Auto Scaling a Kubernetes Deployment using Metrics from a Thunder ADC

By John D. Allen
Global Solutions Architect – Cloud, IoT, & Automation
April, 2022

Table of Contents

Introduction

When using a Thunder ADC as a Cloud Application Endpoint-of-Control, we have the opportunity to collect metrics from the Thunder ADC for a SLB Virtual Server that is controlling access to that Cloud Application. Since all the traffic for that Cloud Application is coming through the Thunder ADC, we have the ability to scale that Cloud Application based on the Throughput rate going to it.

This document will cover a Proof-of-Concept Thunder Cloud Agent (TCA) that is used to "Auto-Scale" the number of Pods running in a Cloud Application up or down to adjust for traffic Throughput. This TCA runs as a Container inside the target Kubernetes Cluster where the Deployment is located. A Rate-per-Pod is configured (in Kbps) and the Thunder ADC is checked on a regular interval. If the current rate is such that there should be more Cloud Application Pods running, the TCA will make an API call to the Kubernetes Cluster to increase the number of Pods running. If the current rate is below the current number of Cloud Application Pods running, the TCA will make another API call to reduce the number of running Pods in the Cloud Application.

This version implements a very simple algorithm to figure out how many Pods of the Cloud Application should be running. It simply takes the Throughput Kbps and divides it by the configured Rate to come up with the desired number of Cloud Application Pods that should be running. If the number of currently running Pods is different from this calculated number, the Kubernetes cluster is contacted and the number of Pods (known as "Replicas" in Kubernetes parlance) is sent to it so that Kubernetes can adjust the number of running Pods either up or down.

It should be noted that this method of scaling works just fine when increasing the number of running Pods but has some flaws when decreasing them. Connections that are "in flight" to the Pods that are suddenly removed by Kubernetes are lost. If the Cloud Application is such that it would just reconnect and try again with one of the other running Pods, this method is still acceptable. If not, then this Proof-of-Concept TCA would need to be expanded to:

1. Select the Cloud Application Pods to be stopped.
2. Make an API call to the Thunder ADC node to tell it to stop sending traffic to those Pods.
3. Watch and wait for the number of connections to those Pods to drop to zero.
4. THEN call Kubernetes to stop those specific Pods.

It should also be noted that this method of shutting down specific Pods only works if the individual Pods are connected to the Thunder ADC – and this can only be done if you are using Network Tunnels such as IPinIP to connect direct down to the Pod Network layer and define each Cloud Application Pod as a Member on the Thunder ADC. You can do this if you are using a Kubernetes CNI that supports a tunneling protocol like IPinIP or VXLAN, such as Calico. The A10 Thunder Kubernetes Connector TCA also supports this type of tunneling.

How we did it

For our TCA, we have a configuration file that contains the Kubernetes Access Token so we can make API calls into our Kubernetes Cluster, and we have the name of a Kubernetes Secret where we have placed our username/password pair we will need to use to make our API calls to the Thunder node. If you are using the Thunder Kubernetes Connector (TKC) – and you really should be! – this Secret has already been defined. To make sure we don't over or under configure the number of Cloud Application Pods that are running, we have a *min_pods* and a *max_pods* defined in our configuration file. Our TCA will prevent API calls to go under or over these values.

Let's take a look at our configuration file:

```
debug: 5
# check_interval is in seconds. How often do you want the program to try
# and make adjustments to the number of running Pods?
check_interval: 10
cmd_timeout: 30
cluster:
  ip: 10.1.1.220
  # K8s API Server Port
  port: 16443
  # K8s Auth Token
  auth_token: Z1pWZHdZcmk5bSt1UXJlbjR2a3plalI0d1VyVURSK2dYbzlNQzFkL2RkWT0K
  # The deployment to scale
  deployment: webserver
  namespace: cyan
  min_pods: 3
  max_pods: 10
thunder:
  # Thunder MGMT IP & Port
  ip: 10.1.1.33
  port: 443
  # This is the K8s Secret where Username/Password to access the Thunder
  # node are stored.
  secret: thunder-access-creds
  secret_namespace: default
  slb: ws-vip
  # This is port '+' protocol. This is the format that aXAPI is looking for.
  slb_port: "80+http"
  # rate is in Kbps
  rate: 20
```

There are two main sections in the configuration file: *cluster* refers to the Kubernetes Cluster where our *deployment* is running. *auth_token* can be found in the *admin.yaml* or whatever configuration file you use on remote systems so that *kubectl* can connect to your Kubernetes Cluster. For example, the *admin.yaml* file for my Cluster looks like this:

