Navigating the Distributed Systems Execution Maze with OpenTracing

Ashlie Martinez, University of Washington PhD student, former intern
Ilya Kislenko (on behalf of Julio López), Member of Technical Stuff
@depohmel/ilya@kasten.io
Our tracing journey...

- Existing Kubernetes app
- Microservices architecture is complex
- Logging and Metrics are not enough
- How to add tracing with minimal disruptions
Kubernetes app footprints

- Microservices pattern
- Golang
- API-first approach
- Code generation
Sample App: Image Gallery

- 2 backend microservices
- golang
- Swagger generated HTTP/JSON APIs
- [https://github.com/kastenhq/demoapp](https://github.com/kastenhq/demoapp)
Distributed Tracing High Level

- Automatically aggregate traces
- Highlights the execution path requests
- Helps pinpoint where failures or slowness occur
- Complements logs and metrics collection tools
Trace and Span what are they?

Trace - is the complete processing of a request. The trace represents the whole journey of a request as it moves through all of the services or components of a distributed system.

Span - is a single step in the total processing of the overall request. Spans are typically named and timed operations.
OpenTracing & Jaeger

OpenTracing:
- CNCF distributed tracing library for Go, C#, Java, and other languages
- Instrument existing code with OpenTracing calls to collect tracing information

Jaeger:
- CNCF UI for visualizing and searching tracing data
- Uses coalesced tracing data stored in a database like Cassandra
- Deployable via helm chart and K8s yaml

Other tracing options: Zipkin, Google OpenCensus
OpenTracing Go SDK

- Each trace collected by a single service is called a “span”
  - Spans can be nested to show one service calling another
- OpenTracing leverages Go’s Context object to carry info about traces
  - Code being traced must propagate Context to be traced
- Information like HTTP status codes or request IDs can be added to traces
  - Allows developers to get more information about the state of the system for that trace
  - Can help the developer associate a specific trace with other debug information like logs
Instrumenting Image Gallery App: Part 1
Add tracing for incoming requests
Use custom middleware to add tracing

Image Catalog

Incoming Request

Middleware

Application Code

HTTP Client for outgoing requests

Other Services
Add tracing to incoming requests for services with custom middleware

```go
func Middleware(next http.Handler) http.Handler {
    // requests that go through it.
    return nethttp.Middleware(opentracing.GlobalTracer(), next,
        nethttp.OperationNameFunc(func(r *http.Request) string {
            return "HTTP " + r.Method + " " + r.URL.String()
        }))
}
```
Instrumenting Image Gallery App: Part 1

And we got 1 lonely trace

HTTP GET /v0/images/412ac325-e0b0-11e8-999a-72fafaaf730c4


Process: client.uuid: 439720b2b21d8ec hostname: metadata:564c:8f944d:df6d5 ip: 10.4.0.41 kubernetes: version: Go:2.14.0
Instrumenting Image Gallery App: Part 2
Add tracing for outgoing requests
Use custom HTTP transport to add tracing

- **Incoming Request**
- **Middleware**
- **Application Code**
- **HTTP Client for outgoing requests**
- **Other Services**
Add tracing to incoming requests for services with custom HTTP transport

```go
func (t *tracingTransport) RoundTrip(r *http.Request) (*http.Response, error) {
    ctx := r.Context()
    span, ctx2 := opentracing.StartSpanFromContext(ctx, "HTTP Request")
    defer span.Finish()

    r.WithContext(ctx2)
    carrier := opentracing.HTTPHeadersCarrier(r.Header)
    span.Tracer().Inject(span.Context(), opentracing.HTTPHeaders, carrier)

    resp, err := t.transport.RoundTrip(r)

    span.SetTag(string(ext.HTTPStatusCode), resp.StatusCode)

    return resp, err
}
```
Add tracing to incoming requests for services with custom HTTP transport

```go
func (t *tracingTransport) RoundTrip(r *http.Request) (*http.Response, error) {
    ctx := r.Context()
    span, ctx2 := opentracing.StartSpanFromContext(ctx, "HTTP Request")
    defer span.Finish()

    r = r.WithContext(ctx2)
    carrier := opentracing.HTTPHeadersCarrier(r.Header)
    span.Tracer().Inject(span.Context(), opentracing.HTTPHeaders, carrier)

    resp, err := t.transport.RoundTrip(r)

    span.SetTag(string(ext.HTTPStatusCode), resp.StatusCode)

    return resp, err
}
```
Add tracing to incoming requests for services with custom HTTP transport

```go
func (t *tracingTransport) RoundTrip(r *http.Request) (*http.Response, error) {
    ctx := r.Context()
    span, ctx2 := opentracing.StartSpanFromContext(ctx, "HTTP Request")
    defer span.Finish()

    r = r.WithContext(ctx2)
    carrier := opentracing.HTTPHeadersCarrier(r.Header)
    span.Tracer().Inject(span.Context(), opentracing.HTTPHeaders, carrier)

    resp, err := t.transport.RoundTrip(r)
    span.SetTag(string(ext.HTTPStatusCode), resp.StatusCode)

    return resp, err
}
```
Now we can see that metadata is calling store service.

