Kubernetes Autoscaling

Balázs Fodor, Felicián Németh, István Pelle, Balázs Sonkoly

2024-11-06

Start the cluster

- Open AWS Academy login page: https://awsacademy.instructure.com/
- Log in
- Start the AWS Academy Learner Lab and open the AWS Management console
- Click on this (CloudFormation) link: https://us-east-1.console.aws.amazon.com/cloudformation/home?region=us-east-1#/stacks/create/review?templateURL=https://vitmac12-resources.s3.amazonaws.com/k3s-multinode.template&stackName=k3s-multinode
- Fill out the parameter field named "`ONeptunCode`"
- If you connect to the internet via any network (e.g., cellular) other than BME's (wired/WiFi, eduroam):
  - Select `0.0.0.0/0` in the dropdown list named `SecurityGroupIngressCidrIp` among the parameters
- At the bottom, accept the three checkboxes, and press `Create`

Access the cluster

- From the outputs tab of the CloudFormation stack
  open the link next to the 0K3sServerSsh key in a new browser tab
- Using EC2 Instance Connect, connect to your Kubernetes cluster
- You can log in even before the cluster is ready, so
  - Check the readiness with `kubectl get pods -A`
  - You should see 5 Pods in Running and 2 Pods in Completed status

Agenda

- App setup + load test
- Debug with Prometheus
- Manual Vertical scaling
- Manual Horizontal scaling
- Grafana setup
- HPA configuration and testing

Install helm

- Helm is a package manager for Kubernetes, helping to manage Kubernetes applications.
- It uses charts, which are packages of pre-configured Kubernetes resources.
- Helm simplifies deployment and management of applications on Kubernetes clusters.
- It allows for easy updates and rollbacks of applications.
- Helm charts can be shared and reused, promoting best practices and consistency.

```
cd edu-cloud-native/k8s-autoscaling
./helm.sh
```

Deploy and observe our test application

Run the following commands the deploy our test application

```
kubectl apply -f deployments/webserver/deployment.yaml
kubectl apply -f deployments/webserver/service.yaml
```

- Discover what Kubernetes objects have been created!

Deploy and observe our test application

Run the following commands the deploy our test application

```
kubectl apply -f deployments/webserver/deployment.yaml
kubectl apply -f deployments/webserver/service.yaml
```

- Discover what Kubernetes objects have been created!
- A service named demo-webserver
- A deployment named demo webserver
  - the deployment created a Pod

Load testing I.

- We want to test the performance of our application
  - We use fortio, which is a load testing tool
- The following command can be used send requests to our test application:

```
kubectl run -i --tty load-generator --rm --image=docker.io/fortio/fortio --restart=Never -- \
load -allow-initial-errors -qps 10 -t 30s -connection-reuse 0:0 -c 4 http://demo-webserver:8080
```

- What does this command do?
  - It uses kubectl to start a Pod name load-generator
  - Which is going to be deleted after the given command returns (–rm)
  - We use the fortio docker image, with the fortio load testing command (second line)
    - the command sends requests at 10 requests per second rate, using 4 connections for 30 seconds
  - Note how we use the domain name of the service to access our webserver
- You will see the following message: "If you don't see a command prompt, try pressing enter." while the load test is running, please ignore it, just wait for the command to be finished
- Observe the response of the command!
  - what is the actual QPS (query per second) our server can serve?
  - what is the average response time?
- Change the -qps parameter to 0 and run the command again!
  - What does this option do?
  - Determines the maximum QPS our application can serve
  - It is not that high (around 60-80 QPS)! Let's try to debug (next slide)

Let's debug the problem with Prometheus

- We install a prometheus monitoring service, using helm:

```
# We need this command for helm to be able to access the config of our Kubernetes cluster
export KUBECONFIG=/etc/rancher/k3s/k3s.yaml
# We create a namespace called monitoring for prometheus (and grafana)
kubectl create namespace monitoring
# Deploy prometheus
cd deployments/prometheus
chmod +x deploy.sh
./deploy.sh
cd ../..
```

- You can access the Prometheus UI using the public IP of your EC2 instance
  - use kubectl to find the port number the prometheus service is exposed to
    - look for the prometheus-server service in the monitoring namespace

Let's debug the problem with Prometheus

- Open the prometheus UI and in the Expression input field, put the following query:

```
(sum(rate(container_cpu_usage_seconds_total{namespace="default", pod=~"demo-webserver.*", container!=""}[1m])) by
↪ (pod))*1000
```

