

Data encoding and transmission

Juan Miguel Carceller and Pip Hamilton

University College London, Imperial College London

March 22, 2021

What is Open Metrics?

- Last time, it was suggested to have a look at Open Metrics as a way of encoding
- It's a standard for sending metrics (telemetry) of a system
- Wire format, independent of the way of transport
- It defines data types, metric types, the information that they contain, how these are named...
- More information:

`https://github.com/OpenObservability/OpenMetrics/blob/main/specification/OpenMetrics.md`

Some examples

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.

Data Types

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.

Booleans

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

Timestamps

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

Strings

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

Label

Some examples

Metric Types

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.

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

Histograms

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 $value_1 \leq value_2 \leq value_3 \leq value_+Inf$. 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.

Advantages

- Open standard
- Well defined data types (for example, histogram)
- Protobuf support

Disadvantages

- Does anyone know about this standard?
- Are these data types enough for what we need?
- It's only a protobuf template with the schema
- Not a very short read

My opinion: It's not better than a standard that we define ourselves since very likely it will be unknown to the person who will maintain the system

Alternative to Open Metrics

- Use ROOT format (without ROOT)
- Prototype not complete but 90% matches an empty ROOT file
- Less than 500 C++ lines, without pulling anything from ROOT
- Can have something working soon

Pros

- ROOT format, well known by many
- Decoding is solved, just install ROOT. No need to maintain encoding and decoding in two different places
- What about compression (the prototype does not have any compression)? From what I have seen ROOT uses libraries that are very common so it should be easy to implement

Cons

- An implementation for each class is required (for example TH1). Possibly solved with good documentation. The developer adding a class or algorithm would have to read instructions with protobuf anyways

Alternative to Open Metrics

```
343 void TFile::WriteHeader()  
344 {  
345     const char *root = "root";  
346     write(fFile, root, 4);  
347     write(fFile, fVersion);  
348     write(fFile, fBEGIN);  
349     if (fVersion < 10000000) {  
350         write(fFile, (int)fEND);  
351         write(fFile, (int)fSeekFree);  
352         write(fFile, fNbytesFree);  
353         write(fFile, nfree);  
354         write(fFile, fNbytesName);  
355         write(fFile, fUnits);  
356         write(fFile, fCompress);  
357         write(fFile, (int)fSeekInfo);  
358         write(fFile, fNbytesInfo);  
359     } else {  
360         write(fFile, fEND);  
361         write(fFile, fSeekFree);  
362         write(fFile, fNbytesFree);  
363         write(fFile, nfree);  
364         write(fFile, fNbytesName);  
365         write(fFile, fUnits);  
366         write(fFile, fCompress);  
367         write(fFile, fSeekInfo);  
368         write(fFile, fNbytesInfo);  
369     }  
370     // fUUID.FillBuffer(buffer);  
371     const char *p;  
372     p = hexfun("000108cf6d0489a411eb87005076af87e100");  
373     write(fFile, p, 18);  
374 }  
375  
376 }
```

Alternative to Open Metrics

```
43 void write(std::ofstream& file, char var)
44 {
45     // var = __builtin_bswap32(var);
46     file.write(reinterpret_cast<const char*>(&var), sizeof(var));
47 }
48 void write(std::ofstream& file, short var)
49 {
50     var = __builtin_bswap16(var);
51     file.write(reinterpret_cast<const char*>(&var), sizeof(var));
52 }
53 void write(std::ofstream& file, int var)
54 {
55     var = __builtin_bswap32(var);
56     file.write(reinterpret_cast<const char*>(&var), sizeof(var));
57 }
58 void write(std::ofstream& file, long long var)
59 {
60     var = __builtin_bswap64(var);
61     file.write(reinterpret_cast<const char*>(&var), sizeof(var));
62 }
63 void write(std::ofstream& file, const char* var)
64 {
65     // var = __builtin_bswap32(var);
66     file.write(var, strlen(var));
67 }
68 void write(std::ofstream& file, const char* var, int size)
69 {
70     // var = __builtin_bswap32(var);
71     file.write(var, size);
72 }
73 void writen(std::ofstream& file, int var, int n)
74 {
75     var = __builtin_bswap32(var);
76     file.write(reinterpret_cast<const char*>(&var), n);
77 }
78 void writen(std::ofstream& file, const char* var, int n)
79 {
80     // var = __builtin_bswap32(var);
81     file.write(var, n);
82 }
```

Transmission

- (In NOvA)
- Files are processed in a few machines, a script runs every X minutes to rsync the products to other machines
- There are cron scripts running in the machines used for monitoring that look for files every Y minutes and then make the plots
- Features: No queues or anything fancy, the system is greedy in the sense it will send whatever is available and it will use for the plots everything that is available
- Similar idea for non-real time monitoring? Do we want something simple or more complex?

Backup