Ops WG
Internet-Draft
Intended status: Standards Track
Expires: May 29, 2021

R. Hartmann, Ed.
Grafana Labs
B. Kochie
GitLab
B. Brazil
Robust Perception
R. Skillington
Chronosphere

November 25, 2020

OpenMetrics, a cloud-native, highly scalable metrics protocol
draft-richih-opsawg-openmetrics-00

Abstract

OpenMetrics specifies today's de-facto standard for transmitting cloud-native metrics at scale, with support for both text representation and Protocol Buffers and brings it into IETF. It supports both pull and push-based data collection.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Created in 2012, Prometheus has been the default for cloud-native observability since 2015. A central part of Prometheus' design is its text metric exposition format, called the Prometheus exposition format 0.0.4, stable since 2014. In this format, special care has been taken to make it easy to generate, to ingest, and to understand by humans. As of 2020, there are more than 700 publicly listed exporters, an unknown number of unlisted exporters, and thousands of native library integrations using this format. Dozens of ingestors from various projects and companies support consuming it.

With OpenMetrics, we are cleaning up and tightening the specification with the express purpose of bringing it into IETF. We are documenting a working standard with wide and organic adoption while introducing minimal, largely backwards-compatible, and well-considered changes. As of 2020, dozens of exporters, integrations, and ingestors use and preferentially negotiate OpenMetrics already.

Given the wide adoption and significant coordination requirements in the ecosystem, sweeping changes to either the Prometheus exposition format 0.0.4 or OpenMetrics 1.0 are considered out of scope.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. Overview

Metrics are a specific kind of telemetry data. They represent a snapshot of the current state for a set of data. They are distinct from logs or events, which focus on records or information about individual events.

OpenMetrics is primarily a wire format, independent of any particular transport for that format. The format is expected to be consumed on a regular basis and to be meaningful over successive expositions.

Implementers MUST expose metrics in the OpenMetrics text format in response to a simple HTTP GET request to a documented URL for a given process or device. This endpoint SHOULD be called "/metrics". Implementers MAY also expose OpenMetrics formatted metrics in other ways, such as by regularly pushing metric sets to an operator-configured endpoint over HTTP.

2.1. Metrics and Time Series

This standard expresses all system states as numerical values; counts, current values, enumerations, and boolean states being common examples. Contrary to metrics, singular events occur at a specific time. Metrics tend to aggregate data temporally. While this can lose information, the reduction in overhead is an engineering trade-off commonly chosen in many modern monitoring systems.

Time series are a record of changing information over time. While time series can support arbitrary strings or binary data, only numeric data is in scope for this RFC.

Common examples of metric time series would be network interface counters, device temperatures, BGP connection states, and alert states.

3. Data Model

This section MUST be read together with the ABNF section. In case of disagreements between the two, the ABNF's restrictions MUST take

precedence. This reduces repetition as the text wire format MUST be supported.

3.1. Data Types

3.1.1. Values

Metric values in OpenMetrics MUST be either floating points or integers. Note that ingestors of the format MAY only support float64. The non-real values NaN, +Inf and -Inf MUST be supported. NaN MUST NOT be considered a missing value, but it MAY be used to signal a division by zero.

3.1.1.1. Booleans

Boolean values MUST follow 1==true, 0==false.

3.1.2. Timestamps

Timestamps MUST be Unix Epoch in seconds. Negative timestamps MAY be used.

3.1.3. Strings

Strings MUST only consist of valid UTF-8 characters and MAY be zero length. NULL (ASCII 0x0) MUST be supported.

3.1.4. Label

Labels are key-value pairs consisting of strings.

Label names beginning with underscores are RESERVED and MUST NOT be used unless specified by this standard. Label names MUST follow the restrictions in the ABNF section.

Empty label values SHOULD be treated as if the label was not present.

3.1.5. LabelSet

A LabelSet MUST consist of Labels and MAY be empty. Label names MUST be unique within a LabelSet.

3.1.6. MetricPoint

Each MetricPoint consists of a set of values, depending on the MetricFamily type.

3.1.7. Exemplars

Exemplars are references to data outside of the MetricSet. A common use case are IDs of program traces.

Exemplars MUST consist of a LabelSet and a value, and MAY have a timestamp. They MAY each be different from the MetricPoints' LabelSet and timestamp.

The combined length of the label names and values of an Exemplar's LabelSet MUST NOT exceed 128 UTF-8 characters. Other characters in the text rendering of an exemplar such as ",= are not included in this limit for implementation simplicity and for consistency between the text and proto formats.

Ingestors MAY discard exemplars.

3.1.8. Metric

Metrics are defined by a unique LabelSet within a MetricFamily. Metrics MUST contain a list of one or more MetricPoints. Metrics with the same name for a given MetricFamily SHOULD have the same set of label names in their LabelSet.

MetricPoints SHOULD NOT have explicit timestamps.

If more than one MetricPoint is exposed for a Metric, then its MetricPoints MUST have monotonically increasing timestamps.

3.1.9. MetricFamily

A MetricFamily MAY have zero or more Metrics. A MetricFamily MUST have a name, HELP, TYPE, and UNIT metadata. Every Metric within a MetricFamily MUST have a unique LabelSet.

3.1.9.1. Name

MetricFamily names are a string and MUST be unique within a MetricSet. Names SHOULD be in snake_case. Metric names MUST follow the restrictions in the ABNF section.

Colons in MetricFamily names are RESERVED to signal that the MetricFamily is the result of a calculation or aggregation of a general purpose monitoring system.

MetricFamily names beginning with underscores are RESERVED and MUST NOT be used unless specified by this standard.

3.1.9.1.1. Suffixes

The name of a MetricFamily MUST NOT result in a potential clash for sample metric names as per the ABNF with another MetricFamily in the Text Format within a MetricSet. An example would be a gauge called "foo_created" as a counter called "foo" could create a "foo_created" in the text format.

Exposers SHOULD avoid names that could be confused with the suffixes that text format sample metric names use.

- Suffixes for the respective types are:
- Counter: '_total', '_created'
- Summary: '_count', '_sum', '_created', '' (empty)
- Histogram: '_count', '_sum', '_bucket', '_created'
- GaugeHistogram: '_gcount', '_gsum', '_bucket'
- Info: '_info'
- Gauge: '' (empty)
- StateSet: '' (empty)
- Unknown: '' (empty)

3.1.9.2. Type

Type specifies the MetricFamily type. Valid values are "unknown", "gauge", "counter", "stateset", "info", "histogram", "gaugehistogram", and "summary".

3.1.9.3. Unit

Unit specifies MetricFamily units. If non-empty, it MUST be a suffix of the MetricFamily name separated by an underscore. Be aware that further generation rules might make it an infix in the text format.

3.1.9.4. Help

Help is a string and SHOULD non-empty. It is used to give a brief description of the MetricFamily for human consumption and SHOULD be short enough to be used as a tooltip.

3.1.9.5. MetricSet

A MetricSet is the top level object exposed by OpenMetrics. It MUST consist of MetricFamilies and MAY be empty.

Each MetricFamily name MUST be unique. The same label name and value SHOULD NOT appear on every Metric within a MetricSet.

There is no specific ordering of MetricFamilies required within a MetricSet. An exposer MAY make an exposition easier to read for humans, for example sort alphabetically if the performance tradeoff makes sense.

If present, an Info MetricFamily called "target" per the "Supporting target metadata in both push-based and pull-based systems" section below SHOULD be first.

3.2. Metric Types

3.2.1. Gauge

Gauges are current measurements, such as bytes of memory currently used or the number of items in a queue. For gauges the absolute value is what is of interest to a user.

A MetricPoint in a Metric with the type gauge MUST have a single value.

Gauges MAY increase, decrease, or stay constant over time. Even if they only ever go in one direction, they might still be gauges and not counters. The size of a log file would usually only increase, a resource might decrease, and the limit of a queue size may be constant.

A gauge MAY be used to encode an enum where the enum has many states and changes over time, it is the most efficient but least user friendly.

3.2.2. Counter

Counters measure discrete events. Common examples are the number of HTTP requests received, CPU seconds spent, or bytes sent. For counters how quickly they are increasing over time is what is of interest to a user.

A MetricPoint in a Metric with the type Counter MUST have one value called Total. A Total is a non-NaN and MUST be monotonically non-decreasing over time, starting from 0.

A MetricPoint in a Metric with the type Counter SHOULD have a Timestamp value called Created. This can help ingestors discern between new metrics and long-running ones it did not see before.

A MetricPoint in a Metric's Counter's Total MAY reset to 0. If present, the corresponding Created time MUST also be set to the timestamp of the reset.

A MetricPoint in a Metric's Counter's Total MAY have an exemplar.

