CSC4160 Assignment 2

Deadline: October 9, 2024, 23:59

Name:

Student ID:

DockerCoins

- DockerCoins is made of 5 services
  - `rng` = web service generating random bytes
  - `hasher` = web service computing hash of POSTed data
  - `worker` = background process calling `rng` and `hasher`
  - `webui` = web interface to watch progress
  - `redis` = data store (holds a counter updated by `worker`)

In this assignment, you will deploy an application called `DockerCoins` which generates a few random bytes, hashes these bytes, increments a counter (to keep track of speed) and repeats forever! You will try to find its bottlenecks when scaling and use HPA (Horizontal Pod Autoscaler) to scale the application. Please follow these instructions carefully!

Environment Setup

The assignment needs to be setup in `AWS`. You should choose `Ubuntu Server 24.04 LTS (HVM), SSD Volume Type` as AMI (Amazon Machine Image) and `m4.large` as the instance type. And you need to configure security group as followed, to make sure that you can access the service of minikube on the web browser later.

Type	Protocol	Port Range	Source	Description
All traffic	All	All	0.0.0.0/0	

Run the following commands to satisfy the requirements.

```
# Install Minikube
$ curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
$ sudo install minikube-linux-amd64 /usr/local/bin/minikube

# Install kubectl
$ curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
$ sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
```

```
# Install Docker
$ sudo apt-get update && sudo apt-get install docker.io -y

# Install conntrack
$ sudo apt-get install -y conntrack

# Install httping
$ sudo apt-get install httping

# Install Helm
$ curl -fsSL -o get_helm.sh
https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3
$ chmod 700 get_helm.sh
$ ./get_helm.sh
```

Running Minikube on EC2 Ubuntu

Add user to “docker” group and switch primary group

```
$ sudo usermod -aG docker $USER && newgrp docker
```

Start Minikube

```
$ minikube start
```

Start the application

You can `git clone` the assignment repo and `cd` into this directory.

- Start the application from the `dockercoins` yaml

```
$ kubectl apply -f dockercoins.yaml
```

- Wait until all the components are running

```
$ kubectl get po
# NAME                      READY   STATUS    RESTARTS   AGE
# hasher-89f59d444-r7v7j    1/1     Running   0          27s
# redis-85d47694f4-m8vnt    1/1     Running   0          27s
# rng-778c878fb8-ch7c2      1/1     Running   0          27s
# webui-7dfbbf5985-24z79    1/1     Running   0          27s
# worker-949969887-rkn48    1/1     Running   0          27s
```

- Check the results from UI

```
$ minikube service webui
# |-----------|-------|-------------|-----------------------------|
# | NAMESPACE | NAME  | TARGET PORT |            URL              |
# |-----------|-------|-------------|-----------------------------|
# | default   | webui |          80 | http://172.31.67.128:30163 |
# |-----------|-------|-------------|-----------------------------|
# 🎉  Opening service default/webui in default browser...
# 👉  http://172.31.67.128:30163
```

- Port forwarding for WebUI

You need to forward connections to a local port to a port on the pod.

```
$ kubectl port-forward --address 0.0.0.0 <webui pod name> <local port>:<pod port>
```

Local port is any number. Pod Port is target port (e.g., 80).

You can access the DockerCoin Miner WebUI on a web browser. The address is <Public IPv4 address>:<local port> (e.g., `3.238.254.199`:`30163`, where `3.238.254.199` is Public IPv4 address of the instance).

Note: kubectl port-forward does not return. To continue with the exercises, you will need to open another terminal.

Bottleneck detection

Workers

Scale the # of workers from 2 to 10 (change the number of `replicas`).

```
$ kubectl scale deployment worker --replicas=3
```

# of workers	**1**	**2**	**3**	**4**	**5**	**10**
hashes/second						

> **Question**: What is the speedup bump when you have 10x workers?
>
> Answer: <You answer goes here>.

Rng / Hasher

After we scaled to 10 workers, the speed bump is not 10x! Something is slowing us down... But what?

Let's keep the number of workers as `10` and use the good old `httping` command for latency detection. Possible bottlenecks for now are `rng` and `hasher`.

```
# Expose the service, since we are detecting the latency outside the k8s cluster
$ kubectl expose service rng --type=NodePort --target-port=80 --name=rng-np
$ kubectl expose service hasher --type=NodePort --target-port=80 --name=hasher-np

# Get the url of the service
$ kubectl get svc rng-np hasher-np

# Find the minikube address
$ kubectl cluster-info
# Kubernetes control plane is running at https://172.31.67.128:8443
# KubeDNS is running at https://172.31.67.128:8443/api/v1/namespaces/kube-
system/services/kube-dns:dns/proxy
# Here, minikube address is 172.31.67.128

# Detect the latency of hasher
$ httping <minikube-address>:<hasher-np-port>

# Detect the latency of rng
$ httping <minikube-address>:<rng-np-port>
```

Service	Hasher	Rng
Latency (ms)		

> **Question**: Which service is the bottleneck?
>
> Answer: <You answer goes here>.

Application monitoring

Collecting metrics with Prometheus

Prometheus is an open-source tool to monitor and collect metrics from applications, allowing users to see and understand important metrics that let them know how well an application is doing.

