Prometheus Client Library Guidelines

This document covers what functionality and API Prometheus client libraries should offer, with the aim of consistency across libraries, making the easy use cases easy and avoiding offering functionality that may lead users down the wrong path.

There are 4 languages supported at the time of writing, so we've gotten a good sense by now of how to write a client. These guidelines aim to help authors of new client libraries produce good libraries.

Conventions

MUST/MUST NOT/SHOULD/SHOULD NOT/MAY have the meanings given in https://www.ietf.org/rfc/rfc2119.txt

In addition ENCOURAGED means that a feature is desirable for a library to have, but it's okay if it's not present. In other words, a nice to have.

Things to keep in mind

- Take advantage of each language's features
- The common use cases should be easy
- The correct way to do something should be the easy way
- More complex use cases should be possible

The common use cases are (in order):

- No-label counters spread liberally around libraries/applications
- Timing functions/blocks of code in Summaries/Histograms
- Gauges to track current states of things (and their limits)
- Monitoring of batch jobs

Overall structure

Clients MUST be written to be callback based internally. Clients SHOULD generally follow the structure described here.

The key class is the Collector. This has a method (typically called ‘collect’) that returns zero or more metrics and their samples. Collectors get registered with a CollectorRegistry. Data is exposed by passing a CollectorRegistry to a class/method/function “bridge”, which returns the metrics in a format Prometheus supports. Every time the CollectorRegistry is scraped it must callback to each of the Collectors’ collect method.

The interface most users interact with are the Counter, Gauge, Summary, and Histogram Collectors. These represent a single metric, and should cover the vast majority of use cases where a user is instrumenting their own code.

More advanced uses cases (such as proxying from another monitoring/instrumentation system) require writing a custom Collector. Someone may also want to write a “bridge” that takes a CollectorRegistry and produces data in a format a different monitoring/instrumentation system understands, allowing users to only have to think about one instrumentation system.

CollectorRegistry SHOULD offer register()/unregister() functions, and a Collector SHOULD be allowed to be registered to multiple CollectorRegistrys.

Client libraries MUST be thread safe.

For non-OO languages such as C, client libraries should follow the spirit of this structure as much as is practical.

Naming

Client libraries SHOULD follow function/method/class names mentioned in this document, keeping in mind the naming conventions of the language they’re working in. For example, set_to_current_time() is good for a method name Python, but SetToCurrentTime() is better in Go and setToCurrentTime() is the convention in Java. Where names differ for technical reasons (e.g. not allowing function overloading), documentation/help strings SHOULD point users towards the other names.

Libraries MUST NOT offer functions/methods/classes with the same or similar names to ones given here, but with different semantics.

Standard metrics types

Counter, Gauge, Summary and Histogram are the primary interface by users.

Counter and Gauge MUST be part of the client library. At least one of Summary and Histogram MUST be offered.

These should be primarily used as file-static variables, that is, global variables defined in the same file as the code they're instrumenting. The client library SHOULD enable this. The common use case is instrumenting a piece of code overall, not a piece of code in the context of one instance of an object. Users shouldn't have to worry about plumbing their metrics throughout their code, the client library should do that for them (and if it doesn't, users will write a wrapper around the library to make it "easier" - which rarely tends to go well).

There MUST be a default CollectorRegistry, the standard metrics MUST by default implicitly register into it with no special work required by the user. There MUST be a way to have metrics not register to the default CollectorRegistry, for use in batch jobs and unittests. Custom collectors SHOULD also follow this.

Exactly how the metrics should be created varies by language. For some (Java, Go) a builder approach is best, whereas for others (Python) function arguments are rich enough to do it in one call.

For example in the Java Simpleclient we have:

```
import io.prometheus.client.Counter;

class YourClass {
  static final Counter requests = Counter.build()
      .name("requests_total")
      .help("Requests.").register();
}
```

This will register requests with the default CollectorRegistry. By calling build() rather than register() the metric won't be registered (handy for unittests), you can also pass in a CollectorRegistry to register() (handy for batch jobs).

Counter

Counter is a monotonically increasing counter. It MUST NOT allow the value to decrease, however it MAY be reset to 0 (such as by server restart).

A counter MUST have the following methods:

inc(): Increment the counter by 1
inc(double v): Increment the counter by the given amount. MUST check that v >= 0.

