Network Observability with Envoy
Network Observability with Envoy

Nic Jackson
Developer Advocate at HashiCorp
1. **Introduction** - create a common vocabulary.
2. **Metrics** - configure and use metrics.
3. **Tracing** - configure and use tracing.
4. **Logging** - collect access logs.
Introduction
The shift from static to dynamic networking

Static Infrastructure
Host-based networking

Dynamic Infrastructure
Service-based networking
The shift from static to dynamic networking

**Static Infrastructure**
Host-based networking

Homogeneous infrastructure with static IPs, primarily north-south traffic, protected by coarse-grained segments.

**TRADITIONAL APPROACH**
- Load balancers to create artificial static IPs
- Firewall sprawl to constrict service traffic
- Configuration management to deploy services

**Dynamic Infrastructure**
Service-based networking

Heterogeneous infrastructure with dynamic IPs, dominated by east-west traffic, without a clear network perimeter.

**CONSUL APPROACH**
- Service Discovery for connectivity
- Service Segmentation for security
- Service Configuration for runtime configuration
Market trend from monoliths to microservices

- Single, Physical Server
- Dynamic Virtual Machines
- Smaller, Ephemeral Containers
Reduced Productivity
Waiting for manual updates to load balancers and firewalls blocks development throughput.

Increased Risk
Firewall rule sprawl is complex to manage and mistakes create security vulnerabilities.

Increased Cost
Load balancers and firewalls are expensive and costly to maintain.
Components

Service Mesh

Control Plane

- Service to service communication policy
- Service Catalog
- CA and x509 certificate generation
- Configuration and proxy management
Components

Service Mesh

Data Plane

- Authorization
- Request tracing
- Traffic shaping
- Load balancing
- Service discovery
- Circuit breaking
- Retry logic
- Networking statistics
Networks are **not** 100% stable and **often** experience transient failure.
You can’t do **Reliability** without **Observability**.
Observability, is it just a buzzword?
Observability is a measure of how well **internal states** of a system can be inferred from knowledge of external outputs.
Observability

- Envoy Statistics
  - Connection data
  - Requests
  - Authentication
  - Control plane data

- Application Statistics
  - Handler timings
  - Errors

- Kubernetes Statistics
  - Pod CPU
  - Pod memory
  - Pod network
  - Cluster health

- Business Analytics
  - Sales
  - Traffic
  - Click throughs
  - Marketing campaigns

- Node Statistics
  - Disk space
  - CPU
  - Network

- Tracing
  - Jaeger

- Health Checks
  - Is application up?

- Log Files
  - Errors
  - Miscellaneous info
Internal and external instrumentation

**INTERNAL**

- **APPLICATION STATISTICS**
  - HEALTH CHECKS
    - is application up?
    - Pingdom
    - Consul
    - Kubernetes
  - NODE STATISTICS
    - disk space
    - cpu
    - network (e.g. Nagios)

**EXTERNAL**

- PROMETHEUS
Metrics
Envoy Architecture

Metrics
Terminology
Listener

Terminology

A listener is a named network location (e.g., port, unix domain socket, etc.) that can be connected to by downstream clients. Envoy exposes one or more listeners that downstream hosts connect to.
Downstream

Terminology

A downstream host connects to Envoy, sends requests, and receives responses.
Upstream

Terminology

An upstream host sends requests from Envoy to other services and returns responses.
Cluster

Terminology

A cluster is a group of logically similar upstream hosts that Envoy connects to. Envoy discovers the members of a cluster via service discovery. The cluster member that Envoy routes a request to is determined by the load balancing policy.
Configuration

Metrics
Envoy metrics.

