RBFS Metric Sampling and Monitoring

Version 20.7.1, 15 July 2020

Registered Address	Support	Sales
26, Kingston Terrace, Princeton, New Jersey 08540, United States		
		+91 80 4850 5445
http://www.rtbrick.com	support@rtbrick.com	sales@rtbrick.com

Table of Contents

1. Basic Concepts

Metric sampling is configured through the CTRLD API. The sampled data is stored in Prometheus, an open source monitoring tool with a built-in time series database (TSDB), and can be queried from the switch using PromQL, the Prometheus Query Language. The CTRLD API also supports programming alert conditions in Prometheus Alert Manager.

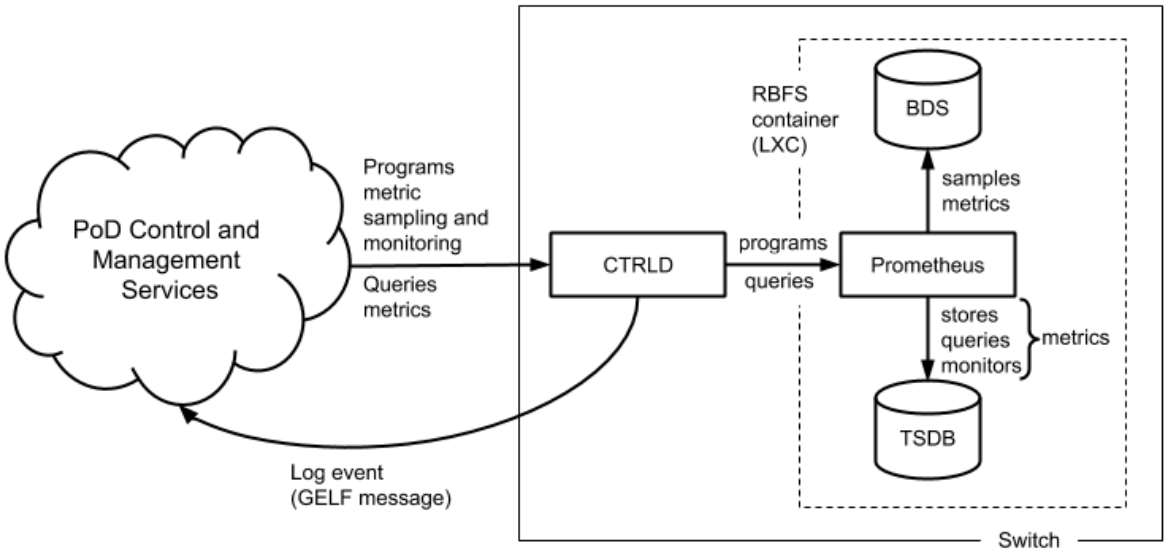

Figure 1. Metric sampling and monitoring overview.

Brick daemons feeding data into BDS are not depicted to keep the drawing simple.

1.1. BDS as Single Point of Truth

The Brick Data Store (BDS) is an object-oriented in-memory database that stores the switch configuration and operational state. BDS objects are typed objects, which means that every object and object attribute is of a certain type. BDS objects are described in schemas and organized in tables. One or more indexes per table exists to query objects. BDS supports sampling values from

- BDS object attributes and from
- BDS table indexes.

Every *numeric* BDS object attribute can be periodically sampled to create a time series of the attribute value. In addition, BDS provides built-in converters for some attribute types that can be converted to numeric values. The bandwidth type is a good example. The bandwidth is stored as a string and consists of a numeric value and a data rate unit, for example, 100.000 Gbps. The built-in converter translates the bandwidth to a numeric value in bits per seconds.

BDS indexes are sampled if the number of objects in a table is of interest. This

tutorial includes examples for object- and index-based metrics.

1.2. Metric Types

There exist two types of metrics:

- gauge
- counters

A gauge metric values are within a certain range and can basically be visualized as is or after applying a simple linear transformation. The value of a gauge metric can increase and decrease. A temperature value is an example for a metric of type gauge.

Counters increase until they are reset manually or by a restart (or by an overflow which is very unlikely to happen because of the length of the data word to store the counter value). The interesting aspect when working with counters is the delta of the count value between two samples, i.e. the derivation of the counter value over time. The derivation of the counter value is still an absolute value that needs to be put into perspective to the available resources to compute the resource utilization in percent. The CPU tick counters are examples for counter metrics.

1.3. Metric Labels

Metric labels separate metric instances from each other. Metric labels have either a static value or is read from a BDS object attribute.

The byte counters, for example, exist for each physical interface. The *ifp_name* label assigns the sampled counter values to the physical interface and is read from the *interface_name* attribute.

1.4. Sampling Rate and Retention Period

The sampling rate is 5 seconds and the retention period is five days. The configuration is built-in to the image and cannot be changed through the CTRLD API.

1.5. Metric Monitoring

Metric monitoring relies on the *Prometheus Alert Manager*. The alert manager notifies CTRLD about all satisfied alert conditions. CTRLD translates the notification to a GELF message /3/ and forwards the message to the log management system. CTRLD exposes an API for programming alert conditions and in turn programs the Prometheus Alert Manager based on the specified alert rules.