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: LS0tLS1C...
```

```
    server: https://10.1.1.220:16443
  name: microk8s-cluster
contexts:
- context:
    cluster: microk8s-cluster
    user: admin
  name: microk8s
current-context: microk8s
kind: Config
preferences: {}
users:
- name: admin
  user:
    token: Z1pWZHdZcmk5bSt1UXJlbjR2a3plalI0d1VyVURSK2dYbzlNQzFkL2RkWT0K
```

And that last line is what I put into my configuration file as the *auth_token*. We also put in our *deployment* and the *namespace* where it resides. The *deployment* value is what is defined in Kubernetes:

```
# kubectl get deployment -n cyan
NAME                 READY   UP-TO-DATE   AVAILABLE   AGE
a10-autoscaler-k8s   1/1     1            1           20h
webserver            3/3     3            3           18d
```

The *thunder* section has our connection and login information, and what Virtual Server and Port we will be watching. *slb* has the name of the SLB Virtual Server that is associated with our Deployment on the Cluster. The *port* field uses the same format that a regular aXAPI call would use. This is usually the port number (80 in our case), a '+' symbol, then the protocol that is defined (http is our example.)

Deploying the a10-autoscaler-k8s TCA

To deploy the Container, we just need a simple Kubernetes Manifest file, like so:

a10-autoscaler.yaml

```
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: a10-autoscaler-k8s-config
  namespace: cyan
data:
  config.yaml: |+
    debug: 5
    # check_interval is in seconds. How often do you want the program to try
    # and make adjustments to the number of running Pods?
    check_interval: 10
    cmd_timeout: 30
    cluster:
      ip: 10.1.1.220
      # K8s API Server Port
      port: 16443
      # K8s Auth Token
      auth_token: Z1pWZHdZcmk5bSt1UXJlbjR2a3plalI0d1VyVURSK2dYbzlNQzFkL2RkWT0K
      # The deployment to scale
```

```
      deployment: "webserver"
      namespace: "cyan"
      min_pods: 3
      max_pods: 10
    thunder:
      # Thunder MGMT IP & Port
      ip: 10.1.1.33
      port: 443
      # This is the K8s Secret where Username/Password to access the Thunder
      # node are stored.
      secret: thunder-access-creds
      secret_namespace: default
      slb: ws-vip
      # This is port '+' protocol. This is the format that aXAPI is looking for.
      slb_port: "80+http"
      # rate is in Kbps
      rate: 20
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: a10-autoscaler-k8s
  namespace: cyan
  labels:
    app: a10-autoscaler-k8s
spec:
  replicas: 1
  selector:
    matchLabels:
      app: a10-autoscaler-k8s
  template:
    metadata:
      labels:
        app: a10-autoscaler-k8s
      name: a10-autoscaler-k8s
    spec:
      containers:
      - name: a10-autoscaler-k8s
        image: 10.1.1.30:5000/a10-autoscaler-k8s:latest
        volumeMounts:
          - name: a10-autoscaler-k8s-config
            mountPath: /config.yaml
            subPath: config.yaml
            readOnly: true
      volumes:
      - name: a10-autoscaler-k8s-config
        configMap:
          name: a10-autoscaler-k8s-config
          items:
          - key: config.yaml
            path: config.yaml
```

Our configuration file is setup as a ConfigMap and attached as a volume to our container. You will need to build the container and place it in a Repository that your Docker Cluster can get to. There is a *Dockerfile* to create the continer in the Github repo where this PDF is located.

Keep in mind that the *a10-autoscaler-k8s* Pod can be placed in any Namespace. I am using something different that *default* for my demo to keep those Pods separated from other Pods running in my *default* Namespace.

Starting the a10-autoscaler-k8s

To start, simply create the deployment:

```
# kubectl create -f a10-autoscaler.yaml
configmap/a10-autoscaler-k8s-config created
deployment.apps/a10-autoscaler-k8s created
```

Once you create the deployment, you should check the logs on the *a10-autoscaler-k8s* Pod to make sure everything is configured correctly.

```
# kubectl -n cyan logs a10-autoscaler-k8s-99c5ff6c7-hsgwf
time="2022-04-12 17:49:17" level=info msg="A10 Kubernetes Autoscaler Starting..."
time="2022-04-12 17:49:17" level=info msg="Connected to Kubernetes Cluster"
time="2022-04-12 17:49:17" level=info msg="Connected to Thunder Device"
```

If both the Kubernetes Cluster and the Thunder Device show as connected, then you are good to go.

The log will also show you when adjustments to the number of running Replicas or Pods were done:

```
time="2022-04-12 17:51:07" level=info msg="Adjusting Deployment 'webserver' to 4
Replicas."
time="2022-04-12 17:51:08" level=info msg="Adjustment of Cluster is Finished"
time="2022-04-12 17:51:17" level=warning msg="Tried to adjust Replicas below minimum.
Adjusting to Minimum."
time="2022-04-12 17:51:17" level=info msg="Adjusting Deployment 'webserver' to 3
Replicas."
time="2022-04-12 17:51:18" level=info msg="Adjustment of Cluster is Finished"
```