Envoy does not label the metrics with the application name, so add tags to be able to differentiate between metrics.

```json
"stats_config": {
  "stats_tags": [
    {
      "tag_name": "local_cluster",
      "fixed_value": "emojify-api-v2"
    },
    
    "use_all_default_tags": true
  }
}
```
Envoy Prometheus Metrics

Metrics

- **1.10** introduces histograms for Prometheus metrics
- Metrics exposed with **unsecured** admin endpoint (/stats/prometheus),
- Exposure of metrics **needs** to be configured with **loopback** route to **avoid** exposing **admin endpoints**
Using metrics
StatsD

- Originally created by Etsy
- Push based metrics
- Lightweight UDP protocol
- **No support for metadata**
## Metrics types

**StatsD**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counter</td>
<td>Increment value, e.g. number of method calls.</td>
</tr>
<tr>
<td>Gauge</td>
<td>Value over time, e.g. CPU consumption, memory usage.</td>
</tr>
<tr>
<td>Timing</td>
<td>Time taken to perform a task, e.g. time take to perform a method call.</td>
</tr>
<tr>
<td>Set</td>
<td>Set of unique values over collection period.</td>
</tr>
</tbody>
</table>
Metrics format

StatsD does not support basic metric labels.

```plaintext
# metric.name:value|type|sample_rate
myservice.mymethod.called:123|c

# metrics output
myservice.service1.mymethod.called
myservice.service2.mymethod.called
myservice.service3.mymethod.called
```
DogStatsD

- Created by DataDog based on StatsD protocol
- Push based metrics
- Lightweight UDP protocol
- **Support for metadata through tags**
Metrics format

DogStatsD

```plaintext
myservice.mymethod.called tags[serviceid:service1]
myservice.mymethod.called tags[serviceid:service2]
myservice.mymethod.called tags[serviceid:service3]
```
Prometheus

- Pull based approach from central server
- Service implements HTTP endpoint exposing metrics
- Supports metadata by default
# Metrics types

**Prometheus**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counter</td>
<td>Cumulative metric, representing a monotonically increasing counter, e.g. number of method calls.</td>
</tr>
<tr>
<td>Gauge</td>
<td>Single numerical value that can arbitrarily go up and down, e.g. CPU consumption.</td>
</tr>
<tr>
<td>Histogram</td>
<td>Samples observations and counts them in configurable buckets, e.g. request timings.</td>
</tr>
</tbody>
</table>
Metrics format

Prometheus

```text
envoy_http_downstream_rq_completed{envoy_http_conn_manager_prefix="ingress_cache"}
```
Choosing a format

- Tagging is **essential** to effectively build dashboards
- Metrics **need** to be tagged with **Metadata** such as pod name, node, etc
Listener

Metrics
## Key service metrics

### Listener - Connections

Every listener has a statistics tree rooted at `<prefix>.listener.<address>`, with the following statistics:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>downstream_cx_total</td>
<td>Counter</td>
<td>Total connections</td>
</tr>
<tr>
<td>downstream_cx_destroy</td>
<td>Counter</td>
<td>Total destroyed connections</td>
</tr>
<tr>
<td>downstream_cx_active</td>
<td>Gauge</td>
<td>Total active connections</td>
</tr>
</tbody>
</table>

[https://www.envoyproxy.io/docs/envoy/latest/configuration/listeners/stats#listener-manager](https://www.envoyproxy.io/docs/envoy/latest/configuration/listeners/stats#listener-manager)
Envoy metrics. 

use_all_default_tags extracts common components from metric names and adds as tags
# The number of established connections to emojify-api-v2 over 30 seconds.
`increase(envoy_listener_downstream_cx_total{local_cluster="emojify-api-v2"}[30s])`

# The number of destroyed connections to emojify-api-v2 over 30 seconds.
`increase(envoy_listener_downstream_cx_destroy{local_cluster="emojify-api-v2"}[30s])`

# The current number of active connections to emojify-api-v2.
`envoy_listener_downstream_cx_active{local_cluster="emojify-api-v2"}`
Total connections

Grafana

New pod started
Key diagnostics metrics

Listener

Every listener has a statistics tree rooted at `<prefix>.listener.<address>`, with the following statistics:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssl.fail_verify_no_cert</td>
<td>Counter</td>
<td>Total TLS connections that failed because of missing client certificate</td>
</tr>
<tr>
<td>ssl.connection_error</td>
<td>Counter</td>
<td>Total TLS connection errors not including failed certificate verifications</td>
</tr>
<tr>
<td>ssl.fail_verify_error</td>
<td>Counter</td>
<td>Total TLS connections that failed CA verification</td>
</tr>
<tr>
<td>ssl.fail_verify_san</td>
<td>Counter</td>
<td>Total TLS connections that failed SAN verification</td>
</tr>
<tr>
<td>downstream_pre_cx_timeout</td>
<td>Counter</td>
<td>Sockets that timed out during listener filter processing</td>
</tr>
<tr>
<td>downstream_pre_cx_active</td>
<td>Gauge</td>
<td>Sockets currently undergoing listener filter processing</td>
</tr>
<tr>
<td>downstream_cx_length_ms</td>
<td>Histogram</td>
<td>Connection length milliseconds</td>
</tr>
</tbody>
</table>

