Distributed Tracing

Understanding Latency

Who am I?

- Michael Würtinger
- Working as a Software Engineer at eGym for ~8 years
- Now Site Reliability Engineer and Go enthusiast

Quick Poll

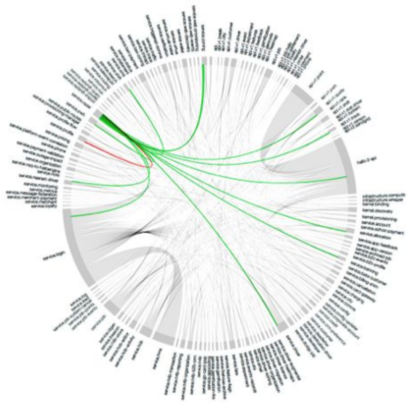
- Who has an idea what distributed tracing is?
- Who is already using distributed tracing in production?

Why do we need tracing?

- Understand where latency comes from
- Understand service dependencies
- Find redundant calls

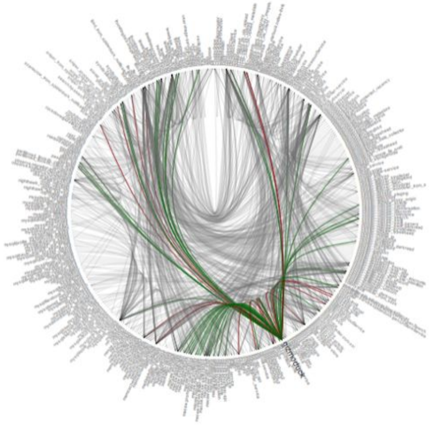
Why are those questions so hard to answer?