2. Temperature Monitoring

The first example in this tutorial samples and monitors temperature values to outline how to work with gauge metrics. Run rtb resmond show sensor to list all available temperature sensors.

Listing 1 - CLI output of temperature values.

```
$ rtb resmond show sensor temperature
+------------+------------------------+------------------------
+------------------------+
 Resource Id  Name                     Status                   Temperature
(millidegree)
+------------+------------------------+------------------------
+------------------------+
 20971520     CPU Core                 PRESENT                  49000
 20971521     LM75-1-48                PRESENT                  33500
 20971522     LM75-2-49                PRESENT                  33000
 20971523     LM75-3-4A                PRESENT                  29000
 20971524     LM75-3-4B                PRESENT                  31000
 20971525     PSU-1 Thermal Sensor 1   PRESENT                  28000
```

This switch has four chassis temperature sensors (LM75), a CPU temperature sensor (CPU Core) and a power supply unit (PSU) temperature sensor (PSU-1 Thermal Sensor 1). A switch typically has two independent power supply units. The second PSU of this switch was not attached in the lab environment.

The temperature is read from the temperature attribute of the *sensor_object* stored in the *global.chassis_0.resource.sensor* BDS table. The sensor object also includes a type (*resource_type* attribute) and a name (*resource_name* attribute). The unit of the temperature is millidegree celsius. An excerpt of the sensor schema definition is listed below:

Listing 2 - Excerpt from BDS sensor object schema definition.

```
      {
        "codepoint": 2,
        "name": "resource_name",
        "type": "string",
        "description": "Name of the resource"
      },
...
      {
        "codepoint": 4,
        "name": "resource_type",
        "type": "string",
        "description": "resource type"
      },
    ...
      {
        "codepoint": 33,
        "name": "temperature",
        "type": "uint32",
        "description": "temperature in millidegree celsius"
      }
```

Contact RtBrick professional services if you need help in finding the BDS table and attribute names.

2.1. Sampling Temperature Sensors

Based on the available sensors it makes sense to create three temperature metrics:

- *chassis_temperature_millicelsius* to sample the chassis temperature
- *cpu_temperature_millicelsius* to sample the CPU temperature and
- *psu_temperature_millicelsius* to sample the PSU temperature.

The CTRLD API exposes the */api/v1/elements/{element}/metrics/{metric_name}* endpoint. A HTTP PUT request to this endpoint configures a metric by either creating a new metric or replacing an existing metric with the specified {*metric_name*}. {*element*} contains the name of the element assigned in the element configuration file and defaults to the container name if no element name was specified. The default container name is rtbrick.

All metrics need to be labeled with the sensor name. In addition, a filter is needed to sample only the sensors for the respective type of temperature. The listings below show the JSON objects to sample the chassis temperature as an example:

Listing 3 - JSON object to configure chassis temperature sampling.

```
{ "metric_name": "chassis_temperature_millicelsius",
  "table_name": "global.chassis_0.resource.sensor",
  "bds_metric_type": "object-metric",
  "metric_type": "gauge",
  "metric_description": "Chassis temperatures in millidegree celsius",
  "attributes": [
      { "attribute_name": "temperature",
        "labels":[
           { "label_name":"sensor",
               "label_value":"resource_name",
             "dynamic":true }
         ]}
  ],
  "filters":[
      { "match_attribute_name":"resource_type",
        "match_type":"exact",
        "match_value":"thermal" },
      { "match_attribute_name":"resource_name",
        "match_type":"regular-expression",
        "match_value":"LM.*"}
  ]}
```

The temperature metric is of type gauge (metric_type) and sample from a BDS object (bds_metric_type). The temperature value shall be sampled, which is of numeric type (uint32, see the schema definition above). The filter section makes sure that only thermal sensors are sampled and also filters for the LM sensors that measure the chassis temperature.

2.2. Querying the Chassis Temperature

The following PromQL query returns the chassis temperature in degree Celsius from the Prometheus running on the switch.

```
chassis_temperature_millicelsius / 1000
```

The next query converts the chassis_temperature from degree Celsius to Fahrenheit:

```
(chassis_stemperature_millicelsius / 1000) * 9 / 5 + 32
```

Both expressions are examples for simple linear transformations of a gauge metric. The queries can be used in Grafana to visualize the chassis temperature time series. The screenshot below shows a chassis temperature panel of a Grafana dashboard:

