Grafana Labs

Implementing distributed traces with eBPF

Monitoring & Observability devroom

Beyla

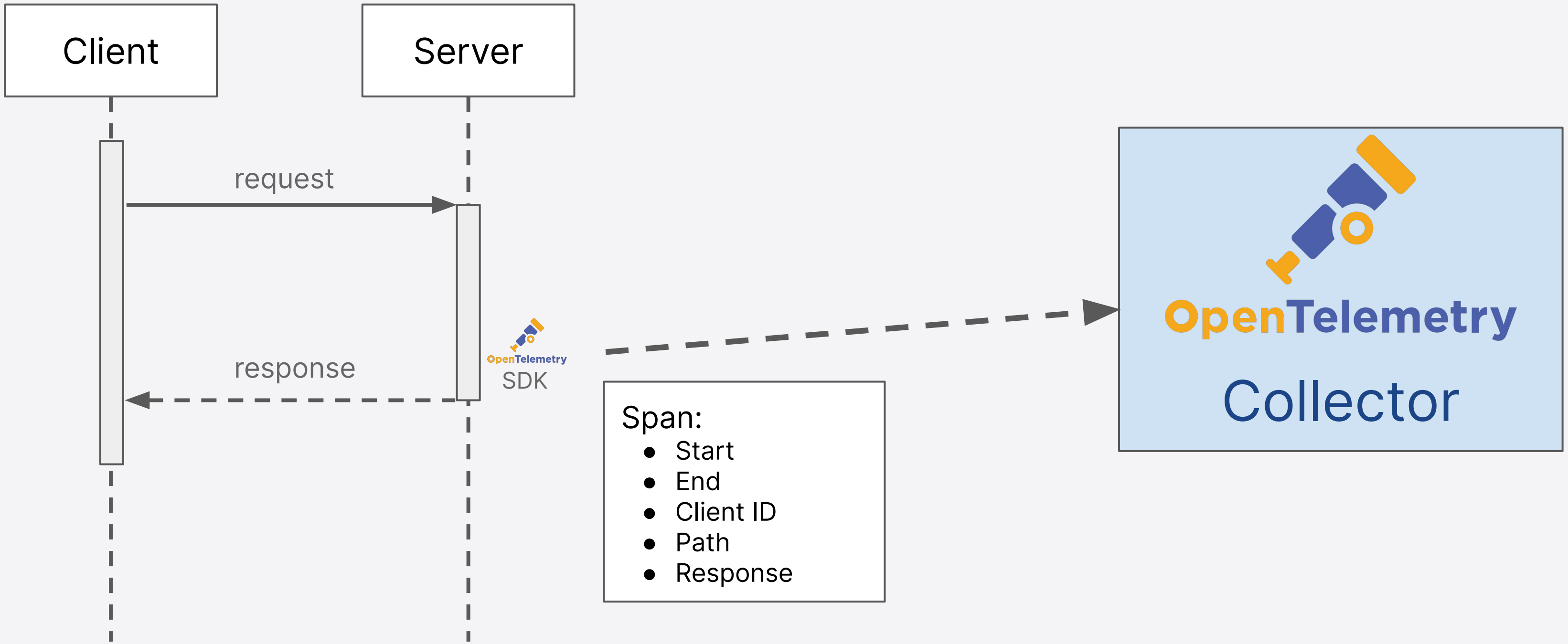
Nikola Grcevski
Mario Macías Lloret

FOSDEM 2024
Brussels, Belgium