We are going to deploy Prometheus on our Kubernetes cluster and see how to query it.

```
# Install prometheus
$ helm repo add prometheus-community https://prometheus-community.github.io/helm-
charts
$ helm install prometheus prometheus-community/prometheus

# Expose service port
$ kubectl expose service prometheus-server --type=NodePort --target-port=9090 --
name=prometheus-server-np
```

Befor proceeding, our metric-server add on is disabled by default. Check it using:

```
$ minikube addons list
```

If its disabled, you will see something like metrics-server: disabled Enable it using:

```
$ minikube addons enable metrics-server
```

We want to collect latency metric via `httplat`, a minimalist Prometheus exporter that exposes the latency of a single HTTP target. (Please replace `<FILL IN>` in the `httplat` yaml as which you need).

```
$ kubectl apply -f httplat.yaml
$ kubectl expose deployment httplat --port=9080

# Check if the deployment is ready
$ kubectl get deploy httplat
# NAME      READY   UP-TO-DATE   AVAILABLE   AGE
# httplat   1/1     1            1           43s
```

Configure Prometheus to gather the detected latency.

```
$ kubectl annotate service httplat \
          prometheus.io/scrape=true \
          prometheus.io/port=9080 \
          prometheus.io/path=/metrics
```

Connect to Prometheus

```
$ kubectl get svc prometheus-server-np
# NAME                    TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)
AGE
# prometheus-server-np   NodePort   10.96.122.34   <none>        80:24057/TCP
87s
```

Port forwarding for Prometheus UI

You need to forward connections to a local port to a port on the pod.

```
$ kubectl port-forward --address 0.0.0.0 <prometheus server pod name> <local
port>:9090
or
```

```
$ kubectl port-forward --address 0.0.0.0 service/prometheus-server-np <local
port>:80
```

Note: Why the difference?

To retrieve the Prometheus server pod name, type the following command into your terminal:

```
$ kubectl get po
```

The pod name you're looking for begins with `prometheus-server-`, such as `prometheus-server-8444b5b7f7-hk6g7`.

Access Prometheus

You can access the Prometheus on a web browser. The address is <Public IPv4 address>:<local port>. Check if `httplab` metrics are available. You can try to graph the following PromQL expression that detects the latency of rng/hasher (as you specified in `httplat.yaml`).

```
rate(httplat_latency_seconds_sum[2m])/rate(httplat_latency_seconds_count[2m])
```

HPA

To solve the bottleneck of the application, you can specify a horizontal pod autoscaler which can scale the number of Pods based on some metrics.

Generating a load test

This is how you generate a load test. You do not need to generate one until further instruction in the next section.

Create a Load Test Job as,

```
file: loadtest-job.yaml
```

```
apiVersion: batch/v1
kind: Job
metadata:
  generateName: loadtest
spec:
  template:
    spec:
      containers:
      - name: siege
        image: schoolofdevops/loadtest:v1
```

```
        command: ["siege",  "--concurrent=15", "--benchmark", "--time=4m",
"http://<FILL IN>"] # FILL IN: rng or hasher
      restartPolicy: Never
  backoffLimit: 4
```

and launch it as

```
kubectl create -f loadtest-job.yaml
```

This will launch a one off Job which would run for 4 minutes.

To get information about the job

```
kubectl get jobs
kubectl describe  job loadtest-xxx
```

replace loadtest-xxx with actual job name.

To check the load test output

```
kubectl logs  -f loadtest-xxxx
```

[replace loadtest-xxxx with the actual pod id.]

[Sample Output]

```
** SIEGE 3.0.8
** Preparing 15 concurrent users for battle.
The server is now under siege...
Lifting the server siege...      done.

Transactions:                  47246 hits
Availability:                 100.00 %
Elapsed time:                 239.82 secs
Data transferred:               1.62 MB
Response time:                  0.08 secs
Transaction rate:             197.01 trans/sec
Throughput:                     0.01 MB/sec
Concurrency:                   14.94
Successful transactions:       47246
Failed transactions:               0
Longest transaction:            0.61
Shortest transaction:           0.00

FILE: /var/log/siege.log
```

```
You can disable this annoying message by editing
the .siegerc file in your home directory; change
the directive 'show-logfile' to false.
```

Create the autoscaling policy.

Please replace `<FILL IN>` in the `hpa` yaml as which you need. Now we are ready to see our HPA in action.

1. Keep monitoring the latency of the rng/hasher (bottleneck component) in Prometheus UI.
2. Generate load test before we apply our HPA.
3. Apply HPA (before our load test job completes):

Type the following command into your terminal:

```
$ kubectl apply -f hpa.yaml
```

The horizontal pod autoscaler in `hpa.yaml` is designed to automatically adjust the number of replicas for a particular deployment (that you need to `<FILL IN>`, either rng or hasher) based on a sample autoscaling policy.

Let's see the details of our hpa. If you still see `<unknown> / 5%`, try to wait for a while.

```
$ kubectl describe hpa <FILL IN> # FILL IN: rng or hasher
```

4. Continuously watch our HPA (for rng or hasher) to see real-time updates of our HPA in action.

```
$ kubectl get hpa <FILL IN> -w # FILL IN: rng or hasher
```

[Sample Output]

```
NAME   REFERENCE        TARGETS      MINPODS   MAXPODS   REPLICAS   AGE
rng    Deployment/rng   cpu: 7%/5%   1         10        6          12m
...
```

> **Screenshot**:
>
> - The Prometheus UI that queries the latency of rng/hasher.
> - The the terminal when we watched HPA in action as it scales out/in the rng/hasher with the increasing/decreasing load.

> **Open question**:

> - Do you think scaling the rng/hasher based on CPU utilization makes sense? If not, what alternative metric could we use instead? Please explain your reasoning.
>
> Answer: <Your answer goes here>.

Grading

In this assignment, you are required to finish the following things:

- Report the performance of DockerCoins with different number of workers (2 marks).
- Detect latency and find the bottleneck component (rng or hasher) (2 marks).
- Upload the screenshot of the HPA in action (2 marks):
  - Prometheus UI
  - Terminal
- Open question (1 mark):
  - A better metric?