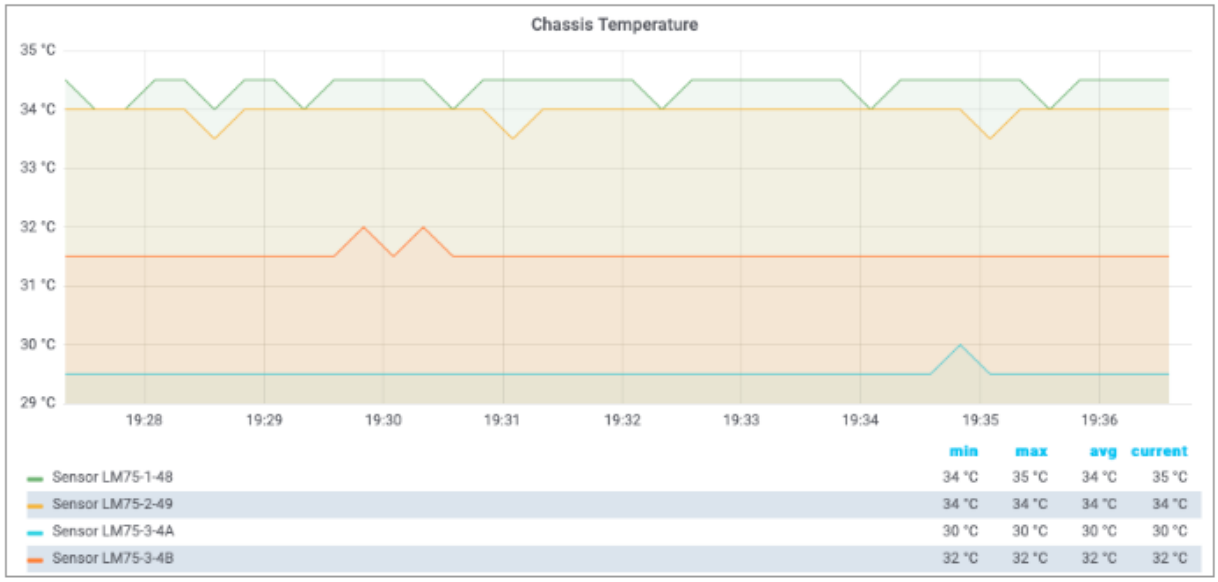

Figure 2. Chassis temperature Grafana panel.

The Grafana dashboard settings for the metrics used in this tutorial can be requested from RtBrick professional services.

2.3. Monitoring Temperature Values

A high temperature can damage the device or shorten its lifetime. Therefore it makes sense to monitor the temperature to get notified about critical temperature values. The alert condition is defined by the acceptable duration of exceeding a specified temperature value, for example, the average chassis temperature is not allowed to exceed 40°C over the last five minutes.

The temperature threshold and evaluation period are example values. The actual values must be taken from the hardware platform documentation or requested from the vendor.

The listing below shows the complete chassis temperature alert rule.

Listing 4 - Chassis temperature alert rule.

```
{ "alert_group": "health",
  "alert_rule_name": "ChassisTemperatureAlert",
  "expr": "(avg_over_time(chassis_temperature_millidegrees[1m]) / 1000) > 40",
  "for": "5m",
  "interval": "1m",
  "description": "The average chassis temperature of {{$labels.element_name}}
     exceeded 40°C for more than 5 minutes.",
  "level": 1,
  "summary": "The chassis temperature exceeded 40°C." }
```

The alert rule evaluates every single minute (interval) whether the average

temperature in the past minute exceeded 40 degrees (expr) and raises an alert if the expression is satisfied for 5 minutes (for), that is, 5 times in a row. The summary field contains a short description of the problem whereas the optional description field contains a more detailed message. The summary is mapped to the *short_message* GELF field and the description is mapped to the *full_message* GELF field. The severity is set to Alert (level). The level attribute values are taken from the GELF format which in turn took it from the Syslog protocol. The table below lists all supported levels:

Table 1. GELF message severity levels

Level	**Description as in RFC 5424**	
	Name	**Comment**
0	Emergency	System is unusable
1	Alert	Action must be taken immediately
2	Critical	Critical conditions
3	Error	Error conditions
4	Warning	Warning conditions
5	Notice	Normal but significant condition
6	Informational	Informational messages
7	Debug	Debug-level messages

Every alert rule has a unique name (alert_rule_name). The PUT operation replaces an existing alert rule with the same name. Every alert rule is assigned to exactly one alert group (alert_group). All alert rules in the same alert group with the same interval setting are evaluated at the same time.

The for attribute is optional. A similar alert rule can be implemented by omitting the for attribute and computing the average temperature over the past five minutes:

Listing 5 - Alternative chassis temperature alert rule.

```
{ "alert_group": "health",
  "alert_rule_name": "ChassisTemperatureAlert",
  "expr": "avg_over_time(chassis_temperature_millidegrees[5m]) / 1000 > 40",
  "interval": "1m",
  "description": "The {{$labels.element_name}} average chassis temperature
     over the past 5 minutes exceeded 40°C.",
  "level": 1,
  "summary": "The chassis temperature exceeded 40°C." }
```

There is a subtle difference between both rules. Consider the following temperature values:

