The Service Mesh
It’s About the Traffic

Oliver Gould  @olix0r
Oliver Gould

Linkerd Lead; Buoyant CTO

@olix0r
@olix0r
@olix0r
@olix0r
Agenda

Why Does Linkerd Exist?

The Trough of Service Mesh Disillusionment

It’s All About the Traffic!
Control Plane
- Discovery
  - ZooKeeper
- Telemetry
  - Zipkin
  - Viz...

Timelines

Users

Finagle (Library)
Any sufficiently complicated microservices arch contains an ad-hoc, informally-specified, bug-ridden, slow implementation of half of Finagle

4:19 AM - 29 Jun 2016 · Manhattan, NY, United States
Service Mesh
Service Mesh: Data Plane
Service Mesh: Control Plane

- Control Plane
- Proxy A
- Proxy B
- Proxy C
An Abridged History of Linkerd

- **2016**: Linkerd 0.1.0
- Twitter-style Operability for Microservices
- Scala (JVM) + Finagle
- Extremely Powerful and Configurable
An Abridged History of Linkerd

- JVM sidecar too heavy for some users
- Difficult to configure
  - High barrier to entry
  - Many different configurations to support
How?

💪 Zero-config “just works”: If you have a functioning K8s app, drop in Linkerd without configuring anything.

💪 Fast and small: proxies should introduce the bare minimum perf and resource cost

💪 Understandable: no magic

Data plane: linkerd2-proxy. Written in Rust. <10MB RSS, <1ms p99. (!!!!)

Control plane: linkerd2. Written in Go. Includes small Prometheus (6 hour window), Grafana, etc.
Linkerd 2.x architecture
Strong Typing

```rust
code
impl<C, S, A, B> svc::Service<http::Request<A>> for Service<S, C>
where
    S: svc::Service<
        http::Request<RequestBody<A, C::Class>>,
        Response = http::Response<B>,
    >,
    A: Payload,
    B: Payload,
    C: ClassifyResponse<Error = h2::Error> + Clone + Default + Send + Sync + 'static,
    C::Class: Hash + Eq,
{
    type Response = http::Response<
        ResponseBody<B, C::ClassifyEos>,
    >;
    type Error = S::Error;
    type Future = ResponseFuture<S::Future, C>;

    fn poll_ready(&mut self) -> Poll<(), Self::Error> {
        self.inner.poll_ready()
    }

    fn call(&mut self, req: http::Request<A>) -> Self::Future {
        let mut req_metrics = self.metrics.clone();
```
No GC: RAII

Resource Acquisition Is Initialization

```rust
impl<B, C> Drop for ResponseBody<B, C> {
    where
    B: Payload,
    C: ClassifyEos<Error = h2::Error>,
    C::Class: Hash + Eq,
{
    fn drop(&mut self) {
        if !self.latency_recorded {
            self.record_latency();
        }
        if let Some(c) = self.classify.take().map(|c| c.eos(None)) {
            self.record_class(c);
        }
    };
```
What does Linkerd do?

👍 **Visibility:** Automatic *golden metrics*: success rates, latencies, throughput

👍 **Reliability:** Load balancing, retries, timeouts, circuit breaking, deadlines

👍 **Security:** Transparent mTLS, cert validation, policy

**Goal:** Move visibility, reliability, and security primitives *into* the infrastructure layer, *out* of the application layer.
Linkerd: Observability

- Rich traffic metrics
  - Request rate, Success rate, latency
  - Across many dimensions
- Request inspection
Linkerd: Reliability

- Latency aware load balancing
- Retries
- Timeouts
Linkerd: Security

- Mutual, cryptographic identity
  - Bootstraps via Kubernetes ServiceAccounts
  - Transparent
  - On by default
An open source **service mesh** and **CNCF** project.

- 24+ months in production
- 3,000+ Slack channel members
- 10,000+ GitHub stars
- 100+ contributors
- Near-weekly edge releases
The Trough of Service Mesh Disillusionment
What Can Go Wrong?

1. Can’t even get it working…
2. Trying to do too many things at once...
3. It’s always the mesh’s fault!
It’s All About the Traffic!
The logical constructs of a Kubernetes deployment:

- **Namespace**
  - **Deployment**
    - **ReplicaSet(s)**
      - **Service**
        - **Pods**
The Service Mesh Interface

What SMI covers

Service Mesh Interface is a specification that covers the most common service mesh capabilities:

- Traffic policy – apply policies like identity and transport encryption across services
- Traffic telemetry – capture key metrics like error rate and latency between services
- Traffic management – shift traffic between different services
Service Mesh Interface (SMI): Q&A with Microsoft's Lachlan Evenson

by
Rags Srinivas

JUN 24, 2019 • 5 MIN READ
**Roadmap**

As of 2.3:

- Telemetry, retries, timeouts, auto-inject, mTLS on by default. All zero config.

2.4

- Traffic shifting (blue-green, canaries), install split.

Mid term:

- Policy, mesh expansion, distributed tracing, lots lots more.
Linkerd v2: How Lessons from Production Adoption Resulted in a Rewrite of the Service Mesh

by William Morgan

reviewed by Daniel Bryant
Join our community!

- github.com/linkerd
- slack.linkerd.io
- @linkerd