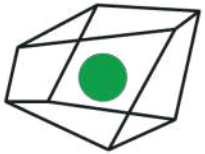


Lightstep

An introduction to OpenTelemetry

CNCF Meetup – 12 May 2022 – v4

Dimitris Finas, Sr Advisory Solution Consultant

Lightstep

Agenda

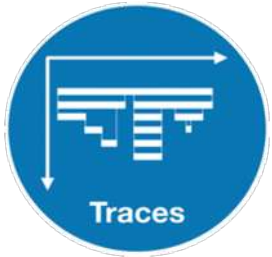
- What is OpenTelemetry?
  - A Brief History
- Distributed Tracing Overview
- OpenTelemetry
  - Instrumentation
  - Collectors
- Q&A
- Put it in action

Lightstep

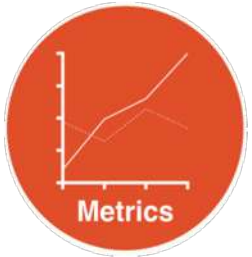
What is OpenTelemetry?

OpenTelemetry is a set of APIs, SDKs, tooling and integrations that are designed for the creation and management of *telemetry data,* such as traces, metrics, and logs.

OpenTelemetry's Mission is to enable effective observability by making high-quality, portable telemetry ubiquitous and vendor-agnostic.



A trace represents a single user's journey across multiple applications and systems (usually microservices).



Numeric data measured at various time intervals (time series data); SLI's (request rate, error rate, duration, CPU%, etc.)

Timestamped records of discrete events that happened within an application or system, such as a failure, an error, or a state transformation

You want more? See Otel vision in https://github.com/open-telemetry/community/blob/main/mission-vision-values.md#otel-mission-vision-and-values

Lightstep

OpenTelemetry - A brief history

2015

Lightstep is founded by Ben Sigelman, Daniel Spoonhower, and Ben Cronin

2016

Lightstep helps to found **OpenTracing**, the first common, portable API for distributed tracing

2017

Jaeger, an open-source tracing tool developed by Yuri Shkuro, is open-sourced by **Uber**

2018

Google releases a new portable instrumentation project called **OpenCensus**

2019

The **OpenTracing** and **OpenCensus** projects announce a merger, forming **OpenTelemetry** to be a part of CNCF

2020

OpenTelemetry is released in beta

Lightstep

Distributed systems - a brief history

2013

Docker debuted to the public in Santa Clara at PyCon in 2013.

2014

Kubernetes was first announced by Google in mid-2014

2014

Lamda, Amazon announces Lambda at AWS reinvent end-2014

2016

Envoy was fully developed at Lyft

2019

The **OpenTracing** and **OpenCensus** projects announce a merger, forming **OpenTelemetry** to be a part of CNCF

2020

OpenTelemetry is released in beta

2005

Dynatrace founded

2008

AppDynamics was founded

2008

New Relic was founded

2010

Datadog was founded

2012

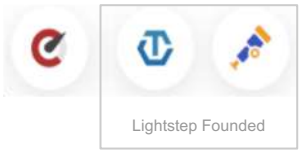
Prometheus Initial Release

2015

Lightstep was founded

Integrations

Standards



Languages

Containers, Platforms and Clouds

Tracers

Data Streaming and Storage

Service Meshes / Proxies

Deployment Automation (CI/CD)

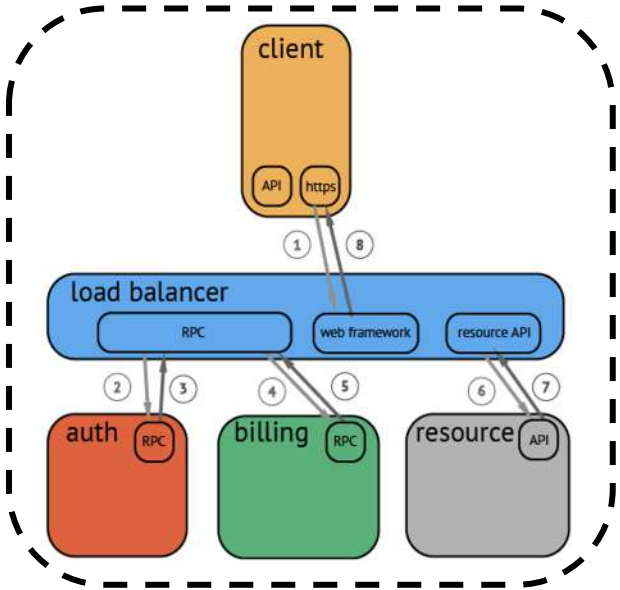
Alerting and Tools

See up to date list in https://opentelemetry.io/registry/

Lightstep

What are **distributed traces**?

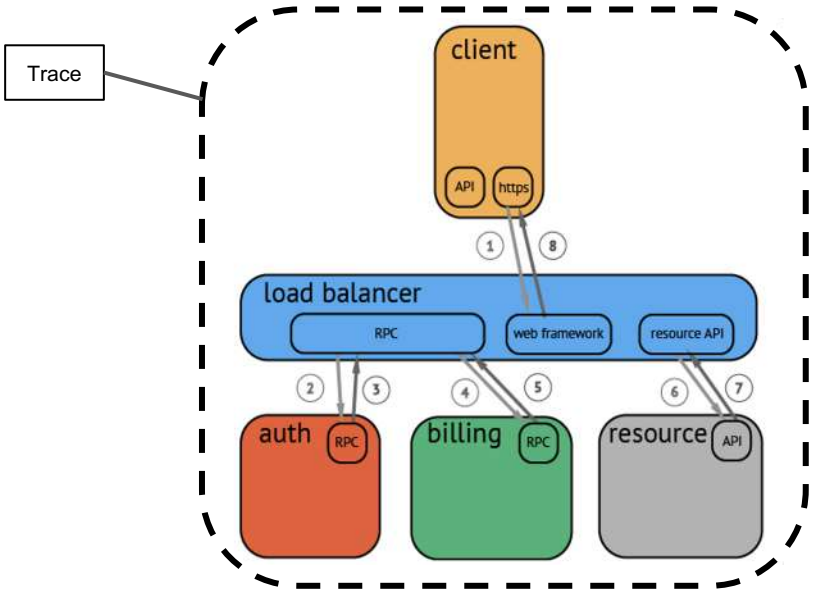
Introduction to distributed traces



A distributed trace provides a view of the life of a request as it travels across multiple hosts and services communicating over various protocols