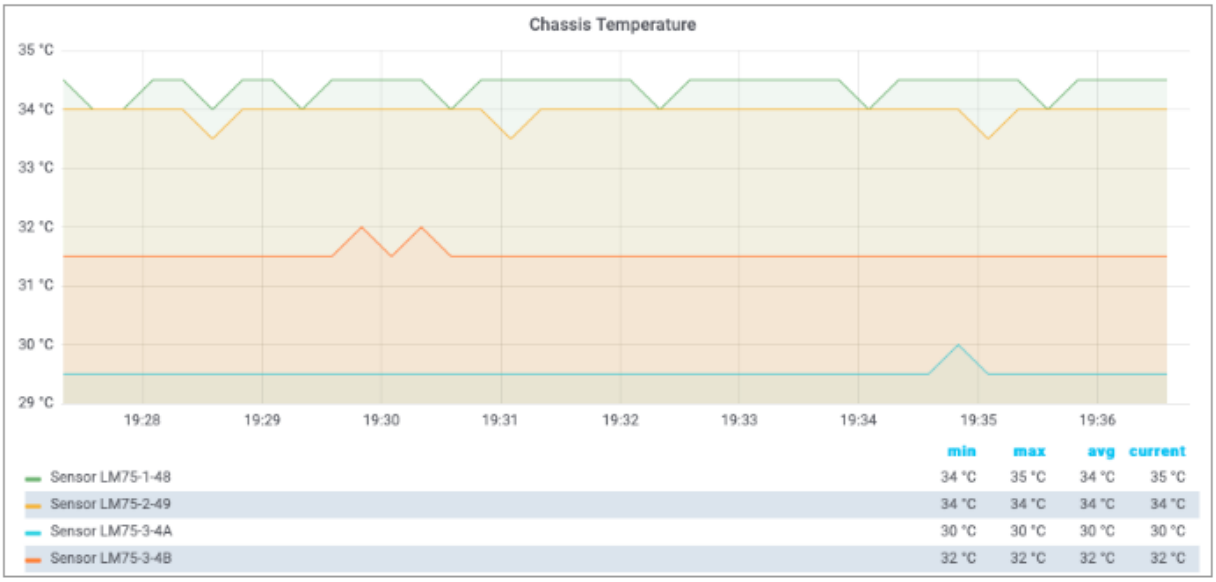

Figure 3. Chassis temperature values.

The first rule does not fire because the threshold is only exceeded for three times, whereas the second rule fires because the average over the past five minutes exceeds 40°C. In fact, the second rule fires an alert albeit the temperature exceeds the alert threshold for 4 minutes only. How about using the min rather than the avg function, i.e. the temperature must exceed the threshold for five minutes? In this case, the rule wouldn't fire an alert if the chassis temperature is wobbling around the threshold.

The first rule aims to mitigate both effects:

- The first rule fires an alert if the chassis temperature is wobbling around the threshold but on average exceeds the threshold five times in a row.
- The first rule does not fire an alert in case of a chassis temperature spike as depicted in Figure 3, because a spike does not satisfy the rule five times in a row.

3. CPU Utilization

The second example measures the CPU utilization to outline how to work with counter metrics. Run rtb resmond show cpu usage to display the current CPU utilization.

Listing 6 - CPU core utilization CLI command.

```
$ rtb resmond show cpu usage
+--------+----------+----------+----------+----------+----------+----------+----------
+----------+
 Name       Total      User       Sys        Nice       I/O-wait   Idle       IRQ
Soft-IRQ
+--------+----------+----------+----------+----------+----------+----------+----------
+----------+
 cpu        4%         2%         2%         0%         0%         95%        0%
0%
 cpu0       1%         0%         1%         0%         0%         99%        0%
0%
 cpu1       16%        12%        4%         0%         0%         83%        0%
0%
 cpu2       3%         2%         0%         0%         0%         96%        0%
0%
 cpu3       10%        1%         9%         0%         0%         89%        0%
0%
 cpu4       2%         1%         1%         0%         0%         97%        0%
0%
 cpu5       4%         0%         4%         0%         0%         96%        0%
0%
 cpu6       3%         3%         0%         0%         0%         97%        0%
0%
 cpu7       0%         0%         0%         0%         0%         100%       0%
0%
+--------+----------+----------+----------+----------+----------+----------+----------
+----------+
```

The CPU provides a set of counters to measure the CPU utilization in jiffies /4/. A jiffy is the duration of a software clock tick, which is platform-dependent. By that, a jiffy is neither a constant period of time nor very meaningful to a human, which is why the counter values need to be put into perspective.

First, it is important to measure the total CPU utilization to see how busy the switch is. Secondly, if the CPU utilization is considerably high, it is interesting to find out which processes cause the high CPU utilization. Both aspects are addressed in this tutorial.

The time spend in user and kernel space needs to be divided by the total amount of available processing time to compute the total CPU utilization:

```
total_cpu_utilization = (total_cpu_user_jiffy + total_cpu_sys_jiffy) /
    (total_cpu_total_jiffy)
```

where

- total_cpu_user_jiffy is the total amount of time spent in user mode in a sampling interval,
- total_cpu_sys_jiffy is the total amount of time spent in kernel mode in a sampling interval and
- total_cpu_total_jiffy is the total amount of computing time available in a sampling interval.

The cpu_total_utilization value is dimensionless. The value range is between 0 and 1. It can be converted into percentage by being multiplied by 100%.

```
total_cpu_utilization_percentage = total_cpu_utilization * 100%
```

The process total load ratio expresses the ratio a process load to the total load:

```
  proc_total_load_ratio =   (proc_cpu_user_jiffy + proc_cpu_sys_jiffy) /
(total_cpu_user_jiffy + total_cpu_sys_jiffy)
```

where

- proc_cpu_user_jiffy is the process time spent in user mode in a sampling interval and
- cpu_sys_proc_jiffy is the process time spent in user mode in a sampling interval

The process_total_load_ratio value is dimensionless. The value range is between 0 and 1. It can be converted into percentage by being multiplied by 100%.

```
proc_total_load_ratio_percentage = proc_total_load_ratio * 100%
```

3.1. Sampling CPU Counters

The CPU counters are located in two different tables. The total CPU utilization can be sampled from the user_cpu_tick, sys_cpu_tick and total_cpu_tick attributes in the global.chassis_0.resource.cpu_usage table. This table contains the total counters but also counters per supported hardware thread (virtual core).

