NPD: Metrics Exporting Option

*Status: Draft | **Under review** | In progress | Done*
Authors: xueweiz@google.com
Last Updated: 2019-05-21

Audience

This doc is for the maintainers/contributors of the open source Kubernetes project node problem detector (NPD). For more background on NPD, the README and the design doc V0 are some good source.

Why NPD needs metrics exporting

Today's node problem detector (NPD) only exports node problems to Kubernetes' Event and Node Condition API by design. This design faces a few challenges:

1. Some problems are better detected at fleet level

We want to have a common mechanism to collect basic VM metrics that are useful to diagnose node problems. Such diagnosis may need VM metrics to be associated with other signals collected outside the VM to determine whether it is indeed a node problem. For this reason, we just want to implement generic metric collection in NPD instead of using these signals to directly determine node problems as we are currently doing through Node Conditions/Events.

2. NPD's dependency on apiserver limits its use cases

We want to be less dependent on k8s control plane so that we can effectively monitor k8s control plane health. Relying on Events/Node Conditions API makes that very hard.

Proposal

Goal

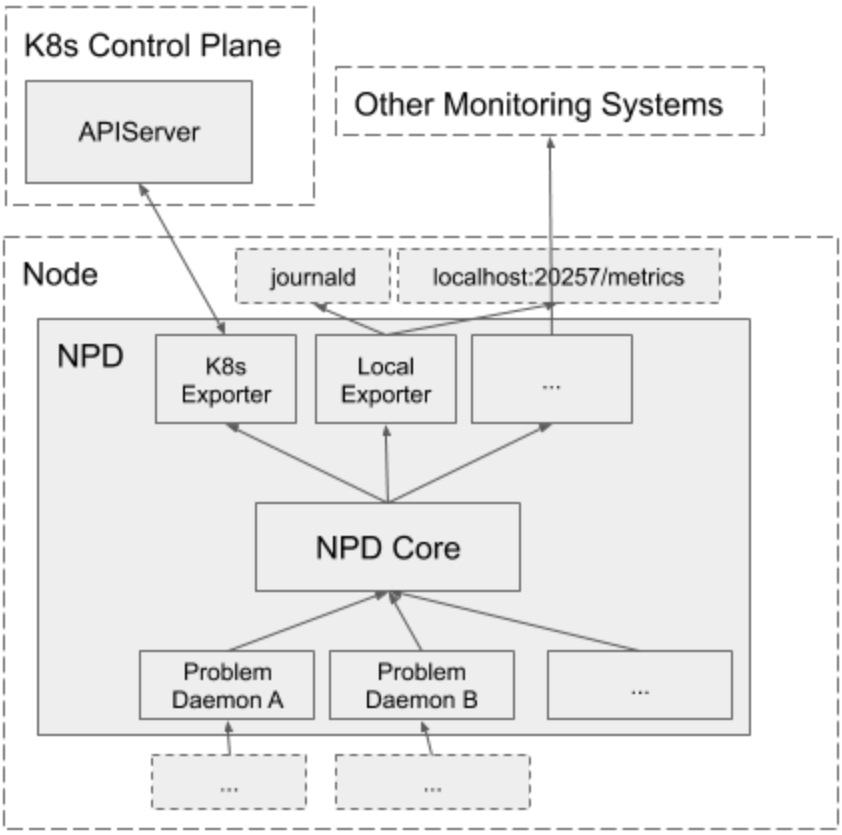
- Give NPD options to report problems and raw system data (e.g. disk IO queue length) in the form of metrics.
- Give NPD options to export metrics locally (i.e. Prometheus).
- Allow easy extensibility of exporting data to other monitoring/alerting systems.
- NPD should maintain backward compatibility with current NPD plugins.

Overview

A few changes will be made to NPD to adopt the metrics mode:

1. We will refactor the data model that NPD uses to represent problems (i.e. Status, which was inherited from Events/Node Conditions API and is not generic enough). The new data model will represent node problems in a generic format that do not have platform dependency. Of course the problems can still be converted into Events and Node Conditions.
2. We will implement flexible exporter plugin registration and configuration, allowing users to use any subset of exporters (e.g. Kubernetes exporter, Prometheus exporter, etc).
3. We will allow NPD core to collect metrics from problem daemons and export locally.
4. We will implement some problem daemons to collect various health-related system stats.
5. We will extend today's problem daemons so that they can report metrics with little config change.
6. We will refactor the code for problem daemon registration, to make it more modular and pluggable.