https://www.envoyproxy.io/docs/envoy/latest/configuration/listeners/stats#listener-manager
Requests HTTP / GRPC
Metrics
# Key metrics

## Listener - Requests HTTP

Every listener has a statistics tree rooted at `<prefix>.http.<address>`, with the following statistics:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>upstream_rq_1xx</td>
<td>Counter</td>
<td>Total 1xx responses</td>
</tr>
<tr>
<td>upstream_rq_2xx</td>
<td>Counter</td>
<td>Total 2xx responses</td>
</tr>
<tr>
<td>upstream_rq_3xx</td>
<td>Counter</td>
<td>Total 3xx responses</td>
</tr>
<tr>
<td>upstream_rq_4xx</td>
<td>Counter</td>
<td>Total 4xx responses</td>
</tr>
<tr>
<td>upstream_rq_5xx</td>
<td>Counter</td>
<td>Total 5xx responses</td>
</tr>
<tr>
<td>downstream_rq_ws_on_non_ws_route</td>
<td>Counter</td>
<td>Total WebSocket upgrade requests rejected by non WebSocket routes</td>
</tr>
<tr>
<td>downstream_rq_time</td>
<td>Histogram</td>
<td>Total time for request and response (milliseconds)</td>
</tr>
<tr>
<td>downstream_rq_timeout</td>
<td>Counter</td>
<td>Total requests closed due to a timeout on the request path</td>
</tr>
</tbody>
</table>

Every listener has a statistics tree rooted at `<prefix>.http.<address>`. with the following statistics:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>downstream_rq_total</td>
<td>Counter</td>
<td>Total requests</td>
</tr>
<tr>
<td>downstream_rq_http1_total</td>
<td>Counter</td>
<td>Total HTTP/1.1 requests</td>
</tr>
<tr>
<td>downstream_rq_http2_total</td>
<td>Counter</td>
<td>Total HTTP/2 requests</td>
</tr>
<tr>
<td>downstream_rq_too_large</td>
<td>Counter</td>
<td>Total requests resulting in a 413 due to buffering an overly large body</td>
</tr>
<tr>
<td>downstream_rq_completed</td>
<td>Counter</td>
<td>Total requests that resulted in a response (e.g. does not include aborted requests)</td>
</tr>
</tbody>
</table>
# The number of requests to emojify-api-v2 over 30 seconds which did not result in an error
increase(envoy_http_downstream_rq_xx{
    local_cluster="emojify-api-v2",
    envoy_response_code_class!="5"
})[30s]

# The number of requests to emojify-api-v2 over 30 seconds which resulted in an error
increase(envoy_http_downstream_rq_xx{
    local_cluster="emojify-api-v2",
    envoy_response_code_class="5"
})[30s]
Total Requests - all listeners for a proxy

Grafana

New pod started
Request Errors

Grafana

New pod started
Metrics queries - Timing
Prometheus

```prometheus
# Upstream Timing
sum(envoy_cluster_upstream_rq_time{
    envoy_cluster_name=~"cluster_emojify_api_v2_sidecar_proxy.*"
} > 0) by (quantile)

sum(rate(envoy_cluster_external_upstream_rq_time_sum{
    envoy_cluster_name=~"cluster_emojify_api_v2_sidecar_proxy.*"
}[30s])) / sum(rate(envoy_cluster_external_upstream_rq_time_count{
    envoy_cluster_name=~"cluster_emojify_api_v2_sidecar_proxy.*"
}[30s]))
```
Request Time

Grafana
Key metrics

Listener - Requests gRPC

The filter emits statistics in the cluster.<route target cluster>.grpc. namespace

