Monitoring Multiple Kubernetes Clusters

Conor Nevin
Oct 31, 2018 · 4 min read

Here at THG we manage Kubernetes clusters for multiple teams. In order to effectively monitor these clusters, we use a single Prometheus instance in each of our datacenters.

Prometheus is an open source monitoring tool that Kubernetes supports out of the box, exposing metrics about cluster health and operations on endpoints in the Prometheus format. Prometheus also supports using the Kubernetes REST API as a source to discover additional metric targets that are running inside the cluster.

Cluster Authentication

First thing we need to do is create a service account that the Prometheus instance will use to authenticate with the cluster.

In order to configure what the service account can access, you'll need to setup a clusterrole and clusterrolebinding. Here is the clusterrole that gives Prometheus read access to each of the resources that we are interested in scraping.

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: prometheus
rules:
- apiGroups:
  - ""
  resources:
  - nodes
  - nodes/proxy
  - services
  - services/proxy
  - endpoints
  - pods
  - pods/proxy
  verbs:
```

```
    - get
    - list
    - watch
  - nonResourceURLs:
    - /metrics
    verbs:
    - get
```

prometheus-cr.yml hosted with ♡ by GitHub view raw

Once we create the clusterrole, we'll need to bind it to the service account by running `kubectl create clusterrolebinding prometheus-querier -clusterrole=prometheus-querier -serviceaccount=kube-system:prometheus`

Now that we've configured our cluster, we need to configure the Prometheus instance.

Prometheus Configuration

Prometheus has an example configuration for scraping Kubernetes; however, it's meant to be run from inside the cluster and assumes default values that won't work outside of the cluster.

Inside the cluster, this is all the configuration required to discover all of the nodes to scrape.

```
- job_name: 'kubernetes-nodes'
  scheme: https
  tls_config:
    ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
  bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token

  kubernetes_sd_configs:
  - role: node

  relabel_configs:
  - action: labelmap
    regex: __meta_kubernetes_node_label_(.+)
  - target_label: __address__
    replacement: kubernetes.default.svc:443
  - source_labels: [__meta_kubernetes_node_name]
    regex: (.+)
    target_label: __metrics_path__
    replacement: /api/v1/nodes/${1}/proxy/metrics
```

prometheus-node-cf-def.yml hosted with ♡ by GitHub view raw

In order to make this work outside of the cluster, we need to point towards the token associated with the service account we created earlier along with the CA (certificate authority) of the cluster and the address of the Kubernetes REST API.

```
    bearer_token_file: <path-to-token>/user_token
    tls_config:
      ca_file: <path-to-ca>/ca.crt

    scheme: https

    kubernetes_sd_configs:
    - api_server: https://<kubernetes-master-url>
      role: node
      bearer_token_file: <path-to-token>/user_token
      tls_config:
        ca_file: <path-to-ca>/ca.crt
```

node-sd-config.yml hosted with ♡ by GitHub view raw

This will allow the Prometheus instance to construct the list of targets that it needs to scrape, but we also need to add the bearer token and CA file to the overarching job so it is able to successfully scrape the metrics.

Now that we're able to make the requests to the cluster, we need to do some relabelling so the Prometheus instance is able to construct the correct external URL to reach the target on.

This relabel config loads every label against the respective node as a Prometheus label, rewrites the target address to the address of the API server and changes the metric path to use the proxy endpoint on the Kubernetes API.

```
    relabel_configs:
    - action: labelmap
      regex: __meta_kubernetes_node_label_(.+)
    - target_label: __address__
      replacement: <kubernetes-master-url>
    - source_labels: [__meta_kubernetes_node_name]
      regex: (.+)
      target_label: __metrics_path__
      replacement: /api/v1/nodes/${1}/proxy/metrics
```

node-relabel-conf.yml hosted with ♡ by GitHub view raw

At the same time, we also add a new static label to every metric that identifies the cluster, so we can easily distinguish between metrics belonging to different clusters. The final configuration looks like this

```
- job_name: 'k8s-nodes'

  bearer_token_file: <path-to-token>/user_token
  tls_config:
    ca_file: <path-to-ca>/ca.crt

  scheme: https

  kubernetes_sd_configs:
  - api_server: https://<kubernetes-master-url>
    role: node
    bearer_token_file: <path-to-token>/user_token
    tls_config:
      ca_file: <path-to-ca>/ca.crt

  relabel_configs:
  - action: labelmap
    regex: __meta_kubernetes_node_label_(.+)
  - target_label: __address__
    replacement: <kubernetes-master-url>
  - source_labels: [__meta_kubernetes_node_name]
    regex: (.+)
    target_label: __metrics_path__
    replacement: /api/v1/nodes/${1}/proxy/metrics
```

prometheus-node-cf.yml hosted with ♡ by GitHub view raw

With this configuration in place, we still need to alter the config for scraping the other targets, node-cadvisor, pods and services. For scraping the cadvisor metrics, all we need to do is duplicate the above config and change the metrics path to `/api/v1/nodes/${1}/proxy/metrics/cadvisor` . For pods we need to use the following relabel config:

```
relabel_configs:
  - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_scrape]
    action: keep
    regex: true
  - target_label: __address__
    replacement: <kubernetes-master-url>
  - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_scheme]
    regex: ^$
```

```
      replacement: http
      target_label: __meta_kubernetes_pod_annotation_prometheus_io_scheme
    - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_port]
      regex: ^$
      replacement: "8080"
      target_label: __meta_kubernetes_pod_annotation_prometheus_io_port
    - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_path]
      regex: (.+)
      replacement: ${1}
      target_label: __metrics_path__
    - source_labels:
      - __meta_kubernetes_namespace
      - __meta_kubernetes_pod_annotation_prometheus_io_scheme
      - __meta_kubernetes_pod_name
      - __meta_kubernetes_pod_annotation_prometheus_io_port
      - __metrics_path__
      regex: (.+);(.+);(.+);(.+);(.+)
      action: replace
      target_label: __metrics_path__
      replacement: /api/v1/namespaces/${1}/pods/${2}:${3}:${4}/proxy${5}
    - action: labelmap
      regex: __meta_kubernetes_pod_label_(.+)
    - source_labels: [__meta_kubernetes_namespace]
      action: replace
      target_label: kubernetes_namespace
    - source_labels: [__meta_kubernetes_pod_name]
      action: replace
      target_label: kubernetes_pod_name
    - source_labels: [__meta_kubernetes_pod_node_name]
      target_label: kubernetes_node_name
    - source_labels: [__meta_kubernetes_pod_name]
      target_label: instance
```

pod-relabel-config.yml hosted with ♡ by GitHub view raw

This configuration will filter the list of all running pods and only scrape those with prometheus.io/scrape=true set as an annotation and then constructs the scrapable address using additional annotations that allow us to configure the port and path of the metrics endpoint.

```
  relabel_configs:
    - source_labels: [__meta_kubernetes_service_annotation_prometheus_io_scrape]
      action: keep
      regex: true
    - target_label: __address__
      replacement: <kubernetes-master-url>
```

```
      - source_labels: [__meta_kubernetes_service_annotation_prometheus_io_scheme]
        regex: ^$
        replacement: http
        target_label: __meta_kubernetes_service_annotation_prometheus_io_scheme
      - source_labels: [__meta_kubernetes_service_annotation_prometheus_io_port]
        regex: ^$
        replacement: "8080"
        target_label: __meta_kubernetes_service_annotation_prometheus_io_port
      - source_labels: [__meta_kubernetes_service_annotation_prometheus_io_path]
        regex: (.+)
        replacement: ${1}
        target_label: __metrics_path__
      - source_labels:
        - __meta_kubernetes_namespace
        - __meta_kubernetes_service_annotation_prometheus_io_scheme
        - __meta_kubernetes_service_name
        - __meta_kubernetes_service_annotation_prometheus_io_port
        - __metrics_path__
        regex: (.+);(.+);(.+);(.+);(.+)
        action: replace
        target_label: __metrics_path__
        replacement: /api/v1/namespaces/${1}/services/${2}:${3}:${4}/proxy${5}
      - action: labelmap
        regex: __meta_kubernetes_service_label_(.+)
      - source_labels: [__meta_kubernetes_namespace]
        action: replace
        target_label: kubernetes_namespace
      - source_labels: [__meta_kubernetes_service_name]
        action: replace
        target_label: kubernetes_service_name
      - source_labels: [__meta_kubernetes_service_name]
        target_label: instance
```

service-relabel-config.yml hosted with ♡ by GitHub view raw

The service configuration is almost identical but constructs a slightly different address using the service annotations.

Configuration Generation/Management

Now that we have all of this in place, we need a way of automatically generating this config for each cluster that we want to scrape, since writing this manually would take far to long. Prometheus doesn't support loading configuration from a directory. It, instead, requires that it is all present in a single file so lets use Ansible to generate the file for us.

Ansible has a module that allows multiple files to be assembled into a larger single file called “assemble” which will make supporting multiple different scrape types much easier. Handily, it also supports a validation step that we can use to verify that our configuration is correct before we overwrite our previous one. By rewriting our configuration changes above into templates we can output one per cluster into a directory and then combine them into a single file.

```
- name: Assemble prometheus config
  assemble:
    src: "{{ prometheus_config_path }}/fragments"
    dest: "{{ prometheus_config_path }}/prometheus.yaml"
    validate: "{{ prometheus_bin_path }}/promtool_{{ prometheus_version }} check config %s"
    backup: yes
  notify:
  - Restart prometheus
```

ansible-assemble.yml hosted with ♡ by GitHub view raw

We now have a functional Prometheus instance, and you should now see a list of targets from the cluster being scrapped automatically.

As part of our cluster setup we install two components into the cluster that give us some additional metrics:

- Kube State Metrics which exposes metrics about the internal state of the various resources inside the cluster
- Node Exporter which exposes basic machine level metrics from each host in the cluster

Now that this is all in place, every time we spin up a new cluster all we need to do is regenerate our Prometheus configuration, and we automatically scrape all the metrics from our new cluster!

Kubernetes Cluster Overview

We're recruiting

Find out about the exciting opportunities at THG here:

https://www.thg.com/careers/

Kubernetes