The JSON objects below enables CPU counter sampling for the three mentioned counters:

Listing 7 - JSON object to enable total CPU utilization counter sampling.

```
{ "metric_name": "total_cpu_total_jiffy",
  "table_name": "global.chassis_0.resource.cpu_usage",
  "bds_metric_type": "object-metric",
  "metric_type": "counter",
  "metric_description": "Total CPU utilization",
  "attributes": [
      { "attribute_name": "total_cpu_tick",
        "labels": [
            { "label_name": "cpu",
              "label_value": "cpu_id",
              "dynamic": true }
        ]}
  ]}
```

Listing 8 - JSON object to enable total user mode CPU utilization counter sampling.

```
{ "metric_name": "total_cpu_user_jiffy",
  "table_name": "global.chassis_0.resource.cpu_usage",
  "bds_metric_type": "object-metric",
  "metric_type": "counter",
  "metric_description": "Total user mode CPU utilization",
  "attributes": [
      { "attribute_name": "user_cpu_tick",
        "labels": [
            { "label_name": "cpu",
              "label_value": "cpu_id",
              "dynamic": true }
        ]}
  ]}
```

Listing 9 - JSON object to enable total kernel CPU utilization counter sampling.

```
{ "metric_name": "total_cpu_sys_jiffy",
  "table_name": "global.chassis_0.resource.cpu_usage",
  "bds_metric_type": "object-metric",
  "metric_type": "counter",
  "metric_description": "Total kernel mode CPU utilization",
  "attributes": [
      { "attribute_name": "sys_cpu_tick",
        "labels": [
            { "label_name": "cpu",
              "label_value": "cpu_id",
              "dynamic": true }
        ]}
]}
```

3.2. Computing Total CPU Utilization From Counter Samples

The Prometheus Query Language /3/ provides functions to work with counters and also allows to put time series into perspective.

Some PromQL functions should be used for gauge metrics only others only for counter metrics.

The PromQL queries below computes the total user, kernel and user + kernel CPU utilization:

```
rate(total_cpu_user_jiffy{cpu="cpu"}[60s])
/ rate(total_cpu_total_jiffy{cpu="cpu"}[60s])

rate(total_cpu_sys_jiffy{cpu="cpu"}[60s])
/ rate(total_cpu_total_jiffy{cpu="cpu"}[60s])

( rate(total_cpu_user_jiffy{cpu="cpu"}[60s])
  + rate(total_cpu_sys_jiffy{cpu="cpu"}[60s]) )
/ rate(total_cpu_total_jiffy{cpu="cpu"}[60s])
```

The rate function computes the delta between two sampled count values. The rate function is optimized for counters and can detect counter resets by being aware that a counter value always increases unless a reset has taken place. The rate function handles counter resets properly. The cpu label filters for the total count values for all virtual cores.

The PromQL query below computes the virtual core utilization:

```
( rate(total_cpu_user_jiffy{cpu!="cpu"}[60s])
  + rate(total_cpu_sys_jiffy{cpu!="cpu"}[60s]))
/ rate(total_cpu_total_jiffy{cpu!="cpu"}[60s])
```

The cpu label identifies the virtual core. The BDS contains count values for each virtual core but also the total count over all virtual cores. The first dashboard queried the total count by filtering for cpu="cpu", whereas the second dashboards fetched the per virtual core counters by filtering for cpu!="cpu", i.e. by excluding the total count over all virtual cores from the result set.

The screenshots below show Grafana dashboard panels to display the computed total CPU utilizations and the utilization of the virtual cores.