<table>
<thead>
<tr>
<th>Service &amp; Operation</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>metadata: HTTP GET /v0/images/412ac325-e0b0-11e8-999a-72fafa730c4</td>
<td>132.91ms</td>
</tr>
</tbody>
</table>

HTTP GET /v0/images/412ac325-e0b0-11e8-999a-72fafa730c4

- Process: client.uid - 0907210232b380ec, hostname - metadata-56c4d8946d8-vd6, ip - 10.4.0.41, jaeger.version - Go-2.14.0
Now we can see that metadata is calling store service
Instrumenting Image Gallery App: Part 3
Instrumenting Image Gallery App: Part 3

Extra tracing in other functions
Instrumenting Image Gallery App: Part 3

Add extra calls to OpenTracing

Incoming Request → Middleware → Application Code → HTTP Client for outgoing requests → Other Services

Image Catalog
Can we reuse the same HTTP transport from the last part?
Can we reuse the same HTTP transport from the last part?

No, not all APIs have the same interface

- K8s API does not take Context object
- mgo does not use Go’s HTTP client
- Official AWS and GCP GO APIs can use the same trick as part 2
func (s *Mongo) GetAllImages(ctx context.Context) (models.ImageList, error) {
    span, _ := opentracing.StartSpanFromContext(ctx, "GetAllImages request")
    defer span.Finish()
    addSpanTags(span)
    err := s.Ping()
    if err != nil {
        return models.ImageList{}, err
    }
    c := s.Conn.DB(dbName).C(collName)
    imgs := models.ImageList{}
    return imgs, c.Find(nil).All(&imgs)
}
Here we can see everything.

```
<table>
<thead>
<tr>
<th>Trace Start: November 4, 2018 7:44 PM</th>
<th>Duration: 142.97ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Services: 2</td>
<td>Total Spans: 6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HTTP GET /v0/images/ced2d81c-00a0-11ed-8d05-862f37c522a8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service: metadata</td>
</tr>
<tr>
<td>Duration: 101.04ms</td>
</tr>
<tr>
<td>Start Time: 8:03ms</td>
</tr>
</tbody>
</table>

**FetchImage request**

- Tags: Service: metadata, DB Provider: MongoDB
- Process: client.uid: 573cd1f7e422998f, hostname: metadata-8477dc4cb6-966f86 |
- `ip`: 10.4.2.46, jaeger.version: Go-2.14.0

**FindImages request**

- Tags: Service: metadata, DB Provider: MongoDB
- Process: client.uid: 573cd1f7e422998f, hostname: metadata-8477dc4cb6-966f86 |
- `ip`: 10.4.2.46, jaeger.version: Go-2.14.0

**HTTP Request**

- Process: client.uid: 573cd1f7e422998f, hostname: metadata-8477dc4cb6-966f86 |
- `ip`: 10.4.2.46, jaeger.version: Go-2.14.0

**HTTP GET /v0/store**

- Process: client.uid: 406681f59b798670, hostname: store-871b9465-8f5b7686 | `ip`: 10.4.1.61, jaeger.version: Go-2.14.0

**ReadData request**

- Tags: Service: store, Store Provider: LocalHDD
- Process: client.uid: 406681f59b798670, hostname: store-871b9465-8f5b7686 | `ip`: 10.4.1.61, jaeger.version: Go-2.14.0
```
Instrumenting Image Gallery App

We saw 3 different ways to add tracing, which are good for different situations:

● Incoming requests -> OpenTracing Middleware
● Outgoing requests -> custom HTTP transport
● Other parts of application -> manual calls to tracing functions
Recap

Today we discussed

● Using OpenTracing Go SDK to add instrumentation microservices
● Instrumenting calls to other services: DB, cloud provider, K8s API
● Installing Jaeger tracing collector and UI in k8s cluster
● Using Jaeger UI to visualize, analyze and dig into traces
Pffff... my logging is amazing!

- **Pro:** Good for collecting detailed information at a single point
- **Con:** Hard to correlate and analyze logs across microservices
- Use tracing to get overview and logging to get details in problem areas
I’m getting sms from Nagios!

- **Pro:** Use alerts to get automated notifications from your monitoring system
- **Cons:** It shows you only service or 1 call, not a specifics.
- **When combined with tracing, request-scoped metrics can be available**
But my service mesh has it?!

- **Pro:** No instrumentation or Context propagation required
- **Con:** Only coarse-grained traces like part 1 of instrumenting our application
- Can be combined with tracing frameworks for better data
My cloud provider do it for me for “free”

Tracing solutions from cloud providers:
● GoogleCloud Stackdriver Trace
● AWS X-Ray

● **Pro:** Almost seamless integration when used in those environments
● **Con:** Most of the time means you locked down to 1 provider
Final Thoughts

Tracing can give insights into system bottlenecks, but need to balance with time spent adding instrumentation

Trade-offs:
- **Pro:** Fine granularity and detailed request information
- **Con:** Additional resource requirements.
  - Request processing in each of the services, and additional network traffic
  - Additional processing and storage requirements for the traces
Questions?
Attributes

Go Gophers

- https://github.com/ashleymcnamara/gophers/blob/master/LICENSE

The Illustrated Children's Guide to Kubernetes