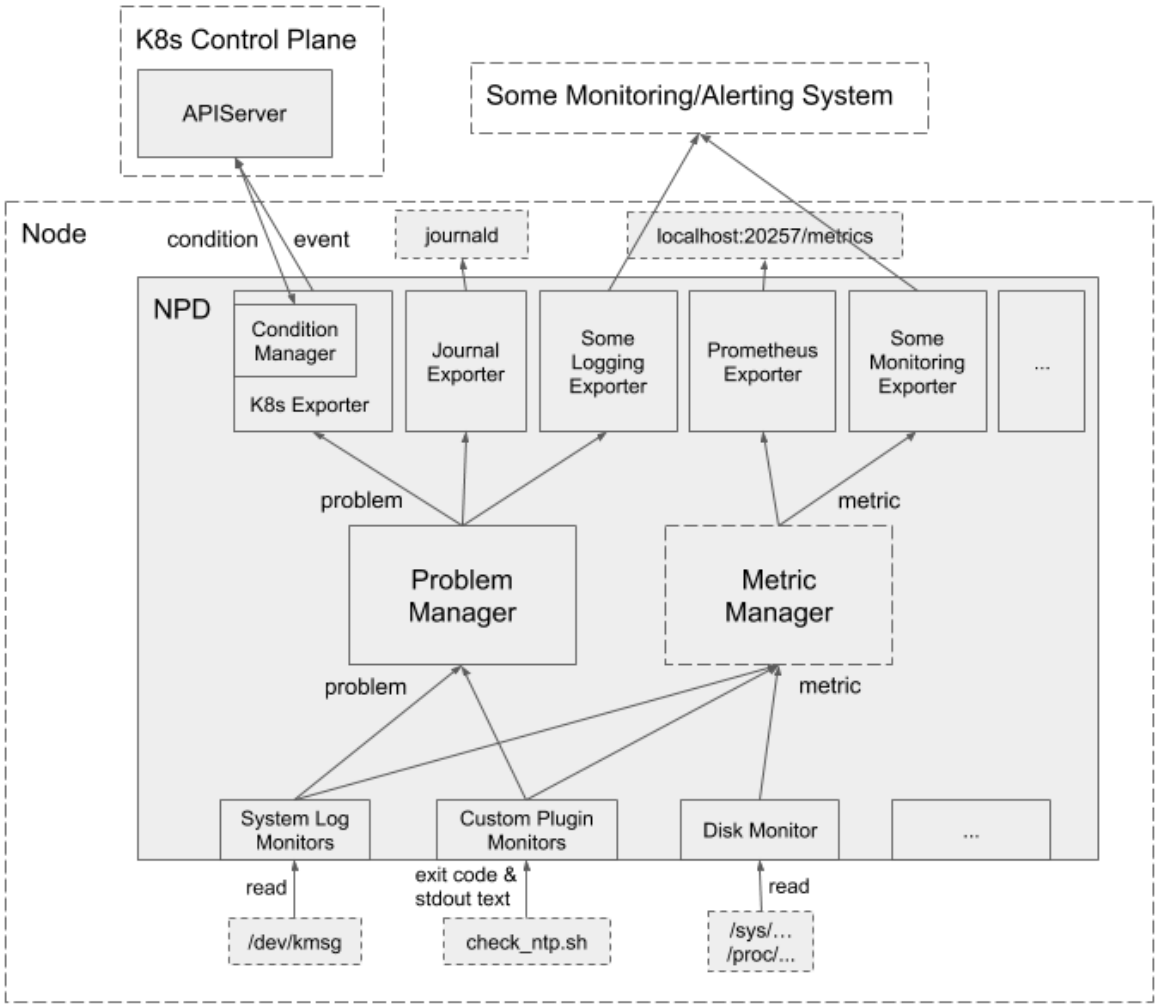
The overview of the new NPD architecture would look like this (details are in below section):



Detailed Design

Detailed NPD architecture

The new NPD architecture would look like below:



Core Data Representation Model

We want to use some minimum set of fundamental data representations to represent node health, which will be the core representations that NPD works with. To interact with various control planes (Events/Node Conditions API in Kubernetes, Prometheus, other monitoring systems, etc), exporters should be responsible for converting the core representations to the control plane format.

Here is the proposed core data representation model:
Metric: Represents a dimension of the current healthiness of the system.
Problem: Represents the debugging information regarding a detected or suspected node problem.

