So You Want to Write an Exporter

Brian Brazil
Founder

Robust Perception

Who am I?

Engineer passionate about running software reliably in production.

- Prometheus Core developer
- Studied Computer Science in Trinity College Dublin.
- Google SRE for 7 years, working on high-scale reliable systems.
- Contributor to many open source projects, including Prometheus, Ansible, Python, Aurora and Zookeeper.
- Founder of Robust Perception, provider of commercial support and consulting for Prometheus.

Ideals

The way to get the most out of Prometheus is to directly instrument your code.

Using the client libraries inside your applications and libraries gives you inclusive monitoring - a view all through your software stack.

That's not always possible though...

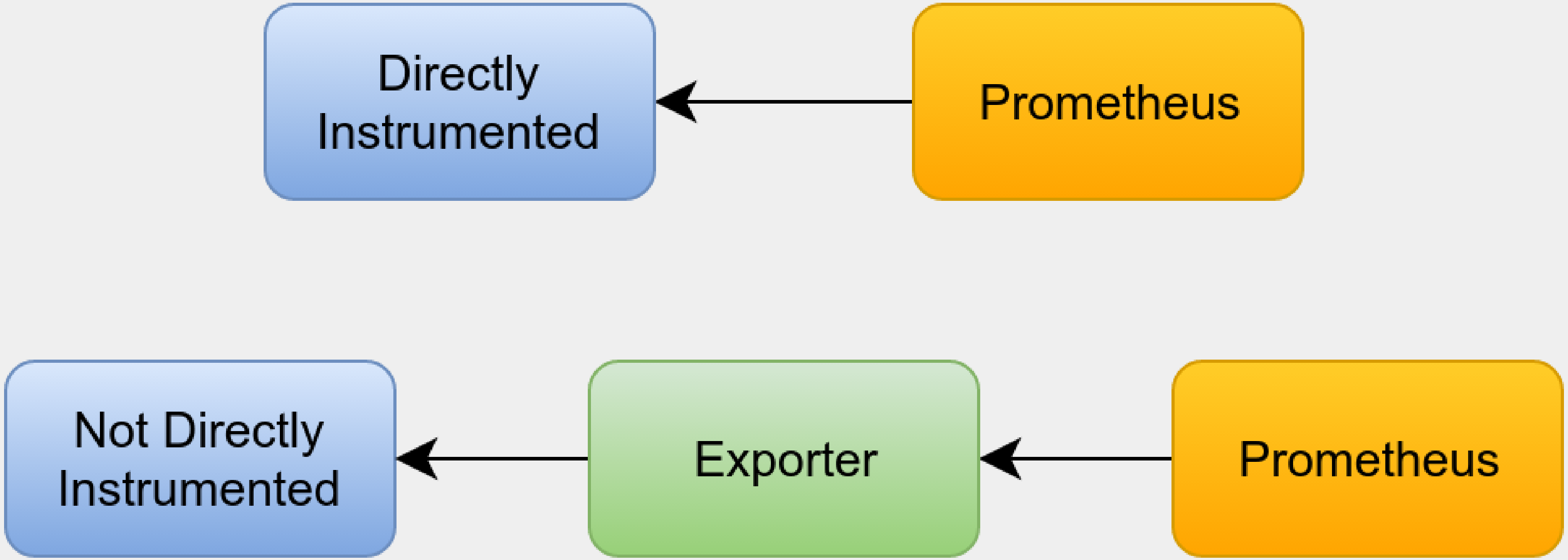
Enter Exporters

There's a small and growing number of 3rd party applications that support Prometheus metrics exposition, but today most tend not to.

Applications expose metrics in a myriad of formats with very little in common.

Thus each requires custom code to convert, which we put in a binary called an exporter.

Where Exporters Fit



Robust Perception

Architecture

We want exporters to behave as much like a directly instrumented app as we can.

This means that scheduling and service discovery belong in Prometheus.

Exporters fetch fresh data on each scraped by Prometheus and return it statelessly.

Each exporter should only talk to one instance of an app. Exceptions are when it's nonsensical to run an exporter beside the app, e.g. SNMP and blackbox.

Exporters should not use the pushgateway, and should not cache unless a query is really expensive (minutes).

Tradeoffs

Direct instrumentation tends to be very black and white in terms of how you should do things.

Exporters on the other hand are very murky, as there's some non-Prometheus data model and philosophy behind the metrics.

You need to decide whether it's okay to spend a lot of effort getting it perfect, or do something that's easier to maintain that's good enough.

I'm going to go into all the rules you should be following for direct instrumentation, and talk about why you might bend them for an exporter.

Metric naming - Units

```
# Bad: no unit
request_latency

# Better: non-base unit
request_latency_milliseconds

# Good: standard base units
request_latency_seconds
```

Prometheus has standardised on base units like seconds and bytes. Convert where practical. Always try put in the unit in use.

Metric naming - Namespaces

```
# Bad: what sort of request?
request_latency_seconds

# Bad: where in the stack?
http_request_latency_seconds

# Good: Ah, when it enters the HTTP server
http_server_request_latency_seconds
```

Someone only vaguely familiar with your system should be able to make a good guess as to what a metric means.

Think libraries, not applications.