Lightstep

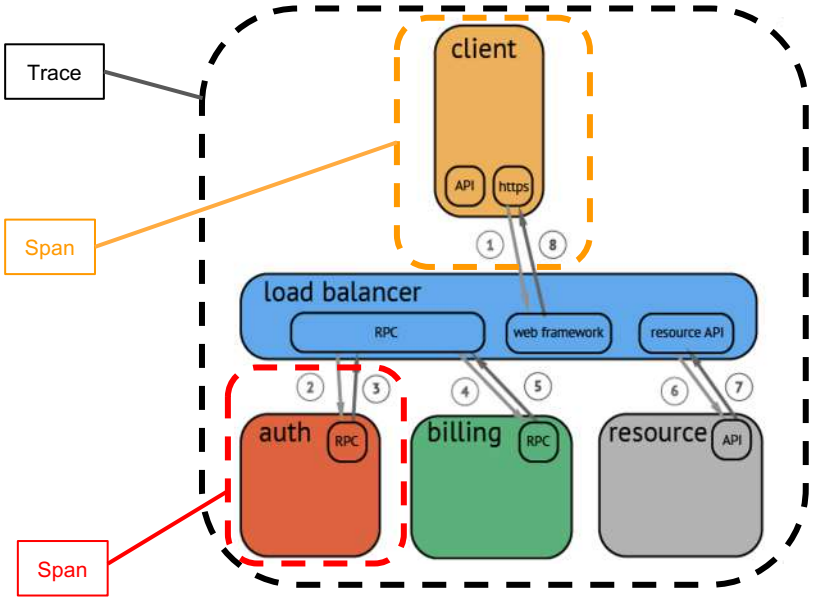
Introduction to distributed traces



A “**trace**” is a view into the request lifecycle as a whole as it moves through a distributed system.

Lightstep

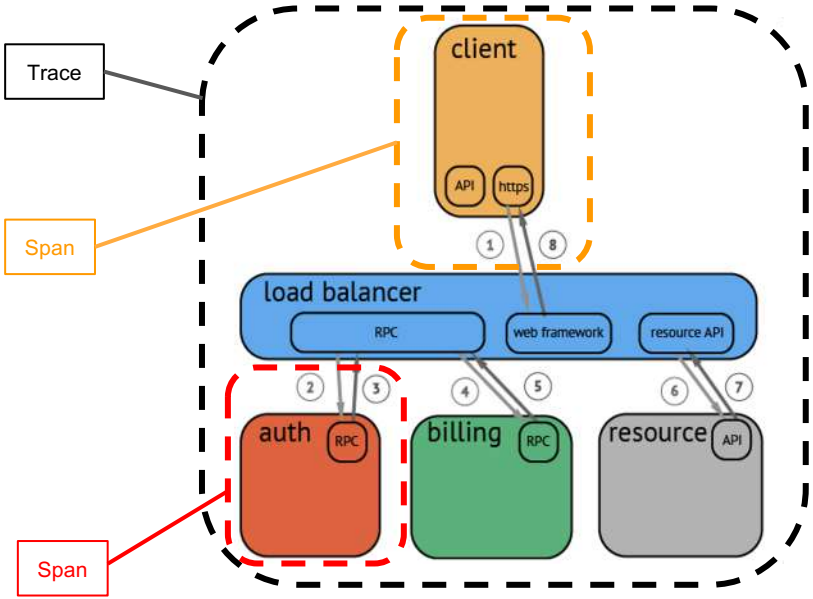
Introduction to distributed traces

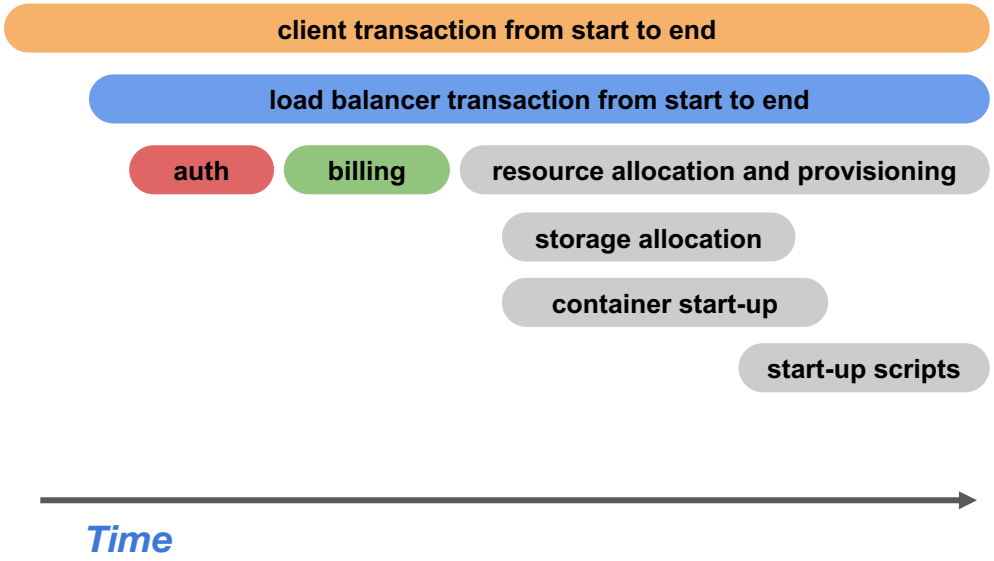


A **trace** is a collection of spans.

Each component of the distributed system contributes a "**span**" - a named, timed operation representing a piece of the workflow.

