based on events`

Logging - recording events

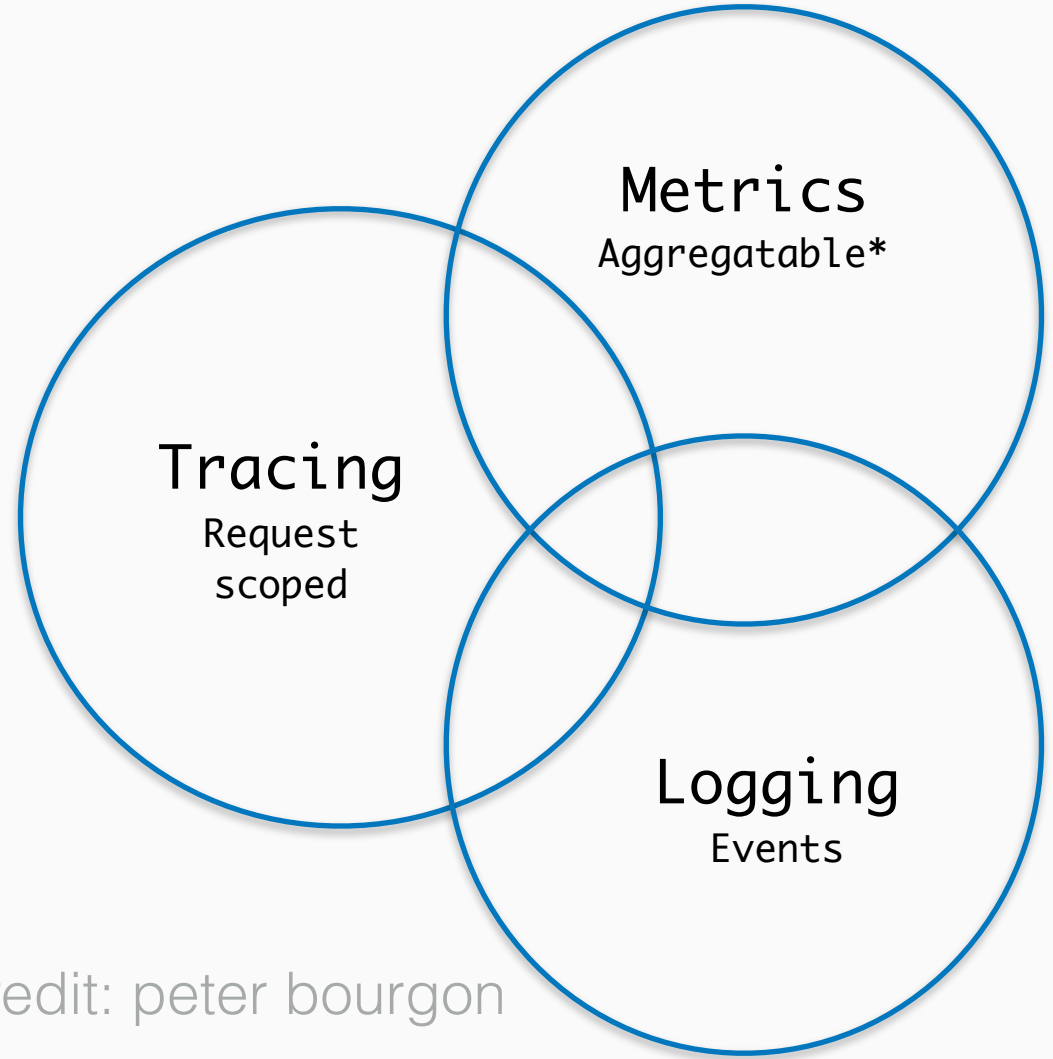
Metrics - data combined from measuring events