<table>
<thead>
<tr>
<th>&lt;grpc service&gt;.&lt;grpc method&gt;.success</th>
<th>Counter</th>
<th>Total successful service/method calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;grpc service&gt;.&lt;grpc method&gt;.failure</td>
<td>Counter</td>
<td>Total failed service/method calls</td>
</tr>
<tr>
<td>&lt;grpc service&gt;.&lt;grpc method&gt;.total</td>
<td>Counter</td>
<td>Total service/method calls</td>
</tr>
</tbody>
</table>

- GRPC does not use HTTP status codes
- Status Codes are part of the Protocol and are reported as individual metrics

https://www.envoyproxy.io/docs/envoy/latest/configuration/listeners/stats#listener-manager
gRPC Bridge Filter

Configuration

```
"filter_chains": [  
  {  
    "filters": [  
      {  
        "name": "envoy.http_connection_manager",  
        "config": {  
          "http_filters": [  
            {  
              "name": "envoy.grpc_http1_bridge",  
              "config": {}  
            },  
            {  
              "name": "envoy.router"  
            }  
          ]  
        }  
      }  
    ]  
  }  
]
```
Metrics queries

Prometheus

# GRPC no errors - Status Code 0
sum(increase(envoy_cluster_grpc_0{
  label_app="emojify-cache"
}[30s])) by (envoy_grpc_bridge_method)

# GRPC no errors - Status Code 5
sum(increase(envoy_cluster_grpc_5{
  label_app="emojify-cache"
}[30s])) by (envoy_grpc_bridge_method)

# gRPC Errors
sum(increase(envoy_cluster_grpc_failure{
  label_app="emojify-cache"
}[30s])) by (envoy_grpc_bridge_method)
gRPC - Success

Grafana

Methods:
Put
Get
Exists
gRPC Error
Grafana

HTTP Response 5xx
Clusters

Metrics
Key metrics

Cluster

Every listener has a statistics tree rooted at `<prefix>.http.<address>`. with the following statistics:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>upstream_rq_timeout</td>
<td>Counter</td>
<td>Total requests that timed out waiting for a response</td>
</tr>
<tr>
<td>upstream_rq_per_try_timeout</td>
<td>Counter</td>
<td>Total requests that hit the per try timeout</td>
</tr>
<tr>
<td>upstream_rq_retry</td>
<td>Counter</td>
<td>Total request retries</td>
</tr>
<tr>
<td>upstream_rq_retry_success</td>
<td>Counter</td>
<td>Total request retry successes</td>
</tr>
<tr>
<td>ejections_active</td>
<td>Counter</td>
<td>Number of currently ejected hosts</td>
</tr>
</tbody>
</table>

https://www.envoyproxy.io/docs/envoy/latest/configuration/listeners/stats#listener-manager
# Retries
sum(increase(envoy_cluster_upstream_rq_retry{envoy_cluster_name=~"cluster_emojify_api_v2_sidecar_proxy.*"}[30s]))

# Timeouts
sum(increase(envoy_cluster_upstream_rq_timeout{envoy_cluster_name=~"cluster_emojify_api_v2_sidecar_proxy.*"}[30s]))
sum(increase(envoy_cluster_upstream_rq_per_try_timeout{envoy_cluster_name=~"cluster_emojify_api_v2_sidecar_proxy.*"}[30s]))

# Outlier Ejection
sum(envoy_cluster_outlier_detection_ejections_active{envoy_cluster_name=~"cluster_emojify_api_v2_sidecar_proxy.*"})
Retries

Grafana

Service Errors

Retry Applied: No errors to user
Timeouts

Grafana
Outlier Ejection

Grafana

New pod started
Constant errors

After a fixed number of consecutive errors endpoint removed from cluster

Envoy retries failing endpoint

Ejection interval increases
Control plane

AuthZ

External Authorization

1. Envoy validates that the connections is allowed by calling the ext_authz filters api (once per new connection).
2. If allowed the request is passed to the upstream service.
3. Send the response to the caller.
Control plane

AuthZ

- External authorization API is normally called when establishing a new connection to an upstream.
- Failed authorization is an indication of a failing control plane, misconfiguration of security policy, or malicious activity.
### Key AuthZ metrics