A counter is ENCOURAGED to have:
A way to count exceptions throw/raised in a given piece of code, and optionally only certain types of exceptions. This is count_exceptions in Python.

Counters MUST start at 0.

Gauge

Gauge represents a value that can go up and down.

A gauge MUST have the following methods:

inc(): Increment the gauge by 1
inc(double v): Increment the gauge by the given amount
dec(): Decrement the gauge by 1
dec(double v): Decrement the gauge by the given amount
set(double v): Set the gauge to the given value

Gauges MUST start at 0, you MAY offer a way for a given gauge to start at a different number.

A gauge SHOULD have the following methods:

set_to_current_time(): Set the gauge to the current unixtime in seconds.

A gauge is ENCOURAGED to have:

A way to track in-progress requests in some piece of code/function. This is track_inprogress in Python.

A way to time a piece of code and set the gauge to its duration in seconds. This is useful for batch jobs. This is startTimer/setDuration in Java and the time() decorator/context manager in Python. This SHOULD match the pattern in Summary/Histogram (though ‘set’ rather than ‘observe’).

Summary

A summary samples observations (usually things like request durations) over sliding windows of time and provides instantaneous insight into their distributions, frequencies, and sums.

A summary MUST NOT allow the user to set “quantile” as a label name, as this is used internally to designate summary quantiles. A summary is ENCOURAGED to offer quantiles as exports, though these can’t be aggregated and tend to be slow. A summary MUST allow not having quantiles, as just _count/_sum is quite useful and this MUST be the default.

A summary MUST have the following methods:

observe(double v): Observe the given amount

A summary SHOULD have the following methods:
Some way to time code for users in seconds. In Python this is the time() decorator/context manager. In Java this is startTimer/observeDuration. Units other than seconds MUST NOT be offered (if a user wants something else, they can do it by hand). This should follow the same pattern as Gauge/Histogram.

Summary _count/_sum MUST start at 0.

Histogram

Histograms allow aggregatable distributions of events, such as request latencies. This is at its core a counter per bucket.

A histogram MUST NOT allow “le” as a user-set label, as “le” is used internally to designate buckets.

A histogram MUST offer a way to manually choose the buckets. Ways to set buckets in a linear(start, width, count) and exponential(start, factor, count) fashion SHOULD be offered. Count MUST exclude the +Inf bucket.

A histogram SHOULD have the same default buckets as other client libraries. Buckets MUST NOT be changeable once the metric is created.

A histogram MUST have the following methods:

observe(double v): Observe the given amount

A histogram SHOULD have the following methods:
Some way to time code for users in seconds. In Python this is the time() decorator/context manager. In Java this is startTimer/observeDuration. Units other than seconds MUST NOT be offered (if a user wants something else, they can do it by hand). This should follow the same pattern as Gauge/Summary.

Histogram _count/_sum and the buckets MUST start at 0.

Further metrics considerations

Providing additional functionality in metrics beyond what's documented above as makes sense for a given language is ENCOURAGED.
If there's a common use case you can make simpler then go for it, as long as it won't encourage undesirable behaviours (such as suboptimal metric/label layouts, or doing computation in the client).

Labels

Labels are one of the most powerful aspects of Prometheus, but easily abused. Accordingly client libraries must be very careful in how labels are offered to users. See http://prometheus.io/docs/practices/instrumentation/#use-labels and http://prometheus.io/docs/practices/instrumentation/#do-not-overuse-labels

Client libraries MUST NOT under any circumstances allow users to have different label names for the same metric for Gauge/Counter/Summary/Histogram or any other Collector offered by the library.
If your client library does validation of metrics at collect time, it MAY also verify this for custom Collectors.

While labels are powerful, the majority of metrics will not have labels. Accordingly the API should allow for labels but not dominate it.

A client library MUST allow for optionally specifying a list of label names at Gauge/Counter/Summary/Histogram creation time. A client library SHOULD support any number of label names. A client library MUST validate that label names meet the requirements in http://prometheus.io/docs/concepts/data_model/#metric-names-and-labels

The general way to provide access to labeled dimension of a metric is via a labels() method that takes either a list of the label values or a map from label name to label value and returns a “Child”. The usual .inc()/.dec()/.observe() etc. methods can then be called on the Child.

