- Contact Us
- Log in
  - Monitor
  - Secure
  - Support
-
  - 日本
  - United States
-

Search for: Search Search

sysdig_

Products Use Cases Open Source Resources Company Start Free Trial

- Products
  - Sysdig Platform
  - Sysdig Secure
  - Sysdig Monitor
  - Getting Started
  - Pricing
  - Start Free Trial
- Use Cases
  - Kubernetes Security
  - Runtime Security
  - Image Scanning
  - IR & Forensics
  - Compliance
  - Kubernetes Monitoring
  - Full Stack Monitoring
  - Troubleshooting
  - Prometheus Monitoring
  - Dashboards
- Open Source
  - Sysdig Contributions
  - Enterprise Falco
  - Prometheus
  - sysdig Inspect
- Environments
  - AWS
  - Google Cloud
  - IBM Cloud
  - Microsoft Azure
  - OpenShift
  - Federal Agencies
- Resources
  - Library
  - Webinars
  - Case Studies
  - Blogs
  - Newsletters
- Company
  - About Us
  - Leadership
  - Trust Center
  - Careers
  - Events
  - Webinars
  - Request a Demo
  - Contact Us

Updated Recommendations for You ^

- Newsroom
  - Newsroom
  - Press Releases
  - News Coverage
  - Media Resources

Search for: Search Search
Start 30-day Free Trial
Contact Sales

Follow us.

Privacy Policy

Updated Recommendations for You ^

Blog Post

How to write a custom Kubernetes scheduler using your monitoring metrics

By Mateo Burillo
on November 3, 2017

This article covers the use case of creating a custom Kubernetes scheduler and implements an example using monitoring

UPDATE: There is a new and more complete implementation of the custom Kubernetes scheduler using Golang.

The default Kubernetes scheduler does a fantastic job for most typical workloads. Starting from Kubernetes 1.6 advanced scheduling features like node or pod affinity, taints and tolerations allows you to configure several pod scheduling policies: in a specific set of nodes (node affinity/anti-affinity), close or far away from other running pods (pod affinity/anti-affinity), or just based on some tags that pods like or dislike (taints and tolerations).

But maybe you have some more specific requirements or would like to use higher level and dynamic application information to map your new pods to the physical nodes. Always striving for extensibility and flexibility, Kubernetes 1.6 introduced multiple scheduler/custom scheduler support as a beta feature.

What if you could use any of the metrics already present in your Kubernetes monitoring system to configure the behaviour of your pod scheduler?

How to write a custom #Kubernetes scheduler using your monitoring metrics

Click to tweet

The following is an example of a custom scheduler using metrics from our Kubernetes monitoring tool: Sysdig Monitor. In Sysdig, all metrics are automatically tagged with Kubernetes metadata, so you can easily do advanced monitoring, alerting, troubleshooting and now, advanced scheduling too.

Coding this scheduler may be a lot simpler that you may imagine. Let's start with a simple example to give you some context and ideas. Say for example that you want to optimize the responsiveness that your users perceive, so you decide that you want to place new web server pods in the physical host that is scoring the best **HTTP response times** at that specific point in time.

Normally, as a prerequisite you would have to instrument your application, but Sysdig collects requests, errors and response times metrics for any application or service without any kind of code instrumentation. But if you wanted to write the scheduler based on the behavior of an internal application metric, Sysdig will get any statsd, JMX or Prometheus metrics for you automagically, awesome! Isn't it?

Configure your pods to use a custom Kubernetes scheduler

First, you need to configure your pods to use a custom scheduler:

```yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    app: nginx
  template:
    metadata:
      name: nginx
      labels:
        app: nginx
    spec:
      schedulerName: sysdigsched
      containers:
      - name: nginx
        image: nginx
        ports:
        - name: http
          containerPort: 80
```

nginx_scheduler.yaml hosted with ❤ by GitHub view raw

This is a very simple vanilla Nginx replicationController. Note that we added `schedulerName: sysdigsched` to the pod definition. Remember that this is a Kubernetes 1.6+ feature, so this config parameter will throw an error when using older versions.

If you push this replicationController to the cluster:

```
$ kubectl create -f nginxrc.yaml
replicationcontroller "nginx" created
```

```
NAME                    READY     STATUS    RESTARTS   AGE
nginx-84cnn             0/1       Pending   0          11s
nginx-ff1dk             0/1       Pending   0          11s
nginx-jq5jk             0/1       Pending   0          11s
```

The pods will never leave the `Pending` state. They require a pod scheduler that doesn't yet exist.

Write your own Kubernetes scheduler!

We are going to use Python for this example. First, you will need to install both Kubernetes and Sysdig python libraries:

```
pip install kubernetes sdcclient
```

Before jumping to the complete example, let's look at the (relevant sections of the) example code.