3.2.3. StateSet

StateSets represent a series of related boolean values, also called a bitset. If ENUMs need to be encoded this MAY be done via StateSet.

A point of a StateSet metric MAY contain multiple states and MUST contain one boolean per State. States have a name which are Strings.

A StateSet Metric's LabelSet MUST NOT have a label name which is the same as the name of its MetricFamily.

If encoded as a StateSet, ENUMs MUST have exactly one Boolean which is true within a MetricPoint.

This is suitable where the enum value changes over time, and the number of States isn't much more than a handful.

EDITOR'S NOTE: This might be better as Consideration

MetricFamilies of type StateSets MUST have an empty Unit string.

3.2.4. Info

Info metrics are used to expose textual information which SHOULD NOT change during process lifetime. Common examples are an application's version, revision control commit, and the version of a compiler.

A MetricPoint of an Info Metric contains a LabelSet. An Info MetricPoint's LabelSet MUST NOT have a label name which is the same as the name of a label of the LabelSet of its Metric.

Info MAY be used to encode ENUMs whose values do not change over time, such as the type of a network interface.

MetricFamilies of type Info MUST have an empty Unit string.

3.2.5. Histogram

Histograms measure distributions of discrete events. Common examples are the latency of HTTP requests, function runtimes, or I/O request sizes.

A Histogram MetricPoint MUST contain at least one bucket, and SHOULD contain Sum, and Created values. Every bucket MUST have a threshold and a value.

Histogram MetricPoints MUST have at least a bucket with an +Inf threshold. Buckets MUST be cumulative. As an example for a metric representing request latency in seconds its values for buckets with thresholds 1, 2, 3, and +Inf MUST follow value_1 <= value_2 <= value_3 <= value_+Inf. If ten requests took 1 second each, the values of the 1, 2, 3, and +Inf buckets MUST equal 10.

The +Inf bucket counts all requests. If present, the Sum value MUST equal the Sum of all the measured event values. Bucket thresholds within a MetricPoint MUST be unique.

Semantically, Sum, and buckets values are counters so MUST NOT be NaN or negative. Negative threshold buckets MAY be used, but then the Histogram MetricPoint MUST NOT contain a sum value as it would no longer be a counter semantically. Bucket thresholds MUST NOT equal NaN. Count and bucket values MUST be integers.

A Histogram MetricPoint SHOULD have a Timestamp value called Created. This can help ingestors discern between new metrics and long-running ones it did not see before.

A Histogram's Metric's LabelSet MUST NOT have a "le" label name.

Bucket values MAY have exemplars. Buckets are cumulative to allow monitoring systems to drop any non-+Inf bucket for performance/anti-denial-of-service reasons in a way that loses granularity but is still a valid Histogram.

EDITOR'S NOTE: The second sentence is a consideration, it can be moved if needed

Each bucket covers the values less and or equal to it, and the value of the exemplar MUST be within this range. Exemplars SHOULD be put into the bucket with the highest value. A bucket MUST NOT have more than one exemplar.

3.2.6. GaugeHistogram

GaugeHistograms measure current distributions. Common examples are how long items have been waiting in a queue, or size of the requests in a queue.

A GaugeHistogram MetricPoint MUST have at least one bucket with an +Inf threshold, and SHOULD contain a Gsum value. Every bucket MUST have a threshold and a value.

The buckets for a GaugeHistogram follow all the same rules as for a Histogram.

The bucket and Gsum of a GaugeHistogram are conceptually gauges, however bucket values MUST NOT be negative or NaN. If negative threshold buckets are present, then sum MAY be negative. Gsum MUST NOT be NaN. Bucket values MUST be integers.

A GaugeHistogram's Metric's LabelSet MUST NOT have a "le" label name.

Bucket values can have exemplars.

Each bucket covers the values less and or equal to it, and the value of the exemplar MUST be within this range. Exemplars SHOULD be put into the bucket with the highest value. A bucket MUST NOT have more than one exemplar.

3.2.7. Summary

Summaries also measure distributions of discrete events and MAY be used when Histograms are too expensive and/or an average event size is sufficient.

They MAY also be used for backwards compatibility, because some existing instrumentation libraries expose precomputed quantiles and do not support Histograms. Precomputed quantiles SHOULD NOT be used, because quantiles are not aggregatable and the user often can not deduce what timeframe they cover.

A Summary MetricPoint MAY consist of a Count, Sum, Created, and a set of quantiles.

Semantically, Count and Sum values are counters so MUST NOT be NaN or negative. Count MUST be an integer.

A MetricPoint in a Metric with the type Summary which contains Count or Sum values SHOULD have a Timestamp value called Created. This can help ingestors discern between new metrics and long-running ones it

did not see before. Created MUST NOT relate to the collection period of quantile values.

Quantiles are a map from a quantile to a value. An example is a quantile 0.95 with value 0.2 in a metric called myapp_http_request_duration_seconds which means that the 95th percentile latency is 200ms over an unknown timeframe. If there are no events in the relevant timeframe, the value for a quantile MUST be NaN. A Quantile's Metric's LabelSet MUST NOT have "quantile" label name. Quantiles MUST be between 0 and 1 inclusive. Quantile values MUST NOT be negative. Quantile values SHOULD represent the recent values. Commonly this would be over the last 5-10 minutes.

3.2.8. Unknown

Unknown SHOULD NOT be used. Unknown MAY be used when it is impossible to determine the types of individual metrics from 3rd party systems.

A point in a metric with the unknown type MUST have a single value.

4. Data transmission & wire formats

The text wire format MUST be supported and is the default. The protobuf wire format MAY be supported and MUST ONLY be used after negotiation.

The OpenMetrics formats are Regular Chomsky Grammars, making writing quick and small parsers possible. The text format compresses well, and protobuf is already binary and efficiently encoded.

Partial or invalid expositions MUST be considered erroneous in their entirety.

4.1. Protocol Negotiation

All ingestor implementations MUST be able to ingest data secured with TLS 1.2 or later. All exposers SHOULD be able to emit data secured with TLS 1.2 or later. ingestor implementations SHOULD be able to ingest data from HTTP without TLS. All implementations SHOULD use TLS to transmit data.

Negotiation of what version of the OpenMetrics format to use is out-of-band. For example for pull-based exposition over HTTP standard HTTP content type negotiation is used, and MUST default to the oldest version of the standard (i.e. 1.0.0) if no newer version is requested.

Push-based negotiation is inherently more complex, as the exposer typically initiates the connection. Producers MUST use the oldest version of the standard (i.e. 1.0.0) unless requested otherwise by the ingestor. Text format

4.1.1. ABNF

ABNF as per RFC 5234

EDITOR'S NOTE: Should we update to RFC 7405, in particular the case insensitive bits?

"exposition" is the top level token of the ABNF.

```
exposition = metricset HASH SP eof [ LF ]

metricset = *metricfamily

metricfamily = *metric-descriptor *metric

metric-descriptor = HASH SP type SP metricname SP metric-type LF
metric-descriptor =/ HASH SP help SP metricname SP escaped-string LF
metric-descriptor =/ HASH SP unit SP metricname SP 1*metricname-char LF

metric = *sample

metric-type = counter / gauge / histogram / gaugehistogram / stateset
metric-type =/ info / summary / unknown

sample = metricname [labels] SP number [SP timestamp] [exemplar] LF

exemplar = SP HASH SP labels SP number [SP timestamp]

labels = "{" [label *(COMMA label)] "}"

label = label-name EQ DQUOTE escaped-string DQUOTE

number = realnumber
; Case insensitive
number =/ [SIGN] ("inf" / "infinity")
number =/ "nan"

timestamp = realnumber

; Not 100% sure this captures all float corner cases.
; Leading 0s explicitly okay
realnumber = [SIGN] 1*DIGIT
realnumber =/ [SIGN] 1*DIGIT ["." *DIGIT] [ "e" [SIGN] 1*DIGIT ]
```

```
 realnumber =/ [SIGN] *DIGIT "." 1*DIGIT [ "e" [SIGN] 1*DIGIT ]


 ; RFC 5234 is case insensitive.
 ; Uppercase
 eof = %d69.79.70
 type = %d84.89.80.69
 help = %d72.69.76.80
 unit = %d85.78.73.84
 ; Lowercase
 counter = %d99.111.117.110.116.101.114
 gauge = %d103.97.117.103.101
 histogram = %d104.105.115.116.111.103.114.97.109
 gaugehistogram = gauge histogram
 stateset = %d115.116.97.116.101.115.101.116
 info = %d105.110.102.111
 summary = %d115.117.109.109.97.114.121
 unknown = %d117.110.107.110.111.119.110

 BS = "\"
 EQ = "="
 COMMA = ","
 HASH = "#"
 SIGN = "-" / "+"

 metricname = metricname-initial-char 0*metricname-char

 metricname-char = metricname-initial-char / DIGIT
 metricname-initial-char = ALPHA / "_" / ":"

 label-name = label-name-initial-char *label-name-char

 label-name-char = label-name-initial-char / DIGIT
 label-name-initial-char = ALPHA / "_"

 escaped-string = *escaped-char

 escaped-char = normal-char
 escaped-char =/ BS ("n" / DQUOTE / BS)

 ; Any unicode character, except newline, double quote, and backslash
 normal-char = %x00-09 / %x0B-21 / %x23-5B / %x5D-D7FF / %xE000-10FFFF
```

4.1.2. Overall Structure

UTF-8 MUST be used. Byte order markers (BOMs) MUST NOT be used. As an important reminder for implementers, byte 0 is valid UTF-8 while, for example, byte 255 is not.

