HPA and METRIC SERVER

HPA

The Horizontal Pod Auto scaler automatically scales the number of pods in a replication controller, deployment, replica set based on observed CPU utilization or memory utilization.

The Horizontal Pod Auto scaler is implemented as a Kubernetes API resource and a controller. The resource determines the behavior of the controller. The controller periodically adjusts the number of replicas in a replication controller or deployment to match the observed average CPU/memory utilization to the target specified by user.

HPA will interact with Metric Server to identify CPU/Memory Utilization of POD.

Kubernetes Metric Server

Metric Server is an application that collects metrics from objects such as pods, nodes according to the state of CPU, RAM and keeps them in time.

Metric-Server can be installed in the system as an addon. You can take and install it directly from the repo.

===Setup Metrics Server===

Git Clone below repo in Kubernetes Master or in kubectl client Machine.

```
$ git clone https://github.com/MithunTechnologiesDevOps/metrics-server.git

$ cd metrics-server

$ kubectl apply -f deploy/1.8+/
```

When the Metric server is installed directly, it is installed under the kube-system namespace

Usage

```
# Display node metrics
$ kubectl top nodes

# Display pod metrics
$ kubectl top pods
```

Refer below link for more details :

https://github.com/kubernetes-sigs/metrics-server

We completed the first part in the article under the first phase, now let's create a simple deployment object and the HPA structure attached to it and observe its behaviour.

Get below manifest files from

https://github.com/MithunTechnologiesDevOps/Kubernates-Manifests/blob/master/horizontal-pod-autoscaler.yml

```
# Deployment
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hpadeployment
  labels:
    name: hpadeployment
spec:
  replicas: 2
  selector:
    matchLabels:
      name: hpapod
  template:
    metadata:
      labels:
        name: hpapod
    spec:
      containers:
        - name: hpacontainer
          image: k8s.gcr.io/hpa-example
          ports:
          - name: http
            containerPort: 80
          resources:
            requests:
              cpu: "100m"
              memory: "64Mi"
            limits:
              cpu: "100m"
              memory: "256Mi"
```

HPA For above deployment(hpadeployment)

```
---
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: hpadeploymentautoscaler
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: hpadeployment
  minReplicas: 2
  maxReplicas: 5
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 40
  - type: Resource
    resource:
     name: memory
     target:
      type: Utilization
      averageUtilization: 40
```

Service

```
---
apiVersion: v1
kind: Service
metadata:
  name: hpaclusterservice
  labels:
    name: hpaservice
spec:
  ports:
    - port: 80
      targetPort: 80
  selector:
    name: hpapod
  type: ClusterIP
```

Deploy all above three kubernetes objects.

==== Execute below commands to increase load====

Now, we will see how the auto scaler reacts to increased load. We will start a container, and send an infinite loop of queries to the php-apache service .

Create A Temp Pod in interactive mod to access app using service name

```
  $ kubectl run -i --tty load-generator --rm  --image=busybox /bin/sh
```

Execute below command in Temp Pod

```
$ while true; do wget -q -O- http://hpaclusterservice; done
```

Open kubectl terminal in another tab and watch kubectl get pods or kubect get hpa to see how the auto scaler reacts to increased load.

```
$ watch kubectl get hpa
```

Requests and Limits

Requests and limits are the mechanisms Kubernetes uses to control resources such as CPU and memory.

Requests are what the container is guaranteed to get. If a container requests a resource, Kubernetes will only schedule it on a node that can give it that resource.

Limits, on the other hand, make sure a container never goes above a certain value. The container is only allowed to go up to the limit, and then it is restricted.

It is important to remember that the limit can never be lower than the request. If you try this, Kubernetes will throw an error and won't let you run the container.

Requests and limits are on a per-container basis. While Pods usually contain a single container, it's common to see Pods with multiple containers as well. Each container in the Pod gets its own individual limit and request, but because Pods are always scheduled as a group, you need to add the limits and requests for each container together to get an aggregate value for the Pod.

Resource Configuration Values

Memory is measured in bytes and expressed as an integer or using a fixed point integer. For example; memory: 1 is 1 byte, memory: 1Mi is 1 mebibyte / megabyte, memory: 1Gi is 1 gibibyte / gigabyte. Memory is not a compressible resource, and there is no throttling. Kubernetes evicts a pod from a node If a node cannot provide sufficient memory.

Note: One thing to keep in mind about CPU requests is that if you put in a value larger than the core count of your biggest node, your pod will

never be scheduled. Let's say you have a pod that needs four cores, but your Kubernetes cluster is comprised of dual core VMs—your pod will never be scheduled.

CPU is measure in millicpus, which is 1/1000th of a CPU core and expressed with integers. For example; cpu: "1" is 1 CPU, cpu: 2000m is 2 CPUs, cpu: "0.75" is 0.75 of a cpu and equivalent to cpu: 750m. CPU is a compressible resource and can be throttled by Kubernetes during contention. CPU contention does not cause pod eviction; instead, the app is slowed down.

Note: If you put in a memory request that is larger than the amount of memory on your nodes, the pod will never be scheduled.