\chapter{Monitoring requirements}

The base monitoring system is readable and understandable representation of the graph. Graphs allow you to see objects beaning monitored and recognize metric from these objects.

The good monitoring graph gives meaningful description, helps quickly to detect and determine issues via representation. This kind of graph should serve as a motivation for action to solve problems.

There are some simple rules, which makes graphing well:

* Consistency. Representation should correctly reflect reality. All objects, which are represented on the graph, must be correlated with a real data on the machine.
* Graphs need to make sense. All lines represented on the chart have to be readable and understandable. Fake or unreadable information could cause problems. Metrics set should be small, one metric represent one object. There is no need to put multiple objects which are does not bind directly to one chart.
* Stacked area vs. multiline area. Not every chart should have the same visual representation of the lines. Depending on the case we can decide which type of area to use. If there are small time series with a high frequency it is better to use multiline area and stacked area on longer time series but with a bigger metric set respectively.
* Understanding graph before starting to analyze it. Since there are going to be multiple charts with a different metrics we need to make sure that every user can understand the meaning of particular graph. Good naming, fulfilled contend and correct positioning are very important.
* Data hierarchy. It is important to define groups, metrics, data points and nested levels on the chart. Groups help to bind similar objects together. Data points give information of time stamps. Metrics is actual graph representation. Nested level is multiple line metrics. All mentioned data should be visible and accessible.
* Clarity. Designing a chart it is important to take into consideration that there are multiple devices with a different screen resolution. Too many lines on limited space will make chart unreadable. If there are lots of charts on one page it makes people confused what a meaning of this page. That is why it is a good practice to create multiple pages with grouped charts.
* Perspective. It is important to put graphs in such perspective so that any deviation will be easily noticeable.
* Appeal. All charts are people oriented, people today like simple and clean appearance of the applications and if an application has lots of charts they need to be with an appropriate design.
* Control and managing. It should be possible for any user to manipulate a graph. Either to change time series or remove some metric, customization of the graph makes the whole application more attractive.

N2Sky uses all this rules to make an application easy to use.

\chapter{Applying Monitoring}

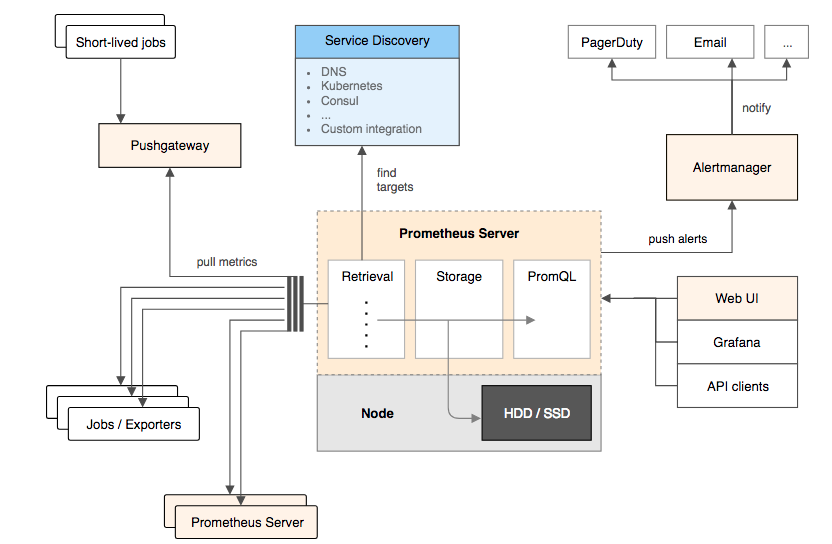
To build continues monitoring system there is need to user a toolkit with an active ecosystem. Searching for a proper toolkit the toolkit should fulfilled some specific requirements that going to be use in N2Sky:

* Proper and self-describable metric name with a key pairs
* Possibility to query metrics and join them in one graph.
* Not resource-intensive
* Support HTTP/HTTPS protocol
* Collect and push to repository time series
* Scalable

After researching it was decided to go for Prometheus Monitoring Toolkit. [1] This tool support all requested requirements. One of most interesting feature is that Prometheus can be used on any UNIX environment. Since in Openstack multiple instance with a different operational system can be created, Prometheus will match exact our needs.

Originally Prometheus was created by SoundCloud team in 2012. [2] The core of monitoring application is Prometheus server, which collects time series data from the moment it was executed on environment. All components are written on Go [3] language and support multiple modules for monitoring different environment metrics.