First, you need to import the Kubernetes and Sysdig API libraries and objects:

```
from kubernetes import client, config, watch
from sdcclient import SdcClient
```

Then, you can initialize the API objects. You will find the *Sysdig Monitor API Token* for your account in the *User Profile* section of your Sysdig Monitor configuration menu:

```
config.load_kube_config()
v1=client.CoreV1Api()
sdclient = SdcClient(<your_sysdig_token>)
sysdig_metric = "net.http.request.time"
metrics = [{ "id": sysdig_metric, "aggregations": { "time": "timeAvg", "group": "avg" } }]
</your_sysdig_token>
```

For this example, we are only going to use the `net.http.request.time` metric, but the `metrics` variable is actually an array, you can easily configure the metric you want to use from an external file or use several metrics to create your custom "node score" function.

Next, you define the scheduler name:

```
scheduler_name = "sysdigsched"
```

This is the name that will be registered on the Kubernetes API, it has to match the pod spec name.

And now, the main loop of the scheduler, it waits for a new event containing an object in `Pending` state and a spec that requires our `scheduler_name`.

```
   w = watch.Watch()
    for event in w.stream(v1.list_namespaced_pod, "default"):
        if event['object'].status.phase == "Pending" and event['object'].spec.scheduler_name == scheduler_name:
            try:
                print "Scheduling " + event['object'].metadata.name
                res = scheduler(event['object'].metadata.name, best_request_time(nodes_available()))
```

Then it calls the scheduler function, assigning this object (the pending pod) to the node with the current best request time.

How do we measure best request time? With a simple call to the Sysdig API:

```
hostfilter = "host.hostName = '%s'" % hostname
start = -60
end = 0
sampling = 60
metricdata = sdclient.get_data(metrics, start, end, sampling, filter = hostfilter)
```

You will query the metrics we declared before, for the last minute, 60 samples. Then it's just a matter of parsing the data and returning the best value.

A test run with some debugging output will produce an output similar to:

```
Scheduling nginxrc-f1k2z
Nodes available: ['kubeworker1', 'kuberworker2']
kubeworker1 (net.http.request.time): 61664.877
kubeworker2 (net.http.request.time): 60456.919
Best node: kubeworker2
```

Here you have the complete scheduler file:

```
#!/usr/bin/env python
```

```
import random
import json

from kubernetes import client, config, watch
from sdcclient import SdcClient

config.load_kube_config()
v1 = client.CoreV1Api()
sdclient = SdcClient(<Your Sysdig API token>)
sysdig_metric = "net.http.request.time"
metrics = [{ "id": sysdig_metric, "aggregations": { "time": "timeAvg", "group": "avg" } }]

scheduler_name = "sysdigsched"


def get_request_time(hostname):
    hostfilter = "host.hostName = '%s'" % hostname
    start = -60
    end = 0
    sampling = 60
    metricdata = sdclient.get_data(metrics, start, end, sampling, filter=hostfilter)
    request_time = float(metricdata[1].get('data')[0].get('d')[0])
    print hostname + " (" + sysdig_metric + "): " + str(request_time)
    return request_time


def best_request_time(nodes):
    if not nodes:
        return []
    node_times = [get_request_time(hostname) for hostname in nodes]
    best_node = nodes[node_times.index(min(node_times))]
    print "Best node: " + best_node
    return best_node


def nodes_available():
    ready_nodes = []
    for n in v1.list_node().items:
            for status in n.status.conditions:
                if status.status == "True" and status.type == "Ready":
                    ready_nodes.append(n.metadata.name)
    return ready_nodes


def scheduler(name, node, namespace="default"):
    body=client.V1Binding()
    target=client.V1ObjectReference()
    target.kind="Node"
    target.apiVersion="v1"
    target.name= node
    meta=client.V1ObjectMeta()
    meta.name=name
    body.target=target
    body.metadata=meta
    return v1.create_namespaced_binding(namespace, body)


def main():
    w = watch.Watch()
    for event in w.stream(v1.list_namespaced_pod, "default"):
        if event['object'].status.phase == "Pending" and event['object'].spec.scheduler_name == scheduler_name:
            try:
                print "Scheduling " + event['object'].metadata.name
                res = scheduler(event['object'].metadata.name, best_request_time(nodes_available()))
            except client.rest.ApiException as e:
                print json.loads(e.body)['message']


if __name__ == '__main__':
    main()
```

SysdigMonitorKubernetesScheduler.py hosted with ❤ by **GitHub** **view raw**

Running Kubernetes scheduler in developer mode and live testing

Let’s try the script manually first.

Copy the file to any of your Kubernetes nodes, before you run the script remember to:

- Install the required Python libraries
- Pass your API token to the `sdclient` object
- You may need to adjust the `available_nodes` method using your custom node labels & taints (i.e. remove `NoScheduled` tainted nodes `node-role.kubernetes.io/master:NoSchedule`

Once you have checked all the items in the list just run it:

```
# python scheduler.py
Nodes available: ['kubeworker1', 'kubeworker2']
kubeworker1 (net.http.request.time): 1202.997
kubeworker2 (net.http.request.time): 1267.912
best node: kubeworker1
Nodes available: ['kubeworker1', 'kubeworker2']
kubeworker1 (net.http.request.time): 1202.997
kubeworker2 (net.http.request.time): 1267.912
best node: kubeworker1
Nodes available: ['kubeworker1', 'kubeworker2']
kubeworker1 (net.http.request.time): 1202.997
kubeworker2 (net.http.request.time): 1267.912
best node: kubeworker1
```