Introduction to distributed traces

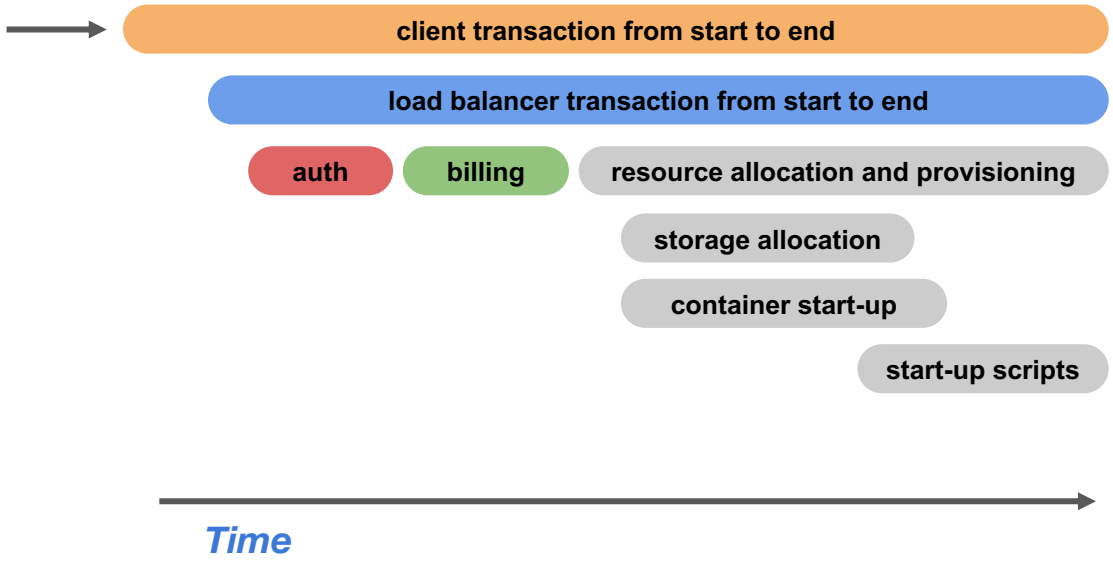




Lightstep

Introduction to distributed traces

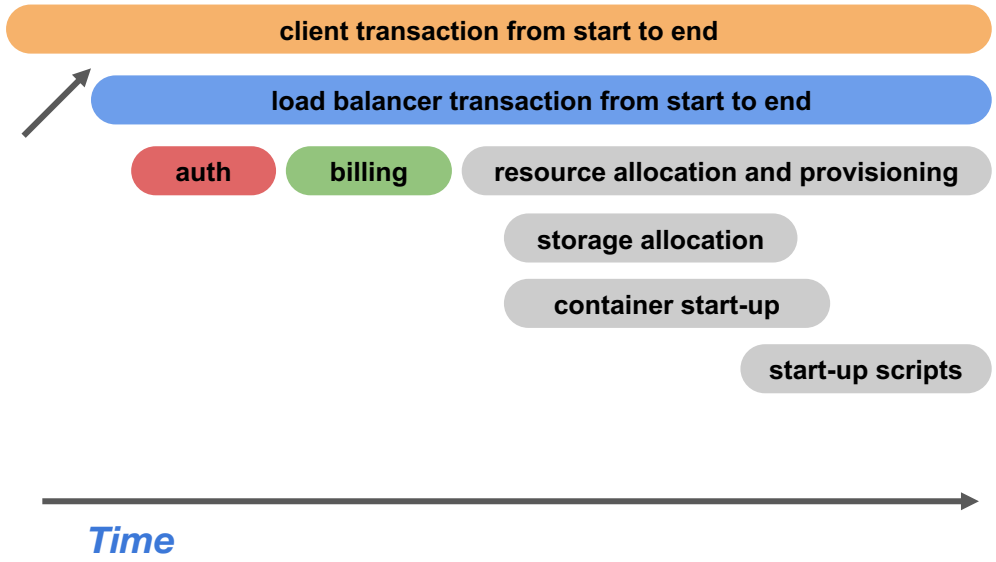
A trace can be visualized as a tree of spans with a "**root**" span at the top.



Introduction to distributed traces

Spans have **parent-child** relationships.

Parent spans can have multiple children.

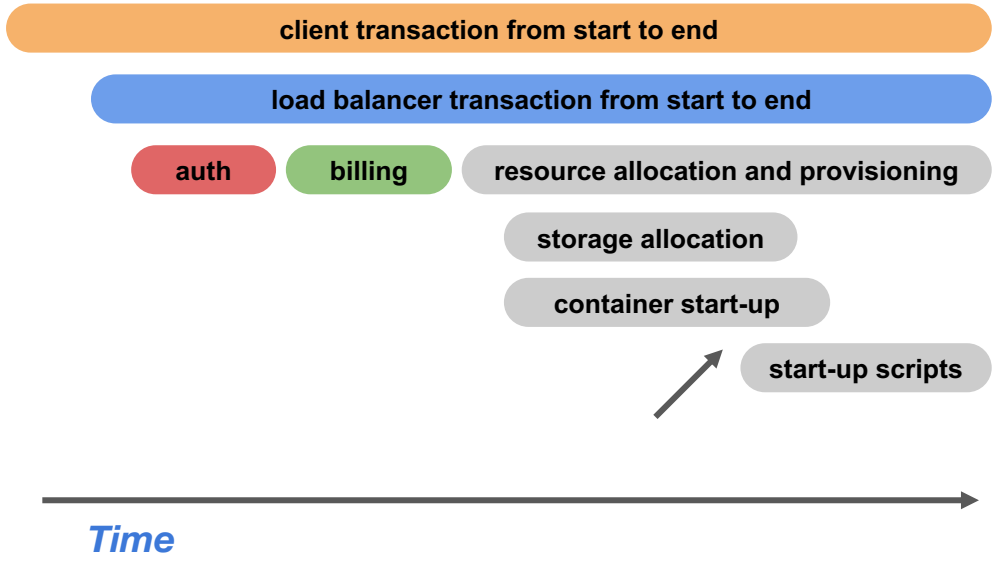


Introduction to distributed traces

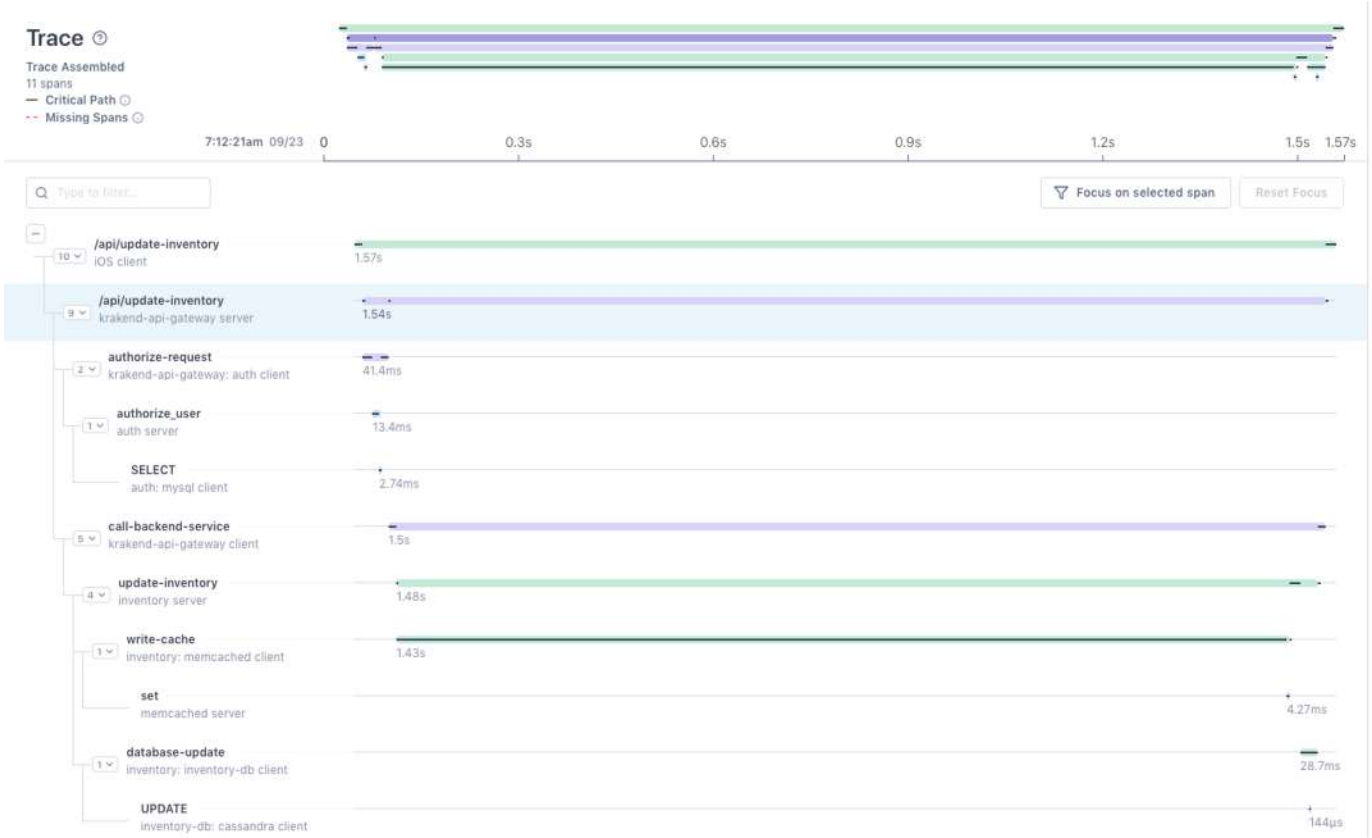
Child spans don't necessarily finish before their parents

