IP Constellation Monitoring Architecture

UMAR YOUSAFZAI: SOFTWARE ENGINEERING INTERN (WINTER 2024), INSTRUCTIONAL PRODUCTS

Description

This document is to outline the monitoring architecture I added for one of the micro services in the Constellation service mesh using Open Telemetry and Apple's in house Mosaic monitoring tool. This architecture to my knowledge is becoming the industry standard and can be replicated by any team at Apple that wants observability into their micro services (among other things).

Prerequisite Readings

To keep this document concise, here are some strongly recommended prerequisite readings to help understand the common concepts in monitoring. These concepts are important to understand how to build and deploy and architecture that can be scaled to tens of services publishing millions of metrics.

- What is OpenTelemetry?
- OpenTelemetry Concepts
- Observability Primer
- Metric types | Prometheus
- Histograms Basics | Mosaic Blog
- Metrics | OpenTelemetry
- Metrics Data Model | OpenTelemetry
- Metrics Python Instrumentation | OpenTelemetry
- Mosaic | Apple Telemetry

Monitoring Purpose

Adding observability to any service may not be a top priority from a feature perspective, but

observability is critical for any system with a very high through put. A good monitoring system can save countless engineering hours when issues arise and help discover issues that were previously unknown (something we discovered in Constellation).

In short, the purpose of a monitoring system is to assess the health of a service in real time. Building on top of this, you can also monitor the service and raise warnings or alerts before potential issues can severely impact the end user.

Architecture

There are three main steps to setting up a monitoring/observability system for any service mesh. The first is to generate metrics from individual services and export them. The second step is to store them in a cloud database for the final step of querying the data, creating visualizations and setting up alerts. This can be further extended to do load testing on the services themselves to evaluate performance and setup benchmarks.

GENERATING METRICS: OPENTELEMETRY (OTEL)

To generate metrics, I utilized OTel an open source standard from the Cloud Native Computing Standard. OTel is quickly becoming the go to industry standard for collecting and exporting telemetry data and emphasizes creating an open source software with no vendor lock in. This is contrast to commercial product offerings such as Datadog which are a mesh of proprietary software and open source libraries.

OpenTelemetry supports metrics, logs and traces. For the purposes of an observability system for teams at Apple, metrics and traces are the most useful since many teams use Splunk for storing logs.

The core building blocks for metrics are counters, gauges and histograms. Counters are useful for counting error occurrences, active requests, total requests etc. Gauges are useful for sampling and recording discrete values. Histograms are useful for summarizing large quantities of data to conserve space during collection. It's very useful for understanding the underlying data distribution when individual data points by themselves are not informative. An example would be visualizing the average latency of thousands of HTTP calls.

Key things to note about using OTel. It's an open source API specification first which is used to build SDKs for different languages. While they may follow the same specifications, the actual run time behavior can differ and developers should be aware of this. The OTel SDK is initialized as the first thing in any app and used to generate instruments to create metrics. When initializing the SDK, a metric reader & periodic exporter is created from which

instruments are created. This Periodic Exporter exports metrics at set configurable intervals from the service.

Cardinality

The most important consideration when generating metrics is cardinality (explained well here: Cardinality is key). A time series consists of a value, a time stamp and a set of attributes for additional information. For example, our `error_counter` metric has an attribute `error_type` to store a string for the error name.

Because time series data is incredibly expensive storage wise, cardinality is the key tradeoff that comes with real time data. For example, a metric with five labels with five unique values can generate up to 3,125 unique time series in the worst case. Scaling this across a few pods for one service can easily cause a singular metric for one service to report tens of thousands of time series.

And while libraries such as OTel can scrape up to two million datapoints for an individual export, this tradeoff of the entropy of a metric type is the most important consideration when instrumenting a service.

METRICS EXPORT: OPENTELEMETRY PROTOCOL (OTLP)

Metrics are exporter from the Periodic Exporter in the OTLP protocol. Metrics can be exported via either gRPC or HTTP but it is highly recommended to use gRPC (and the only protocol Mosaic supports for ingestion). gRPC is high performant (up to 10x faster than HTTP) and supports bidirectional streaming which is useful since the OTLP protocol requires a server response for each batch of metrics exported.

gRPC is slightly harder to work with and debug in case of any network issues due to its enforcement of protocol buffers. Developers should just be aware of this and become familiar with tools for health checks and debugging in case of any issues.

Delta vs Cumulative Temporality

Apart from the network protocol used to export metrics, the data stream is the other important aspect to note. This is explained well here: Metrics Data Model | OpenTelemetry.

In short, cumulative temporality is when metrics are sent with cumulative values with the starting timestamp of when data collection began. For a counter metric this means that the most recent counter value would be exported, even if the value didn't change.

Delta temporality is when metrics are exported with respect to the change in values. For a counter metric, the deltas of the counter value would be reported and then aggregated in the time series database .

In general, cumulative temporality is more memory intensive. If there is a time period where metrics are dropped, cumulative temporality results in a loss of resolution in time whereas for delta temporality results in a loss of actual data. Delta temporality is usually preferred because it scales very well when aggregating the same metric from multiple different sources (ex: kube pods).

For further reading: OpenTelemetry metrics: Delta vs. Cumulative temporality trade-offs

METRICS INGESTION: MOSAIC

The generation and export of metrics are the most time consuming aspect of setting up an observability system since it requires a deep understanding of service behavior from developers/service owners. Again, it is important that service owners generate correct metrics to prevent making queries on garbage data (https://xkcd.com/2295/).