OK, seems that `kuberworker1` had better HTTP response times, so the three `Pending` pods are now running there:

```
$ kubectl get pods -o wide
NAME                 READY     STATUS    RESTARTS   AGE       IP               NODE
nginx-84cnn          1/1       Running   0          1h        10.244.1.19      kubeworker1
nginx-ff1dk          1/1       Running   0          1h        10.244.1.18      kubeworker1
nginx-jq5jk          1/1       Running   0          1h        10.244.1.20      kubeworker1
```

You can use any HTTP load generator to dramatically increase the load in one of the nodes, for example httperf:

```
kubeworker1:~$ httperf --server 127.0.0.1 --port 32768 --uri / --num-conn 20000 --num-cal 100000 --rate 200 --timeout 5
```

Scaling the replicationController will automatically generate more pods to be allocated:

```
$ kubectl scale rc nginx --replicas=5
replicationcontroller "nginx" scaled
```

But if we look at the output form our script, `kuberworker2` has now much better responsiveness than `kuberworker1`, currently under `httperf` stress:

```
Nodes available: ['kubeworker1', 'kubeworker2']
kubeworker1 (net.http.request.time): 17241.543
kubeworker2 (net.http.request.time): 1176.621
best node: kubeworker2
Nodes available: ['kubeworker1', 'kubeworker2']
kubeworker1 (net.http.request.time): 17241.543
kubeworker2 (net.http.request.time): 1176.621
best node: kubeworker2
 Your scheduler is allocating new pods in the most responsive node, just as you wanted.
```

Deploy your Kubernetes scheduler as a pod

A better solution is to containerize and orchestrate your scheduler rather than executing a script manually the Kubernetes nodes.

You will need a couple of changes to the script, on line 10:

```
config.load_incluster_config() # instead of config.load_kube_config()
```

And then, on line 12, load the token from a file:

```
sdclient = SdcClient(open("/etc/sysdigtoken/token.txt","r").read().rstrip())
```

From example-kubernetes-scheduler Github repository you can download some template files that you can use as an starting point:

- `Dockerfile`: to build the scheduler container image
- `scheduler.py`: the modified script file to run as a Docker container in a pod
- `sysdig-account.yaml`: credentials for Kubernetes RBAC
- `scheduler.yaml`: replicationController to launch the scheduler pod (fill with your container image name)

```
$ kubectl create secret generic sysdig-token --from-file=./token.txt
$ kubectl create -f sysdig-account.yaml
$ kubectl create -f scheduler.yaml
$ kubectl get pods

pythonscheduler-js8gg   1/1       Running   2          2h
```

Your custom Kubernetes scheduler is ready to go!

Custom Kubernetes scheduler – Golang implementation

During KubeCon EU 2018, we presented a newer and more complete Golang version of the Python code above. You will find the source code and usage instructions here.

This implementation still cannot be considered production ready, however, it has some relevant improvements over the Python version: * Metrics cache & metrics reuse * Failover and failover recovery * Async event handling and scheduling

Further thoughts

This is a relatively simple PoC example, if you really plan to code your own production-level scheduler:

- Declare and properly manage all the possible exception conditions.
- An scheduler has to be fast, benchmark the time it takes to pick a node, average and outliers, maybe you want to use Sysdig Tracers for that?
- If your code returns an error or is taking too long, you can always code a fallback to the default Kubernetes scheduler, much better than having orphaned pending pods.

A few more use case examples for writing a custom Kubernetes scheduler, we are sure you can come up with your own:

- Schedule backup / storage related pods on nodes with low IO latency and plenty of free HD space.
- Avoid hosts with a high error rate (net error, http error, IO error, etc), you may also set an alarm in Sysdig Monitor.
- Avoid scheduling new pods in a host running container that have specific security incidents. Yes, Sysdig can also look at what your containers are doing from a security point of view with Sysdig Secure: container run-time security and forensics product.

We hope you found this example useful when diving deep in customizing your Kubernetes cluster behavior. For more deep dives and clear visibility on what your Kubernetes and your containers are doing, check out our Sysdig Monitor and Sysdig Secure products and start a free trial yourself.

Post navigation

Previous Post
Next Post

Stay up to date

Sign up to receive our newest.

Submit
*
Subscription: General Marketing:
□Also keep me informed of Sysdig news + updates

Related Posts

Monitoring Kubernetes (part 1)

Read more

Container Metadata – Understanding Metrics, Labels, & Tags

Read more

Prometheus metrics / OpenMetrics code instrumentation.

Updated Recommendations for You ^

Read more
Start 30-day Free Trial
Contact Sales

Follow us.

- Country
  - 日本
  - United States

Products

- Sysdig Platform
- Sysdig Secure
- Sysdig Monitor

Support

- Support Log-in
- Submit Ticket
- Sysdig Status

Company

- About Us
- Leadership
- Contact Us
- Sitemap

Privacy Policy

Updated Recommendations for You