This can be seen with asynchronous children

Real world example: An rpc call timeout out - the parent span finished earlier than the hanging child



Visualizing a trace



Service
krakend-api-gateway server

Operation
/api/update-inventory

Share

Tags & Logs | Workflow Links | Details

Tags

client.platform	iOS
customer	kicksuit
http.method	POST
http.status_code	200
runinfo.host	component-5
service.version	v4.6.1
span.kind	server
user.agent	Mozilla/5.0 (iPhone; CPU iPhone OS 12_2 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) Mobile/15E148

Logs

8µs	context_deadline: "2020-09-23T14:12:51Z"
8µs	▸ payload: 4 keys

Lightstep

Visualizing a trace

Each span also has **span context**

Span context is composed of ***attributes (tags)*** and ***events (logs)***. These are added during instrumentation.

Attributes allow you to search and segment your data within LightStep

Events (logs) add information that is useful during root cause and debugging

OpenTelemetry Instrumentation

Lightstep

OpenTelemetry support of dev languages

LANGUAGE	TRACE STATUS*	INSTRUMENTATION MANUAL/AUTO**
C++	stable	manual
C# / .NET	stable	manual & auto
Erlang / Elixir	stable	manual
Go	stable	manual
Java	stable	manual & auto
Javascript / Node	stable	manual & auto
Javascript / Browser	stable	manual & auto

LANGUAGE	TRACE STATUS*	INSTRUMENTATION MANUAL/AUTO**
PHP	pre-alpha	manual
Python	stable	manual & auto
Ruby	stable	manual & auto
Rust	beta	manual
Swift	beta	manual

You want to know more?
See https://opentelemetry.io/docs/instrumentation/

(*) Trace implementation status as of end of April 2022
(**) Automatic instrumentation means quick wins as no need to update existing code to collect some data
Supported languages versions:

- .NET & .NET Framework all supported versions except .NET Fwk v3.5 as https://github.com/open-telemetry/opentelemetry-dotnet
- Java > v1.8
- NodeJS >v10 as https://github.com/open-telemetry/opentelemetry-js
- Python, only latest versions as of https://github.com/open-telemetry/opentelemetry-python

OTel Instrumentation Steps

- Configure Tracer
- Create Spans
  - Decorate Spans
- Context Propagation

Configure Tracer

```
const opentelemetry = require('@opentelemetry/api');
const { NodeTracerProvider } = require('@opentelemetry/node');
const { SimpleSpanProcessor } = require('@opentelemetry/tracing');
const { CollectorTraceExporter } =  require('@opentelemetry/exporter-collector');

const collectorOptions = {
  serviceName: '<Service Name>', // Provide a service name
  url: 'https://<Satellite Host>:<Satellite Port>/api/v2/otel/trace', // URL and port to Lightstep Satellite
  headers: {
    'Lightstep-Access-Token': 'YOUR_TOKEN' //Lightstep Access Token
  },
};

// Create an exporter for sending span data
const exporter = new CollectorTraceExporter(collectorOptions);
// Create a provider for activating and tracking spans
const tracerProvider = new NodeTracerProvider({
  plugins: {
    http: {
      enabled: true,
      // You may use a package name or absolute path to the file.
      path: '@opentelemetry/plugin-http',
      // Enable HTTP Plugin to automatically inject/extract context into HTTP headers
    }
  }
});
// Configure a span processor for the tracer
tracerProvider.addSpanProcessor(new SimpleSpanProcessor(exporter));
// Register the tracer
tracerProvider.register();
```

Declare Dependencies

Declare Destination Options

Declare Plugins

Lightstep

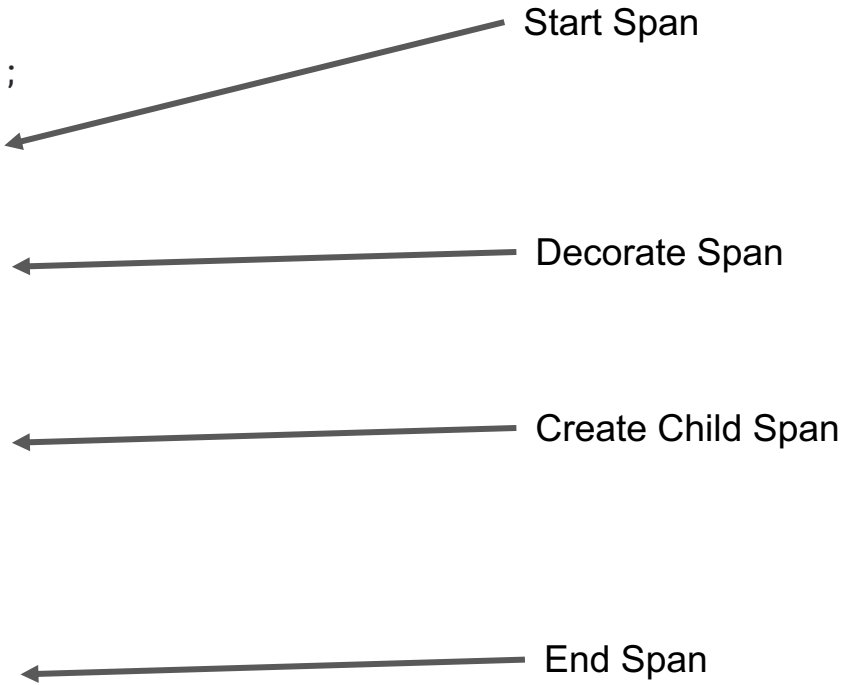
Send Telemetry To OpenTelemetry Collector

```
...
const collectorOptions = {
  serviceName: '<Service Name>', // Provide a service name
  url: 'https://<Collector Host>:<Collector Port>', // URL and port to OTel Collector
};

const exporter = new CollectorTraceExporter(collectorOptions);

...
```

Only difference between sending traffic to an OTel Collector and a Lightstep Satellite is the address and Access Token

Create Spans

```
const tracer = opentelemetry.trace.getTracer();

const span = tracer.startSpan('foo');
span.setAttribute('platform', 'osx');
span.setAttribute('version', '1.2.3');
span.addEvent('event in foo');

const childSpan = tracer.startSpan('bar', {
 parent: span,
});

childSpan.end();
span.end();
```