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

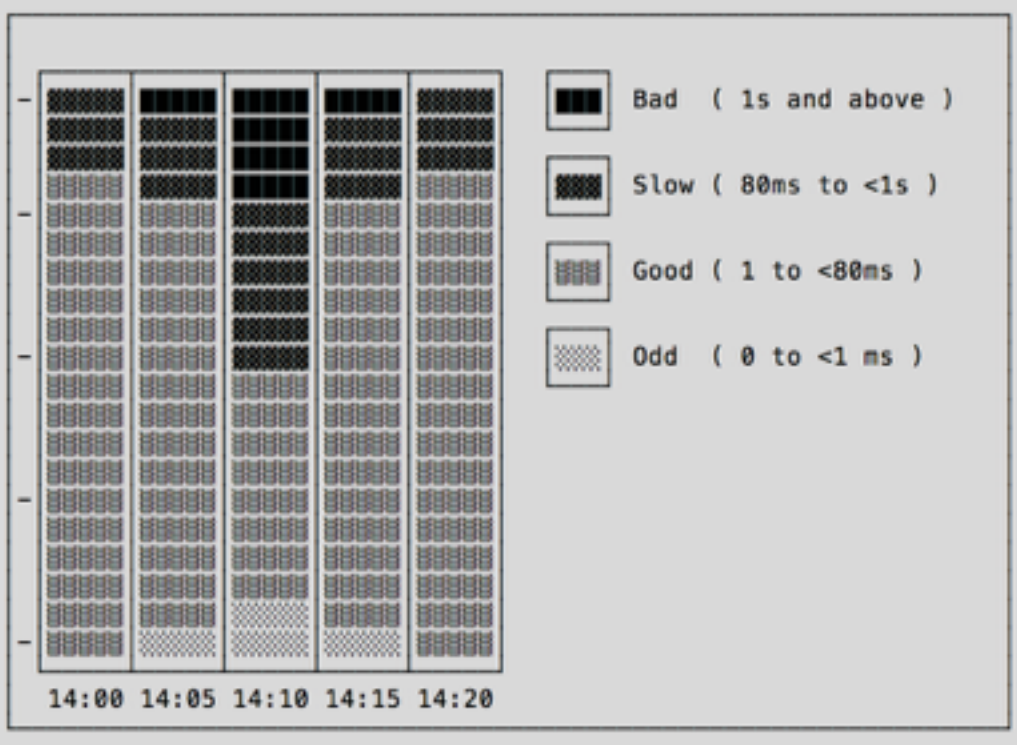
- Log - event (response time)
- Metric - value (response time)
- Trace - tree (response time)

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****

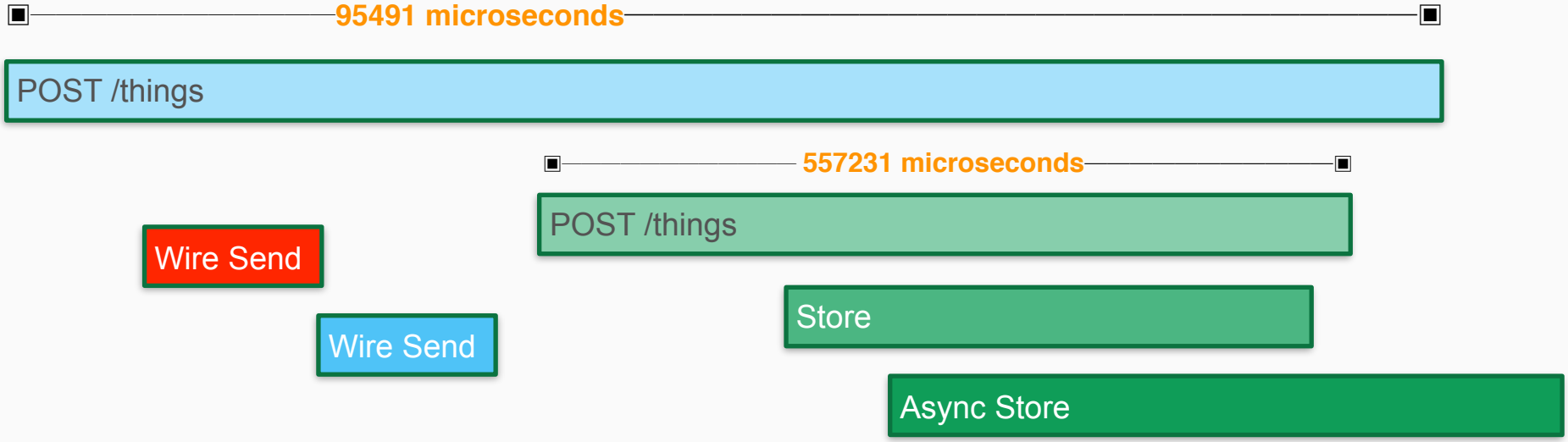
Look! this request took 95 milliseconds!

