Observability 3 ways
Logging, Metrics and Tracing

@adrianfcole
works at Pivotal
works on Zipkin
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
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/2017:11:07:07 +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 11:07?
Alert on max, performance tune to high percentiles.
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
How do you write timing code?

- **Log** - time and write formatted or structured logs
- **Metric** - time and store the number
- **Trace** - start, propagate and finish a “span”

**Jargon alert!** span == one operation in the call graph
Logging response time

long tookMs = TimeUnit.NANOSECONDS.toMillis(System.nanoTime() - startNs);

logger.log("<-- " + response.code() + ' ' + response.message() + ' ' + response.request().url() + " (" + tookMs + "ms" + (!logHeaders ? ", " + bodySize + " body" : ")")");

Find the thing you want and time it, format the result into a log statement.
def apply(request: Req, service: Service[Req, Rep]): Future[Rep] = {
    val sample = Timer.start(Clock.SYSTEM)

    service(request).respond { response =>
        sample.stop(
            Metrics.timer("request.latency", "code", response.status())
        )
    }
}
Tracing response time

```java
Span span = handler.handleReceive(extractor, httpRequest);
try {
    chain.doFilter(httpRequest, httpResponse);
} finally {
    servlet.handleAsync(handler, httpRequest, httpResponse, span);
}
```

Create and **manage** a span. Pass it on via **headers**
Impact of timing code

• Log - ubiquitous apis, but requires coordination
• Metric - easy, but least context
• Trace - hardest, as identifiers must be passed within and between services
Should you write timing code?

- Frameworks usually have metrics built-in
- Many frameworks have tracing built-in
- Lots of edge cases in this sort of code!
How to not see tracing code?

• Buddy - another process intercepts yours
• Agent - code patches code
• Framework - code intercepts or configures code
Use a service mesh to trace around your services
Agent tracing

if ("spark/webserver/JettyHandler".equals(className)) {
    ClassPool cp = new ClassPool();
    cp.appendClassPath(new LoaderClassPath(loader));

    CtClass ct = cp.makeClass(new ByteArrayInputStream(classfileBuffer));

    CtMethod ctMethod = ct.getDeclaredMethod("doHandle");
    ctMethod.insertBefore("{ $4.setHeader("TraceId", MagicTraceId.get()); }");

    return ct.toByteArray();
}
Framework Tracing

```java
@Configuration
@AutoConfigureAfter(TraceAutoConfiguration.class)
@ConditionalOnClass(HystrixCommand.class)
@ConditionalOnBean(Tracer.class)
public class SleuthHystrixAutoConfiguration {

    @Bean
    SleuthHystrixConcurrencyStrategy sleuthHystrixConcurrencyStrategy(
            Tracer tracer, TraceKeys traceKeys) {
        return new SleuthHystrixConcurrencyStrategy(tracer, traceKeys);
    }
}
```

Framework code configures libraries
How is timing data shipped?

- Log - pull raw events into a parsing pipeline
- Metric - report duration buckets near-real time
- Trace - report spans near-real time
Parsing latency from events

input {
  file {
    path => "/var/log/http.log"
  }
}

filter {
  grok {
    match => { "message" => "%{IP:client} %{WORD:method} %{URIPATHPARAM:request} %{NUMBER:bytes} %{NUMBER:duration}" }
  }
}

Identify the pattern and parse into indexable fields
define boundaries up front...

boundaries[0] = 1; // 0 to < 1ms
boundaries[1] = 1000; // 1ms to < 1s
boundaries[2] = 50000; // 1s to < 50s

add values by incrementing count in a bucket

for (int i = 0; i < boundaries.length; i++) {
    if (duration < boundaries[i]) {
        bucket[i]++;
        return;
    }
}
bucket[boundaries.length]++; // overflow!
Shipping spans

Spans represent operations and are structured.

```json
{
    "traceId": "aa",
    "id": "6b",
    "name": "get",
    "timestamp": 1483945573944000,
    "duration": 95491,
    "annotations": [
        --snip--
    ]
}
```

Spans represent operations and are structured.
How timing data grows

• Log - grows with traffic and verbosity
• Metric - fixed wrt traffic
• Trace - grows with traffic
Means to reduce volume

- Log - don’t log irrelevant data, filtering
- Metric - read-your-writes, coarser grain
- Trace - sampling, but needs to be consistent

Each have different retention, too!
Stitching all 3 together
Correlating Metrics and Tracing Data

https://medium.com/observability/want-to-debug-latency-7aa48ecbe8f7
Leverage strengths while understanding weaknesses

- **Log** - monoliths, black boxes, exceptional cases
- **Metric** - identify patterns and/or alert
- **Trace** - distributed services “why is this slow”
Was this helpful?

If so, thank folks who helped with this!

@peterbourgon  https://peter.bourgon.org/blog/

@basvanbeek  @bogdandrutu  @jeanneretph

@munroenic  @jon_k_schneider  @felix_b  @coda

If not, blame me, @adrianfcole  

@abhik5ingh