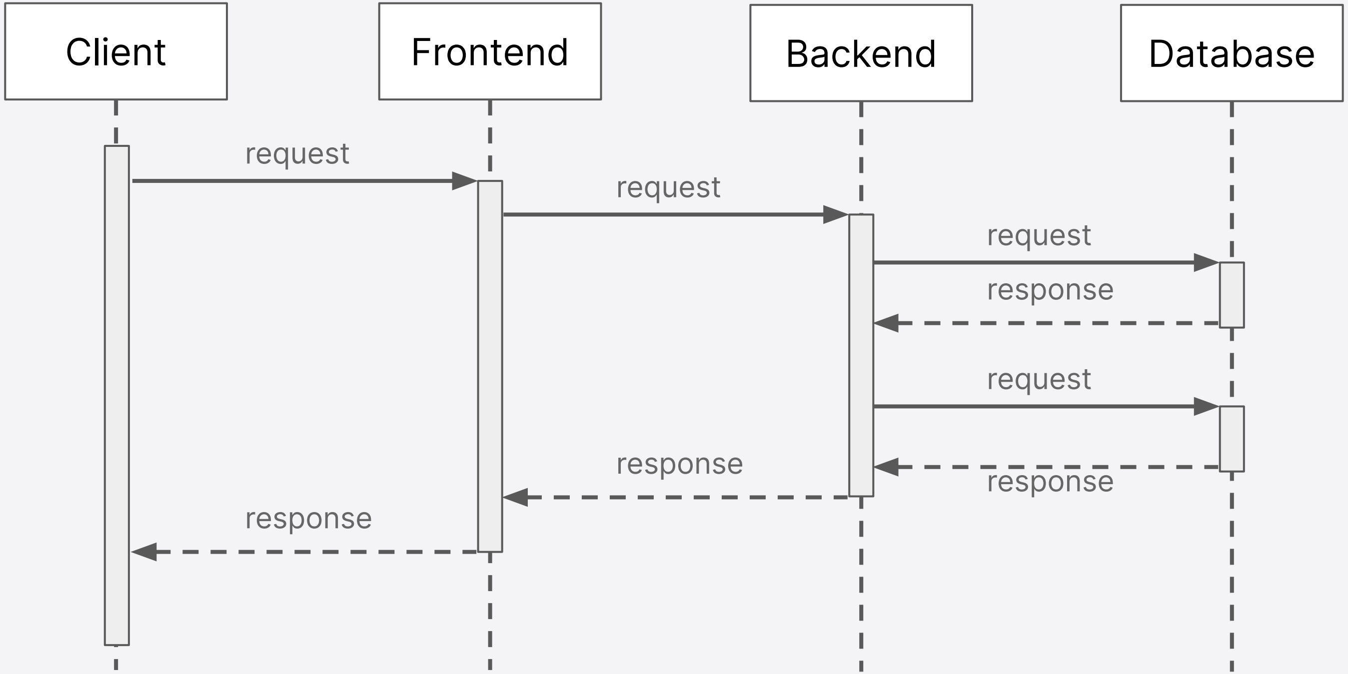
Contents

- Quick introduction to what is distributed tracing
- How is distributed tracing done with OpenTelemetry
- Distributed traces with Beyla (eBPF)
- DEMO