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

introduction
understanding latency
distributed tracing
zipkin
demo
propagation
wrapping up

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

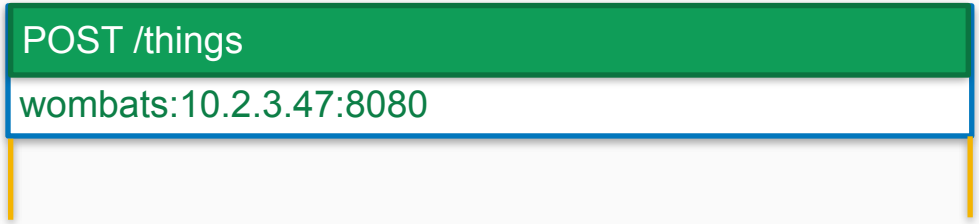
A **Span** is an individual operation that took place. A span contains **timestamped events** and **tags**.

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

A Span is an individual operation

Operation



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

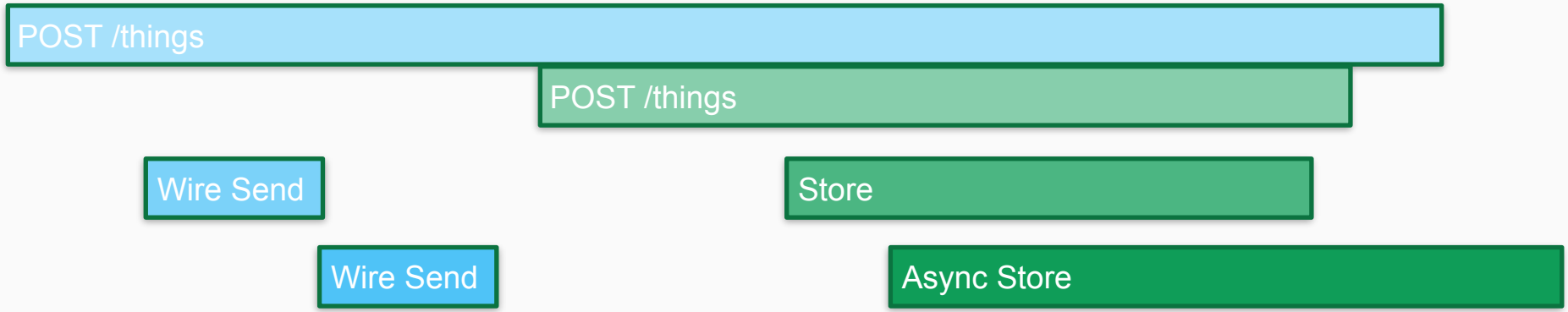
Wire Send

Store

Wire Send

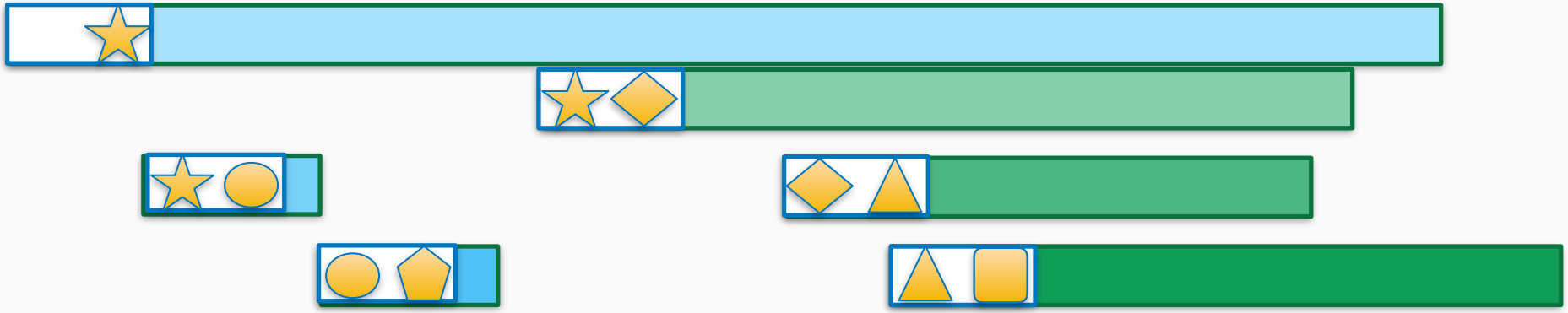
Async Store

Tracers record time, duration and host