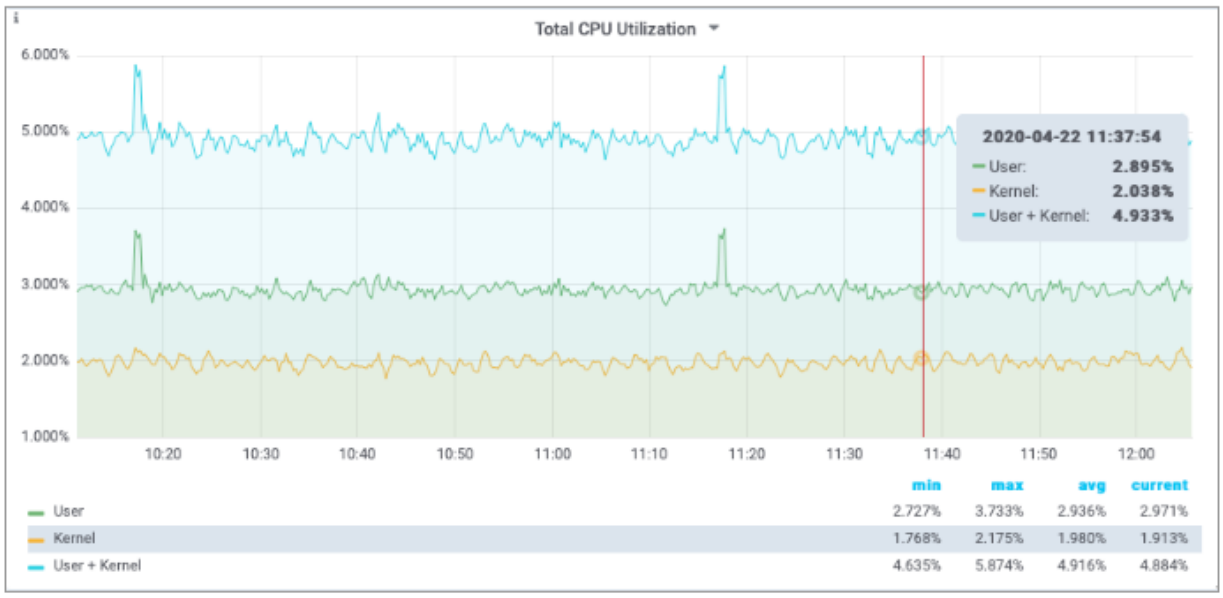

Figure 4. Total CPU utilization Grafana panel.

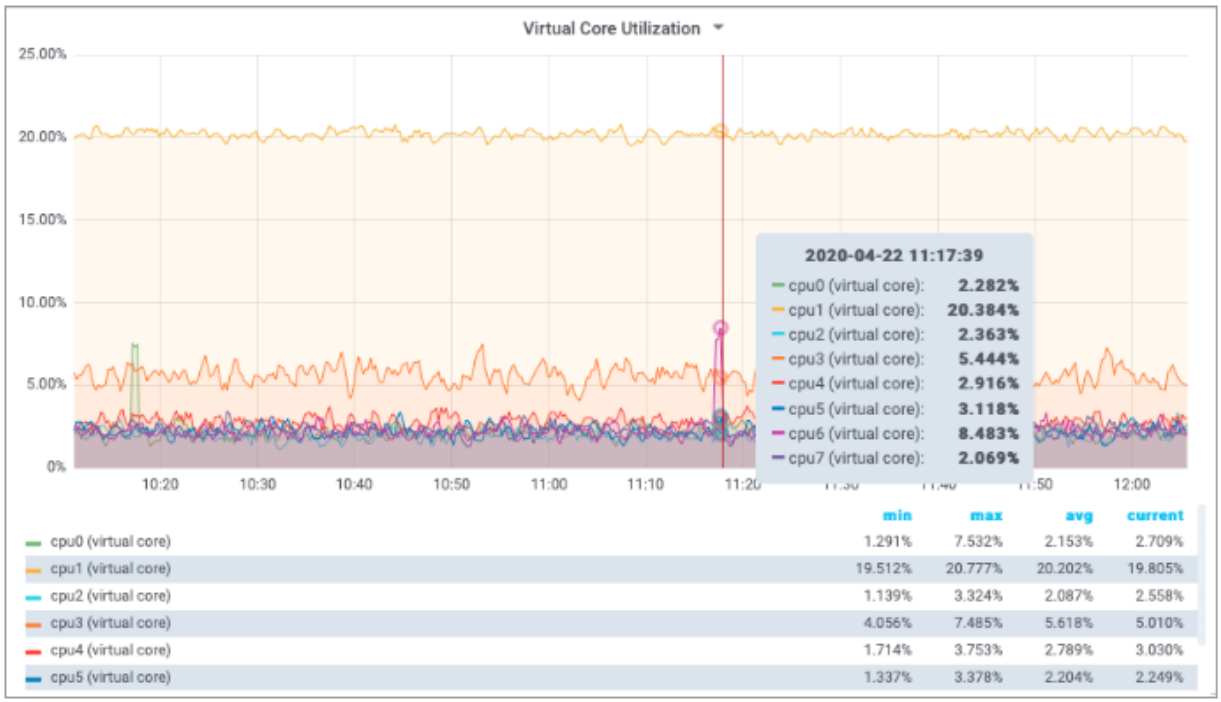

Figure 5. Virtual core utilization Grafana panel.

3.3. Sampling Process CPU Counters

The next step is to compute the per process CPU utilization. This requires to sample the process utilization counters of each process and put them into perspective of the total CPU counters.

The process CPU usage can be read from the cpu_user and cpu_sys attributes in the *global.chassis_0.resource.proc_usage* table. The process name can be read from the *process_name* attribute. The listings below configure user mode and kernel mode CPU utilization sampling per process:

Listing 10 - JSON object to enable process kernel mode CPU utilization sampling.

```
{ "metric_name": "proc_cpu_sys_jiffy",
  "table_name": "global.chassis_0.resource.proc_usage",
  "bds_metric_type": "object-metric",
  "metric_type": "counter",
  "metric_description": "Process user mode CPU utilization",
  "attributes": [
      { "attribute_name": "cpu_sys",
        "labels": [
            { "label_name": "process",
              "label_value": "process_name",
              "dynamic": true }
        ] }
   ]}
```

Listing 11 - JSON object to enable process user mode CPU utilization sampling.

```
{ "metric_name": "proc_cpu_user_jiffy",
  "table_name": "global.chassis_0.resource.proc_usage",
  "bds_metric_type": "object-metric",
  "metric_type": "counter",
  "metric_description": "Process user mode CPU utilization",
  "attributes": [
      { "attribute_name": "cpu_user",
        "labels": [
            { "label_name": "process",
              "label_value": "process_name",
              "dynamic": true }
        ] }
   ]}
```

3.4. Computing Process CPU Utilization From Counter Samples

The PromQL query puts the CPU counters of each process into perspective of the total CPU utilization.

```
( rate(proc_cpu_sys_jiffy[60s]) + rate(proc_cpu_user_jiffy[60s]) )
/ scalar(rate(total_cpu_total_jiffy{cpu="cpu"}[60s]))
```

The scalar function converts the one-dimensional total_cpu_total vector to a scalar to put the CPU process utilization into perspective.