Tracer Span Methods

- tracer.startSpan(name, {parent: , …})
  - This method returns a child of the specified span.
- api.context.with(api.setSpan(api.context.active(), name))
  - Starts a new span, sets it to be active. Optionally, can get a reference to the span.
- api.trace.getSpan(api.context.active())
  - Used to access & add information to the current span
- span.addEvent(msg)
  - Adds structured annotations (e.g. "logs") about what is currently happening.
- span.setAttributes(core.Key(key).String(value)...)
  - Adds a structured, typed attribute to the current span. This may include a user id, a build id, a user-agent, etc.
- span.end()
  - Often used with defer, fires when the unit of work is complete and the span can be sent

Context Propagation

- Distributed context is an abstract data type that represents collection of entries.
- Each key is associated with exactly one value.
- It is serialized for propagation across process boundaries
- Passing around the context enables related spans to be associated with a single trace.
- W3C TraceContext is the de-facto standard.
- Context propagation can be automatically handled by using HTTP plugin
- Manual propagation code can be found here: https://open-telemetry.github.io/opentelemetry-js/classes/propagationapi.html

Putting It All Together

```
const opentelemetry = require('@opentelemetry/api');
const { NodeTracerProvider } = require('@opentelemetry/node');
const { SimpleSpanProcessor } = require('@opentelemetry/tracing');
const { CollectorTraceExporter } =  require('@opentelemetry/exporter-collector');

const collectorOptions = {
  serviceName: '<Service Name>', // Provide a service name
  url: 'https://<Satellite Host>:<Satellite Port>/api/v2/otel/trace', // URL and port to
Lightstep Satellite
  headers: {
    'Lightstep-Access-Token': 'YOUR_TOKEN' //Lightstep Access Token
  },
};

// Create an exporter for sending span data
const exporter = new CollectorTraceExporter(collectorOptions);
// Create a provider for activating and tracking spans
const tracerProvider = new NodeTracerProvider({
  plugins: {
    http: {
      enabled: true,
      // You may use a package name or absolute path to the file.
      path: '@opentelemetry/plugin-http',
      // Enable HTTP Plugin to automatically inject/extract context into HTTP headers
    }
  }
});
```

Continued…..

```
// Configure a span processor for the tracer
tracerProvider.addSpanProcessor(new
SimpleSpanProcessor(exporter));
// Register the tracer
tracerProvider.register();

const tracer = opentelemetry.trace.getTracer();

const span = tracer.startSpan('foo');
span.setAttribute('platform', 'osx');
span.setAttribute('version', '1.2.3');
span.addEvent('event in foo');

const childSpan = tracer.startSpan('bar', {
 parent: span,
});

childSpan.end();
span.end();
```

Lightstep

SDKs, Exporters, and Collector Services

- OpenTelemetry's **SDK** implements trace & span creation.
- An **exporter** can be instantiated to send the data collected by OpenTelemetry to the backend of your choice.
  - E.g. Jaeger, Lightstep, etc.
- OpenTelemetry **collector** proxies data between instrumented code and backend service(s). The exporters can be reconfigured without changing instrumented code.

Appendix

- NodeJS OpenTelemetry API Docs: https://open-telemetry.github.io/opentelemetry-js/
- Context Propagation API Docs: https://open-telemetry.github.io/opentelemetry-js/classes/propagationapi.html
- Plugins:
  - Core: https://github.com/open-telemetry/opentelemetry-js/tree/master/packages
  - Contrib: https://github.com/open-telemetry/opentelemetry-js-contrib/tree/master/plugins/node

Appendix (Cont'd): Basic Data Formats

JSON formatted info, output when span.end() was called.

```
"SpanContext": {
            "TraceID": "9850b11fa09d4b5fa4dd48dd37f3683b",
            "SpanID": "1113d149cfffa942",
            "TraceFlags": 1
},
"ParentSpanID": "e1e1624830d2378e",
"SpanKind": "internal",
"Name": "dbHandler/database",
"StartTime": "2019-11-03T10:52:56.903919262Z",
"EndTime": "2019-11-03T10:52:56.903923338Z",
"Attributes": [],
"MessageEvents": null,
"Links": null,
"Status": 0,
"HasRemoteParent": false,
"DroppedAttributeCount": 0,
"DroppedMessageEventCount": 0,
"DroppedLinkCount": 0,
"ChildSpanCount": 0
```

Appendix (Cont'd): Attributes & MessageEvents


```
"Attributes": [
            {
                        "Key": "http.host",
                        "Value": {
                                    "Type": "STRING",
                                    "Value": "opentelemetry-instructor.glitch.me"
                        }
            },
            {
                        "Key": "http.status_code",
                        "Value": {
                                    "Type": "INT64",
                                    "Value": 200
                        }
            }
],
"MessageEvents": [
            {
                        "Message": "annotation within span",
                        "Attributes": null,
                        "Time": "2019-11-03T10:52:56.903914029Z"
            }
],
```

OpenTelemetry Collectors

Lightstep

What is the OpenTelemetry Collector?

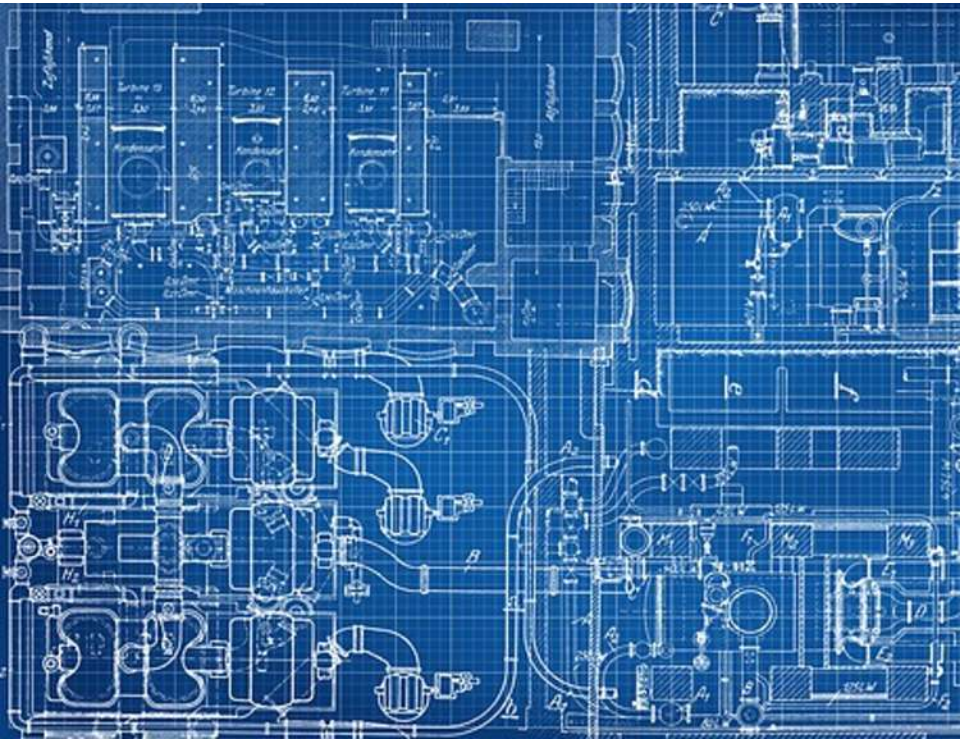
The OTel Collector is an independent binary process, written in Go, that acts as a ‘universal agent’ for the OpenTelemetry ecosystem.