The content type MUST be: application/openmetrics-text; version=1.0.0; charset=utf-8

Line endings MUST be signalled with line feed (\n) and MUST NOT contain carriage returns (\r). Expositions MUST end with EOF and SHOULD end with 'EOF\n'.

An example of a complete exposition: ~~~~ # TYPE acme_http_router_request_seconds summary # UNIT acme_http_router_request_seconds seconds # HELP acme_http_router_request_seconds Latency though all of ACME's HTTP request router. acme_http_router_request_seconds_sum{path="/api/v1",method="GET"} 9036.32 acme_http_router_request_seconds_count{path="/api/v1",method="GET"} 807283.0 acme_http_router_request_seconds_created{path="/api/v1",method="GET"} 1605281325.0 acme_http_router_request_seconds_sum{path="/api/v2",method="POST"} 479.3 acme_http_router_request_seconds_count{path="/api/v2",method="POST"} 34.0 acme_http_router_request_seconds_created{path="/api/v2",method="POST"} 1605281325.0 # TYPE go_goroutines gauge # HELP go_goroutines Number of goroutines that currently exist. go_goroutines 69 # TYPE process_cpu_seconds counter # UNIT process_cpu_seconds seconds # HELP process_cpu_seconds Total user and system CPU time spent in seconds. process_cpu_seconds_total 4.20072246e+06 # EOF ~~~~

4.1.2.1. Escaping

Where the ABNF notes escaping, the following escaping MUST be applied Line feed, '\n' (0x0A) -> literally '\n' (Bytecode 0x5c 0x6e) Double quotes -> '"' (Bytecode 0x5c 0x22) Backslash -> '\' (Bytecode 0x5c 0x5c)

4.1.2.2. Numbers

Integer numbers MUST NOT have a decimal point. Examples are "23", "0042", and "1341298465647914".

Floating point numbers MUST be represented either with a decimal point or using scientific notation. Examples are "8903.123421" and "1.89e-7". Floating point numbers MUST fit within the range of a 64-bit floating point value as defined by IEEE 754, but MAY require so many bits in the mantissa that results in lost precision. This MAY be used to encode nanosecond resolution timestamps.

Arbitrary integer and floating point rendering of numbers MUST NOT be used for "quantile" and "le" label values as in section "Canonical Numbers". They MAY be used anywhere else numbers are used.

4.1.2.2.1. Considerations: Canonical Numbers

Numbers in the "le" label values of histograms and "quantile" label values of summary metrics are special in that they're label values, and label values are intended to be opaque. As end users will likely directly interact with these string values, and as many monitoring systems lack the ability to deal with them as first-class numbers, it would be beneficial if a given number had the exact same text representation.

Consistency is highly desirable, but real world implementations of languages and their runtimes make mandating this impractical. The most important common quantiles are 0.5, 0.95, 0.9, 0.99, 0.999 and bucket values representing values from a millisecond up to 10.0 seconds, because those cover cases like latency SLAs and Apdex for typical web services. Powers of ten are covered to try to ensure that the switch between fixed point and exponential rendering is consistent as this varies across runtimes. The target rendering is equivalent to the default Go rendering of float64 values (i.e. %g), with a .0 appended in case there is no decimal point or exponent to make clear that they are floats.

Exposers MUST produce output for positive infinity as +Inf.

Exposers SHOULD produce output for the values 0.0 up to 10.0 in 0.001 increments in line with the following examples: 0.0 0.001 0.002 0.01 0.1 0.9 0.95 0.99 0.999 1.0 1.7 10.0

Exposers SHOULD produce output for the values 1e-10 up to 1e+10 in powers of ten in line with the following examples: 1e-10 1e-09 1e-05 0.0001 0.1 1.0 100000.0 1e+06 1e+10

Parsers MUST NOT reject inputs which are outside of the canonical values merely because they are not consistent with the canonical values. For example 1.1e-4 must not be rejected, even though it is not the consistent rendering of 0.00011.

Exposers SHOULD follow these patterns for non-canonical numbers, and the intention is by adjusting the rendering algorithm to be consistent for these values that the vast majority of other values will also have consistent rendering. Exposers using only a few particular le/quantile values could also hardcode. In languages such as C where a minimal floating point rendering algorithm such as

Grisu3 such as Grisu3 is not readily available, exposers MAY use a different rendering.

A warning to implementers in C and other languages that share its printf implementation: The standard precision of %f, %e and %g is only six significant digits. 17 significant digits are required for full precision, e.g. "printf("%.17g", d)".

4.1.2.3. Timestamps

Timestamps SHOULD NOT use exponential float rendering for timestamps if nanosecond precision is needed as rendering of a float64 does not have sufficient precision, e.g. 1604676851.123456789.

4.1.3. MetricFamily

There MUST NOT be an explicit separator between MetricFamilies. The next MetricFamily MUST be signalled with either metadata or a new sample metric name which cannot be part of the previous MetricFamily.

MetricFamilies MUST NOT be interleaved.

4.1.3.1. MetricFamily metadata

There are four pieces of metadata: The MetricFamily name, TYPE, UNIT and HELP. An example of the metadata for a counter Metric called foo is:

```
# TYPE foo counter
```

If no TYPE is exposed, the MetricFamily MUST be of type Unknown.

If a unit is specified it MUST be provided in a UNIT metadata line. In addition, an underscore and the unit MUST be the suffix of the MetricFamily name.

A valid example for a foo_seconds metric with a unit of "seconds": ~~~~ # TYPE foo_seconds counter # UNIT foo_seconds seconds ~~~~

An invalid example, where the unit is not a suffix on the name: ~~~~ # TYPE foo counter # UNIT foo seconds ~~~~

It is also valid to have: ~~~~ # TYPE foo_seconds counter ~~~~

If the unit is known it SHOULD be provided.

The value of a UNIT or HELP line MAY be empty. This MUST be treated as if no metadata line for the MetricFamily existed.

```
# TYPE foo_seconds counter
# UNIT foo_seconds seconds
# HELP foo_seconds Some text and \n some \" escaping
```

There MUST NOT be more than one of each type of metadata line for a MetricFamily. The ordering SHOULD be TYPE, UNIT, HELP.

Aside from this metadata and the EOF line at the end of the message, you MUST NOT expose lines beginning with a #.

4.1.3.2. Metric

Metrics MUST NOT be interleaved.

See the example in "Text format -> MetricPoint". Labels A sample without labels or a timestamp and the value 0 MUST be rendered either like:

```
bar_seconds_count 0
```

or like

```
bar_seconds_count{} 0
```

Label values MAY be any valid UTF-8 value, so escaping MUST be applied as per the ABNF. A valid example with two labels: ~~~~ bar_seconds_count{a="x",b="escaping" example \n "} 0 ~~~~

The rendering of values for a MetricPoint can include additional labels (e.g. the "le" label for a Histogram type), which MUST be rendered in the same way as a Metric's own LabelSet.

4.1.4. MetricPoint

MetricPoints MUST NOT be interleaved.

A correct example where there were multiple MetricPoints and Samples within a MetricFamily would be:

```
# TYPE foo_seconds summary
# UNIT foo_seconds seconds
foo_seconds_count{a="bb"} 0 123
foo_seconds_sum{a="bb"} 0 123
foo_seconds_count{a="bb"} 0 456
foo_seconds_sum{a="bb"} 0 456
foo_seconds_count{a="ccc"} 0 123
foo_seconds_sum{a="ccc"} 0 123
foo_seconds_count{a="ccc"} 0 456
foo_seconds_sum{a="ccc"} 0 456
```

An incorrect example where Metrics are interleaved:

```
# TYPE foo_seconds summary
# UNIT foo_seconds seconds
foo_seconds_count{a="bb"} 0 123
foo_seconds_count{a="ccc"} 0 123
foo_seconds_count{a="bb"} 0 456
foo_seconds_count{a="ccc"} 0 456
```

An incorrect example where MetricPoints are interleaved:

```
# TYPE foo_seconds summary
# UNIT foo_seconds seconds
foo_seconds_count{a="bb"} 0 123
foo_seconds_count{a="bb"} 0 456
foo_seconds_sum{a="bb"} 0 123
foo_seconds_sum{a="bb"} 0 456
```

4.1.5. **Metric types**

4.1.5.1. **Gauge**

The Sample MetricName for the value of a MetricPoint for a MetricFamily of type Gauge MUST NOT have a suffix.

