Adding Labels to Collectd Metrics

First Proposed: 2018-09-17
Last Updated: 2020-07-15
Authors: Manoj Srivastava <srivasta@google.com>, Florian Forster <octo@collectd.org>
Status: Draft
Reviewed by: <*Sunku Ranganath*>

Objective

1. Create a more powerful and more flexible way to identify metrics.
2. Align collectd with other efforts, such as OpenMetrics, Prometheus, and possibly Metrics 2.0.

Background

The data collected by collectd are time series, consisting of data points, timestamps, and an identity. We'll use the term "metric" to refer to all time series data with the same identity. For example, the amount of time spent by the CPU 0 in the "idle" state would be a metric.

Version 5 of collectd uses six fields to identify a metric:

1. Host
   The name of the (physical or logical) device a metric belongs to
2. Plugin
   The name of the collectd plugin (think: subsystem) that created the metric
3. Plugin Instance (optional)
   The name of the subsystem if there is more than one. For example, the "CPU" plugin may report the usage of each CPU separately and set *plugin_instance* to `"0"`, `"1"`, …
4. Type
   A name that refers to a metric type that is defined in the *types.db* file. Via this indirection, the value type (GAUGE, DERIVE, and COUNTER) and the number and names of "Data Sources" (see 6.) are defined. For example, "total_bytes" is defined as having one "Data Source" called "value" of type DERIVE (cumulative integer).
5. Type Instance (optional)
   Name differentiating multiple metrics of the same type in the same subsystem. For example, the "CPU" plugin may report the time that CPU 0 spent in the "user", "system", and "idle" states by storing the state in the *type_instance* field.
6. Data Source (semi-optional)
   The data structure collectd 5 uses to hold metrics, `value_list_t`, can hold multiple metrics. This feature is deprecated and by convention the single data source is called "value". In that common case, the data source name doesn't provide additional information to the metric's identity.

Metrics have names that identify the time series (usually based on some combination of where they are being collected from, like hostname, or what the time series represents, like load average or memory usage). Currently, collectd uses a fixed 3-5 tuples (three of them are fixed and mandatory, and only two fields are flexible and up to the plugin author) are too inflexible (a service might be better served by sharding metrics by component, or region, or other notional dimensions than the hard coded set in collectd). Using more descriptive, and user defined, set of labels to create the identity of the metric might be more natural for users. Labels make selecting, filtering and grouping much easier. Most modern monitoring systems have mechanisms to query for metrics based on arbitrary labels as long as the metrics carry the labels with them. This functionality is currently missing from the collectd data model, and this proposal is aimed at adding that. Examples are detailed in the background section

Mathematically speaking, the performance service level indicators of real world services have multiple, and potentially different, dimensions of interest. Adding arbitrary, and variable number of labels to define dimensions for the metric allows the system being monitored to specify what the reported dimensionality should be.

This proposal is to

1. Enhance the value list object to accommodate metric labels
2. Make a corresponding adjustment to the wire format
3. Detail how the changes can be done in a backward and forward compatible manner to allow for a gradual transition. (no flag days)

A metric (time series datastream) is usually associated with a number of attributes. These attributes are usually quite service specific and not usually generalizable. Commonly, attributes contribute to the identity of the metric; if these attributes change, then a separate time series data stream is being identified. The number, and nature, of this attribute set may differ from service to service and between deployments, so hard coded constraints may prevent the user's ability to differentiate between metrics naturally and in a manner that fits the entity being measured

> Examples: `client_type` = **Premium|normal**; `service_catgory` = **Frontend|backend**, `server` = **Primary|hot backup**

The collectd data model is that a metric is a time series datastream (each datum in the stream has a timestamp and a value) uniquely identified by a 5-tuple[1] (2 of those elements are optional). Only one of the elements defines the general feature of the system being reported (*type*). Any combination of the other elements contributes to the identity of the metric, but only along a rigid, predefined set of axes. Any user specified breakdown of the metric into multiple, narrower dimensions, or along a different set of axes, is not possible. The constraint of using host and plugin name to define the dimensions that the service owners care about is at odds with real world services with multiple, and potentially different, dimensions of interest. For example, the virt plugin is an excellent example of a plugin that is limited by the current approach.