Introduction: isolated spans



Introduction: isolated spans

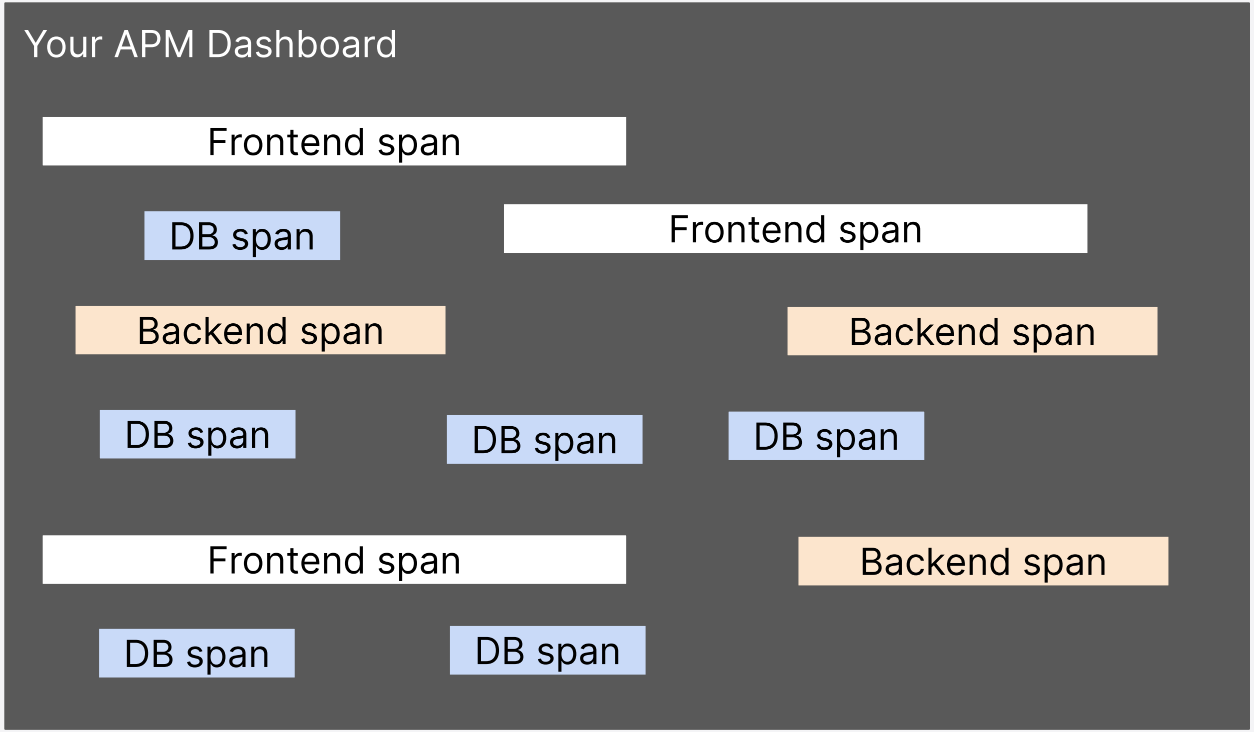


Introduction: isolated spans

Not

the most

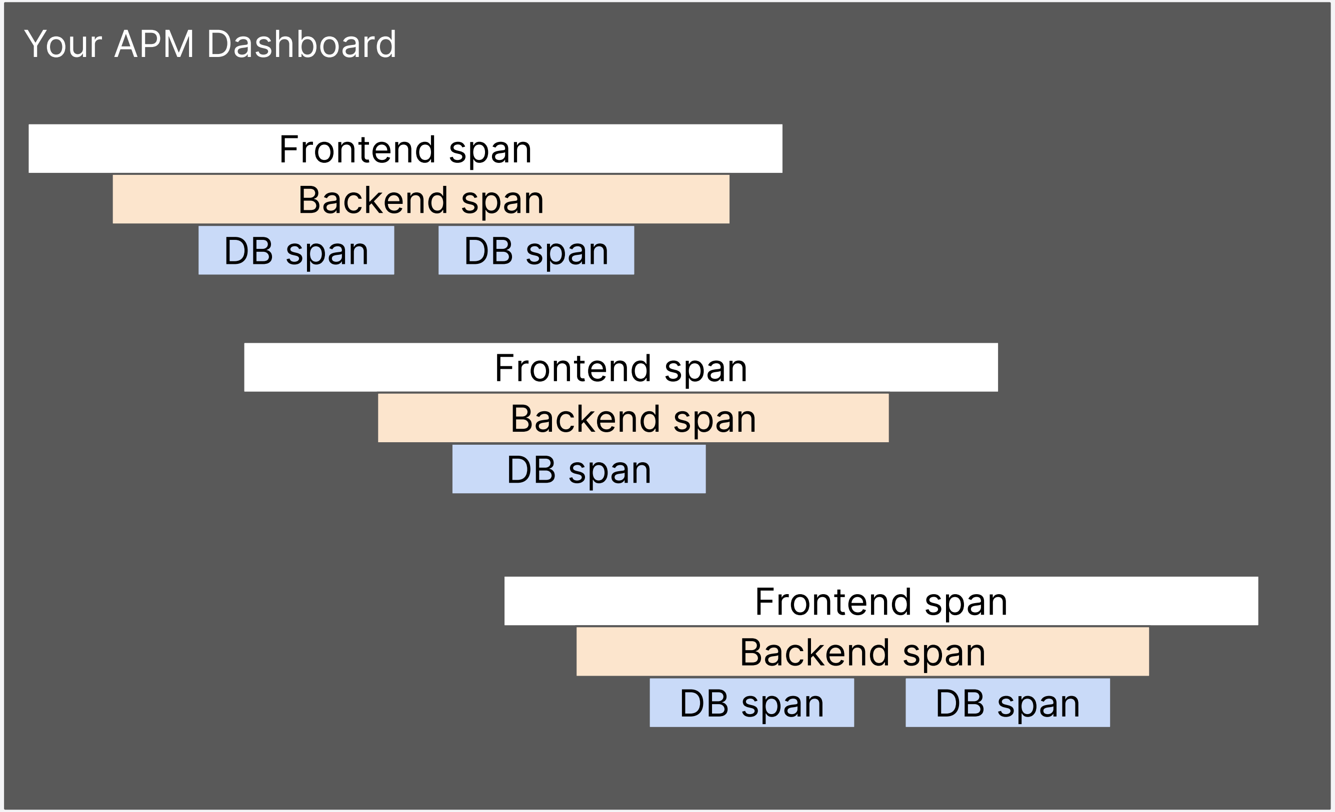
useful



Introduction: distributed traces

We want to see

the **context**



How is context propagated between services?

- Each new request gets unique 16 hex character **SpanID**
- W3C defines a request header field called “traceparent”

00	5fe865607da112abd799ea8108c38bcd	4c59e9a913c480a3	01
Vendor Info	TraceID	Parent SpanID	Flags

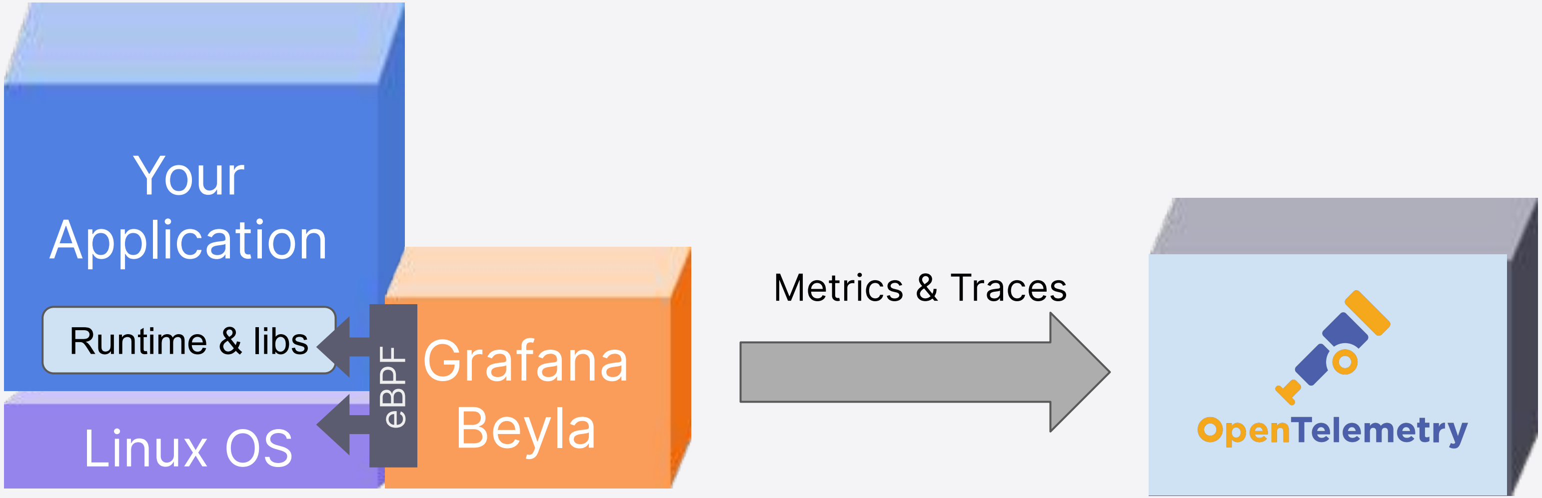
- The **TraceID** is common for all spans of one trace
- This traceparent value is propagated through outgoing header calls

How to propagate context (pseudocode)

```
service frontend(request, response) {
  traceparent = request.header[“traceparent”]
  span.start(traceparent)

  /* do stuff */

  backend.call(headers = {
     “traceparent”: traceparent
  })

  /* do stuff */

  response.ok().render()
  span.end()
}
```

Can be injected by your instrumentation SDK or agent

Beyla native eBPF auto-instrumentation



eBPF

- JIT Virtual Machine at the Linux Kernel
- Can hook your probe programs to multiple events of the Kernel, libraries and user-space programs
- Lets you see (and even modify) the runtime memory

Providing spans information with Beyla

- Language-level (Go)
  - Hook uprobe at the start and end of any ServeHttp(Request, Response) function
- Kernel-level (other languages)
  - Hook kprobes and kretprobes at several kernel functions and libraries (sys_accept, tcp_recvmsg, tcp_sendmsg, etc...)