An example MetricFamily with a Metric with no labels and a MetricPoint with no timestamp: ~~~~ # TYPE foo gauge foo 17.0 ~~~~

An example of a MetricFamily with two Metrics with a label and MetricPoints with no timestamp: ~~~~ # TYPE foo gauge foo{a="bb"} 17.0 foo{a="ccc"} 17.0 ~~~~

An example of a MetricFamily with no Metrics: ~~~~ # TYPE foo gauge ~~~~

An example with a Metric with a label and a MetricPoint with a timestamp: ~~~~ # TYPE foo gauge foo{a="b"} 17.0 1520879607.789 ~~~~

An example with a Metric with no labels and MetricPoint with a timestamp: ~~~~ # TYPE foo gauge foo 17.0 1520879607.789 ~~~~

An example with a Metric with no labels and two MetricPoints with timestamps: ~~~~ # TYPE foo gauge foo 17.0 123 foo 18.0 456 ~~~~

4.1.5.2. Counter

The MetricPoint's Total Value Sample MetricName MUST have the suffix "_total". If present the MetricPoint's Created Value Sample MetricName MUST have the suffix "_created".

An example with a Metric with no labels, and a MetricPoint with no timestamp and no created: ~~~~ # TYPE foo counter foo_total 17.0 ~~~~

An example with a Metric with no labels, and a MetricPoint with a timestamp and no created: ~~~~ # TYPE foo counter foo_total 17.0 1520879607.789 ~~~~

An example with a Metric with no labels, and a MetricPoint with no timestamp and a created: ~~~~ # TYPE foo counter foo_total 17.0 foo_created 1520430000.123 ~~~~

An example with a Metric with no labels, and a MetricPoint with a timestamp and a created: ~~~~ # TYPE foo counter foo_total 17.0 1520879607.789 foo_created 1520430000.123 1520879607.789 ~~~~

Exemplars MAY be attached to the MetricPoint's Total sample.

4.1.5.3. StateSet

The Sample MetricName for the value of a MetricPoint for a MetricFamily of type StateSet MUST NOT have a suffix.

StateSets MUST have one sample per State in the MetricPoint. Each State's sample MUST have a label with the MetricFamily name as the label name and the State name as the label value. The State sample's value MUST be 1 if the State is true and MUST be 0 if the State is false.

An example with the states "a", "bb", and "ccc" in which only the value b is enabled and the metric name is foo:

```
# TYPE foo stateset
foo{foo="a"} 0
foo{foo="bb"} 1
foo{foo="ccc"} 0
```

An example of an "entity" label on the Metric: ~~~~ # TYPE foo stateset foo{entity="controller",foo="a"} 1.0 foo{entity="controller",foo="bb"} 0.0 foo{entity="controller",foo="ccc"} 0.0 foo{entity="replica",foo="a"} 1.0 foo{entity="replica",foo="bb"} 0.0 foo{entity="replica",foo="ccc"} 1.0 ~~~~

4.1.5.4. Info

The Sample MetricName for the value of a MetricPoint for a MetricFamily of type Info MUST have the suffix "_info". The Sample value MUST always be 1.

An example of a Metric with no labels, and one MetricPoint value with "name" and "version" labels: ~~~~ # TYPE foo info foo_info{name="pretty name",version="8.2.7"} 1 ~~~~

An example of a Metric with label "entity" and one MetricPoint value with "name" and "version" labels: ~~~~ # TYPE foo info foo_info{entity="controller",name="pretty name",version="8.2.7"} 1.0 foo_info{entity="replica",name="prettier name",version="8.1.9"} 1.0 ~~~~

Metric labels and MetricPoint value labels MAY be in any order.

4.1.5.5. Summary

If present, the MetricPoint's Sum Value Sample MetricName MUST have the suffix "_sum". If present, the MetricPoint's Count Value Sample MetricName MUST have the suffix "_count". If present, the MetricPoint's Created Value Sample MetricName MUST have the suffix "_created". If present, the MetricPoint's Quantile Values MUST specify the quantile measured using a label with a label name of "quantile" and with a label value of the quantile measured.

An example of a Metric with no labels and a MetricPoint with Sum, Count and Created values: ~~~~ # TYPE foo summary foo_count 17.0 foo_sum 324789.3 foo_created 1520430000.123 ~~~~

An example of a Metric with no labels and a MetricPoint with two quantiles: ~~~~ # TYPE foo summary foo{quantile="0.95"} 123.7 foo{quantile="0.99"} 150.0 ~~~~

Quantiles MAY be in any order.

4.1.5.6. Histogram

The MetricPoint's Bucket Values Sample MetricNames MUST have the suffix "_bucket". If present, the MetricPoint's Sum Value Sample MetricName MUST have the suffix "_sum". If present, the MetricPoint's Created Value Sample MetricName MUST have the suffix "_created". If and only if a Sum Value is present in a MetricPoint, then the MetricPoint's +Inf Bucket value MUST also appear in a Sample with a MetricName with the suffix "_count".

Buckets MUST be sorted in number increasing order of "le", and the value of the "le" label MUST follow the rules for Canonical Numbers.

An example of a Metric with no labels and a MetricPoint with Sum, Count, and Created values, and with 12 buckets. A wide and atypical but valid variety of "le" values is shown on purpose: ~~~~ # TYPE foo histogram foo_bucket{le="0.0"} 0 foo_bucket{le="1e-05"} 0 foo_bucket{le="0.0001"} 5 foo_bucket{le="0.1"} 8 foo_bucket{le="1.0"} 10 foo_bucket{le="10.0"} 11 foo_bucket{le="100000.0"} 11 foo_bucket{le="1e+06"} 15 foo_bucket{le="1e+23"} 16 foo_bucket{le="1.1e+23"} 17 foo_bucket{le="+Inf"} 17 foo_count 17 foo_sum 324789.3 foo_created 1520430000.123 ~~~~

4.1.5.6.1. Exemplars

Exemplars without Labels MUST represent an empty LabelSet as {}.

An example of Exemplars showcasing several valid cases: The "0.01" bucket has no Exemplar. The 0.1 bucket has an Exemplar with no Labels. The 1 bucket has an Exemplar with one Label. The 10 bucket has an Exemplar with a Label and a timestamp. In practice all buckets SHOULD have the same style of Exemplars. ~~~~ # TYPE foo histogram foo_bucket{le="0.01"} 0 foo_bucket{le="0.1"} 8 # {} 0.054 foo_bucket{le="1"} 11 # {trace_id="KOO5S4vxi0o"} 0.67 foo_bucket{le="10"} 17 # {trace_id="oHg5SJYRHA0"} 9.8 1520879607.789 foo_bucket{le="+Inf"} 17 foo_count 17 foo_sum 324789.3 foo_created 1520430000.123 ~~~~

4.1.5.7. GaugeHistogram

The MetricPoint's Bucket Values Sample MetricNames MUST have the suffix "_bucket". If present, the MetricPoint's Sum Value Sample MetricName MUST have the suffix "_gsum". If present, the MetricPoint's Created Value Sample MetricName MUST have the suffix "_created". If and only if a Sum Value is present in a MetricPoint, then the MetricPoint's +Inf Bucket value MUST also appear in a Sample with a MetricName with the suffix "_gcount".

Buckets MUST be sorted in number increasing order of "le", and the value of the "le" label MUST follow the rules for Canonical Numbers.

An example of a Metric with no labels, and one MetricPoint value with no Exemplar with no Exemplars in the buckets: ~~~~ # TYPE foo gaugehistogram foo_bucket{le="0.01"} 20.0 foo_bucket{le="0.1"} 25.0 foo_bucket{le="1"} 34.0 foo_bucket{le="10"} 34.0 foo_bucket{le="+Inf"} 42.0 foo_gcount 42.0 foo_gsum 3289.3 foo_created 1520430000.123 ~~~~

4.1.5.8. Unknown

The sample metric name for the value of the MetricPoint for a MetricFamily of type Unknown MUST NOT have a suffix.

An example with a Metric with no labels and a MetricPoint with no timestamp: ~~~~ # TYPE foo unknown foo 42.23 ~~~~

4.2. Protobuf format

4.2.1. Overall Structure

Protobuf messages MUST be encoded in binary and MUST have "application/openmetrics-protobuf; version=1.0.0" as their content type.

All payloads MUST be a single binary encoded MetricSet message, as defined by the OpenMetrics protobuf schema.