Here is an example *problem* object:

```
{
   "severity":"permanent",
   "timestamp":"2012-10-31 15:50:13.793654 +0000 UTC",
   "type":"KernelDeadlock",
   "reason":"AUFSUmountHung",
   "message":"[...]task umount.aufs:xxxx blocked for more than 180
seconds.[...]"
}
```

For metrics, we plan to delegate the aggregation and reporting work to some vendor-neutral generic metric library (e.g. Prometheus, OpenCensus). That's why the "Metric Manager" box in the NPD architecture graph is dotted box.
The format of the metric will depend on what library we use. Let's take disk IO queue length as an example. It should eventually be reported as a metrics like this (shown in Prometheus format):

```
disk_queue_len{"device": "/dev/sda1"} 133
```

And here is how we would define it in OpenCensus library:

```
// keyDevice defines a TagKey, which is essentially converted to a
metric label. e.g. {"device": "/dev/sda1"}
keyDevice, _ = tag.NewKey("device")

// mQueueLen defines a Measure.
// Whenever a new data point is collected, a Measurement should be made
to this measure via:
// stats.Record(ctx, mQueueLen.M(133))
mQueueLen = stats.Int64("disk-queue-len", "The disk IO queue length",
"1")

// vDiskQueueLen defines a View.
// The view specifies how should measurements be aggregated over time.
// i.e. should we add them together? should we take the last value?
//
// In this example,
// "disk-queue-len" is the metric name that eventually will be shown to
the users.
// mQueueLen specifies the measure.
// "The disk IO queue length" is the helper text that will be shown to
the users.
```

```
// []tag.Key{keyDevice} specifies that this metric has a "device" label.
// view.LastValue() specifies that we want the last measurement to
overwrite previous measurements.
vDiskQueueLen = &view.View{
      Name:        "disk-queue-len",
      Measure:     mQueueLen,
      Description: "The disk IO queue length",
      TagKeys:     []tag.Key{keyDevice},
      Aggregation: view.LastValue(),
}
```

So when a real data point comes, all that's needed to report the measurement:

```
ctx, _ := tag.New(context.Background(), tag.Insert(keyDevice,
"/dev/sda1"))
stats.Record(ctx, mQueueLen.M(133))
```

All the aggregation and exporting will happen automatically afterwards.

Eventually, all data reported from problem daemons should be collected by NPD core in the form of *metrics* and *problems*.

Intuitive view of how the transition looks like on Kubernetes

Today, when NPD detects a permanent problem that makes the node unavailable for pods, it will set a Node Condition, e.g.:

```
{
   "type":"KernelDeadlock",
   "status":"True",
   "transition":"2012-10-31 15:50:13.793654 +0000 UTC",
   "reason":"AUFSUmountHung",
   "message":"[...]task umount.aufs:xxxx blocked for more than 180
seconds.[...]"
}
```

And when NPD detects a temporary problem that has limited impact on pod but is informative, it will report an event, e.g.:

```
{
   "severity":"warn",
   "timestamp":"2012-10-31 15:50:13.793654 +0000 UTC",
```

```
   "reason":"OOMKilling",
   "message":"[...]Kill process 677 dockerd…[...]"
}
```

After the proposed changes, both permanent problems and temporary problems will be presented as “problems” in NPD core:

```
{
   "severity":"permanent",
   "timestamp":"2012-10-31 15:50:13.793654 +0000 UTC",
   "type":"KernelDeadlock",
   "reason":"AUFSUmountHung",
   "message":"[...]task umount.aufs:xxxx blocked for more than 180
seconds.[...]"
}
```

```
{
   "severity":"temporary",
   "timestamp":"2012-10-31 15:50:13.793654 +0000 UTC",
   "type":"",
   "reason":"OOMKilling",
   "message":"[...]Kill process 677 dockerd…[...]"
}
```

Node health status of interest can be reported as metrics as well. Say if we are interested in tracking both problems above, an option is to show them as counter metrics, and put the problem type in the metric labels (shown as Prometheus format):

```
problem_counter{"name": "KernelDeadlock", "reason": "AUFSUmountHung"}  1
problem_counter{"name": "OOMKilling"}  2
```

Basically the problem type will be shown in metric labels.

And Kubernetes exporter will export these problems to Kubernetes in the form of Events / Node Conditions:

```
{
   "severity":"warn",
   "timestamp":"2012-10-31 15:50:13.793654 +0000 UTC",
   "reason":"KernelDeadlock-AUFSUmountHung",
   "message":"[...]task umount.aufs:xxxx blocked for more than 180
seconds.[...]"
}
```

```
{
   "severity":"warn",
   "timestamp":"2012-10-31 15:50:13.793654 +0000 UTC",
   "reason":"OOMKilling",
   "message":"[...]Kill process 677 dockerd…[...]"
}
```

Extendable Exporters

The new NPD design features configurable/plugable exporters. Users may pick a set of exporters to use according to the environment the node runs in, and exporters will be registered to the NPD core. All exporters should have an option to be disabled at compile time (to keep NPD small).