At a high level, the goals of the collector are…

- Collect, process, and export telemetry data in a highly performant and stable manner.
- Support multiple types of telemetry.
- Highly extensible and customizable, but with an easy OOB experience.

The collector provides separation of concerns between Developers and Operations/SRE teams.

Architecture of the Collector



The collector has 4 major components -

- Receivers
- Processors
- Exporters
- Extensions

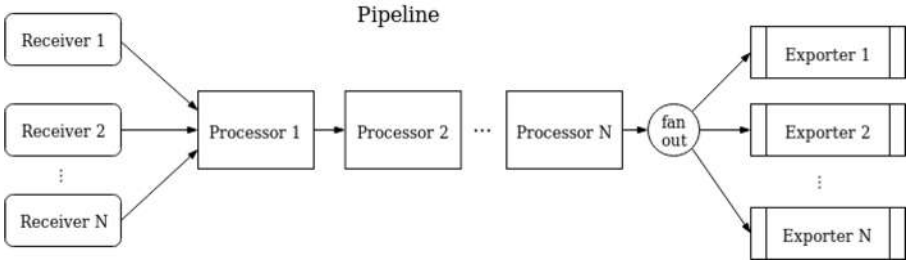
Collectors receive, process, and forward telemetry to export destinations.

These can be part of one, or more, *pipelines*.

Pipelines

Telemetry data flows through the collector, from receiver to exporter. Multiple pipelines in a single collector are configurable, but each must operate on a single *telemetry type* - either traces, or metrics.

All data received by a pipeline will be processed by all processors and exported to all exporters.



YAML

```
service:
  pipelines: # section that can contain multiple subsections, one per pipeline
    traces:  # type of the pipeline
      receivers: [opencensus, jaeger, zipkin]
      processors: [tags, tail_sampling, batch, queued_retry]
      exporters: [opencensus, jaeger, stackdriver, zipkin]
```

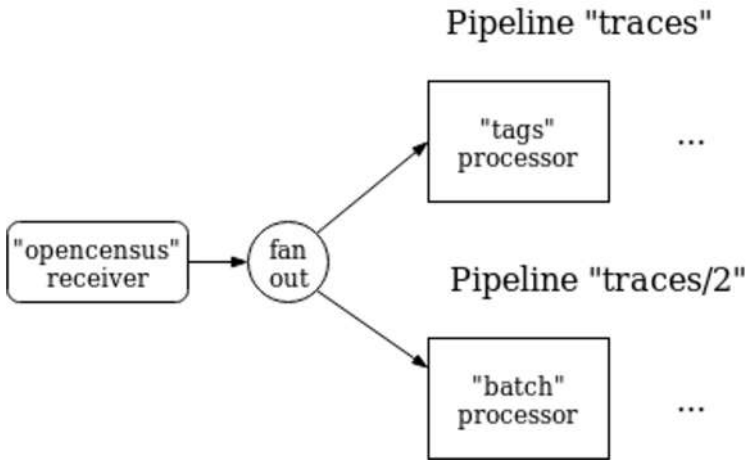
Lightstep

Receivers

Receivers listen on a single port for telemetry data. A single receiver can send to multiple pipelines.

Currently, the collector supports the following receivers -

- OTLP
- OpenCensus
- Fluentd Forward
- Jaeger
- Zipkin
- Prometheus
- Host Metrics



Receivers, Cont’d.

The collector *also* supports third-party/contributed receivers, which include:

- Amazon X-Ray
- Wavefront
- SignalFX
- Carbon
- CollectD
- Redis
- Kubernetes/Kubelet

Many of these are metrics receivers, but some (xray, SAPM) are trace receivers.

YAML

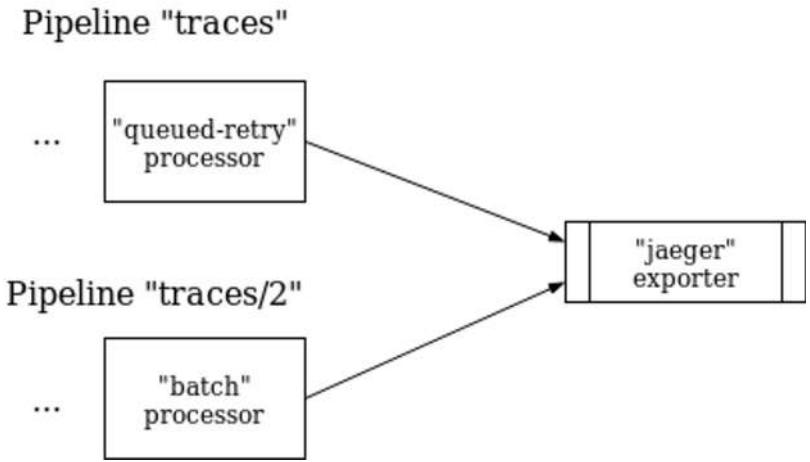
```yaml
Receivers:
  otlp:
    grpc:
      endpoint: "0.0.0.0:55678"

service:
  pipelines:
    traces:  # a pipeline of “traces” type
      receivers: [otlp]
      processors: [tags, tail_sampling, batch, queued_retry]
      exporters: [otlp]
```

Lightstep

Exporters

Typically, an exporter will forward all data it receives to a network endpoint, but can also write to a file or console.

Similar to receivers, a single exporter may be included in multiple pipelines.



YAML

```yaml
exporters:
    jaeger:
        protocols:
        grpc:
            endpoint: "0.0.0.0:14250"
    otlp:
        endpoint: <YOUR_LOAD_BALANCER_DNS_NAME_OR_IP_ADDRESS>
        headers: {"lightstep-access-token":"<YOUR_ACCESS_TOKEN>"}
service:
  pipelines:
    traces:  # a pipeline of “traces” type
      receivers: [OTLP]
      processors: [tags, tail_sampling, batch, queued_retry]
      exporters: [jaeger, otlp]
```

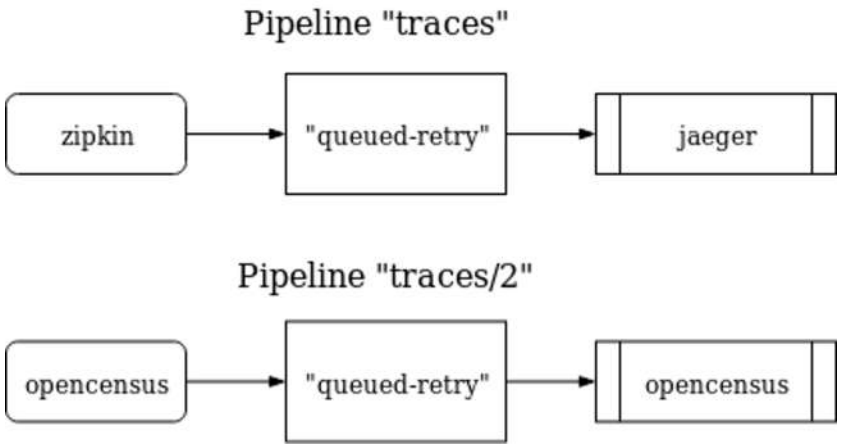
Lightstep

Processors

Processors are very powerful!