---

[1] host "/" plugin ["-" plugin instance] "/" type ["-" type instance]

There is an emerging set of conventions, standards and concepts around time series metrics metadata. That site states that there are pretty good time series collection agents, storage and dashboards. But linking a time series to nothing more than a string "name" and maybe a few tags without further metadata is profoundly limiting. Metrics 2.0 aims for self-describing, standardized metrics using orthogonal tags for every dimension. "metrics" being the pieces of information that point to, and describe time series of data. Most modern monitoring platforms (Stackdriver, Prometheus) allow for key-value tuples to be associated with metric identity, and allow filters to select data points from the metric series for further manipulation based on selectors applied to these tuples

A side benefit of adding labels would be that it would be very trivial to write metric ingestion read plugins that collect data from prometheus exporters, making available a rich set of plugins for collectd based systems. If we also update the network protocol to the open metrics standard (essentially the Prometheus exposition format) then collectd will be able to natively understand data from Prometheus exporters.

Requirements and Scale

Additionally, the changes should not perceptibly impact the memory, CPU, or bandwidth usage of the collectd daemon. The wire protocol for collectd data should be backwards compatible. The changes to collectd need to be rigorously tested, and a case would need to be made to the free software community in order to push these changes upstream.

Reference Implementation

There is a work in progress implementation in the `metric-label-support` branch on Github. You can find a pull request referencing that branch here:

> https://collectd.org/3506

Design Ideas

The proposal is to create a new `metric_t` data structure in parallel with, and as an eventual replacement for, the collectd core value_list_t data structure. Instead of a single string identifier, the metric_t data structure has a set of labels that define the identity of the metric the reported data belongs to.

The current `value_list_t` data structure looks like this:

```
union value_u {
  counter_t counter;
  gauge_t gauge;
  derive_t derive;
  absolute_t absolute;
```

```
};
typedef union value_u value_t;

struct value_list_s {
  value_t *values;
  size_t values_len;
  cdtime_t time;
  cdtime_t interval;
  char host[DATA_MAX_NAME_LEN];
  char plugin[DATA_MAX_NAME_LEN];
  char plugin_instance[DATA_MAX_NAME_LEN];
  char type[DATA_MAX_NAME_LEN];
  char type_instance[DATA_MAX_NAME_LEN];
  meta_data_t *meta;
};
typedef struct value_list_s value_list_t;
```

Adding a new data structure (`metric_t`) to replace the `value_list_t` struct also allows for the possibility of simplifying it: remove the concept of "*data sources*" so that each metric contains a single `value_t` (in other words no multi-valued metric like `value_list_t` allows) and include the "metric type" (usually either `GAUGE` or `DERIVE`) eliminates the need for pre-defined "types").

Example:

```
union value_u {
  counter_t counter;
  gauge_t gauge;
  derive_t derive;
  absolute_t absolute;
};
typedef union value_u value_t;
struct id_label_s;
typedef struct id_label_s id_label_t;
struct id_label_s {
  char *key;
  Char *value;
  id_label_t *next;
};
struct identity_s {
  Char *name;
  id_label_t *head;

  pthread_mutex_t lock;
};
```

```
typedef struct identity_s identity_t;

struct metric_s {
  value_t value;
  Int value_ds_type;
  cdtime_t time;
  cdtime_t interval;
  identity_t *identity;
};
typedef struct metric_s metric_t;
```

Plugins consuming metrics, most notably the write plugins, need to be updated to accept the new data type and do something sane. For read plugins we'd introduce a compatibility layer that maps `value_list_t → metric_t` which would be used during development (allowing us to upgrade one read plugin at a time without breaking the "main" branch). This same mechanism would also be used to "*upgrade*" v5 metrics received by a v6 server. Compatibility the other way around (a v6 client sending metrics to a v5 server) is not feasible.