To understand the nature of Prometheus it is necessary to explain its architecture.



The core Prometheus server pull all metrics from jobs which are instrumented, if the service is unavailable for instrumentation it can be pulled from push gateway. All metrics and logs data is stored locally so there is no distributed storage. It is possible to query this data to retrieve more specific information about particular metrics of joint metrics. N2Sky uses Prometheus API to build own customized dashboards.

The common components of Prometheus architecture:

* The Prometheus server. This is the base element in the whole architecture. The server include services which collection, storing and retrieving nodes. The principle is scrapping or pulling. It means that the data fetched with some interval, which can me configured and stored accordantly as a time series. Prometheus support different modules, each module represents some node. The nodes expose these ports that Prometheus uses for retrieving the data. For example in N2Sky we are using Node Exporter Module which gives possibility to collect almost all essential data like CPU, RAM, HDD/SSD etc.
* Push gateway. There are some nodes, which are not exposing these endpoints. In this case collection of the data throw Push gateway is possible. Prometheus short-lived jobs are executed to capture the data and convert it to the time series that can be used by Prometheus.
* Alert Manager. Prometheus support the module which responsible for alerting in case of deviations. The alerts can be send over multiple channels like SMS, Email etc. More information about alert manager is located in Alert System chapter.

As it was mentioned before all data stored locally, it stored as a time series. Every time series is a stream of time stamped values. Every time stamp has labels, which represent additional information about metric. The labels make metrics to be multi-dimensional.

Follow example represent notation for metric:

node\_filesystem\_avail {method="GET", endpoint="/api/posts", status="200"}

After executing this query the data will be retrieved from logs and represent as a time series:

| **Element** | **Value** |
| --- | --- |
| node\_filesystem\_avail{device="/dev/sda2",fstype="ext4",instance="localhost:9100",job="node",mountpoint="/boot"} | 327512064 |
| node\_filesystem\_avail{device="/dev/sda5",fstype="ext4",instance="localhost:9100",job="node",mountpoint="/"} | 53616574464 |
| node\_filesystem\_avail{device="/dev/sdb1",fstype="ext4",instance="localhost:9100",job="node",mountpoint="/home"} | 906128850944 |
| node\_filesystem\_avail{device="/dev/sdb2",fstype="ext4",instance="localhost:9100",job="node",mountpoint="/var"} | 417015595008 |
| node\_filesystem\_avail{device="/dev/sdb2",fstype="ext4",instance="localhost:9100",job="node",mountpoint="/var/lib/docker/aufs"} | 417015595008 |
| node\_filesystem\_avail{device="/dev/sdb2",fstype="ext4",instance="localhost:9100",job="node",mountpoint="/var/lib/docker/plugins"} | 417015595008 |
| node\_filesystem\_avail{device="/dev/sdb3",fstype="ext4",instance="localhost:9100",job="node",mountpoint="/opt"} | 279920013312 |
| node\_filesystem\_avail{device="/dev/sdb4",fstype="ext4",instance="localhost:9100",job="node",mountpoint="/usr/local"} | 186646953984 |
| node\_filesystem\_avail{device="gvfsd-fuse",fstype="fuse.gvfsd-fuse",instance="localhost:9100",job="node",mountpoint="/run/user/1000/gvfs"} | 0 |
| node\_filesystem\_avail{device="none",fstype="aufs",instance="localhost:9100",job="node",  mountpoint="/var/lib/docker/aufs/mnt/37b40f519e26a974577833d0668a34e04134e5fd2537f1fb4c2fe6b7b0d53539"} | 417015595008 |
| node\_filesystem\_avail{device="nsfs",fstype="nsfs",instance="localhost:9100",job="node",mountpoint="/run/docker/netns/1-szz5t6pxd8"} | 0 |
| node\_filesystem\_avail{device="nsfs",fstype="nsfs",instance="localhost:9100",job="node",mountpoint="/run/docker/netns/618ed9d7179f"} | 0 |
| node\_filesystem\_avail{device="nsfs",fstype="nsfs",instance="localhost:9100",job="node",mountpoint="/run/docker/netns/ingress\_sbox"} | 0 |
| node\_filesystem\_avail{device="shm",fstype="tmpfs",instance="localhost:9100",job="node",  mountpoint="/var/lib/docker/containers/be36a80809d4fc59a1f8c3bd0da32b6e753e1ab7902d2676d50527d9e4458a74/shm"} | 67108864 |
| node\_filesystem\_avail{device="tmpfs",fstype="tmpfs",instance="localhost:9100",job="node",mountpoint="/run"} | 1237848064 |
| node\_filesystem\_avail{device="tmpfs",fstype="tmpfs",instance="localhost:9100",job="node",mountpoint="/run/lock"} | 5238784 |
| node\_filesystem\_avail{device="tmpfs",fstype="tmpfs",instance="localhost:9100",job="node",mountpoint="/run/user/1000"} | 1248632832 |
| node\_filesystem\_avail{device="tmpfs",fstype="tmpfs",instance="localhost:9100",job="node",mountpoint="/run/user/123"} | 1248690176 |