4.2.1.1. Version

The protobuf format MUST follow the proto3 version of the protocol buffer language.

4.2.1.2. Strings

All string fields MUST be UTF-8 encoded.

4.2.1.3. Timestamps

Timestamp representations in the OpenMetrics protobuf schema MUST follow the published google.protobuf.Timestamp [timestamp] message. The timestamp message MUST be in Unix epoch seconds as an int64 and a non-negative fraction of a second at nanosecond resolution as an int32 that counts forward from the seconds timestamp component. It MUST be within 0 to 999,999,999 inclusive.

4.2.2. Protobuf schema

```
syntax = "proto3";

// The OpenMetrics protobuf schema which defines the protobuf wire
// format.
// Ensure to interpret "required" as semantically required for a valid
// message.
// All string fields MUST be UTF-8 encoded strings.
package openmetrics;

import "google/protobuf/timestamp.proto";

// The top-level container type that is encoded and sent over the wire.
message MetricSet {
  // Each MetricFamily has one or more MetricPoints for a single Metric.
  repeated MetricFamily metric_families = 1;
}

// One or more Metrics for a single MetricFamily, where each Metric
// has one or more MetricPoints.
message MetricFamily {
  // Required.
  string name = 1;

  // Optional.
  MetricType type = 2;

  // Optional.
  string unit = 3;

  // Optional.
  string help = 4;

  // Optional.
  repeated Metric metrics = 5;
}

// The type of a Metric.
enum MetricType {
  // Unknown must use unknown MetricPoint values.
  UNKNOWN = 0;
  // Gauge must use gauge MetricPoint values.
  GAUGE = 1;
  // Counter must use counter MetricPoint values.
  COUNTER = 2;
  // State set must use state set MetricPoint values.
  STATE_SET = 3;
```

```
  // Info must use info MetricPoint values.
  INFO = 4;
  // Histogram must use histogram value MetricPoint values.
  HISTOGRAM = 5;
  // Gauge histogram must use histogram value MetricPoint values.
  GAUGE_HISTOGRAM = 6;
  // Summary quantiles must use summary value MetricPoint values.
  SUMMARY = 7;
}

// A single metric with a unique set of labels within a metric family.
message Metric {
  // Optional.
  repeated Label labels = 1;

  // Optional.
  repeated MetricPoint metric_points = 2;
}

// A name-value pair. These are used in multiple places: identifying
// timeseries, value of INFO metrics, and exemplars in Histograms.
message Label {
  // Required.
  string name = 1;

  // Required.
  string value = 2;
}

// A MetricPoint in a Metric.
message MetricPoint {
  // Required.
  oneof value {
    UnknownValue unknown_value = 1;
    GaugeValue gauge_value = 2;
    CounterValue counter_value = 3;
    HistogramValue histogram_value = 4;
    StateSetValue state_set_value = 5;
    InfoValue info_value = 6;
    SummaryValue summary_value = 7;
  }

  // Optional.
  google.protobuf.Timestamp timestamp = 8;
}

// Value for UNKNOWN MetricPoint.
message UnknownValue {
```

```
  // Required.
  oneof value {
    double double_value = 1;
    int64 int_value = 2;
  }
}

// Value for GAUGE MetricPoint.
message GaugeValue {
  // Required.
  oneof value {
    double double_value = 1;
    int64 int_value = 2;
  }
}

// Value for COUNTER MetricPoint.
message CounterValue {
  // Required.
  oneof total {
    double double_value = 1;
    uint64 int_value = 2;
  }

  // The time values began being collected for this counter.
  // Optional.
  google.protobuf.Timestamp created = 3;

  // Optional.
  Exemplar exemplar = 4;
}

// Value for HISTOGRAM or GAUGE_HISTOGRAM MetricPoint.
message HistogramValue {
  // Optional.
  oneof sum {
    double double_value = 1;
    int64 int_value = 2;
  }

  // Optional.
  uint64 count = 3;

  // The time values began being collected for this histogram.
  // Optional.
  google.protobuf.Timestamp created = 4;

  // Optional.
```

```
  repeated Bucket buckets = 5;

  // Bucket is the number of values for a bucket in the histogram
  // with an optional exemplar.
  message Bucket {
    // Required.
    uint64 count = 1;

    // Optional.
    double upper_bound = 2;

    // Optional.
    Exemplar exemplar = 3;
  }
}

message Exemplar {
  // Required.
  double value = 1;

  // Optional.
  google.protobuf.Timestamp timestamp = 2;

  // Labels are additional information about the exemplar value
  // (e.g. trace id).
  // Optional.
  repeated Label label = 3;
}

// Value for STATE_SET MetricPoint.
message StateSetValue {
  // Optional.
  repeated State states = 1;

  message State {
    // Required.
    bool enabled = 1;

    // Required.
    string name = 2;
  }
}

// Value for INFO MetricPoint.
message InfoValue {
  // Optional.
  repeated Label info = 1;
}
```

```
// Value for SUMMARY MetricPoint.
message SummaryValue {
  // Optional.
  oneof sum {
    double double_value = 1;
    int64 int_value = 2;
  }

  // Optional.
  uint64 count = 2;

  // The time sum and count values began being collected for this
  // summary.
  // Optional.
  google.protobuf.Timestamp created = 3;

  // Optional.
  repeated Quantile quantile = 4;

  message Quantile {
    // Required.
    double quantile = 1;

    // Required.
    double value = 2;
  }
}
```

5. Design Considerations

5.1. Scope

OpenMetrics is intended to provide telemetry for online systems. It runs over protocols which do not provide hard or soft real time guarantees, so it can not make any real time guarantees itself. Latency and jitter properties of OpenMetrics are as imprecise as the underlying network, operating systems, CPUs, and the like. It is sufficiently accurate for aggregations to be used as a basis for decision-making, but not to reflect individual events.

Systems of all sizes should be supported, from applications that receive a few requests an hour up to monitoring bandwidth usage on a 400Gb network port. Aggregation and analysis of transmitted telemetry should be possible over arbitrary time periods.

It is intended to transport snapshots of state at the time of data transmission at a regular cadence.

5.1.1. Out of scope

How ingestors discover which exposers exist, and vice-versa, is out of scope for and thus not defined in this standard.

5.2. Extensions and Improvements

This first version of OpenMetrics is based upon well established and de facto standard Prometheus text format 0.0.4, deliberately without adding major syntactic or semantic extensions, or optimisations on top of it. For example no attempt has been made to make the text representation of Histogram buckets more compact, relying on compression in the underlying stack to deal with their repetitive nature.

This is a deliberate choice, so that the standard can take advantage of the adoption and momentum of the existing user base. This ensures a relatively easy transition from the Prometheus text format 0.0.4.

It also ensures that there is a basic standard which is easy to implement. This can be built upon in future versions of the standard. The intention is that future versions of the standard will always require support for this 1.0 version, both syntactically and semantically.

We want to allow monitoring systems to get usable information from an OpenMetrics exposition without undue burden. If one were to strip away all metadata and structure and just look at an OpenMetrics exposition as an unordered set of samples that should be usable on its own. As such, there are also no opaque binary types, such as sketches or t-digests which could not be expressed as a mix of gauges and counters as they would require custom parsing and handling.

This principle is applied consistently throughout the standard. For example a MetricFamily's unit is duplicated in the name so that the unit is available for systems that don't understand the unit metadata. The "le" label is a normal label value, rather than getting its own special syntax, so that ingestors don't have to add special histogram handling code to ingest them. As a further example, there are no composite data types. For example, there is no geolocation type for latitude/longitude as this can be done with separate gauge metrics.

5.3. Units and Base Units

For consistency across systems and to avoid confusion, units are largely based on SI base units. Base units include seconds, bytes,

joules, grams, meters, ratios, volts, amperes, and celsius. Units should be provided where they are applicable.

For example, having all duration metrics in seconds, there is no risk of having to guess whether a given metric is nanoseconds, microseconds, milliseconds, seconds, minutes, hours, days or weeks nor having to deal with mixed units. By choosing unprefixed units, we avoid situations like ones in which kilomilliseconds were the result of emergent behaviour of complex systems.

As values can be floating point, sub-base-unit precision is built into the standard.

Similarly, mixing bits and bytes is confusing, so bytes are chosen as the base. While Kelvin is a better base unit in theory, in practice most existing hardware exposes Celsius. Kilograms are the SI base unit, however the kilo prefix is problematic so grams are chosen as the base unit.

While base units SHOULD be used in all possible cases, Kelvin is a well-established unit which MAY be used instead of Celsius for use cases such as color or black body temperatures where a comparison between a Celsius and Kelvin metric are unlikely.