Prometheus differentiates between vectors and scalars. Algebraic operations between two vectors, like the addition of the proc_cpu_sys and the proc_cpu_user vectors above, require that both vectors have the same labels. Otherwise no data points are returned by Prometheus, because a built-in filter excludes all items with different labels from the computation.

The screenshot below shows a Grafana panel to display the total CPU utilization of each brick daemon.

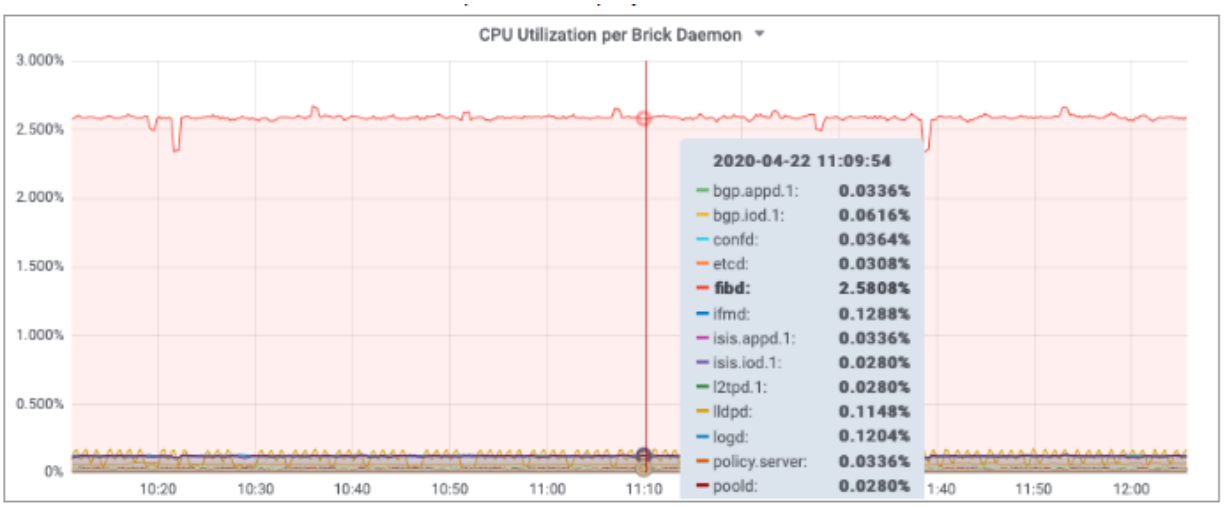

Figure 6. Brick daemon CPU utilization.

4. PPPoE Session Count

Sampling the number of active PPPoE sessions is an example for an index based metric. The PPPoE sessions are stored in the local.pppoe.ppp.sessions table. The active session count can be determined by sampling the active-entry-count attribute of the pppoe_ppp_session_index. The listing below shows the JSON object to sample the active PPPoE sessions:

Listing 12 - JSON object to enable PPPoE session count sampling

```
{ "metric_name": "pppoe_session_count",
  "table_name": "local.pppoe.ppp.sessions",
  "bds_metric_type": "index-metric",
  "index_name": "pppoe_ppp_session_index",
  "metric_type": "gauge",
  "metric_description": "session count",
  "attributes": [
      { "attribute_name": "active-entry-count"}
  ]}
```

The active PPPoE session count can increase and decrease. Therefore the metric type must be gauge rather than counter.

Run rtb pppoed.1 show datastore schema table table-name local.pppoe.ppp.sessions to inspect the definition of the local.pppoe.ppp.sessions table.

Listing 13 - Excerpt of the table definition CLI output

```
$ rtb pppoed.1 show datastore schema table table-name  local.pppoe.ppp.sessions
...
        "index"   : [
        {
            "name"      : "pppoe_ppp_session_index",
            "type"      : "bplus",
            "immutable" : true ,
            "key"       : [
                "ifp_name",
                "outer_vlan",
                "inner_vlan",
                "client_mac",
                "session_id"
            ]
        },
...
```

5. Metric Management

A HTTP GET request to the /api/v1/elements/{element}/metrics CTRLD API endpoint lists all metrics configured on the switch. A HTTP GET request to /api/v1/elements/{element}/metrics/{metric_name} returns the complete metric settings. {element} is the assigned element name and {metric_name} contains the name of the requested metric.

Metric sampling is stopped by sending a HTTP DELETE request to the */api/v1/elements/{element}/metrics/{metric_name}* CTRLD API endpoint to remove the metric settings from the switch configuration. More information can be found in the CTRLD API /1/.

