HAProxy Ingress Controller for Kubernetes

by Andjelko Iharos | Dec 12, 2017 | KUBERNETES, TECH | 3 comments

Cloud-based applications have seen a great uptake in recent years, and that is especially true for microservices-based apps and related orchestration frameworks.

These types of applications create needs for load balancing in new contexts, and here at HAProxy Technologies, we are always happy to empower our community with solutions to help it grow. We have developed a number of features over the past year that enable easier integration of HAProxy with Kubernetes and other dynamic environments. The relevant features include hitless reloads, dynamic configuration without reloading using the HAProxy Runtime API, and DNS records for service discovery.

With HAProxy already being well known for its reliability, extensible features, advanced security, and leadership in performance amongst all free and commercial software load balancers, the Kubernetes user community is also supplementing our efforts with several Ingress Controller implementations that use HAProxy at their core, and we are committed to providing them with expert advice and code contributions to help them make the best use of HAProxy's most advanced features.

In this blog post we are going to show you how HAProxy can improve performance and out-of-the-box features for microservices-based applications by using one of these HAProxy Ingress Controllers.

Getting Traffic into Kubernetes

Kubernetes started out by relying on LBaaS capabilities of cloud container engines like GKE and AWS to provide connectivity between users and microservices components. But the community soon realized that there was a need for a more dynamic, independent, and portable way of providing the same functionality. Hence, the Kubernetes Ingress project was started.

Ingress provides an implementation-independent definition of rules that govern the flow of traffic between users and components of a service. This feature is most easily demonstrated through L7 routing. With mappings of end user consumable URLs to specific microservices, Ingress gives users the choice of Ingress Controller that will actually implement the routing. While the option to use GKE or AWS always exists, standalone Kubernetes deployments have gained the option to use any load balancer with an accompanying Ingress Controller.

Dynamic Scaling with HAProxy

The Ingress Controller provides functionality required to satisfy Ingress rules, and it also has to contend with the dynamic nature of the microservices for which it is routing the traffic.

HAProxy is extremely fast and resource-efficient allowing you to get the most out of your infrastructure and minimize latencies in high-traffic scenarios. It also brings an almost endless list of options for tuning and customization. HAProxy's features like dynamic scaling and reconfiguration without reloading are also very valuable in this use case as Kubernetes pods are often spawned, terminated, and migrated in quick bursts and in high amounts, especially during deployments.

In our previous blog posts titled "Dynamic Scaling for Microservices with the HAProxy Runtime API" and "DNS for Service Discovery in HAProxy" we have covered two possible methods for dynamic scaling. In this blog post we are going to take a look at how these play a role in a Kubernetes HAProxy Ingress Controller implementation.

We will use the HAProxy Ingress Controller implementation available at jcmoraisjr/haproxy-ingress. It is a project to which HAProxy Technologies has contributed code that enables the Ingress Controller to take advantage of the HAProxy Runtime API. (Another useful HAProxy Ingress Controller implementation that you could look into would be appscode/voyager.)

Runtime API in Kubernetes Controller

Starting off with one of the examples provided in the Ingress Controllers documentation, we will deploy and enable two HTTP services with one of them serving as a default fallback.

This could be done by preparing files ingress-default-deployment.yaml and http-svc-deployment.yaml as follows:

ingress-default-deployment.yaml:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    run: ingress-default-backend
  name: ingress-default-backend
spec:
  replicas: 1
  selector:
    matchLabels:
      run: ingress-default-backend
  template:
    metadata:
      labels:
        run: ingress-default-backend
    spec:
      containers:
        - name: ingress-default-backend
          image: gcr.io/google_containers/defaultbackend:1.0
          ports:
          - containerPort: 8080

---
apiVersion: v1
kind: Service
metadata:
  labels:
    run: ingress-default-backend
  name: ingress-default-backend
  namespace: default
spec:
  ports:
  - name: port-1
    port: 8080
    protocol: TCP
    targetPort: 8080
  selector:
    run: ingress-default-backend
```

http-svc-deployment.yaml:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    run: http-svc
  name: http-svc
spec:
  replicas: 2
  selector:
    matchLabels:
      run: http-svc
  template:
    metadata:
      labels:
        run: http-svc
    spec:
      containers:
        - name: http-svc
          image: gcr.io/google_containers/echoserver:1.3
          ports:
          - containerPort: 8080

---
apiVersion: v1
kind: Service
metadata:
  labels:
    run: http-svc
  name: http-svc
  namespace: default
spec:
  ports:
    - name: port-1
      port: 8080
      protocol: TCP
      targetPort: 8080
  selector:
    run: http-svc
```

And applying the configuration:

```
$ kubectl apply -f ingress-default-deployment.yaml
deployment "ingress-default-backend" created
service "ingress-default-backend" created

$ kubectl apply -f http-svc-deployment.yaml
deployment "http-svc" created
service "http-svc" created
```

As a next step, we will set up the Ingress rules for L7 routing with the URL "/app" for hostnames, hostname "foo.bar" routed to the pods running the app "http-svc", and all other URLs routed to pods running the app "ingress-default-backend".