Ratios are the base unit, not percentages. Where possible, raw data in the form of gauges or counters for the given numerator and denominator should be exposed. This has better mathematical properties for analysis and aggregation in the ingestors.

Decibels are not a base unit as firstly, deci is a SI prefix and secondly, bels are logarithmic. To expose signal/energy/power ratios exposing the ratio directly would be better, or better still the raw power/energy if possible. Floating point exponents are more than sufficient to cover even extreme scientific uses. An electron volt (~1e-19 J) all the way up to the energy emitted by a supernova (~1e44 J) is 63 orders of magnitude, and a 64-bit floating point number can cover over 2000 orders of magnitude.

If non-base units can not be avoided and conversion is not feasible, the actual unit should still be included in the metric name for clarity. For example, joule is the base unit for both energy and power, as watts can be expressed as a counter with a joule unit. In practice a given 3rd party system may only expose watts, so a gauge expressed in watts would be the only realistic choice in that case.

Not all MetricFamilies have units. For example a count of HTTP requests wouldn't have a unit. Technically the unit would be HTTP requests, but in that sense the entire MetricFamily name is the unit.

Going to that extreme would not be useful. The possibility of having good axes on graphs in downstream systems for human consumption should always be kept in mind.

5.4. Statelessness

The wire format defined by OpenMetrics is stateless across expositions. What information has been exposed before MUST have no impact on future expositions. Each exposition is a self-contained snapshot of the current state of the exposer.

The same self-contained exposition MUST be provided to existing and new ingestors.

A core design choice is that exposers MUST NOT exclude a metric merely because it has had no recent changes, or observations. An exposer must not make any assumptions about how often ingestors are consuming expositions.

5.5. Exposition Across Time and Metric Evolution

Metrics are most useful when their evolution over time can be analysed, so accordingly expositions must make sense over time. Thus, it is not sufficient for one single exposition on its own to be useful and valid. Some changes to metric semantics can also break downstream users.

Parsers commonly optimize by caching previous results. Thus, changing the order in which labels are exposed across expositions SHOULD be avoided even though it is technically not breaking This also tends to make writing unit tests for exposition easier.

Metrics and samples SHOULD NOT appear and disappear from exposition to exposition, for example a counter is only useful if it has history. In principle, a given Metric should be present in exposition from when the process starts until the process terminates. It is often not possible to know in advance what Metrics a MetricFamily will have over the lifetime of a given process (e.g. a label value of a latency histogram is a HTTP path, which is provided by an end user at runtime), but once a counter-like Metric is exposed it should continue to be exposed until the process terminates. That a counter is not getting increments doesn't invalidate that it still has its current value. There are cases where it may make sense to stop exposing a given Metric; see the section on Missing Data.

In general changing a MetricFamily's type, or adding or removing a label from its Metrics will be breaking to ingestors.

A notable exception is that adding a label to the value of an Info MetricPoints is not breaking. This is so that you can add additional information to an existing Info MetricFamily where it makes sense to be, rather than being forced to create a brand new info metric with an additional label value. ingestor systems should ensure that they are resilient to such additions.

Changing a MetricFamily's Help is not breaking. For values where it is possible, switching between floats and ints is not breaking. Adding a new state to a stateset is not breaking. Adding unit metadata where it doesn't change the metric name is not breaking.

Histogram buckets SHOULD NOT change from exposition to exposition, as this is likely to both cause performance issues and break ingestors and cause. Similarly all expositions from any consistent binary and environment of an application SHOULD have the same buckets for a given Histogram MetricFamily, so that they can be aggregated by all ingestors without ingestors having to implement histogram merging logic for heterogeneous buckets. An exception might be occasional manual changes to buckets which are considered breaking, but may be a valid tradeoff when performance characteristics change due to a new software release.

Even if changes are not technically breaking, they still carry a cost. For example frequent changes may cause performance issues for ingestors. A Help string that varies from exposition to exposition may cause each Help value to be stored. Frequently switching between int and float values could prevent efficient compression.

5.6. NaN

NaN is a number like any other in OpenMetrics, usually resulting from a division by zero such as for a summary quantile if there have been no observations recently. NaN does not have any special meaning in OpenMetrics, and in particular MUST NOT be used as a marker for missing or otherwise bad data.

5.7. Missing Data

There are valid cases when data stops being present. For example a filesystem can be unmounted and thus its Gauge Metric for free disk space no longer exists. There is no special marker or signal for this situation. Subsequent expositions simply do not include this Metric.

5.8. Exposition Performance

Metrics are only useful if they can be collected in reasonable time frames. Metrics that take minutes to expose are not considered useful.

As a rule of thumb, exposition SHOULD take no more than a second.

Metrics from legacy systems serialized through OpenMetrics may take longer. For this reason, no hard performance assumptions can be made.

Exposition SHOULD be of the most recent state. For example, a thread serving the exposition request SHOULD NOT rely on cached values, to the extent it is able to bypass any such caching

5.9. Concurrency

For high availability and ad-hoc access a common approach is to have multiple ingestors. To support this, concurrent expositions MUST be supported. All BCPs for concurrent systems SHOULD be followed, common pitfalls include deadlocks, race conditions, and overly-coarse grained locking preventing expositions progressing concurrently.

5.10. Metric Naming and Namespaces

EDITOR'S NOTE: This section might be good for a BCP paper.

We aim for a balance between understandability, avoiding clashes, and succinctness in the naming of metrics and label names. Names are separated through underscores, so metric names end up being in "snake_case".

To take an example "http_request_seconds" is succinct but would clash between large numbers of applications, and it's also unclear exactly what this metric is measuring. For example, it might be before or after auth middleware in a complex system.

Metric names should indicate what piece of code they come from. So a company called A Company Manufacturing Everything might prefix all metrics in their code with "acme_", and if they had a HTTP router library measuring latency it might have a metric such as "acme_http_router_request_seconds" with a Help string indicating that it is the overall latency.

It is not the aim to prevent all potential clashes across all applications, as that would require heavy handed solutions such as a global registry of metric namespaces or very long namespaces based on

DNS. Rather the aim is to keep to a lightweight informal approach, so that for a given application that it is very unlikely that there is clash across its constituent libraries.

Across a given deployment of a monitoring system as a whole the aim is that clashes where the same metric name means different things are uncommon. For example acme_http_router_request_seconds might end up in hundreds of different applications developed by A Company Manufacturing Everything, which is normal. If Another Corporation Making Entities also used the metric name acme_http_router_request_seconds in their HTTP router that's also fine. If applications from both companies were being monitored by the same monitoring system the clash is undesirable, but acceptable as no application is trying to expose both names and no one target is trying to (incorrectly) expose the same metric name twice. If an application wished to contain both My Example Company's and Mega Exciting Company's HTTP router libraries that would be a problem, and one of the metric names would need to be changed somehow.

As a corollary, the more public a library is the better namespaced its metric names should be to reduce the risk of such scenarios arising. acme_ is not a bad choice for internal use within a company, but these companies might for example choose the prefixes acmeverything_ or acorpme_ for code shared outside their company.

After namespacing by company or organisation, namespacing and naming should continue by library/subsystem/application fractally as needed such as the http_router library above. The goal is that if you are familiar with the overall structure of a codebase, you could make a good guess at where the instrumentation for a given metric is given its metric name.

For a common very well known existing piece of software, the name of the software itself may be sufficiently distinguishing. For example bind_ is probably sufficient for the DNS software, even though isc_bind_ would be the more usual naming.

Metric names prefixed by scrape_ are used by ingestors to attach information related to individual expositions, so should not be exposed by applications directly. Metrics that have already been consumed and passed through a general purpose monitoring system may include such metric names on subsequent expositions. If an exposer wishes to provide information about an individual exposition, a metric prefix such as myexposer_scrape_ may be used. A common example is a gauge myexposer_scrape_duration_seconds for how long that exposition took from the exposer's standpoint.

Within the Prometheus ecosystem a set of per-process metrics has emerged that are consistent across all implementations, prefixed with process_. For example for open file ulimits the MetricFamiles process_open_fds and process_max_fds gauges provide both the current and maximum value. (These names are legacy, if such metrics were defined today they would be more likely called process_fds_open and process_fds_limit). In general it is very challengings to get names with identical semantics like this, which is why different instrumentation should use different names.

Avoid redundancy in metric names. Avoid substrings like "metric", "timer", "stats", "counter", "total", "float64" and so on - by virtue of being a metric with a given type (and possibly unit) exposed via OpenMetrics information like this is already implied so should not be included explicitly. You should not include label names of a metric in the metric name for the same reasons, and in addition subsequent aggregation of the metric by a monitoring system could make such information incorrect.

Avoid including implementation details from other layers of your monitoring system in the metric names contained in your instrumentation. For example a MetricFamily name should not contain the string "openmetrics" merely because it happens to be currently exposed via OpenMetrics somewhere, or "prometheus" merely because your current monitoring system is Prometheus.