Note: Openmetrics states that label_value can be any sequence of UTF-8 characters, but the backslash (`\`), double-quote (`"`), and line feed characters have to be escaped as `\\`, `\"`, and `\n`, respectively[2].

Binary Protocol

The binary protocol used by the *network plugin* needs to be extended to support labels. collectd's binary protocol is very lightweight, which is very useful for low power devices, such as the ESP8266 (implementation). However, the *network plugin* is expected to play a smaller role going forward, primarily serving (some) backwards compatibility and for low-power devices. Servers and other devices with sufficient compute capacity are expected to transition to an OpenTelemetry or OpenMetrics based approach, likely using TLS+TCP.

For supporting metrics, we need to add serialization for the following:

- metric name
- metric type
- labels
- gauge value
- counter value

TODO(octo): add details for the above.

---

[2] https://prometheus.io/docs/instrumenting/exposition_formats/#text-based-format

Plain text protocol

The plain text protocol used by the *exec* and *unixsock* plugin (input, parsed) as well as the *amqp, amqp1, write_http,* and *write_kafka* plugin (output, generated) needs to be extended to support labels. The successor to the `PUTVAL` command is especially important because it is the most widely used command.

PUTMETRIC

The new `PUTMETRIC` command serializes a metric. When parsed, i.e. when reading a user-provided metric, it has reasonable defaults for all options, so that simple cases stay easy.

Synopsis:

```
PUTMETRIC <metric_name> [options…] <metric_value>
```

Options:

The following options are accepted by the `PUTMETRIC` command:

- **`type=`**`(UNTYPED|GAUGE|COUNTER)`
  Sets the metric type. Defaults to `UNTYPED`, which is handled equivalent to `GAUGE`, i.e. it expects a floating point metric value.
- **`label:`***`<name>=<value>`*
  Sets the label *`<name>`* to *`<value>`*. Multiple (different) labels may be specified.
- **`time=`***`<seconds>`*
  Sets the timestamp of the metric. Fractional seconds are accepted to specify sub-second precision. If omitted or zero, the current time will be used.
- **`interval=`***`<seconds>`*
  Sets the interval in which the metric is read. Fractional seconds are accepted to specify sub-second precision. If omitted or zero, the interval in the plugin's context will be used.

Example:

```
PUTMETRIC example_metric \
  type=GAUGE             \
  label:name="value"      \
  time=1594793126        \
  interval=10.0          \
  42.0
```

Text Based Format

This text based exposition format is the currently supported Prometheus binary format (as of version 2.0). There is a client library (with bindings in C++) for generating the text proto. There is no available library for consuming the textproto, but there is an EBNF description of the format, allowing to create a compatible C language parser using flex+bison for example.

With a 2 byte header and a length field (so that older collectd daemons ignore this fragment) this can easily be supported going forward while retaining compatibility if pre-6.0 collectd instances.

Protobuf based Format

Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data – think XML, but smaller, faster, and simpler. You define how you want your data to be structured once, then you can use special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of languages. There are bindings to produce and consume protocol buffers available in C++, which should also be trivially useable for collectd.

It would be better if we had a different type identifier, which, along with a length part, will allow for compatibility with older collectd instances, and also allow support for both text and protobuf format data moving forward.

Glossary

Annotations:

> A list of key-value pairs attached to individual data points that are part of a time series. The annotations are associated with an individual. In metric 2.0 terms, these are *Extrinsic* properties. Extrinsic properties allow to include information about a metric (in the network proto, in the db, etc), which might change, without changing the metric identifier. I.e. it's metadata. Some examples: You can include the source of a metric (filename and line number), so you know who to contact in case the metric source goes berserk. You can include which agent the data is coming from, without being forced to recreate graphs when you switch to a different agent. You can even include comments if the tags are not sufficient.

Serialized Identifier

> A unique identifier for the metric which is created from the set of labels (key-value pairs) by a stable recipe, and that will be identical to the identifier as used by collectd currently, at least during the transition period. Change a value and you get a different series name.