- this Prometheus query displays the CPU usage of our Pods in millicore (1000 millicore = 1 CPU core)
- Choose the Graph tab in the UI to display the data
- Execute the previous load test again (with `-qps 0`), and follow the CPU usage in the Prometheus UI
  - This time set the `-t` parameter to 120s, to test for two minutes
- What do you see in the Prometheus UI?
  - Our Pod's CPU usage can not surpass 150 millicores
- Check the deployment's configuration!
  - Do you see something that can cause this?
  - In the resources section, under limits, there is a CPU limitation configured to 150 millicores
  - What options do we have if we want to be able to serve more requests?
    - Vertical and horizontal scaling

Vertical scaling

- The observed QPS is not good enough for us, so let's try to increase the resources assigned to our service!
- Let's change the configured 150 millicores CPU limit to 300 millicores
- Use the `EDITOR=nano kubectl edit deployment <deployment_name>` command to modify the configuration of the deployment
  - What parameters should be set?
  - Find the resources section and set the CPU limit to 300
- Execute the load test again!
  - What changes?
  - The served QPS value became twice as big
  - Observe that the max CPU usage in the Prometheus UI is 300
- Change back the CPU limit to 150

Horizontal scaling

- Let's scale our service horizontally
- We still want to have 300 millicores instead of 150, but this time, we achieve this by making a replica of our Pod

```
kubectl scale deployment <deployment_name> --replicas=2
```

- use `kubectl get pods` to verify the number of Pods
- when both Pods are ready, execute the load test again!
- What changes?
  - We had a similar performance as in the case of vertical scaling
  - Observe that the max CPU usage in the Prometheus UI is 300
- Change back the number of replicas to 1

```
kubectl scale deployment <deployment_name> --replicas=1
```

Grafana

- Grafana is an open-source platform for monitoring and observability.
- It provides a powerful and flexible dashboard for visualizing metrics from various data sources.
- In Kubernetes, Grafana is often used in conjunction with Prometheus to monitor cluster and application performance.
- Grafana can display metrics such as CPU usage, memory usage, and network traffic in real-time.
- It supports alerting, allowing users to set up notifications based on specific conditions.
- Grafana's dashboards can be customized and shared, making it easier to collaborate and maintain visibility into system health.

Deploying grafana

- We will use grafana for a better monitoring experience

```
cd deployments/grafana
chmod +x deploy.sh
./deploy.sh
cd ../..
```

- If the Grafana install fails with `Kubernetes cluster unreachable... connection refused` (excerpt)
  - Rerun `export KUBECONFIG=/etc/rancher/k3s/k3s.yaml`
- You can access the Grafana UI using the public IP of your EC2 instance
  - use kubectl to find the port number the grafana service is exposed to
    - look for the grafana service in the monitoring namespace
- Open the grafana UI in your web browser
  - you can log in using the username "admin"
  - the password is displayed as the last log of the deploy.sh script
- Go to the Connections → Add new connection option in the left sidebar
- Look for Prometheus
- Select it and then choose "Add new data source"
- In the Connection section, set the "Prometheus server url" to "`http://prometheus-server.monitoring`"
- At the bottom, click "save and test"

Add grafana dashboard

- You can find a grafana dashboard JSON config file here: https://raw.githubusercontent.com/hsnlab/edu-cloud-native/refs/heads/main/k8s-autoscaling/deployments/grafana/service-dashboard.json
- Copy the content
- In your Grafana, go to Dashboards, select "New", and choose "Import"
- Copy the content of the JSON file into the input field, and hit "Load"
- As a result, a preconfigured Grafana dashboard is created
- Examine the different visualizations

Configure HPA

- Observe the content of the deployments/webserver/hpa.yaml file
- What are the configurations we are using?
  - What is the requested amount of CPU?
  - Around what CPU usage (in millicores) will HPA initiate scaling?
    - $150 \cdot 0.5 = 75$ millicores
- Deploy your HPA config using

```
kubectl apply -f deployments/webserver/hpa.yaml

# wait a 1-2 minutes
# get the details of you HPA using these commands:
kubectl get hpa
kubectl describe hpa
```

Dynamically scale the application

- Open the following file: `deployments/fortio/qps_test.sh`
- Observe the load pattern we plan to send!
  - it is a 10 minute long measurement

```
chmod +x deployments/fortio/qps_test.sh
./deployments/fortio/qps_test.sh
```

- Follow what happens with your application in the Grafana dashboard
  - you should see how HPA scales your application based on the CPU usage

Dynamically scale the application II

- If you have 10 more minutes!
- It is maybe an overkill to set the CPU threshold to 50%
  - this means that the average CPU usage of the Pods is 50%
  - maybe we can use the resources more efficently
- Try changing it to 70% in the HPA configuration, and execute the measurement script again
- Are there any differences compared to the 50% case?
  - check in the Grafana dashboard when the instances started
  - how the replica counts changed
  - how the served QPS changed
  - e.g. overall, we used less resources in the 70% case