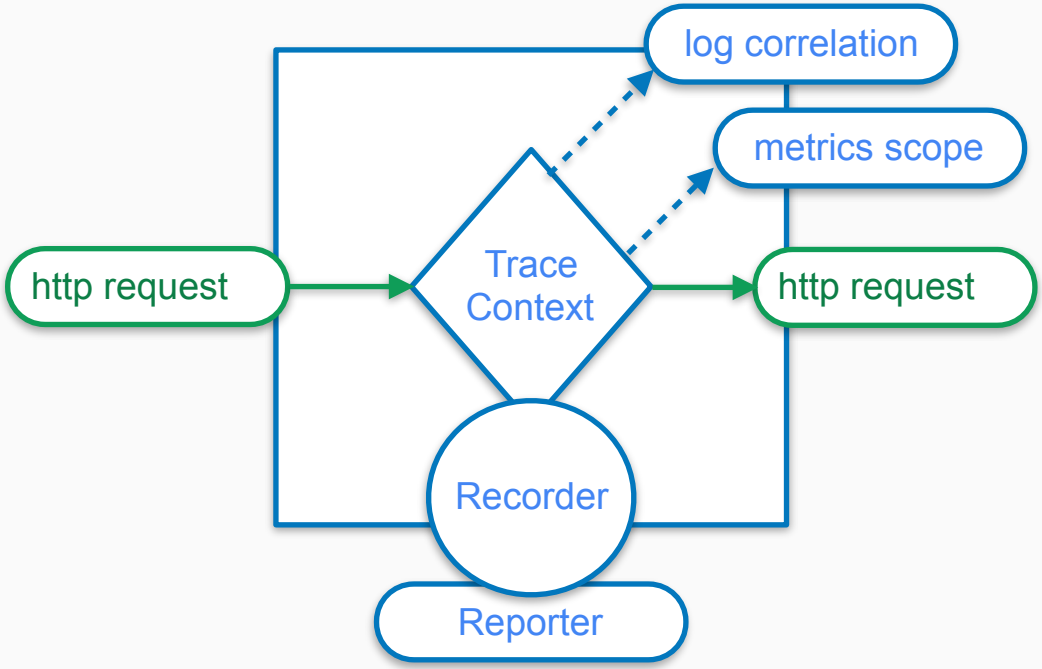
Tracers send trace data out of process

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

Example Tracer Flow



Tracers usually live in your application

Tracers execute in your production apps! They are written to not log too much, and to not cause applications to crash.

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

- have instrumentation or sampling policy to manage volume

- often include opinionated instrumentation of layers such as HTTP

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

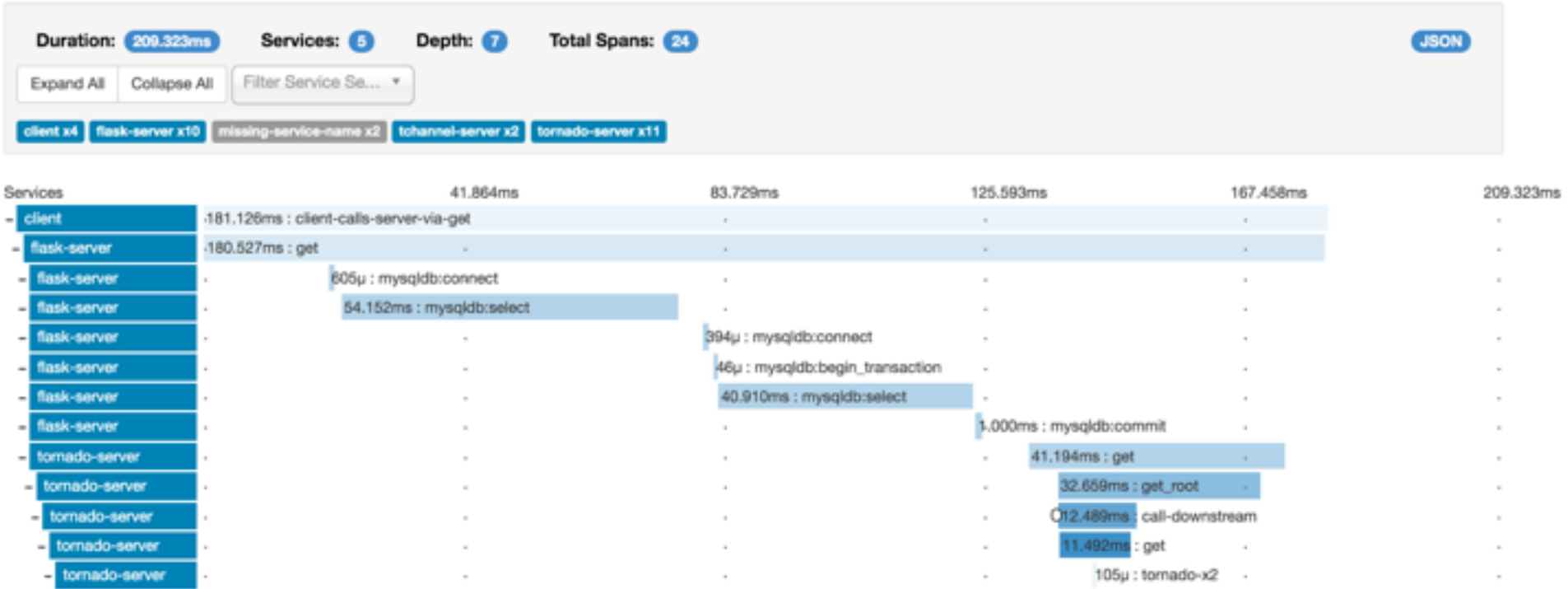
- Find who's calling deprecated services