The Child returned by labels() SHOULD be cacheable by the user, to avoid having to look it up again - this matters in latency-critical code.
Metrics with labels SHOULD support a remove() method with the same signature as labels() that will remove a Child from the metric no longer exporting it, and a clear() method that removes all Children from the metric. These invalidate caching of Children.

There SHOULD be a way to initialize a given Child with the default value, usually just calling labels(). No-labels metrics MUST always be initialized. See http://prometheus.io/docs/practices/instrumentation/#avoid-missing-metrics

Metric names

Metric names must match the restrictions in http://prometheus.io/docs/concepts/data_model/#metric-names-and-labels. As with label names, this MUST be met for uses of Gauge/Counter/Summary/Histogram and in any other Collector offered with the library.

Many client libraries offer setting the name in three parts: namespace_subsystem_name of which only the “name” is mandatory.

Dynamic/generated metric names or subparts of metric names MUST be discouraged, except when a custom Collector is proxying from other instrumentation/monitoring systems. Generated/dynamic metric names are a sign that you should be using labels instead.

Metric descriptions/help

Gauge/Counter/Summary/Histogram MUST require metric descriptions/help to be provided. Any custom Collectors provided with the client libraries MUST have descriptions/help on their metrics.

It is suggested to make it a mandatory argument, but not to check that it’s of a certain length as if someone really doesn’t want to write docs we’re not going to convince them otherwise. Collectors offered with the library (and indeed everywhere we can within the ecosystem) SHOULD have good metric descriptions, to lead by example.

Standard and runtime collectors

Client libraries SHOULD offer what they can of the Standard exports, documented at https://docs.google.com/document/d/1Q0MXWdwp1mdXCzNRak6bW5LLVylVRXhdi7_21Sg15xQ/edit

In addition, client libraries are ENCOURAGED to also offer whatever makes sense in terms of metrics for their language's runtime (e.g. Garbage collection stats).

These SHOULD be implemented as custom Collectors, and registered by default on the default CollectorRegistry. There SHOULD be a way to disable these, as there are some very niche use cases where they get in the way.

Exposition

Clients MUST implement one of the exposition formats documented at http://prometheus.io/docs/instrumenting/exposition_formats/.

Clients MAY implement more than one format. There SHOULD be a human readable format offered.

If in doubt, go for the text format. It doesn't have a dependency (protobuf), tends to be easy to produce, is human readable and the performance benefits of protobuf are not that significant for most use cases.

Reproducible order of the exposed metrics is ENCOURAGED (especially for human readable formats) if it can be implemented without a significant resource cost.

Unit tests

Client libraries SHOULD have unit tests covering the core instrumentation library and exposition.

Client libraries are ENCOURAGED to offer ways that make it easy for users to unit-test their use of the instrumentation code. For example, the CollectorRegistry.get_sample_value in Python.

Packaging and Dependencies

Ideally, a client library can be included in any application to add some instrumentation, without having to worry about it breaking the application.

Accordingly, caution is advised when adding dependencies to the client library. For example, if a user adds a library that uses a Prometheus client that requires version 1.4 of protobuf but the application uses 1.2 elsewhere, what will happen?

It is suggested that where this may arise, that the core instrumentation is separated from the bridges/exposition of metrics in a given format. For example, the Java simpleclient “simpleclient” module has no dependencies, and the “simpleclient_servlet” has the HTTP bits.

Performance Considerations

As client libraries must be thread-safe, some form of concurrency control is required and consideration must be given to performance on multi-core machines and applications.

In our experience the least performant is mutexes.
Processor atomic instructions tend to be in the middle, and generally acceptable.
Approaches that avoid different CPUs mutating the same bit of RAM work best, such as the DoubleAdder in Java’s simpleclient. There is a memory cost though.

As noted above, the result of labels() should be cacheable. The concurrent maps that tend to back metric with labels tend to be relatively slow. Special-casing metrics without labels to avoid labels()-like lookups can help a lot.

Metrics SHOULD avoid blocking when they are being incremented/decremented/set etc. as it’s undesirable for the whole application to be held up while a scrape is ongoing.

Having benchmarks of the main instrumentation operations, including labels, is ENCOURAGED.

Resource consumption, particularly RAM, should be kept in mind when performing exposition. Consider reducing the memory footprint by streaming results, and potentially having a limit on the number of concurrent scrapes.