Automatic context propagation with Beyla

```
service frontend(request, response) {
  /* do stuff */

  backend.call(headers = {
     "content-type": ...
     ...
  })

  /* do stuff */

  response.ok().render()
}
```

Read memory with eBPF

```
traceparent = request.header["traceparent"]
span.start(traceparent)
```

Propagate context

Write user space memory from eBPF

```
"traceparent": traceparent
```

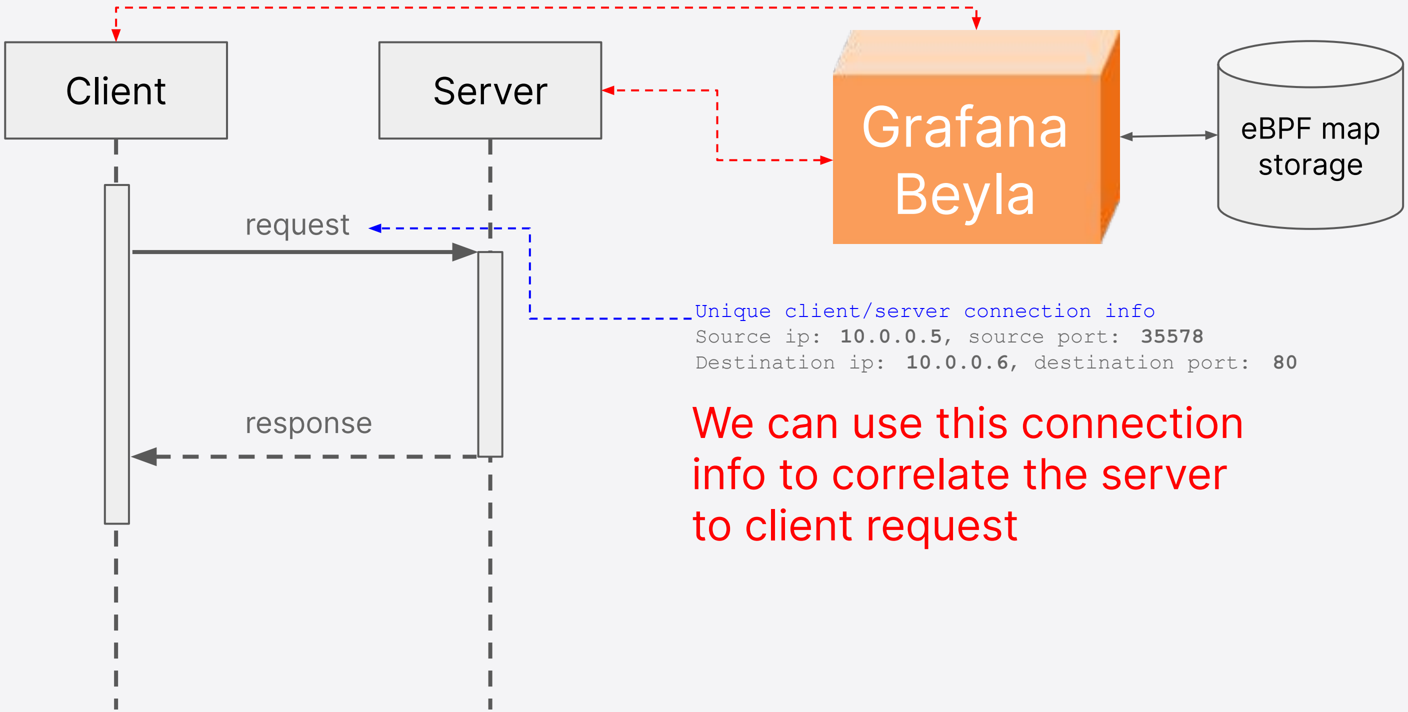
```
span.end()
```

- Deal with runtime-managed memory
- Deal with limited-size preallocated buffers
- Deal with operating system protections