- Reduce time spent on triage

Zipkin

introduction
understanding latency
distributed tracing
zipkin
demo
propagation
wrapping up

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.

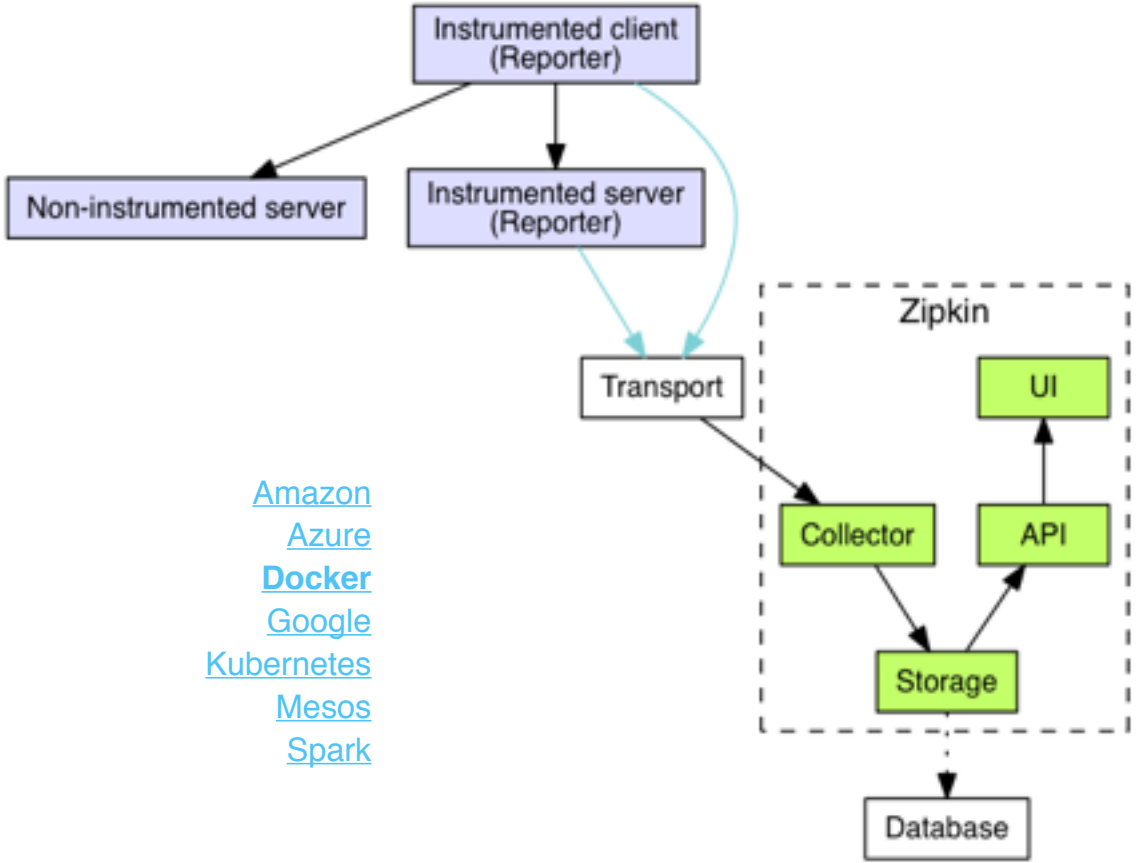
https://github.com/openzipkin

