Kubernetes Metric Conventions

Author: Fabian Reinartz

This document references and outlines general guidelines for metric instrumentation in Kubernetes components. Components are instrumented using the Prometheus Go client library. For non-Go components. Libraries in other languages are available.
The metrics are exposed via HTTP in the Prometheus metric format, which is open and well-understood by a wide range of third party applications and vendors outside of the Prometheus eco-system.

The general instrumentation advice from the Prometheus documentation applies. This document reiterates common pitfalls and some Kubernetes specific considerations.

Prometheus metrics are cheap as they have minimal internal memory state. Set and increment operations are thread safe and take 10-25 nanoseconds (Go & Java) depending on contention. Thus, instrumentation can and should cover all operationally relevant aspects of an application, internal and external.

Instrumentation types

Components have metrics capturing events and states that are inherent to their application logic. Examples are request and error counters, request latency histograms, or internal garbage collection cycles. Those metrics are instrumented directly in the application code.

Secondly, there are business logic metrics. Those are not about observed application behavior but abstract system state, such as desired replicas for a deployment. They are not directly instrumented but collected from otherwise exposed data.
In Kubernetes they are generally capture in the kube-state-metrics repository, which reads them from the API server.
For this types of metric exposition, the exporter guidelines apply additionally.

Naming

Metrics added directly by application or package code should have a unique name. This avoids collisions of metrics added via dependencies. They also clearly distinguish metrics collected with different semantics. This is solved through prefixes:

<component_name>_<metric>

For example, suppose the kubelet instrumented its HTTP requests but also uses an HTTP router providing its own implementation. Both expose metrics on total http requests. They should be distinguishable as in:

```
kubelet_http_requests_total{path="/some/path",status="200"}
routerpkg_http_requests_total{path="/some/path",status="200",method="GET"}
```

As we can see they expose different labels and thus a naming collision would not have been possible to resolve even if both metrics counted the exact same requests.

Resource objects that occur in names should inherit the spelling that is used in kubectl, i.e. daemon sets are `daemonset` rather than `daemon_set`.

Dimensionality & Cardinality

Metrics can often replace more expensive logging as they are time-aggregated over a sampling interval. The multidimensional data model enables deep insights and all metrics should use those label dimensions where appropriate.

A common error that often causes performance issues in the ingesting metric system is considering dimensions that inhibit or eliminate time aggregation by being too specific. Typically those are user IDs or error messages. More generally: one should know a comprehensive list of all possible values for a label at instrumentation time.

Notable exceptions are exporters like kube-state-metrics, which expose per-pod or per-deployment metrics, which are theoretically unbound. However, they have a reasonable upper bound for a given size of infrastructure they refer to.

In general, “external” labels like pod or node name do not belong into the instrumentation itself. They are to be attached to metrics by the collecting system that has the external knowledge. (blog post)

Normalization

Metrics should be normalized with respect to their dimensions. They should expose the minimal set of labels everyone of which provides additional information. Labels that are composed from values of different labels are not desirable. For example:

```
example_metric{pod="abc",container="proxy",container_long="abc/proxy"}
```

It often seems feasible to add additional meta information about an object to all metrics about that object, e.g.:

```
kube_pod_container_restarts{namespace=...,pod=...,container=...}
```

A common use case is wanting to look at such metrics w.r.t to the node the pod is scheduled on. So it seems convenient to add a “node” label.

```
kube_pod_container_restarts{namespace=...,pod=...,container=...,node=...}
```

This however only caters to one specific query use case. There are many more meta infos that could be added effectively blowing up the instrumentation. They are also not guaranteed to be stable over time. What if pods at some point can be live migrated?
Those pieces of information should be normalized into an info-level metric (blog post), which is always set to 1. For example:

```
kube_pod_info{pod=...,namespace=...,pod_ip=...,host_ip=..,node=..., ...}
```

The metric system can later denormalize those along the identifying labels “pod” and “namespace” labels. This leads to...

Resource Referencing

It is often desirable to correlate different metrics about a common object, such as a pod.
Label dimensions can be used to match up different metrics. This is most easy if label names and values are following a common pattern. For metrics exposed by the same application, that often happens naturally.

For a system composed of several independent, and also pluggable components, it makes sense to set cross-component standards to allow easy querying in metric systems without extensive post-processing of data.
In Kubernetes, those are mainly the resource objects such as deployments, pods, or services and the namespace they belong to

For example, a metric specifying the pod it applies to could happen in various forms:

```
example_metric_aaa{pod_name="example-app-5378923", pod_namespace="default"}
example_metric_bbb{pod_name="example-app-5378923", namespace="default"}
example_metric_ccc{pod="example-app-5378923", namespace="default"}
example_metric_ddd{pod="default/example-app-5378923"}
```

As namespace is not specific to the referenced object (given the regular case of there being only/all are about the same namespace), prefixing the “namespace” label name does not add value.
One should refrain from concatenating independent dimensions into an opaque string, which eliminates the last option.
“pod” and “pod_name” seem equally feasible, with the latter one being arguably more explicit. Resource names are only unique at a given time, which is implicitly given in metric data. Thus the following should be consistent:

```
example_metric_ccc{pod="example-app-5378923", namespace="default"}
```

Example: Selecting by UUID even though it’s not part of regular metrics.

kube_pod_restarts and on(namespace, pod) kube_pod_info{uuid=”ABC”}