Initially, we plan to introduce below exporters:

- k8s-exporter: reports problems as Node Conditions and Events.
- journal-exporter: reports problems to systemd-journald.
- prometheus-exporter: reports metrics locally in Prometheus format.

Note that k8s-exporter and journal-exporter practically already exist today, since NPD already logs problems locally as well as reports to Kubernetes apiserver. We just need some refactoring on them to adopt the new model.

More exporter plugins can be added in the future to support other environments.

Metrics Collection Design

We plan to use OpenCensus library for the metrics collection. OpenCensus supports flexible metrics collection, aggregation and exporting ability.

Under the OpenCensus library, the end-to-end experience of adding a metrics looks like this:

1. Define a measurement, which could be a counter (e.g. NPD uptime), a diff (e.g. number of crashes in last minute), a gauge (e.g. disk bandwidth in last minute), etc…
2. Define a view from the measurement, which specifies the method for aggregating the measurement. For example count, sum, or overwrite last value.
3. Attach an exporter, which could be Prometheus, Datadog, Stackdriver, etc…
4. When the problem daemon has a data point (e.g.number of crashes in last minute), the problem daemon only need to report the data point to the measurement.
5. OpenCensus library then handles the aggregation and exporting work.

Using OpenCensus library saves us the burden of implementing the “Metric Manager” component in the new architecture. And comparing to Prometheus library, OpenCensus has the advantage of the built-in exporter support for many control planes.

Collecting System Stats

As part of this work, we will make problem daemons to collect various system stats that are important to machine health, e.g. some stats in /proc/meminfo, /proc/slabinfo, etc. But we’d like to avoid writing code to parse through these files, because the parsing logic may be platform dependant and very error-prone.

Our plan is to use the gopsutil library inside the problem daemons to handle all the parsing logic.

Extending Today’s Problem Daemon to Report Metrics

Users have requested to add support for Prometheus metrics, see this GitHub issue. It is very useful if all currently detected problems can also be reported in metrics form:

- It allows various alerting systems to plug into the monitoring backend and create alerts.
- It allows people to use NPD to monitor Kubernetes master node as well.

And of course, we hope that the additional metrics support does not require users to change their existing config files (examples).

Our current proposal is to convert all problems into counters on currently supported problem daemons **by default**:

```
problem_counter{"name": "KernelDeadlock"}  0
problem_counter{"name": "DockerRestart"}  17
```

This can be done by extending the System Log Monitor and Custom Plugin Monitor a little bit. Today when problem daemons detect node problems, they will report a Status object to NPD core, which contains a list of detected problems. We just need to add a small logic in the problem daemons, to make them also report a number indicating how many new problems has happened. And the measurement will automatically be added up and reported.

And we plan to support some more customized behaviors using the config file. For example:

- Report the metric as gauge, rather than counter.
- Allow user to change the metric name.
- ...

This way, people can then build queries and create alerts based on these metrics, and reuse all their current NPD configurations. And when they need more customized metrics, they can change the config file to do that.

More Pluggable Problem Daemon

Today NPD only supports two types of problem daemon, and part of their initialization logic is hardcoded in NPD's main function.

In the future, since we are allowing NPD to do more data collection from various places (i.e. disk/CPU/memory/network/entropy), we expect more problem daemons to be added. Specific initialization for every problem daemon in main function won't be scalable.

We plan to allow problem daemon registration, and unify the initialization logic. Basically each problem daemon will have an initialization handler like this:

```
func NewFooMonitor(configPath string) types.Monitor
```

Problem daemons should register their initialization handler using the init() function, which will construct a map mapping problem daemon names to the initialization handlers:

```
map[string]func(string) types.Monitor
```

So NPD core does not need any logic about the problems daemons, it will just need to parse the NPD options to get the name and config file path of the problem daemons, and then use the initialization handlers to initialize them.

This flexible problem daemon registration is also useful if people want to maintain custom problem daemons that must be compiled together with NPD:

- The custom problem daemons may not be suitable for the OSS NPD project because of vendor/platform specific logic, private APIs, etc.
- The custom problem daemons may not fit with the custom plugin monitor. In which case it must be compiled into NPD.

With a flexible problem daemon registration, these users would have the option to first fork NPD and maintain a list of problem daemons in their repository, and then do periodic rebase on top of NPD master without significant merge conflicts.