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

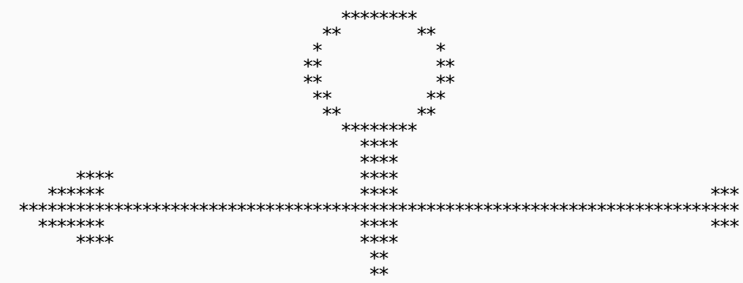
Tracing is new for a lot of folks.

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
```



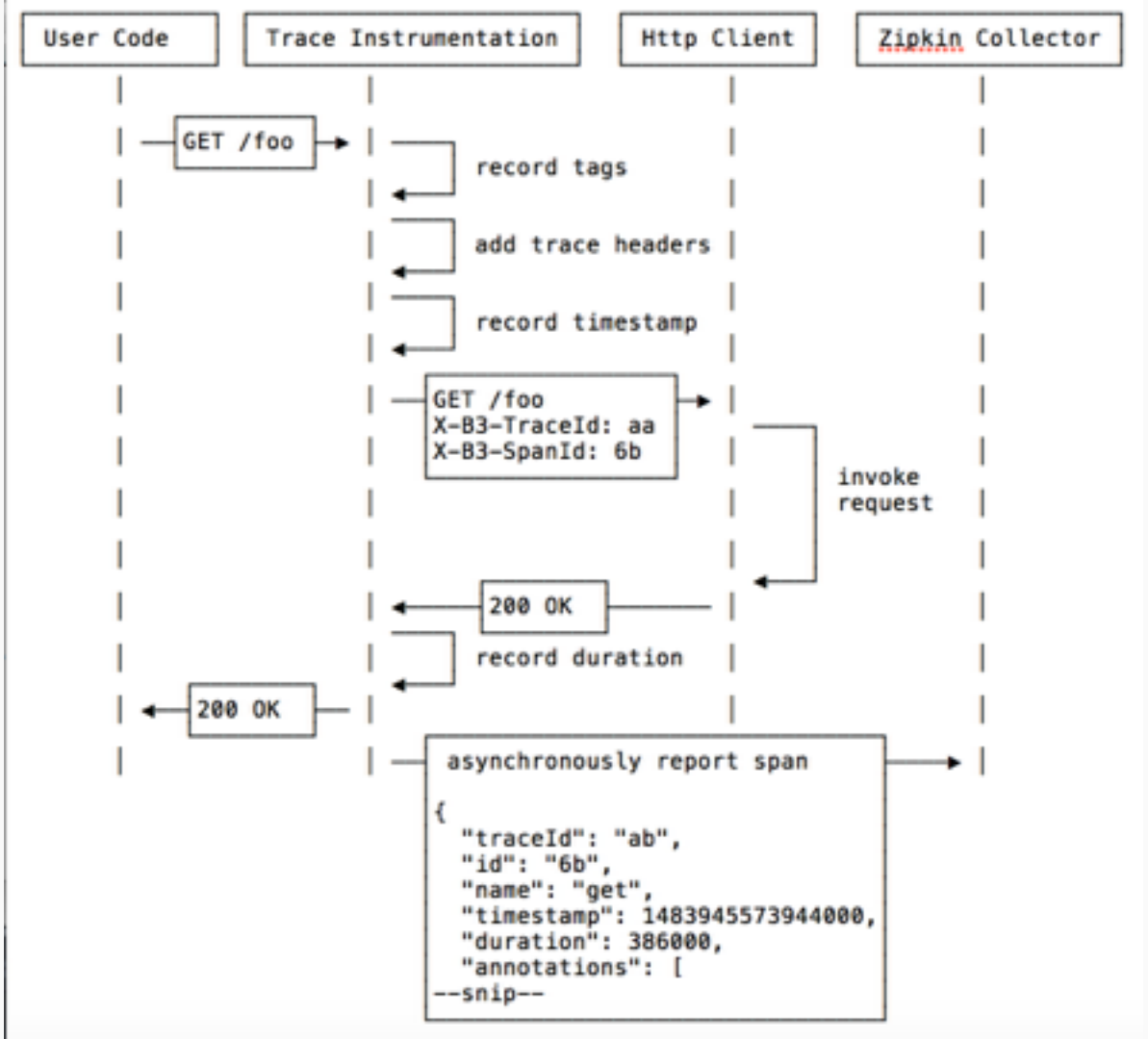
```
:: Powered by Spring Boot ::        (v1.5.4.RELEASE)