This could be done by preparing file http-svc-ingress.yaml as follows:

http-svc-ingress.yaml:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: app
spec:
  rules:
  - host: foo.bar
    http:
      paths:
      - path: /app
```

```
        backend:
          serviceName: http-svc
          servicePort: 8080
      - path: /
        backend:
          serviceName: ingress-default-backend
          servicePort: 8080
```

And applying the configuration:

```
$ kubectl apply -f http-svc-ingress.yaml
ingress "app" created
```

Before we start the HAProxy Ingress Controller service, we need to tune its configuration slightly to enable the option “dynamic-scaling” and to set “backend-server-slots-increment” to “4” in this example:

haproxy-configmap.yaml:

```
apiVersion: v1
data:
    dynamic-scaling: "true"
    backend-server-slots-increment: "4"
kind: ConfigMap
metadata:
  name: haproxy-configmap
```

And applying the configuration:

```
$ kubectl apply -f haproxy-configmap.yaml
configmap "haproxy-configmap" created
```

The option “backend-server-slots-increment” will allow us to tune Ingress Controller behavior in response to fluctuations in the number of pods running. With the setting of “4” in this example, there will always be up to 4 available slots in a backend definition to accommodate adding extra pods before HAProxy will increase the number of available slots. As pods are shut down and more than 4 slots in a backend definition become available, they will be removed to reduce memory consumption. For larger deployments, this setting should be increased accordingly and setting it to hundreds will have no negative impact.

Finally, we can create the HAProxy Ingress Controller deployment:

haproxy-ingress-deployment.yaml

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    run: haproxy-ingress
  name: haproxy-ingress
spec:
  replicas: 1
  selector:
    matchLabels:
      run: haproxy-ingress
  template:
    metadata:
      labels:
        run: haproxy-ingress
    spec:
      containers:
      - name: haproxy-ingress
        image: quay.io/jcmoraisjr/haproxy-ingress
        args:
        - --default-backend-service=default/ingress-default-backend
        - --default-ssl-certificate=default/tls-secret
        - --configmap=$(POD_NAMESPACE)/haproxy-configmap
        - --reload-strategy=native
        ports:
        - name: http
          containerPort: 80
        - name: https
          containerPort: 443
        - name: stat
          containerPort: 1936
        env:
        - name: POD_NAME
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
        - name: POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
```

And apply the configuration:

```
$ kubectl apply -f haproxy-ingress-deployment.yaml
deployment "haproxy-ingress" created
```

For this example we will also use an explicit service definition to allow client traffic to reach our Ingress Controller. Alternatively, this functionality can be left to external load balancer setups with access to both public network and internal Kubernetes network (a good place for HAProxy or a LBaaS) while retaining the L7 routing of the Ingress Controller.

haproxy-ingress-svc.yaml:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    run: haproxy-ingress
  name: haproxy-ingress
  namespace: default
spec:
  externalIPs:
    - 10.245.1.4
  ports:
  - name: port-1
    port: 80
    protocol: TCP
    targetPort: 80
  - name: port-2
    port: 443
    protocol: TCP
    targetPort: 443
  - name: port-3
    port: 1936
    protocol: TCP
    targetPort: 1936
  selector:
    run: haproxy-ingress
```

And apply the configuration:

```
$ kubectl apply -f haproxy-ingress-svc.yaml
```

Checking that the L7 routing works as expected, we can see that the URLs below /app are being served by the echo pod, and all others are served by the default pod.

```
$ curl -s -XGET -H 'Host: foo.bar' 'http://10.245.1.4:80/app'
CLIENT VALUES:
client_address=10.246.97.6
command=GET
real path=/app
query=nil
request_version=1.1
request_uri=http://foo.bar:8080/app

SERVER VALUES:
server_version=nginx: 1.9.11 - lua: 10001

HEADERS RECEIVED:
accept=*/*
host=foo.bar
user-agent=curl/7.47.0
x-forwarded-for=10.246.97.1
BODY:
-no body in request-

$ curl -s -XGET -H 'Host: foo.bar' 'http://10.245.1.4:80/xyz'
default backend - 404
```

Most of the above is entirely familiar to regular users of Kubernetes and Ingress, with the notable exception of adding the two options related to dynamic scaling in the HAProxy configmap.

Taking a look at the HAProxy status page on external_ip:1936, we can see the two active pods of http-svc and two empty slots.

With this, we can now add and remove pods of a service without HAProxy needing to be reloaded as long as the number of pods is within a multiple of the configured “backend-server-slots-increment” value (“4” in our example). When the set of active pods changes, the Ingress Controller issues commands over the HAProxy Runtime API to update the backend definition. At the same time, it records the new configuration in the configuration file for easier inspection and as a failsafe in the case of an unexpected reload.