One of the requirements to our monitoring system is a scalability. One of the greatest features of Prometheus is that event if environment going to be overloaded it will generate the same amount of metrics anyway. Hence the amount of events is independent on amount on generated time series.

Talking about requirements it is important to mention here that there is possibility to build joint metrics namely build multiple time series using this kind of metric:

* Counter. The counter is a metric is representing a simple numerical value which can be incremented but not inverse. One of the typical examples is a number of expectations to be occurred.
* Gauge. The gauge is a metric, which also represents a simple numerical value like counter, but it is bidirectional. It means that this value can be decremented. The common example is CPU usage, which can go up and down.
* Histogram. The histogram is a metric, which represent observations. It is stored as a bucket, which can be pulled. Any bucket can be configured depending on the need. It can be sum of values or count of events, which are observed.
* Summary. The summary is a metric, which is similar to histogram but it calculates configurable quantities.

\chapter{Applying Monitoring to N2Sky}

Prometheus supports query language, which is a key feature for this tool. The Prometheus query language, or promql, is an expressive.

With Prometheus the self-described metric name can be choose. Prometheus converts all metric so that every human can understand what exactly particular metric means. Lets take a previos example with a metric “node\_filesystem\_avail”. This metric will show the folders on root and available memory on each of it.

The response looks like this:

|  |  |
| --- | --- |
| node\_filesystem\_avail{device="/dev/sdb4",fstype="ext4",instance="localhost:9100",job="node",mountpoint="/usr/local"} | 186646953984 |

It means that on “/usr/local” 186.6 GB is available.

There is also possibility to check a response code, which especially useful for alerting.

node\_filesystem\_avail {status="500"}

This request returns some response code 500 namely internal server error.

For building proper dashboard for monitoring it is important to provide customization that is why Prometheus supports time duration:

* **s** - seconds
* **m** - minutes
* **h** - hours
* **d** - days
* **w** - weeks
* **y** - years

Using the time duration with an offset it is possible to get exact metric on demand.

Building query with a Prometheus can bring lots of advantages. For example there is query which use counter with a available node file system metric:

topk(

3, sum(

rate(

api\_http\_requests\_total{status=500}[1h]

) )

by (endpoint)

)

This query looks already complicated, but it is just one of the first examples.

Building N2Sky user interface was expired by applying easy to use approach. The idea behind is to make user experience more simple to use

In N2Sky was developed monitoring service, which use micro services approach like in entire application. It was decided to get rid of complex queries and provide some intuitive way of creating metrics.

First of all the time range add a complexity. It was decided that the user should give only time interval and step. Lets say the user want to see CPU load for a last hour with a step 30 seconds. It makes creation of metric more intuitive, no more range like “from”, “to” and type of ranges. All this can be solved with a one simple request.

Second part is a storing of metric. Instead of every time build a query the monitoring service saving requested by user metric. In this case every user will get his own customized metric.

The service uses Mongo DB for storing the metric configuration. Every collections has it own schema. When a user make a request to save a metric the schema have to be filled with a requested by user data.

Lets take a look on some simple example:

POST /user/dashboard/monitoring

{

user: admin

metric: node\_filesystem\_avail,

delay: 1,

delaytype: hour,

step: 30,

steptype: second,

server: PROD\_HOST ,

show: ["all", “Openstack”],

selectedServerId : 1,

selectedServerName : Openstack

}

GET /monitoring/:minus/:type

\chapter{Monitoring dashlet Design}

Since there are multiple machine and services to monitor there was need to create a dedicated dashboard design. At first lets take a look on the environments we have to monitor:

* Openstack Machine. It is dedicated machine, our cloud base for development and running instances.
* Openstack Instances. Virtual machines with a different OS.
* Docker Containers. Virtual machines in the openstack instances.

One of the most important part in application design is to maximize reusability of the components.

[1] <https://prometheus.io/docs/introduction/overview/>

[2] https://soundcloud.com/