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/v1/services|jq .
[
  "zipkin-server"
]
```

How data gets to Zipkin —>

Looks easy right?



The most popular Zipkin Java tracers

- **Spring Cloud Sleuth** - automatic tracing for Spring Boot
  - Includes many common spring integrations

- **Brave** - OpenZipkin's java library and instrumentation
  - Layers under projects like Ratpack, Dropwizard, Play

`Tracing is polyglot. There are others in c#, go, php, etc.`

Other tracing libraries

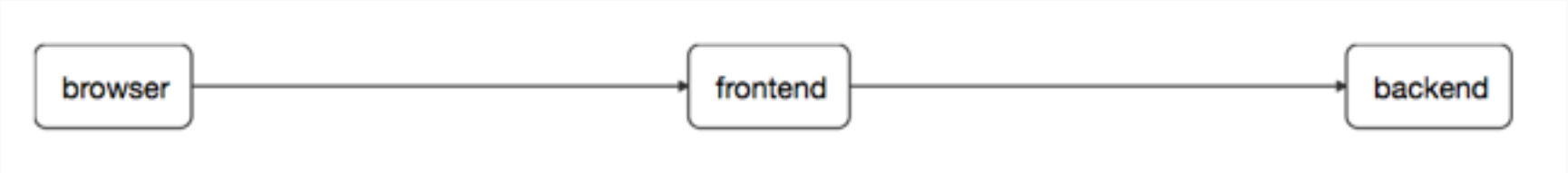
- **Apache HTrace** - tracing system for data services
  - a plugin sends HTrace traces to Zipkin
- **OpenTracing** - trace instrumentation library api definitions
  - some implementations are compatible w/ Zipkin tracers
- **OpenCensus** - Observability SDK (metrics, tracing, tags)
  - plugins in various languages for Zipkin data and B3 headers

Demo

introduction
understanding latency
distributed tracing
zipkin
demo
propagation
wrapping up

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

introduction
understanding latency
distributed tracing
zipkin
demo
propagation
wrapping up

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);
}
```

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

Extra goodies

Material about interop

Instrumentation

Instrumentation records behavior of a request. It extracts trace context from incoming messages, passes it through the process and injects that onto outgoing messages.

Library lock-in can happen regardless of tracing interop. Expose the least surface area possible, ideally none.

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.

Data Formats

Tracing systems have one or more data formats they accept and can emit. As long as propagation works, you can join across multiple systems via data export or conversion.

Zipkin and Census define models for import and export into different systems like Google Stackdriver or Amazon X-Ray.

Instrumentation lock-in example: OpenTracing

OpenTracing is a library api that aims to prevent lock-in by encouraging people to share it. Sounds nice, but there are serious lock-in risks

* You are limited to those implementing it, the versions they pin to, and how well it is tested
* The propagation part api thrashes, leading to custom extensions.
* Culture doesn't encourage automatic tracing, rather using api directly (even for logging!)
* Some use the term OpenTracing to describe incompatible dialects

OpenTracing can be safely used an exporter plug-in. Bridge to it with caution

Instrumentation lock-in example: Brave

Brave is a Zipkin v2 library that aims to prevent lock-in by focus on plugin interop and its established ecosystem. Sounds nice, but there are serious lock-in risks

* Intra process propagation works with explicit keys, not prefixed ones
* Data format doesn't currently support multiple parents, nor >128bit trace IDs
* Besides zipkin-php, there aren't yet other languages with similar library design
* Currently, no APM supports Brave injection

Only use Brave if your system is dapper-based. Don't use Brave as a logging api.

Instrumentation lock-in example: Agent

Agents connect to your process and implicitly add tracing to it. The are commonly used in APM (Application Performance Management), and usually do more than tracing.

Agents do not require user code changes, so have least lock-in.