Metric naming - Measuring latency

```
# Bad: Latency calculated on client
http_server_request_latency_seconds

# Good: Prometheus Histogram/Summary-style counters
http_server_request_latency_seconds_count
http_server_request_latency_seconds_sum
```

If the data is available to calculate latency in Prometheus, expose it.

Robust Perception

Metric naming - Ratios

```
# Bad: Non-base units, incorrect units
http_server_requests_failed_percentage
http_server_requests_failed_percentage  # Actually a ratio


# Better: Base units
http_server_requests_failed_ratio

# Best: Raw data as Counters (or Gauges, depending)
http_server_requests_total
http_server_requests_failed_total
```

Robust Perception

Metric naming - Suffixes

```
# Bad: Counter without suffix
http_server_requests

# Good: Correct counter suffix
Http_server_requests_total

# Bad: Using Counter/Summary/Histogram suffixes on a Gauge
http_server_queued_requests_count
http_server_queued_requests_total
# Good: Gauge has no suffix
http_server_queued_requests
```

Robust **Perception**

Metric naming - mixed types

You'll sometimes come across a list of metrics, some are Counters others Gauges. For example in SHOW GLOBAL STATUS, MySQL has:

```
threads_created
threads_running
```

You could go through the 497 entries and classify them by hand, and add `_total`

Or you could just leave them as is and expose them as Untyped.

This is a tradeoff you have to make.

Metrics - General

Don't do math to create metrics, handle it on the Prometheus side.

Don't put the labels of a metric in the name e.g. `_by_type`, it'll make no sense when that label is aggregated away.

`snake_case` is standard, often not practical to convert in exporters.

`process_` and `scrape_` prefixes are reserved.

Labels

If there are useful labels then it's good to extract them.

Don't add labels just because something looks related or "label-like".

Taking another example from MySQL:

```
Innodb_buffer_pool_pages_data
Innodb_buffer_pool_pages_dirty
Innodb_buffer_pool_pages_flushed
Innodb_buffer_pool_pages_free
Innodb_buffer_pool_pages_latched
Innodb_buffer_pool_pages_misc
```

Labels - Partition

Timeseries within a metric shouldn't overlap.

A sum or average across the metric should make sense.

```
my_metric{l="a"}
my_metric{l="b"}
my_metric{l="c"}
my_metric{l="d+a"}      # Not a partition
my_metric{l="total"}  # Breaks aggregation: Remove.
my_metric{}             # Breaks aggregation: Just No.
```

Robust **Perception**

Labels - The Table Exception

Sometimes data shouldn't be a label, but it's the only sane way to work with on the PromQL end.

Consider hardware temperature sensors.

You can't sum or average temperatures (Joules are what you want).

They're fairly non-standard and there's hundreds of types, so putting the adapter types into the metric would not allow for usable dashboards.

So having one metric with an "adapter" and "sensor" label makes sense.

Labels - Failure and Caches

It's better to have separate Counters for total and failures, rather than a `result` label. Easier to work with in PromQL.

Similarly for caches, export hits and total as Counters.

Don't break out latency metrics by success/failure with a label.

It'll cause confusion when latency goes up due to failures timing out but you can't see it in success-only graphs (yes, people will do that and forget the subtlety in an emergency).

Labels - General Advice

Don't have a static label returned on all metrics, use machine role approach or target labels depending on what you're doing.

All time series of a metric should have consistent label names.

For clarity and to reduce chance of collisions avoid le,quantile, instance, job, type, id, cluster, datacenter, az, zone, region, service, team, env, group etc. as label names.

Avoid time series that aren't always set.

If you're unsure, don't use a label.

Metrics to drop

Some metrics aren't useful.

Stddev, Min and Max are statistically useless. And over what time period anyway?

Machine metrics (e.g. filesystem, memory disk) - use the Node exporter.

For JMX, the jmx_exporter already covers process and JVM stats.

Metrics with high cardinality or churn - they're too expensive.

Consider carefully whether to keep quantiles.

Exporters - Failed scrapes

Sometimes talking to the application won't work. How to deal with that?

You could return a 500, which will mean up will be 0. Simple.

If your exporter can still expose some useful metrics in this case (e.g. process stats), you could have a **`myexporter_up`** metric which the user has to check for 0/1.

Useful to have a **`myexporter_scrape_duration_seconds`**.

Robust **Perception**

Exporters - other tips

Keep config minimal and simple. Just a flag with the app endpoint is best.

Don't ask about or try to expose timestamps. It's very unlikely you need to.

There's a port registry in the Prometheus server's github wiki. Grab one.

Always use ConstMetrics and their equivalents in other client libraries.

Robust Perception

Example

```
from prometheus_client import start_http_server
from prometheus_client.core import GaugeMetricFamily, REGISTRY

class MyCollector(object):
  def collect(self):
    # Your code here
    yield GaugeMetricFamily('my_gauge', 'Help text', value=42)

REGISTRY.register(MyCollector())
start_http_server(1234)
while True: time.sleep(1)
```

Resources

Official Docs: https://prometheus.io/docs/instrumenting/writing_exporters/

Example: http://www.robustperception.io/writing-a-jenkins-exporter-in-python/

Robust Perception Blog: www.robustperception.io/blog

Queries: prometheus@robustperception.io