Anatomy of a Prometheus Client Library

Brian Brazil
Founder

Robust Perception

Who am I?

- One of the main developers of Prometheus
- Author of Prometheus: Up&Running
- Founder of Robust Perception

Client Libraries

Client Libraries are what you use to instrument your code with metrics, and expose those metrics to Prometheus.

Over the past 5ish years we've largely figured out what a client library should, and shouldn't, do.

This talk will look at how client libraries are designed.

Direct Instrumentation

Client libraries are intended to be used via "Direct Instrumentation", that is you use the base Gauge/Counter/Summary/Histogram classes directly in your own application code.

This is contrasted with "Custom Collectors", which are used to write exporters.

Keep Simple Things Simple

One principle of instrumentation is that it should be easy to use.

While most time series have labels, most metrics don't. So the label-less case should be the easiest:

```
FAILS = Counter('my_failures_total', 'Description')
FAILS.inc(4)
```

The metric declaration should be at file/class level.

Robust **Perception**

Things to Note - 1

The usage is to create the counter, keep a reference to it, and then to increment it. Not this:

```
prometheus.inc("counter_name", 4)
```

Firstly this is slower, as you'd have to look up a thread-safe map for the metric.

Robust Perception

Things to Note - 2

Secondly, it would encourages sharing metrics across files which you should ~never do.

Remember: Different file, different metric name.

Thirdly the Prometheus approach ensures the metric is initialised and has a value, even if never gets increments/used. Missing time series are hard to deal with.

Robust **Perception**

Things to Note - 3

Another thing is that a counter can be incremented by values other than one. This allows tracking bytes processed etc.

Finally, this is all the code. There's no need for you to worry about concurrency or other bookkeeping.

That's a lot of design for two lines of code:

```
FAILS = Counter('my_failures_total', 'Description')
FAILS.inc(4)
```

Robust Perception

Other Types

In addition to the Counter, there's also a Gauge, Summary, and Histogram.

Internally these are all* just a set of floating point numbers that get add/subtracted/set. Each individual event is not saved. For example a counter internally is essentially:

```
def inc(self, v=1):
  self.value += v
```

* Summary quantiles are more complicated.

Summary and Histogram

A Summary is two counters, and potentially a set of quantiles calculated on the client side.

A Histogram is two counters, plus a counter per bucket. In the exposition format the histogram is cumulative, though it's usually kept non-cumulative in memory to reduce contention.

Histogram buckets must be pre-chosen, with a reasonable default.

Robust **Perception**

Types of Types

Metric types in Prometheus are defined by their semantics, not by their APIs as is the case in something like DropWizard.

In DropWizard a Counter can be incremented/decremented by 1, whereas a Gauge is a callback.

By contrast, both of those cases covered by the Gauge type in Prometheus. Callbacks we'll cover later.

Timing and Units

There is no "Timer" type in Prometheus, as it is only a special case of tracking how big an event is, or how many bytes it used. All of these use the Summary or Histogram.

As timing is a common use case, there are utilities for that:

```
with a_summary.time():
  pass # Your code here
```

Seconds are always used, as they are the base unit for time.

Labels

I should probably cover one of the key features of Prometheus. Labels are key-value pairs that in addition to the metric name identify a metric. For example:

```
FAILS = Counter('my_failures_total', 'Description',
  labels=['path'])
FAILS.labels('/foo').inc(4)
```

Label names must be known in advance, if not you're probably doing event logging.

Child

The return value of `FAILS.labels('/foo')` is called a "Child".

Internally each metric with labels has a thread-safe map that it stores these in. If a `labels` call is for a labelset that doesn't exist yet, one will be created.

It is safe to cache Child objects, and it saves a lookup on each event. Not always possible though.

Robust **Perception**

Child Lifecycle

You should try to ensure that time series don't appear and disappear, as that's difficult to deal with. In an ideal world you you would initialise all potential label values in advance:

```
FAILS.labels('/foo')
FAILS.labels('/bar)
```

There is a `remove` method, however it is to be avoided outside of unit tests. A time series should exist from when a process starts until it terminates.

Robust **Perception**

Registry

A metric object in isolation is rarely useful. While the exact details depend on the client library, metrics are typically registered into a "Registry".

Registries do sanity checks, such as for duplicate metrics.

Registries do not produce the exposition format. They can be asked to call all the collectors registered with them, and provide them with the resultant metrics.

Collectors

The standard Counter&friends are just one type of collector. They only return one metric run.

You can also have "Custom Collectors" which can return as many as they like, and do things not sane with direct instrumentation such as setting counters. Custom collectors are for pulling data from other instrumentation systems.

All collectors are callback based.

Robust **Perception**

Standard Exports and Runtime Stats

One example of a custom collector are the "Standard Exports".
These are metrics such as process_cpu_seconds_total and process_fds_open which have been standardised across client libraries and typically exposed by default.

Client libraries also usually expose metrics about their runtime, such as garbage collection stats and runtime versions.

The Default Registry

As part of keeping simple things simple, a global default registry is typically used.

You can use a custom registry, if you have more complex use cases. The PushGateway or unittests for example.

This design is debated. Global mutable state has been abused greatly over the decades, however that doesn't mean that all global mutable state is automatically bad.

Exposition

Finally, we get to exposition. Producing output that Prometheus can parse.

Exposition code typically accepts a HTTP request for /metrics, asks the registry for metrics, and then renders the exposition format. This is a public API of the registry, so can produce other formats and work in other ways (e.g. PGW).

Prometheus clients don't lock you in!

Missing Features

In line with overall Prometheus design, there are various features explicitly not present:

- Setting a metric name prefix on all exposed metrics
- Adding a label to all exposed metrics
- Varying labelnames within a metric
- Being able to set a Counter
- Pushing regularly to the PushGateway
- Set a timestamp on direct instrumentation

The Future

The OpenMetrics format is coming, and with it new types.

Info and Enum will make those patterns easier to use.
GaugeHistogram clarifies a rare case for custom collectors.

This will largely be transparent, and look very similar to today.

However! Counter time series that are missing _total will now have it.

Robust Perception

Resources

Client Library Guidelines: https://prometheus.io/docs/instrumenting/writing_clientlibs/

Book: http://shop.oreilly.com/product/0636920147343.do

Robust Perception Blog: www.robustperception.io/blog

Queries: prometheus@robustperception.io

Robust Perception