Luckily, Apple is building a great tool for telemetry ingestion and visualization called Mosaic. It's designed to provide teams with extensive control over their raw data and support industry standard features such as distributed tracing.

For this architecture, Mosaic now supports the ingestion of OTLP data via gRPC. This means that the Periodic Exporter in an OTel SDK can directly publish metrics to a gRPC endpoint.

To try out Mosaic, it's first important to understand common terminology (primarily workspace vs namespace). Teams can first create a 90 day namespace in a test workspace called `playground-gala` .

For metrics publication to the gRPC endpoint, mutual TLS (mTLS) is required. This means that users need to generate a client certificate and client key (explained here). Note that this process is simplified in Kube where each pod already has access to a client certificate chain and a client key than can be referenced for mTLS. This volume is already provided and is called `identity-certs`. In your kube config file, add the following `volume` and `volumeMount`:

```
      volumes:
      - name: identity-certs
        emptyDir: {}
      volumeMounts:
       - mountPath: /certs
```

```
          name: identity-certs
          readOnly: true
```

Once your team is ready to publish your metrics to a permanent namespace, you can be onboarded to a global workspace such as `swe-global` or `swe-sd` . Please reach out to the Mosaic team for this. You will need to create a very small AD group that will be granted admin access to this workspace to publish metrics and will need to generate certificates for mTLS accordingly.

Mosaic CLI:

Mosaic CLI is an easy tool to test out metric publication and to validate if metrics are being published. One useful query is `mosaic get utilization [workspace] [namespace]` which lists the utilization of all the metrics ingested. Useful to determine if a particular metric name was published.

METRICS VISUALIZATION & ALERTS: GRAFANA

This final part of this system is visualizing and querying your time series data. Mosaic is integrated with Grafana to provide the ability to create complex queries for visualizations.

Necessary prerequisites for learning to query data is PromQL. Some useful queries are listed here: Querying examples | Prometheus.

Histogram queries are slightly complicated to understand due to the nature of the underlying data. This is a useful resource to grasp a better understanding: Prometheus Histograms. Run that past me again? | andykuszyk.github.io.

Below is an example of some of our dashboard. The metrics below are all based on a monotonically increasing counter metric to track errors and retries. With that we are able to apply PromQL functions to determinate rate and cumulative errors (and filter by type).

Alerting:

Mosaic alerts are also based on PromQL queries. More information on setting up alerts is here: Mosaic Alerting. It's important to benchmark your service mesh before setting up alerts to avoid erroneous alerts that are filtered or ignored by developers.

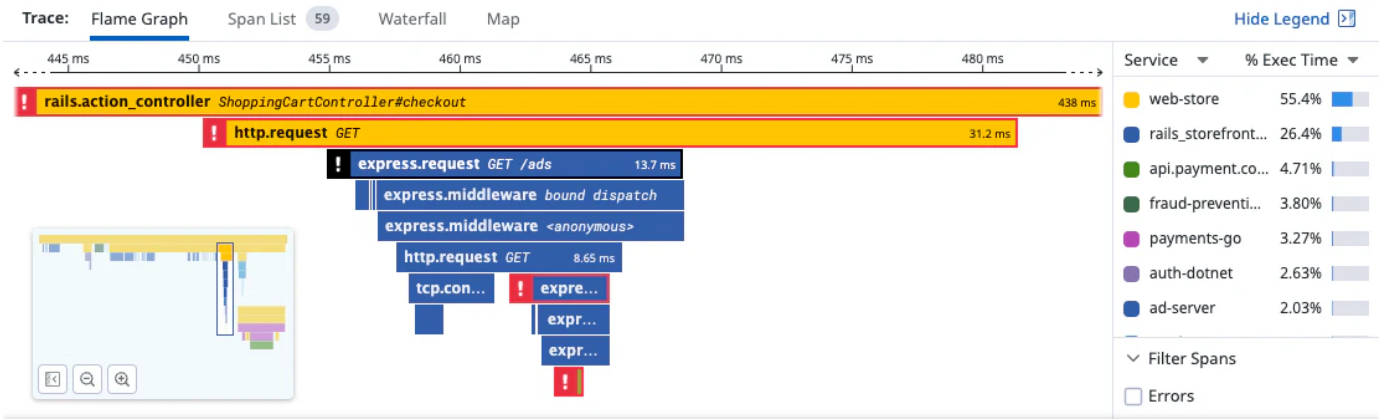
SCALE:

Scale is important for time series data due to the nature of recording events real time. From our testing for one service, we were able to reach 300k active time series in one week. Therefore, it is crucial to balance real time data monitoring with how much data needs to be published. A good explanation of this issue is here: Cardinality is key Robust Perception | Prometheus Monitoring Experts.

TLDR: Metrics should be the first line for real time observability followed by event stores and logs that can supplement with additional information.

The Next Frontier: Distributed Tracing

While the current architecture I built is solely focused on metrics, the next generation of real time observability is distributed tracing: Traces | OpenTelemetry. At a high level, distributed tracing can map the exact path of a request as it goes service to service. This is an incredibly powerful tool to have near total observability into every aspect of your service mesh. An example visualization from Datadog is below:



While OTel supports creating traces and spans for distributed tracing, its ingestion is not currently supported by Mosaic. However, it is slated to come by the end of this year at which point developers can easily begin adding this feature to their services.