450+ microservices



500+ microservices

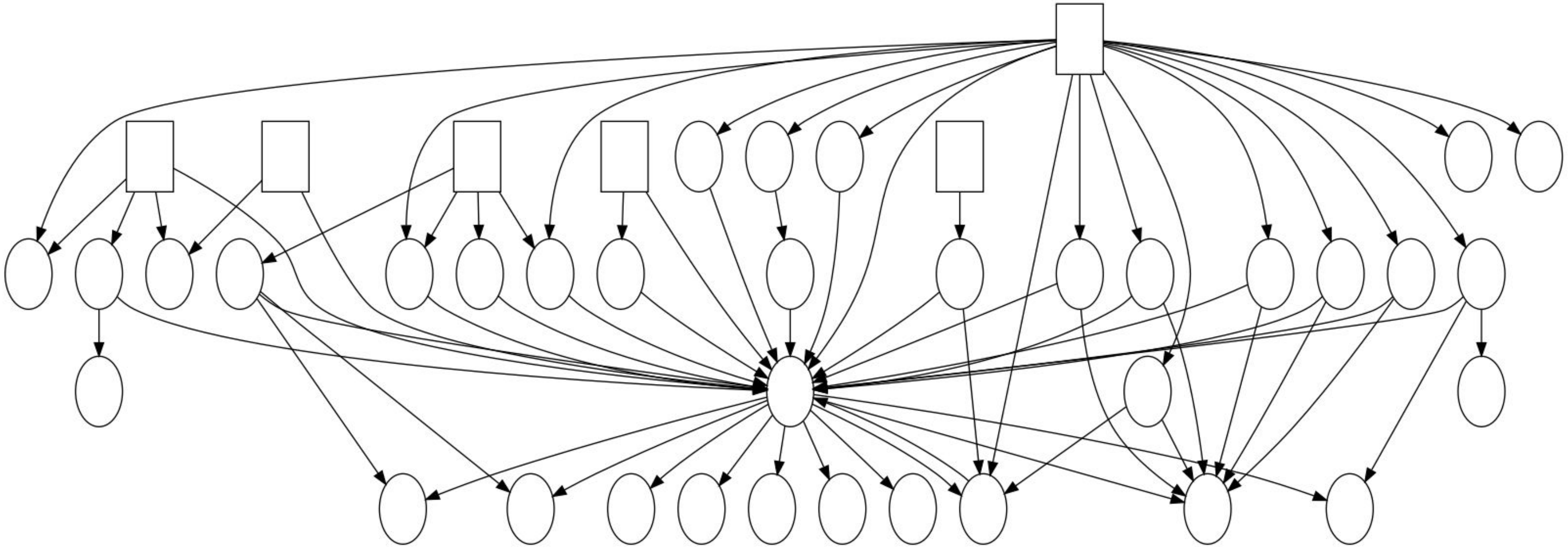
500+ microservices



HAIL O

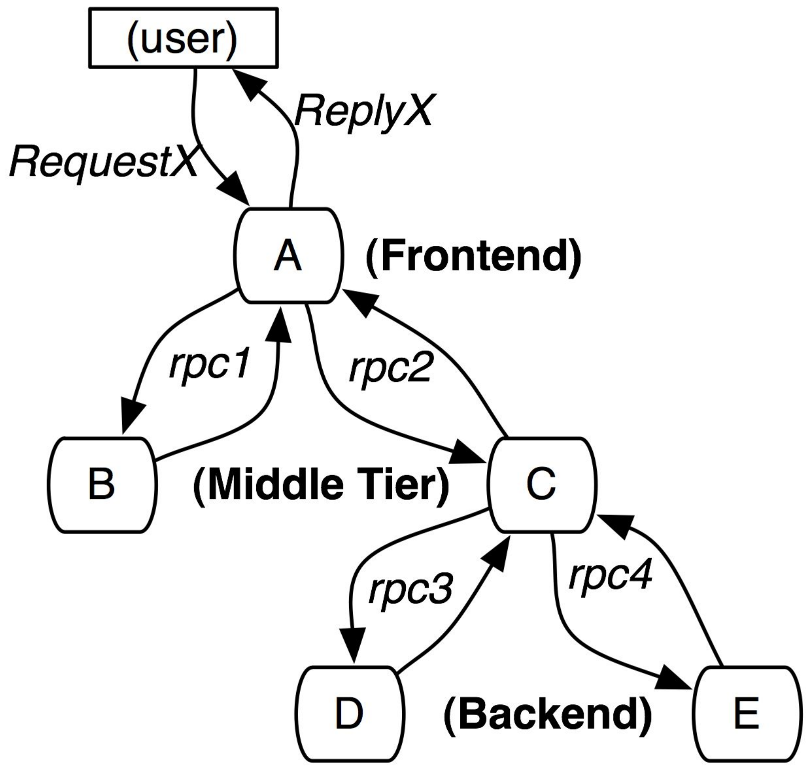
NETFLIX

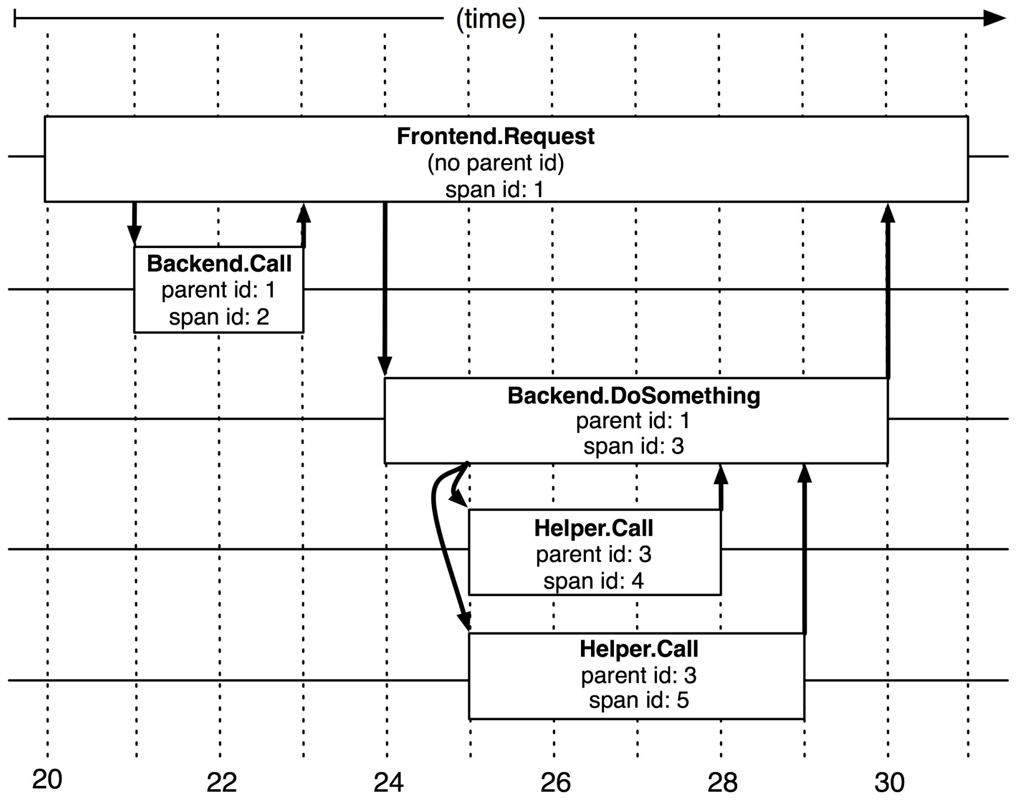
Situation at eGym