They run sequentially, and receive all data from the processor before them. A processor can transform telemetry data, delete it, etc.

A single processor configuration can apply to multiple pipelines, but each pipeline gets a unique instance of the processor.



YAML

```yaml
processors:
  queued_retry:
    size: 50
    per-exporter: true
    enabled: true
service:
  pipelines:
    traces:  # a pipeline of “traces” type
      receivers: [zipkin]
      processors: [queued_retry]
      exporters: [jaeger]
    traces/2:  # another pipeline of “traces” type
      receivers: [opencensus]
      processors: [queued_retry]
      exporters: [opencensus]
```

Lightstep

Processors Deep Dive

Receivers, Exporters, and Extensions aren't super interesting on the face of things - they receive or emit data, or simply give you some introspection on the collector instance.

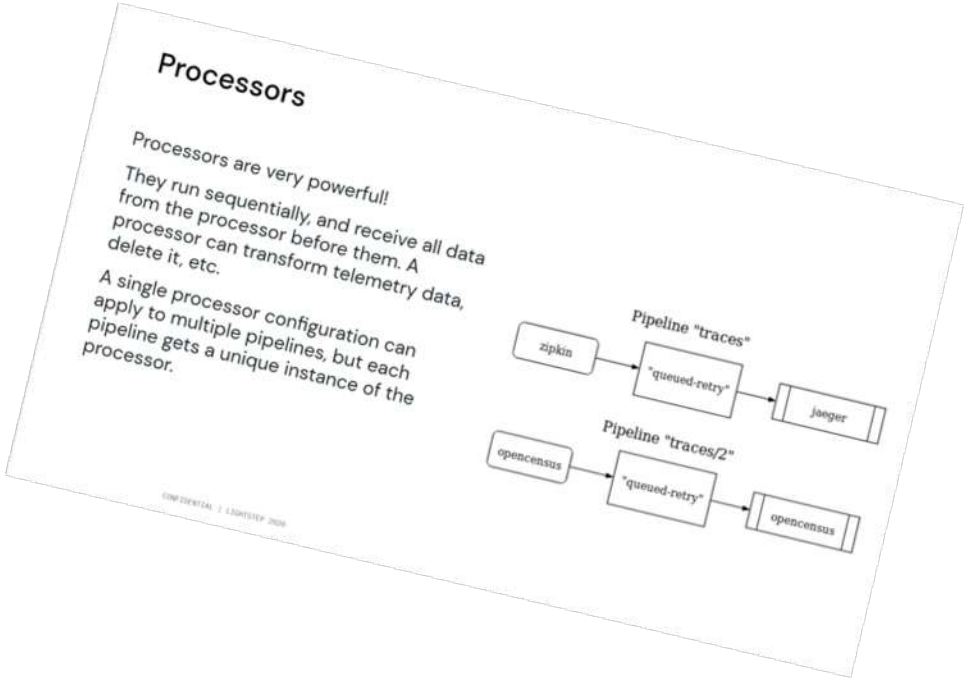
Processors, however, can do a *lot*. Let's talk about them.

Processors Ordering

Remember this?

Processors one run after the other, so the order that you define them matters.

This means that limiting and sampling should be first, and batching/transforms should be later.



Memory Limiter

The memory limiter is used to prevent a collector from going OOM.

It does this by allowing the user to configure and tune GC on the collector.

This is not a replacement for sizing/tuning collector memory/cpu.

```
processors:
  memory_limiter:
    ballast_size_mib: 2000
    check_interval: 5s
    limit_mib: 4000
    spike_limit_mib: 500
```

Lightstep

Sampling Processors

Out of the box, the collector supports a *probabilistic sampler* and a *tail-based sampler*.

Probabilistic supports OpenTracing priority sampling hints as well as trace ID based hashing.

Tail sampling allows for user-defined policies such as string/number attributes or desired rate.

```
processors:
  probabilistic_sampler:
    hash_seed: 22
    sampling_percentage: 15.3
  tail_sampling:
    decision_wait: 10s
    num_traces: 100
    expected_new_traces_per_sec: 10
    policies:
      [
          {
            name: test-policy-1,
            type: always_sample
          },
          {
            name: test-policy-2,
            type: numeric_attribute,
            numeric_attribute: {key: key1, min_value: 50, max_value: 100}
          },
          {
            name: test-policy-3,
            type: string_attribute,
            string_attribute: {key: key2, values: [value1, value2]}
          },
          {
            name: test-policy-4,
            type: rate_limiting,
            rate_limiting: {spans_per_second: 35}
        }
      ]
```

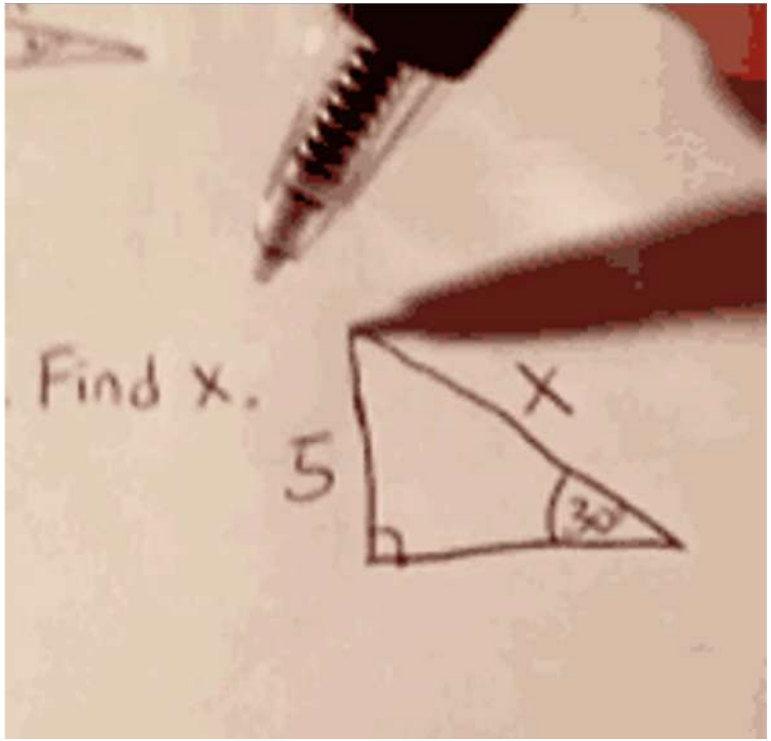
Attribute and Span Processors

Attribute processors can modify the attributes of a span; insert, update, upsert, delete, and extract based on attribute names or regular expressions.

Span processors can modify a span name or attributes based on a span name.