Propagating context: writing propagation in memory

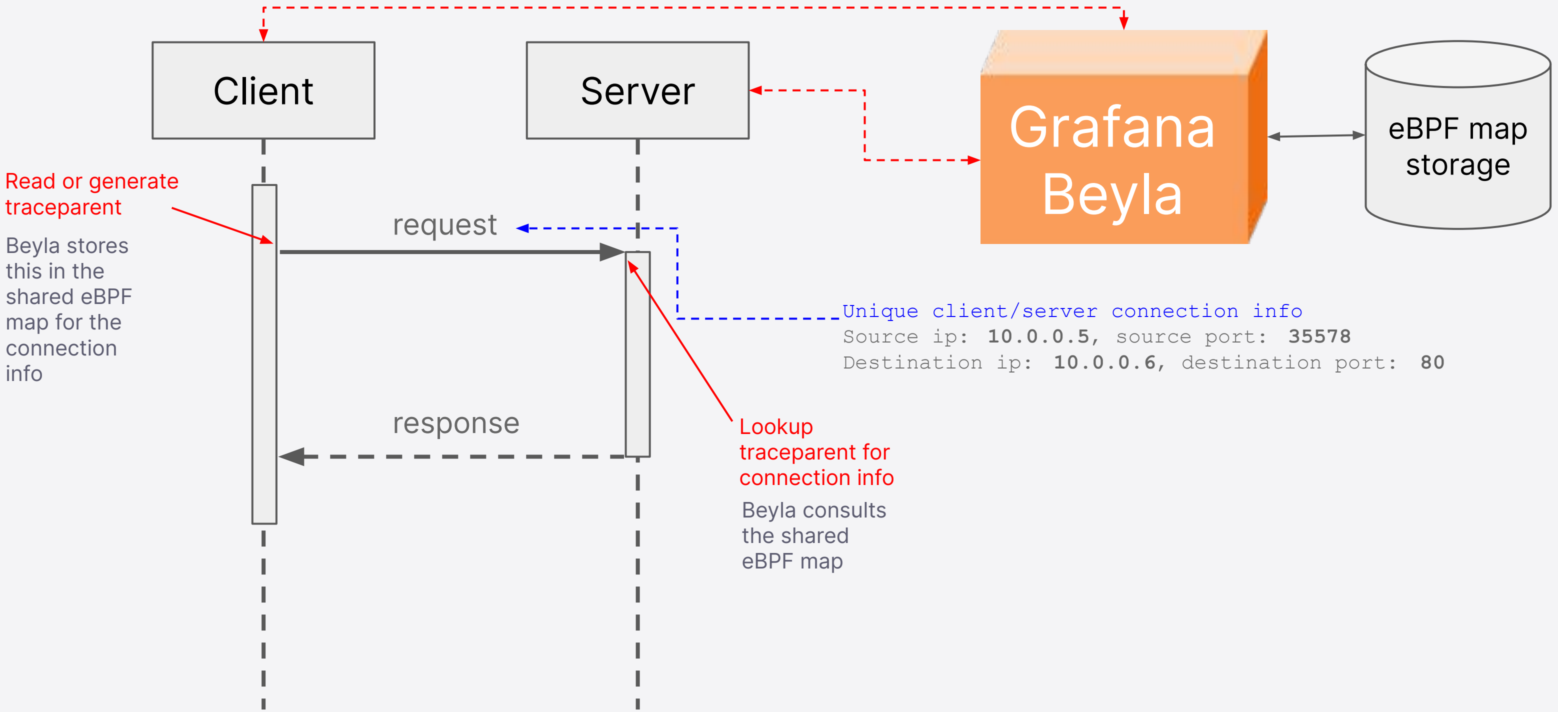
- For Go
- Tracks goroutine child parent relationships for async calls
- Writes traceparent into outgoing request headers