~30 services with ~50 interaction points

What is distributed tracing?



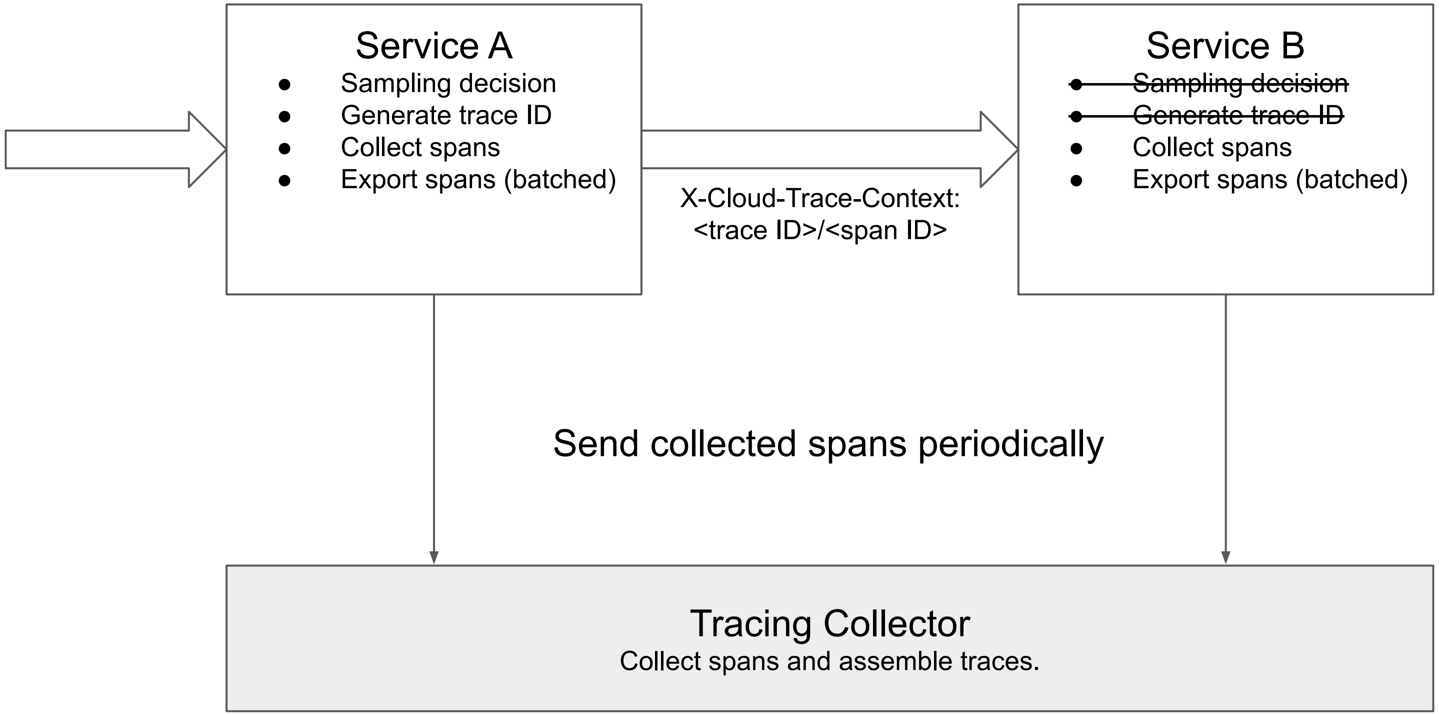


Source: https://ai.google/research/pubs/pub36356

What is distributed tracing? (cont'd)