Identifier

> A string that uniquely identifies the metric (time series). In Metric 2.0 terms, this is what is called *intrinsic* properties. Currently a sealizewd 5-tuple (two of the tuples are optional). The proposed change would replace the 5-tuples with a arbitrary list of labels, along with serializing the label values in a stable order.
>
> The metric identifier is composed by a stable ordering of the key attributes separated by '/'. For instance, the current implementation of this proposal uses

Plugin Name + '/' + Type Name + '/' + Data source Name

This is not a serialization of the label key-value tuples, which would also be a part of the identity iof te metric beyond the identifier or name itself

Labels:

A list of key-value pairs that help to uniquely identify a metric beyond just the metric name. Conventionally, labels can be *target labels* (example: __host__) which often are related to where the metric comes from, and *metric labels*, that shard the metric value into parts.

Metric:

A list of timestamp/value/data tuples (a time series data), uniquely identified by the metric identifier, and values of the Labels

Alternatives Considered

We could break backward compatibility with a new major version. All the core plugins would be patched to the new label based identifier on a separate branch, and the branch would be merged in for the new release. This will break out of tree modules, and will increase TOIL for collectd and distribution maintainers.

If it is desirable to maintain backwards and forwards compatibility, and not gating an incompatible change behind a major version update, at the expense of prohibiting major changes by policy until the next major version update, the following process is feasible. Collectd has a huge installed base and there might be value in rolling out these features in a manner that minimizes disruption to user operations. This reduces the need to have a flag day, and also does not require that every read or write plugin has to change is going to be a non-starter. The exec plugins would be updated along with the core functionality changes (adding identity defining labels) to maintain backwards compatibility.

Instead of creating new `metrics_t` data structure, we could just add the list of identity defining labels and values in a new member of the `value_list_t` structure. The `value_list_t` data structure of collectd core should be enhanced (appended) with a list of labels that can be serialized to define the identifier for the metric the reported data belongs to. Initially, this list can be seeded by the initializer to be identical to the current default identifier. However, since the key labels are arbitrary, the names of the keys should be discoverable. Consider that the serialized identifier is a '/' separated list of label values (in order to maintain backwards compatibility with the monolithic identifier in the transition phase), the resulting serialized identifier would be unintelligibly opaque unless we know what the label keys the values correspond to. Proposal: Add a field in the *`value_list`* "*IDLabels*", whose value would be a '/' separated list of key labels. The write plugins can then decide how, if at all, this field is exposed.

For example, with a stable ordering of keys, the identifier remains the same through the transition period:

- leeloo/cpu-0/cpu-idle
  - 001HOST=leelo 002Plugin=cpu-0 003Name=cpu-idle
    - IDLabels=001HOST/002Plugin/003Name;
    - Serialized identifier=leeloo/cpu-0/cpu-idle
- alyja/memory/memory-used
  - 001HOST=alyja 002Plugin=memory 003Name=memory-used
    - IDLabels=001HOST/002Plugin/003Name;
    - Serialized identifier=alyja/memory/memory-used
- leeloo/load/load
  - 001HOST=leelo 002Plugin=load 003Name=load
    - IDLabels=001HOST/002Plugin/003Name;
    - Serialized identifier=leeloo/load/load

Incrementally / As-needed, modify the read and write plugins to start using the labels instead of the monolithic identifier. The monolithic identifier should exist in parallel with the label list for a period long enough to allow for the plugin maintainers to switch (at least one major release). Optionally, allow for a boolean member of the value_list structure (defaulting to false) for read plugins to indicate to the write plugins to use the label list instead of the monolithic identifier earlier, at which point the compatibility monolithic identifier will be ignored, and the key and corresponding values specified in the label list will be used to define the metric identifier. This allows individual read plugins to message the write plugin on which mechanism to use to determine the metric identifier, easing the gradual transition. It also is a means to know how many read plugin have transitioned. No flag days would be needed to transition to the new mechanism,

The identifier initializer function, the IDLabels initializer function, and the marshaling function (see below) shall be added to core utilities for collectd

Serializing the labels for backwards compatibility

During the transition period, while the labels and the monolithic identifier co-exist, there should be a reproducible and stable ordering of the labels to ensure a ready bidirectional marshalling/unmarshalling process. The proposal is:

1. Labels and values must not contain either '/' or '=' symbols.
2. The monolithic identifier is created by ordering the label keys lexicographically, and the identifier created by concatenating the values associated with the ordered keys separated by '/'.. This maintains backwards compatibility with the current format until the end of the transition period.