**AuthZ**

The network filter outputs statistics in the config.ext_authz. namespace, with the following statistics:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td>Counter</td>
<td>Total responses from the filter.</td>
</tr>
<tr>
<td>error</td>
<td>Counter</td>
<td>Total errors contacting the external service.</td>
</tr>
<tr>
<td>denied</td>
<td>Counter</td>
<td>Total responses from the authorizations service that were to deny the traffic.</td>
</tr>
<tr>
<td>failure_mode_allowed</td>
<td>Counter</td>
<td>Total requests that were error(s) but were allowed through because of failure_mode_allow set to true.</td>
</tr>
<tr>
<td>ok</td>
<td>Counter</td>
<td>Total responses from the authorization service that were to allow the traffic.</td>
</tr>
<tr>
<td>cx_closed</td>
<td>Counter</td>
<td>Total connections that were closed.</td>
</tr>
<tr>
<td>active</td>
<td>Gauge</td>
<td>Total currently active requests in transit to the authorization service.</td>
</tr>
</tbody>
</table>
Metrics queries
Prometheus

# Successful AuthZ
increase(envoy_ext_authz_connect_authz_ok{local_cluster="emojify-api-v2"}[1m])

# AuthZ Denied
increase(envoy_ext_authz_connect_authz_denied{local_cluster="emojify-api-v2"}[1m])
AuthZ OK

Grafana

Cached Authorization - no metrics

New pod started
AuthZ Failed

Grafana

AuthZ failure, either: Misconfiguration or Attack
What is tracing?

Distributed tracing, also called distributed request tracing, is a method used to profile and monitor applications, especially those built using a microservices architecture. Distributed tracing helps pinpoint where failures occur and what causes poor performance.

https://opentracing.io/docs/overview/what-is-tracing/
Configuration

Tracing
Tracing Cluster Configuration

```
"load_assignment": {
    "cluster_name": "cluster_tracing_honeycomb_opentracing_proxy_9411",
    "endpoints": [
        {
            "lb_endpoints": [
                {
                    "endpoint": {
                        "address": {
                            "socket_address": {
                                "address": "honeycomb-opentracing-proxy",
                                "port_value": 9411,
                                "protocol": "TCP"
                            }
                        }
                    }
                }
            ],
            "name": "cluster_tracing_honeycomb_opentracing_proxy_9411"
        }
    ]
}
```
Tracing Configuration

Configuration

```json
{
  "http": {
    "config": {
      "collector_cluster": "cluster_tracing_honeycomb_opentracing_proxy_9411",
      "collector_endpoint": "/api/v1/spans"
    },
    "name": "envoy.zipkin"
  }
}
```
Trace - HTTP Post

honeycomb.io

Public Ingress

Route upstream to API

External upstream gRPC
Handling tracing spans
var otHeaders = []string{
    "x-request-id",
    "x-b3-traceid",
    "x-b3-spanid",
    "x-b3-parentspanid",
    "x-b3-sampled",
    "x-b3-flags",
    "x-ot-span-context"
}
var headers http.Header
for _, h := range otHeaders {
    if v := r.Header.Get(h); len(v) > 0 { headers.Add(h, v) }
}

return headers
Adding Headers

HTTP

```go
headers := createHeadersFromRequest(r)

req.Header = headers

resp, err := http.DefaultClient.Do(req)
if err != nil {
    return
}
```
Adding Headers

gRPC

```golang
var otHeaders = []string{
    "x-request-id",
    "x-b3-traceid",
    "x-b3-spanid",
    "x-b3-parentspanid",
    "x-b3-sampled",
    "x-b3-flags",
    "x-ot-span-context"
}
var pairs []string
for _, h := range otHeaders {
    if v := r.Header.Get(h); len(v) > 0 {
        pairs = append(pairs, h, v)
    }
}
md := metadata.Pairs(pairs...)
return metadata.NewOutgoingContext(context.Background(), md)
```
Adding Headers

gRPC

```go
// create a grpc context containing the parent span metadata
cxt := createGRPCContextFromRequest(r)

resp, err := e.emojify.Create(cxt, &wrappers.StringValue{Value: u.String()})
if err != nil {
    return
}
```
Logging
Logging

Loki