- **Trace**
  - represents one operation from the user's perspective
  - is a *tree* of spans
- **Span** represents a *unit of work* in a service and consists of
  - Human readable name
  - Start and end timestamp
  - Parent span ID (optional)
  - Application specific annotations (optional)
- A span without parent is called a *root span*

How are traces collected?



Source: https://ai.google/research/pubs/pub36356

Historic overview

- 2010 Distributed tracing first described in the Dapper paper by Google
- 2012 Work begins on OpenZipkin at Twitter
- 2016 OpenZipkin 1.0.0
- 2017 Google open sources OpenCensus tracing/instrumentation library
- 2017 Uber open sources Jaeger (tracing server)

Tracing eGym

OpenCensus

Modern planet scale observability:
distributed tracing and monitoring for your services

Stackdriver

Why?

- We prefer hosted solutions
- We run on GCP so Stackdriver is the obvious choice
- OpenCensus is the recommended way to ingest traces into Stackdriver

A word about OpenCensus

- OpenCensus is a universal instrumentation library
- It supports tracing, metrics and in the future maybe even logging
- Why? Because all those things belong together ⇒ they provide observability
- OpenCensus is designed to have minimal overhead
  - Unless metrics are actually collected in "views" they have practically zero cost
  - Allows to add metrics generously

How to add tracing to a Go service (Step 1)

```go
import (
	"contrib.go.opencensus.io/exporter/stackdriver"
	"github.com/sirupsen/logrus"
	"go.opencensus.io/trace"
	"go.opencensus.io/stats/view"
)

exporter, err := stackdriver.NewExporter(stackdriver.Options{
	ProjectID: "tracing-demo",
	MetricPrefix: "custom.googleapis.com/",
})
if err != nil {
	logrus.Fatal("stackdriver: ", err)
}
trace.RegisterExporter(exporter) // tracing
view.RegisterExporter(exporter)  // metrics
```

How to add tracing to a Go service (Step 2)

```go
import (
	"go.opencensus.io/trace"
)

fraction := 1.0 // sample everything (do not use this in production)
// fraction := 0.01 // sample 1% (a value more suitable for production)
trace.ApplyConfig(trace.Config{DefaultSampler:
trace.ProbabilitySampler(fraction)})
```

How to add tracing to a Go service (Step 3)

```go
import (
	"go.opencensus.io/plugin/ochttp"
	"contrib.go.opencensus.io/exporter/stackdriver/propagation"
)

// HTTP
handler := &ochttp.Handler{Handler: mux, Propagation: &propagation.HTTPFormat{}}

// gRPC
s := grpc.NewServer(grpc.StatsHandler(&ocgrpc.ServerHandler{}))
```

Full example:

https://bitbucket.org/egym-com/tracing-example/src/master/

What about the client side?

```go
import (
	"net/http"
	"go.opencensus.io/plugin/ochttp"
)

// HTTP
client := &http.Client{Transport: &ochttp.Transport{}}

// gRPC
conn, err := grpc.Dial(address, grpc.WithStatsHandler(&ocgrpc.ClientHandler{}))
```

Demo Time

Demo

Thank you

Resources

- Demo: https://bitbucket.org/egym-com/tracing-example/
- Dapper Paper: https://ai.google/research/pubs/pub36356
- Observability of Distributed Systems: https://www.youtube.com/watch?v=SoZZzB-yTOk
- Reach out to me
  - mw@egym.de
  - @mwuertinger
  - https://www.linkedin.com/in/wuertinger/

Find these slides online: https://goo.gl/GyGEvo