Black-box context propagation

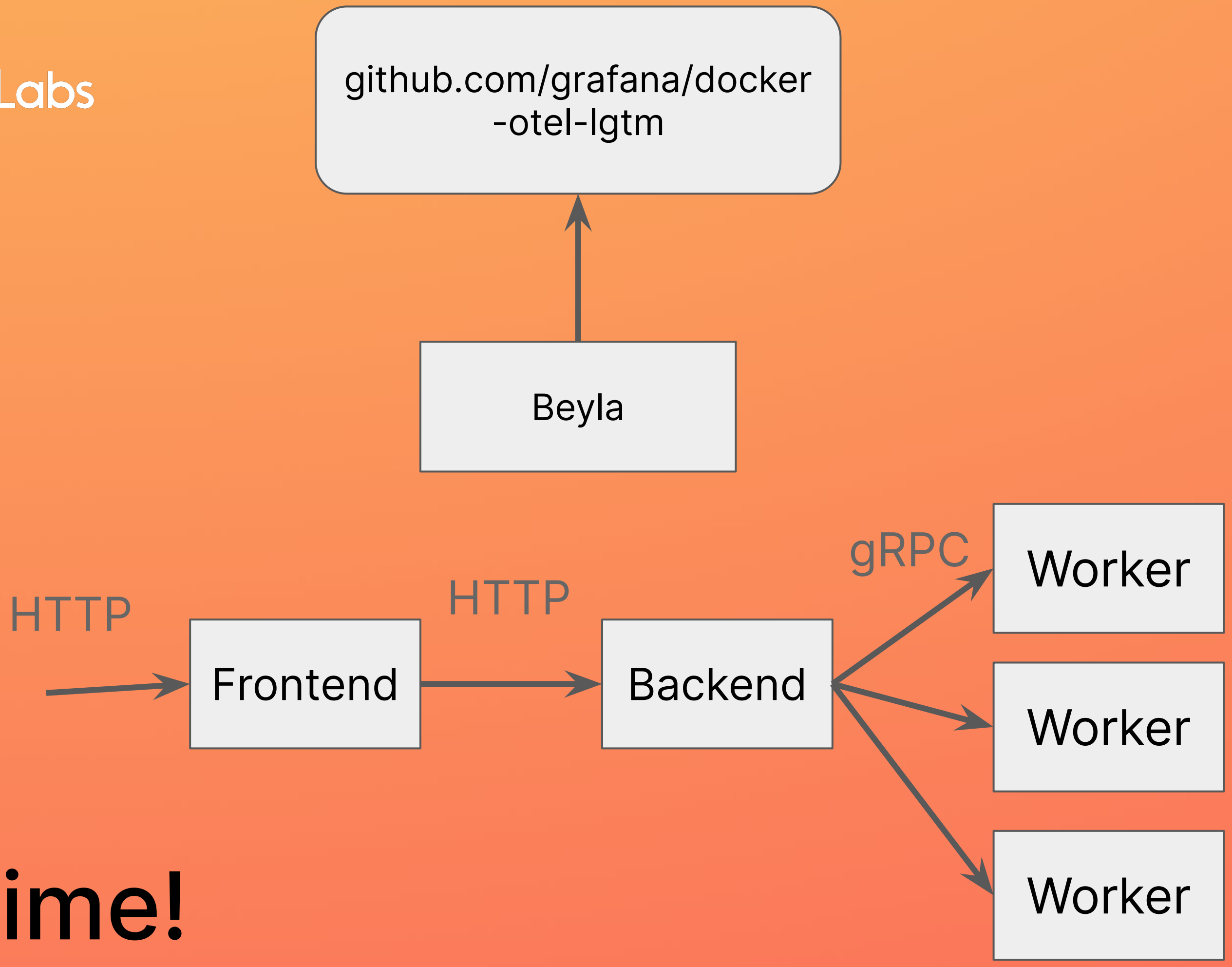


We can use this connection info to correlate the server to client request

Black-box context propagation



Grafana Labs



Demo time!

Summary

- By using eBPF we can capture distributed traces with some limitations
- Using eBPF requires almost no effort from the developer/operator
- Combining eBPF kernel packet tracing with language level support can get us to fully automatic distributed traces

Thank you!

Connect with us at

https://github.com/grafana/beyla