Implementing distributed traces with eBPF

Monitoring & Observability devroom

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

Client

Server

request

response

OpenTelemetry SDK

Collector

Span:
- Start
- End
- Client ID
- Path
- Response
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”

![Traceparent Value]

- The **TraceID** is common for all spans of one trace
- This traceparent value is propagated through outgoing header calls
How to propagate context (pseudocode)

```plaintext
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

- Your Application
- Linux OS
- Runtime & libs
- Grafana
- Beyla

Metrics & Traces

OpenTelemetry
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

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

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

  /* do stuff */

  response.ok().render()
}

Read memory with eBPF

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

Write user space memory from eBPF

```
"traceparent": traceparent
```

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

Deal with
```
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.

Unique client/server connection info
Source ip: 10.0.0.5, source port: 35578
Destination ip: 10.0.0.6, destination port: 80
Black-box context propagation

Client ➔ Server ➔ Grafana

Read or generate traceparent
Beyla stores this in the shared eBPF map for the connection info

Unique client/server connection info
Source ip: 10.0.0.5, source port: 35578
Destination ip: 10.0.0.6, destination port: 80

Lookup traceparent for connection info
Beyla consults the shared eBPF map

eBPF map storage
Demo time!

github.com/grafana/docker-otel-lgtm

Beyla

Frontend

Backend

HTTP

Worker

Worker

Worker

HTTP

gRPC
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