5.11. Label Namespacing

For label names no explicit namespacing by company or library is recommended, namespacing from the metric name is sufficient for this when considered against the length increase of the label name. However some minimal care to avoid common clashes is recommended.

There are label names such as region, zone, cluster, availability_zone, az, datacenter, dc, owner, customer, stage, service, team, job, instance, environment, and env which are highly likely to clash with labels used to identify targets which a general purpose monitoring system may add. Try to avoid them, adding minimal namespacing may be appropriate in these cases.

The label name "type" is highly generic and should be avoided. For example for HTTP-related metrics "method" would be a better label name if you were distinguishing between GET, POST, and PUT requests.

While there is metadata about metric names such as HELP, TYPE and UNIT there is no metadata for label names. This is as it would be bloating the format for little gain. Out-of-band documentation is one way for exposers could present this their ingestors.

5.12. Metric Names versus Labels

There are situations in which both using multiple Metrics within a MetricFamily or multiple MetricFamilies seem to make sense. Summing or averaging aMetricFamily should be meaningful even if it's not always useful. For example, mixing voltage and fan speed is not meaningful.

As a reminder, OpenMetrics is built with the assumption that ingestors can process and perform aggregations on data.

Exposing a total sum alongside other metrics is wrong, as this would result in double-counting upon aggregation in downstream ingestors. ~~~~ wrong_metric{label="a"} 1 wrong_metric{label="b"} 6 wrong_metric{label="total"} 7 ~~~~

Labels of a Metric should be to the minimum needed to ensure uniqueness as every extra label is one more that users need to consider when determining what Labels to work with downstream. Labels which could be applied many MetricFamilies are candidates for being moved into _info metrics similar to database [normalization]. If virtually all users of a Metric could be expected to want the additional label, it may be a better trade-off to add it to all MetricFamilies. For example if you had a MetricFamily relating to different SQL statements where uniqueness was provided by a label containing a hash of the full SQL statements, it would be okay to have another label with the first 500 characters of the SQL statement for human readability.

Experience has shown that downstream ingestors find it easier to work with separate total and failure MetricFamiles rather than using {result="success"} and {result="failure"} Labels within one MetricFamily. Also it is usually better to expose separate read & write and send & receive MetricFamiles as full duplex systems are common and downstream ingestors are more likely to care about those values separately than in aggregate.

All of this is not as easy as it may sound. It's an area where experience and engineering trade-offs by domain-specific experts in both exposition and the exposed system are required to find a good balance. Metric and Label Name Characters

OpenMetrics builds on the existing widely adopted Prometheus text exposition format and the ecosystem which formed around it. Backwards compatibility is a core design goal. Expanding or contracting the set of characters that are supported by the Prometheus text format would work against that goal. Breaking backwards compatibility would have wider implications than just the

wire format. In particular, the query languages created or adopted to work with data transmitted within the Prometheus ecosystem rely on these precise character sets. Label values support full UTF-8, so the format can represent multi-lingual metrics.

5.13. Types of Metadata

Metadata can come from different sources. Over the years, two main sources have emerged. While they are often functionally the same, it helps in understanding to talk about their conceptual differences.

"Target metadata" is metadata commonly external to an exposer. Common examples would be data coming from service discovery, a CMDB, or similar, like information about a datacenter region, if a service is part of a particular deployment, or production or testing. This can be achieved by either the exposer or the ingestor adding labels to all Metrics that capture this metadata. Doing this through the ingestor is preferred as it is more flexible and carries less overhead. On flexibility, the hardware maintenance team might care about which server rack a machine is located in, whereas the database team using that same machine might care that it contains replica number 2 of the production database. On overhead, hardcoding or configuring this information needs an additional distribution path.

"Exposer metadata" is coming from within an exposer. Common examples would be software version, compiler version, or Git commit SHA.

5.13.1. Supporting Target Metadata in both Push-based and Pull-based Systems

In push-based consumption, it is typical for the exposer to provide the relevant target metadata to the ingestor. In pull-based consumption the push-based approach could be taken, but more typically the ingestor already knows the metadata of the target a-priori such as from a machine database or service discovery system, and associates it with the metrics as it consumes the exposition.

OpenMetrics is stateless and provides the same exposition to all ingestors, which is in conflict with the push-style approach. In addition the push-style approach would break pull-style ingestors, as unwanted metadata would be exposed.

One approach would be for push-style ingestors to provide target metadata based on operator configuration out-of-band, for example as a HTTP header. While this would transport target metadata for push-style ingestors, and is not precluded by this standard, it has the disadvantage that even though pull-style ingestors should use their

own target metadata, it is still often useful to have access to the metadata the exposer itself is aware of.

The preferred solution is to provide this target metadata as part of the exposition, but in a way that does not impact on the exposition as a whole. Info MetricFamilies are designed for this. An exposer may include an Info MetricFamily called "target" with a single Metric with no labels with the metadata. An example in the text format might be: ~~~~ # TYPE target info # HELP target Target metadata targe t_info{env="prod",hostname="myhost",datacenter="sdc",region="europe", owner="frontend"} 1 ~~~~

When an exposer is providing this metric for this purpose it SHOULD be first in the exposition. This is for efficiency, so that ingestors relying on it for target metadata don't have to buffer up the rest of the exposition before applying business logic based on its content.

Exposers MUST NOT add target metadata labels to all Metrics from an exposition, unless explicitly configured for a specific ingestor. Exposers MUST NOT prefix MetricFamily names or otherwise vary MetricFamily names based on target metadata. Generally, the same Label should not appear on every Metric of an exposition, but there are rare cases where this can be the result of emergent behaviour. Similarly all MetricFamily names from an exposer may happen to share a prefix in very small expositions. For example an application written in the Go language by A Company Manufacturing Everything would likely include metrics with prefixes of acme_, go_, process_, and metric prefixes from any 3rd party libraries in use.

Exposers can expose exposer metadata as Info MetricFamilies.

The above discussion is in the context of individual exposers. An exposition from a general purpose monitoring system may contain metrics from many individual targets, and thus may expose multiple target info Metrics. The metrics may already have had target metadata added to them as labels as part of ingestion. The metric names MUST NOT be varied based on target metadata. For example it would be incorrect for all metrics to end up being prefixed with staging_ even if they all originated from targets in a staging environment).

5.14. Client Calculations and Derived Metrics

Exposers should leave any math or calculation up to ingestors. A notable exception is the Summary quantile which is unfortunately required for backwards compatibility. Exposition should be of raw values which are useful over arbitrary time periods.

As an example, you should not expose a gauge with the average rate of increase of a counter over the last 5 minutes. Letting the ingestor calculate the increase over the data points they have consumed across expositions has better mathematical properties and is more resilient to scrape failures.

Another example is the average event size of a histogram/summary. Exposing the average rate of increase of a counter since an application started or since a Metric was created has the problems from the earlier example and it also prevents aggregation.

Standard deviation also falls into this category. Exposing a sum of squares as a counter would be the correct approach. It was not included in this standard as a Histogram value because 64bit floating point precision is not sufficient for this to work in practice. Due to the squaring only half the 53bit mantissa would be available in terms of precision. As an example a histogram observing 10k events per second would lose precision within 2 hours. Using 64bit integers would be no better due to the loss of the floating decimal point because a nanosecond resolution integer typically tracking events of a second in length would overflow after 19 observations. This design decision can be revisited when 128bit floating point numbers become common.

Another example is to avoid exposing a request failure ratio, exposing separate counters for failed requests and total requests instead.

5.15. Number Types

For a counter that was incremented a million times per second it would take over a century to begin to lose precision with a float64 as it has a 53 bit mantissa. Yet a 100 Gbps network interface's octet throughput precision could begin to be lost with a float64 within around 20 hours. While losing 1KB of precision over the course of years for a 100Gbps network interface is unlikely to be a problem in practice, int64s are an option for integral data with such a high throughput.

Summary quantiles must be float64, as they are estimates and thus fundamentally inaccurate.

5.16. Exposing Timestamps

One of the core assumptions of OpenMetrics is that exposers expose the most up to date snapshot of what they're exposing.

While there are limited use cases for attaching timestamps to exposed data, these are very uncommon. Data which had timestamps previously attached, in particular data which has been ingested into a general purpose monitoring system may carry timestamps. Live or raw data should not carry timestamps. It is valid to expose the same metric MetricPoint value with the same timestamp across expositions, however it is invalid to do so if the underlying metric is now missing.

Time synchronization is a hard problem and data should be internally consistent in each system. As such, ingestors should be able to attach the current timestamp from their perspective to data rather than based on the system time of the exposer device.