This can be used for more precise data scrubbing, creating attributes from span names, hashing attribute fields, and more.

Note: Resource attributes are modified with the Resource processor.

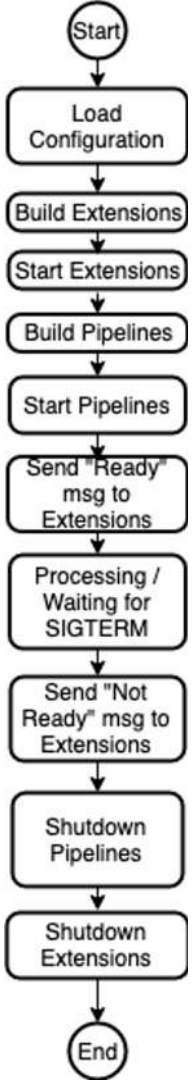


Lightstep

Extensions

Functionality can be added to the collector through Extensions.

Currently, this is used for health checks and diagnostic information.



Lightstep

```yaml
# Example of the extensions available with the core Collector. The list below
# includes all configurable options and their respective default value.
extensions:
  health_check:
    port: 13133
  pprof:
    endpoint: "localhost:1777"
    block_profile_fraction: 0
    mutex_profile_fraction: 0
  zpages:
   endpoint: "localhost:55679"

# The service lists extensions not directly related to data pipelines, but used
# by the service.
service:
  # extensions lists the extensions added to the service. They are started
  # in the order presented below and stopped in the reverse order.
  extensions: [health_check, pprof, zpages]
```

Lightstep

```yaml
receivers:
  otlp:
    protocols:
      grpc:
        endpoint: "0.0.0.0:55680"
      http:
        endpoint: "0.0.0.0:55681"
processors:
  batch:
  queued_retry:
extensions:
  health_check: {}
exporters:
  otlp:
    endpoint: "ingest.lightstep.com:443"
    headers:
      "lightstep-access-token": "{{ .Values.lightstepKey }}"
service:
  extensions: [health_check, zpages]
  pipelines:
    traces:
      receivers: [otlp]
      processors: [batch, queued_retry]
      exporters: [otlp]
```

This can be any receiver, not just OTLP. Remember, receivers can be part of multiple pipelines!

Batch and Retry processors are recommended to reduce connections and decrease size of outgoing reports.

Other export options include compression (gzip), etc. If satellites are in Single Project mode, no headers needed, just endpoint!

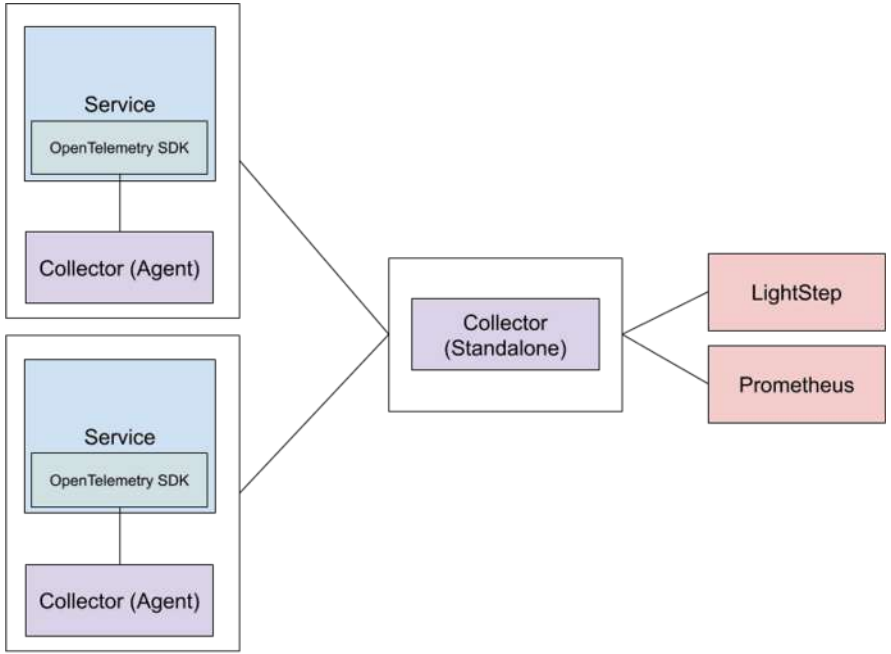
Putting It All Together

Collector Deployment Strategies

A goal of the collector is ‘5 Minutes to Value’, and part of this is easy support for a variety of deployment strategies.

The collector can be run as an ‘agent’ or as a standalone (or pool of standalone) processes.

Eventually, the collector will support distribution packages for popular targets (rpm, docker, windows…) and automatic ingest of environment metrics/tags (AWS env resources, k8s, etc.)



Further Reading: Collectors

Most collector components are well-documented, and you can find individual READMEs in each sub-directory of the https://github.com/open-telemetry/opentelemetry-collector repository.

Third-party extensions, processors, receivers, and exporters are available in the https://github.com/open-telemetry/opentelemetry-collector-contrib repository.

A Kubernetes Operator is available at https://github.com/open-telemetry/opentelemetry-operator, but tends to lag development significantly and so I don't currently recommend it.

Further Reading: OpenTelemetry

- OpenTelemetry documentation https://opentelemetry.io/docs/ and https://opentelemetry.lightstep.com/
- OpenTelemetry registry to get list of supported technologies and projects: https://opentelemetry.io/registry/
- How to choose your Observability solution: https://medium.com/dzerolabs/unpacking-observability-how-to-choose-an-observability-vendor-aa0e6d80b71d

Q&A

Put it in action

- Instrument your first application for distributed traces visibility
- Deploy your first Collector
- Connect and see results in different backends

Module Lab and Activities (1/2)

- **Activity**
  - Collect distributed traces from your application
  - Use Otel Collector to receive traces
  - See results in console output
- **Lab 1.1**
  - **Time**: 15m
  - Clone the git application repository
  - Add auto-instrumentation to NodeJS application
  - Test and see results in console output
- **Lab 1.2**
  - **Time**: 10m
  - Deploy Collector as container
  - Send traces to Collector
  - Test and see results in Collector output and debug

Module Lab and Activities (2/2)

- **Activity**
  - Send traces to different Backends
  - Add custom attributes and log events
  - Test with volume and see results in backends
- **Lab 2.1**
  - **Time**: 10m
  - Create your Lightstep account
  - Update Collector to send traces to Jaeger and Lightstep
  - Test and see results in the backends
- **Lab 2.2**
  - **Time**: 15m
  - Add custom attributes and log events
  - Start some volume tests
  - See correlated analysis in Lightstep backend

Step by Step instructions

Prerequites: you must have docker desktop installed

- git clone https://github.com/dimitrisfinas/nodejs-microservices-example.git
- cd nodejs-microservices-example
- docker-compose up --build

Once deployment finished, check application works:

- Open Browser on ”http://127.0.0.1:4000/api/data”

Then, follow instructions from file “**INSTALLING_OTEL.md**”

All URLs to test application are available from “**README.md**” in “## Starting the microservices application” section