6. Grafana Dashboards

Grafana can visualize time series data from Prometheus /5/. Grafana can query the Prometheus instance on RBFS by using CTRLD as proxy:

```
http://<SWITCH_MGMT_IP>:19091/api/v1/rbfs/elements/rtbrick/services/PROMETHEUS/proxy/
```

The downside of this approach is that a Prometheus datasource needs to be created in Grafana for every switch. In addition, all dashboards must be created per switch too, because a dashboard panel can operate on a single datasource only. Fortunately, Prometheus can federate data from other Prometheus instances /5/. By that, all sampled metrics get accessible through a single Prometheus instance. In combination with Grafana dashboard variables, a dashboard can be configured to access all existing switches.

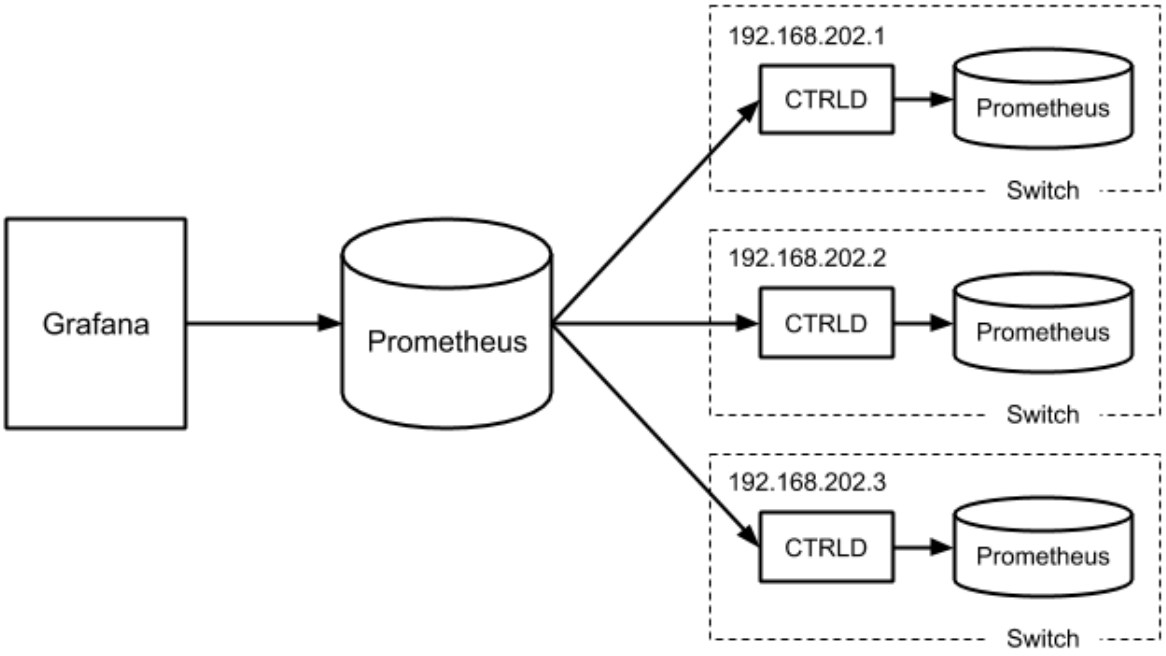

Figure 7. Prometheus federation.

The federating Prometheus instance can assign new label names. This allows to assign a unique element_name label value, if the element name is not specified on the switches and defaults to rtbrick. The listing below shows an excerpt of the Prometheus configuration to federate data from other Prometheus instances.

Listing 14 - Excerpt of a Prometheus federation configuration

```
scrape_configs:
- job_name: federate
  static_configs:
  - targets: ["192.168.202.1:19091"]
    labels:
      element_name: "l1.pod2"
      __metrics_path__:
"/api/v1/rbfs/elements/rtbrick/services/PROMETHEUS/proxy/federate"
  - targets: ["192.168.202.2:19091"]
    labels:
      element_name: "s1.pod2"
      __metrics_path__:
"/api/v1/rbfs/elements/rtbrick/services/PROMETHEUS/proxy/federate"
  - targets: ["192.168.202.3:19091"]
    labels:
      element_name: "bl1.pod2"
      __metrics_path__:
"/api/v1/rbfs/elements/rtbrick/services/PROMETHEUS/proxy/federate"
```

The remaining step is to create a single datasource in Grafana to query the federated time series data.

7. Summary

This tutorial outlines how to configure metric sampling and monitoring in RBFS. Providing a full introduction to Grafana, Prometheus and the Prometheus Query Language would go beyond the scope of this tutorial. However, we mentioned some pitfalls and key aspects for working with PromQL and Grafana and recommend looking up more information in the Grafana and Prometheus documentations.

A postman collection to work with RBFS metrics and Grafana dashboards, including the dashboards this tutorial refers to, can be requested from RtBrick.

8. References

- Switch Management API Overview
  https://documents.rtbrick.com/index_api.html
- Querying Prometheus
  https://prometheus.io/docs/prometheus/latest/querying/basics/
- Overview of time and timers
  http://man7.org/linux/man-pages/man7/time.7.html
- GELF - Graylog Extended Logging Format
  https://docs.graylog.org/en/3.2/pages/gelf.html
- Grafana Documentation
  https://grafana.com/docs/grafana/latest/
- Prometheus Federation
  https://prometheus.io/docs/prometheus/latest/federation/