With timestamped metrics it is not generally possible to detect the time when a Metric went missing across expositions. However with non-timestamped metrics the ingestor can use its own timestamp from the exposition where the Metric is no longer present.

All of this is to say that, in general, MetricPoint timestamps should not be exposed, as it should be up to the ingestor to apply their own timestamps to samples they ingest.

5.16.1. Tracking When Metrics Last Changed

Presume you had a counter my_counter which was initialized, and then later incremented by 1 at time 123. This would be a correct way to expose it in the text format: ~~~~ # HELP my_counter Good increment example # TYPE my_counter counter my_counter_total 1 ~~~~ As per the parent section, ingestors should be free to attach their own timestamps, so this would be incorrect: ~~~~ # HELP my_counter Bad increment example # TYPE my_counter counter my_counter_total 1 123 ~~~~

In case the specific time of the last change of a counter matters, this would be the correct way: ~~~~ # HELP my_counter Good increment example # TYPE my_counter counter my_counter_total 1 # HELP my_counter_last_increment_timestamp_seconds When my_counter was last incremented # TYPE my_counter_last_increment_timestamp_seconds gauge # UNIT my_counter_last_increment_timestamp_seconds seconds my_counter_last_increment_timestamp_seconds 123 ~~~~

By putting the timestamp of last change into its own Gauge as a value, ingestors are free to attach their own timestamp to both Metrics.

Experience has shown that exposing absolute timestamps (epoch is considered absolute here) is more robust than time elapsed, seconds since, or similar. In either case, they would be gauges. For

example ~~~~ # TYPE my_boot_time_seconds gauge # HELP my_boot_time_seconds Boot time of the machine # UNIT my_boot_time_seconds seconds my_boot_time_seconds 1256060124 ~~~~

Is better than ~~~~ # TYPE my_time_since_boot_seconds gauge # HELP my_time_since_boot_seconds Time elapsed since machine booted # UNIT my_time_since_boot_seconds seconds my_time_since_boot_seconds 123 ~~~~

Conversely, there are no best practice restrictions on exemplars timestamps. Keep in mind that due to race conditions or time not being perfectly synced across devices, that an exemplar timestamp may appear to be slightly in the future relative to a ingestor's system clock or other metrics from the same exposition. Similarly it is possible that a "_created" for a MetricPoint could appear to be slightly after an exemplar or sample timestamp for that same MetricPoint.

Keep in mind that there are monitoring systems in common use which support everything from nanosecond to second resolution, so having two MetricPoints that have the same timestamp when truncated to second resolution may cause an apparent duplicate in the ingestor. In this case the MetricPoint with the earliest timestamp MUST be used.

5.17. Thresholds

Exposing desired bounds for a system can make sense, but proper care needs to be taken. For values which are universally true, it can make sense to emit Gauge metrics for such thresholds. For example, a data center HVAC system knows the current measurements, the setpoints, and the alert setpoints. It has a globally valid and correct view of the desired system state. As a counter example, some thresholds can change with scale, deployment model, or over time. A certain amount of CPU usage may be acceptable in one setting and undesirable in another. Aggregation of values can further change acceptable values. In such a system, exposing bounds could be counter-productive.

For example a the maximum size of a queue may be exposed alongside the number of items currently in the queue like: ~~~~ # HELP acme_notifications_queue_capacity The capacity of the notifications queue. # TYPE acme_notifications_queue_capacity gauge acme_notifications_queue_capacity 10000 # HELP acme_notifications_queue_length The number of notifications in the queue. # TYPE acme_notifications_queue_length gauge acme_notifications_queue_length 42 ~~~~

5.18. Size Limits

This standard does not prescribe any particular limits on the number of samples exposed by a single exposition, the number of labels that may be present, the number of states a stateset may have, the number of labels in an info value, or metric name/label name/label value/help character limits.

Specific limits run the risk of preventing reasonable use cases, for example while a given exposition may have an appropriate number of labels after passing through a general purpose monitoring system a few target labels may have been added that would push it over the limit. Specific limits on numbers such as these would also not capture where the real costs are for general purpose monitoring systems. These guidelines are thus both to aid exposers and ingestors in understanding what is reasonable.

On the other hand, an exposition which is too large in some dimension could cause significant performance problems compared to the benefit of the metrics exposed. Thus some guidelines on the size of any single exposition would be useful.

ingestors may choose to impose limits themselves, for in particular to prevent attacks or outages. Still, ingestors need to consider reasonable use cases and try not to disproportionately impact them. If any single value/metric/exposition exceeds such limits then the whole exposition must be rejected.

In general there are three things which impact the performance of a general purpose monitoring system ingestion time series data: the number of unique time series, the number of samples over time in those series, and the number of unique strings such as metric names, label names, label values, and HELP. ingestors can control how often they ingest, so that aspect does not need further consideration.

The number of unique time series is roughly equivalent to the number of non-comment lines in the text format. As of 2020, 10 million time series in total is considered a large amount and is commonly the order of magnitude of the upper bound of any single-instance ingestor. Any single exposition should not go above 10k time series without due diligence. One common consideration is horizontal scaling: What happens if you scale your instance count by 1-2 orders of magnitude? Having a thousand top-of-rack switches in a single deployment would have been hard to imagine 30 years ago. If a target was a singleton (e.g. exposing metrics relating to an entire cluster) then several hundred thousand time series may be reasonable. It is not the number of unique MetricFamilies or the cardinality of individual labels/buckets/statesets that matters, it is the total

order of magnitude of the time series. 1,000 gauges with one Metric each are as costly as a single gauge with 1,000 Metrics.

If all targets of a particular type are exposing the same set of time series, then each additional targets' strings poses no incremental cost to most reasonably modern monitoring systems. If however each target has unique strings, there is such a cost. As an extreme example, a single 10k character metric name used by many targets is on its own very unlikely to be a problem in practice. To the contrary, a thousand targets each exposing a unique 36 character UUID is over three times as expensive as that single 10k character metric name in terms of strings to be stored assuming modern approaches. In addition, if these strings change over time older strings will still need to be stored for at least some time, incurring extra cost. Assuming the 10 million times series from the last paragraph, 100MB of unique strings per hour might indicate a use case for then the use case may be more like event logging, not metric time series.

There is a hard 128 UTF-8 character limit on exemplar length, to prevent misuse of the feature for tracing span data and other event logging.

6. Security Considerations

Implementors MAY choose to offer authentication, authorization, and accounting; if they so choose, this SHOULD be handled outside of OpenMetrics.

All exposer implementations SHOULD be able to secure their HTTP traffic with TLS 1.2 or later. If an exposer implementation does not support encryption, operators SHOULD use reverse proxies, firewalling, and/or ACLs where feasible.

Metric exposition should be independent of production services exposed to end users; as such, having a /metrics endpoint on ports like TCP/80, TCP/443, TCP/8080, and TCP/8443 is generally discouraged for publicly exposed services using OpenMetrics.

7. IANA Considerations

While currently most implementations of the Prometheus exposition format are using non-IANA-registered ports from an informal registry at [PrometheusPorts], OpenMetrics can be found on a well-defined port.

The port assigned by IANA for clients exposing data is <9099 requested for historical consistency>.

If more than one metric endpoint needs to be reachable at a common IP address and port, operators might consider using a reverse proxy that communicates with exposers over localhost addresses. To ease multiplexing, endpoints SHOULD carry their own name in their path, i.e. "/node_exporter/metrics". Expositions SHOULD NOT be combined into one exposition, for the reasons covered under "Supporting target metadata in both push-based and pull-based systems" and to allow for independent ingestion without a single point of failure.

OpenMetrics would like to register two MIME types, "application/openmetrics-text" and "application/openmetrics-proto".

EDITOR'S NOTE: "application/openmetrics-text" is in active use since 2018, "application/openmetrics-proto" is not yet in active use.

EDITOR'S NOTE: We would like to thank Sumeer Bhola, but kramdown 2.x does not support "Contributor:" any more so we will add this by hand once consensus has been achieved.

8. References

8.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

8.2. Informative References

[normalization]
"Database Normalization", n.d., <https://en.wikipedia.org/wiki/Database_normalization>.

[PrometheusPorts]
"Prometheus informal port allocation", n.d., <https://github.com/prometheus/prometheus/wiki/Default-port-allocations>.

[timestamp]
"Go Timestamp ProtoBuf", n.d., <https://github.com/protocolbuffers/protobuf/blob/2f6a7546e4539499bc08abc6900dc929782f5dcd/src/google/protobuf/timestamp.proto>.

Authors' Addresses

Richard Hartmann (editor)
Grafana Labs

Email: richih@richih.org

Ben Kochie
GitLab

Email: superq@gmail.com

Brian Brazil
Robust Perception

Email: brian.brazil@gmail.com

Rob Skillington
Chronosphere

Email: rob.skillington@gmail.com