```
$ kubectl scale deployment http-svc --replicas=3
deployment "http-svc" scaled

$ kubectl exec haproxy-ingress-1373648123-dqkkc pidof haproxy
38

$ kubectl scale deployment http-svc --replicas=1
deployment "http-svc" scaled

$ kubectl exec haproxy-ingress-1373648123-dqkkc pidof haproxy
38

$ kubectl scale deployment http-svc --replicas=4
deployment "http-svc" scaled

$ kubectl exec haproxy-ingress-1373648123-dqkkc pidof haproxy
38
```

As mentioned, scaling above or below the initial multiple of “backend-server-slots-increment” will cause HAProxy to adjust the number of free slots for bursting pods.

```
$ kubectl scale deploy http-svc --replicas=9
deployment "http-svc" scaled

$ kubectl exec haproxy-ingress-1373648123-dqkkc pidof haproxy
68
```

Rolling Updates

DevOps teams are often in charge of releasing new application code in production. To help with that, Kubernetes comes with a tool called “rolling update” which will move the pods to the new desired version.

In essence, it works by starting a new pod with the new version of the application, and once it is up and running, the pod with the old version is shut down. Repeating this operation for each pod that is delivering the application could be tedious, but thanks to its controller, HAProxy can perform rolling updates with no effort!

The command to trigger a rolling update could be as follows (assuming that the version "1.4" of echoserver exists):

```
$ kubectl set image deployment http-svc http-svc=gcr.io/google_containers/echoserver:1.4
```

Conclusion

HAProxy <3 Microservices!

The recent release of HAProxy version 1.8 contains additional functionality added specifically for using DNS SRV records to implement dynamic scaling using DNS.

While using DNS SRV records for configuring backend servers is not Kubernetes-specific, Kube-DNS is perfectly capable of providing this information to the controller, and our solution for making use of it with the same Ingress Controller is coming soon!

If you would like to use HAProxy Enterprise Edition with Kubernetes and get extra benefits such as a fully supported HAProxy installation, Real Time Dashboard, and management and security-focused enterprise add-ons, please see our HAProxy Enterprise Edition – Trial Version or contact us for expert advice.

Stay tuned!

SHARE THIS ARTICLE

Tags: configuration, Ingress Controller, Kubernetes, microservices

3 Comments

Giovanni Silva on May 21, 2018 at 4:43 pm

Reply

One of the HAproxy ingress community projects with good maturity is https://github.com/appscode/voyager

Is backed by a company and opensource

pedro navajas on July 8, 2018 at 5:19 pm

Reply

can you help me ? I make this Ingress, but the Pod is CrashLoopBackOff. This is the pod's log:

It seems the cluster it is running with Authorization enabled (like RBAC) and there is no permissions for the ingress controller. Please check the configuration

Baptiste Assmann on November 6, 2018 at 5:35 am

Reply

You have to configure RBAC for the ingress controller.
Check this link https://github.com/jcmoraisjr/haproxy-ingress/tree/master/examples/rbac

This site uses Akismet to reduce spam. Learn how your comment data is processed.

Comments will display after being approved by the moderator.

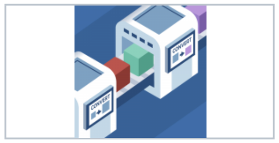
RELATED STORIES

HAPROXY WEBINAR

[On-Demand Webinar] Introduction to HAProxy ACLs: Building Rules for Dynamically Routing Requests, Redirecting Users and Blocking Malicious Traffic

The Four Essential Sections of an HAProxy Configuration

COMPANY CONTACT US GET HAPROXY GET HAPROXY BLOG CUSTOMER LOGIN


Introduction to HAProxy ACLs

PRODUCTS SOLUTIONS SUPPORT RESOURCES

Solutions

Load Balancing

High Availability

Application Acceleration

Security

Microservices

Products

HAProxy Enterprise Edition

Community vs Enterprise Edition

ALOHA Hardware Appliance

ALOHA Virtual Appliance

Software vs Appliance

+1 (844) 222-4340 contact@haproxy.com

Connect With Us

Support

Customer Support Portal

Support Options

Professional Services

Community Mailing List

Partners

Partner Programs

Find a Partner

Company

About Us

Contact Us

Events

Careers

User Reference

Resources

HAProxy Enterprise Edition Documentation

ALOHA Appliance Documentation

Content Library

Blog