The extant labels can be given keys like "*001-Hostname*", "*002-Plugin*", "*003-Plugin-Instance*", …

At the end of the transition period (which perhaps should coincide with a new collectd version), the backward compatibility would be phased out.

The identifier initializer function, the IDLabels initializer function, and the marshaling function (see below) shall be added to core utilities for collectd.

Please note that unless the transition policy that maintains the label based policy resolves into identical serialization as the monolithic identifier had this change will not be backward compatible, since the identity of the metric will change. With the transition policy in place, this results in a backwards compatible wire format.

Custom Binary Protocol

We could add another stanza to the existing binary protocol, with a new type identifier that will make older collectd daemons ignore this. This part consists of:

1. Type (2 bytes)
2. Length (16 bit field)
3. Flag (8bit field)
4. Number of labels (16 bit field)
5. Key length (16 bit field)
6. Key String
7. Value length (16 bit field)
8. Value (string)
9. Repeat fields 5-8 for all the labels (number of labels is specified in FIELD 3)

The flag contains one bit to mean "incremental update". When set, this bit instructs the decoder to copy the previous fields and only update / add the fields present in the "part" being parsed. Other bits will be specified later, as the need arises.. This allows the protocol to (like it does now) only resend fields, for example the plugin name, when it changed from the previous metric. This allows, for example, to efficiently encode the CPU plugin's metrics (which differ in plugin instance and type instance).

Non Identity Defining Annotations

A related concept is annotations attached to the data points: annotations are much like labels, in that they can be used to filter data, but adding annotations does not change the metric identity (in other words, it does not create a new time series). An example is that if I am counting resource utilization (CPU, for example), and am canarying a new kernel version, I can annotate the data with the kernel version. At this point it would be sub-optimal to create a new time series, and lose the history needed for SLOs, but it would be desirable to compare utilization patterns between the machines with different annotations. If identifiers are primary keys, annotations are secondary keys associated with the data point. We could change the collectd model to include the concept of data point annotations that do not change the identity of the metric itself.

We could add another set of key-value tuples (annotations) to the value_list structure. This can be set to NULL by the initializer function, and the absence of annotations should be a no-op. Utility functions to add

key=value annotations, and to remove annotations with a specific key shall be added to the list of utility function. The text format (also used by the exec plugin, will be changed to

<timestamp> <serialized identifier> <value> <annotations>

The annotations list is serialized into a comma separated list of key=value tuples (this implies that annotation, either keys or values, cannot contain unquoted commas or equals signs). Individual write plugin authors determine if the target destination has a concept related to labels and annotations, and if not, ignore the unsupported parts (so the annotation set might not get pushed by every write plugin). The binary protocol would need to be enhanced in a similar manner as labels for annotations, with an identical layout.

We could also try to overload the *value_list* metadata set, and publish that to the targets.This technically qualifies as annotations that does not modify the identity of the metric. This would involve redacting the bits of metadata already in use internally, and would probably cause confusion. It is cleaner to carve out a specific part in the binary protocol for labels, distinct from a field already in use for other purposes that is not reflected in the binary protocol. Additionally, I'm not aware of a time series database / monitoring system making the identifying label/non-identifying annotation distinction. Also, the CPU / kernel version example above works despite the kernel version being part of the metric's identity. Without a compelling use-case, I'll argue to keep it simple, i.e. have *identifying* labels only.

References

- Metrics 2.0
- Open Metrics
- Collectd binary protocol
- Prometheus data model

Feedback

If you read this document please provide your short general feedback in the section below

Email/Date	Comment
sunku.ranganath@intel.com/01/28/2020	Very well thought out proposal. There might be discussion required as to how this would really help or impede use cases of Collectd users. This is really good start.