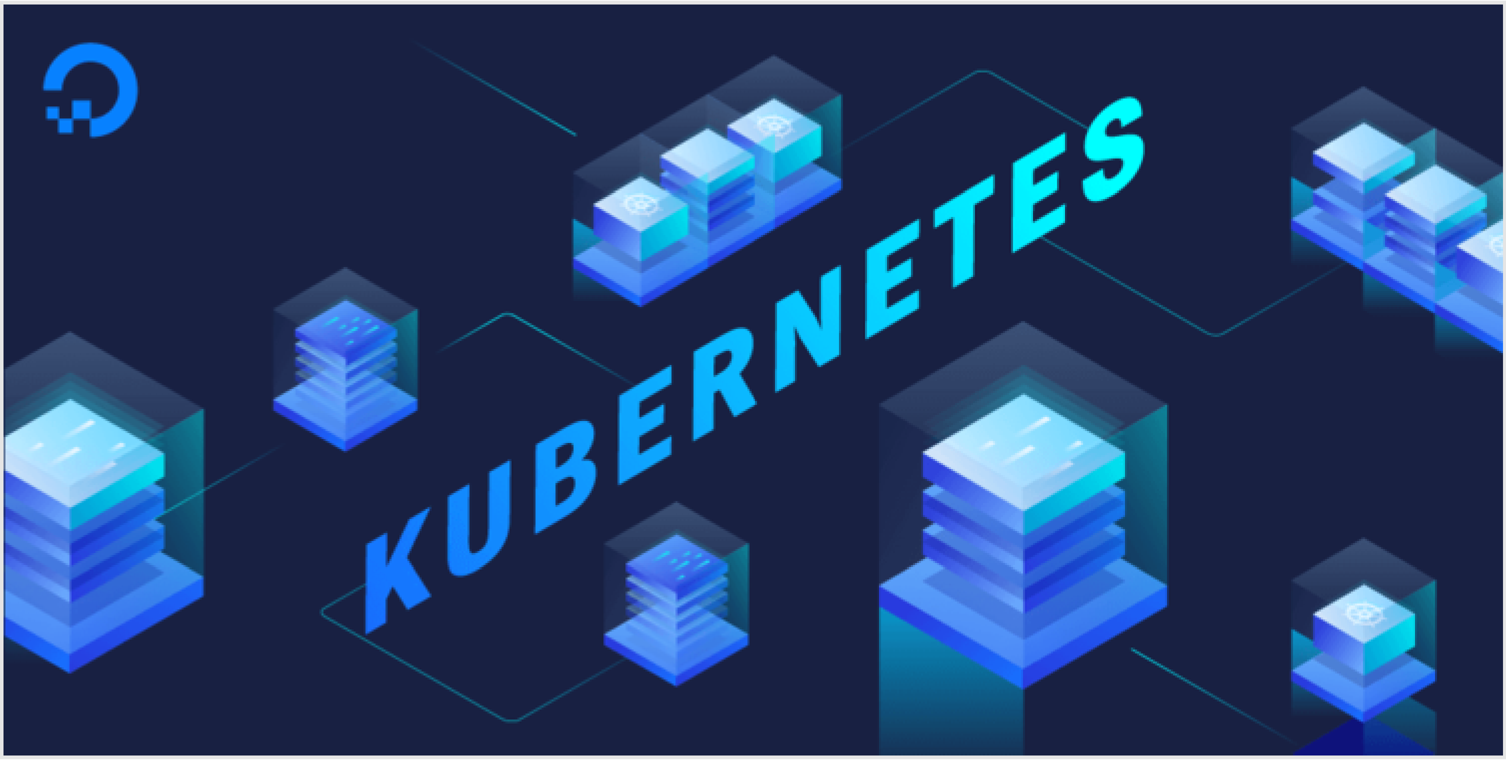

How To Inspect Kubernetes Networking

Posted July 23, 2018 20.3k KUBERNETES NETWORKING

By: Brian Boucheron

Introduction

Kubernetes is a container orchestration system that can manage containerized applications across a cluster of server nodes. Maintaining network connectivity between all the containers in a cluster requires some advanced networking techniques. In this article, we will briefly cover some tools and techniques for inspecting this networking setup.

These tools may be useful if you are debugging connectivity issues, investigating network throughput problems, or exploring Kubernetes to learn how it operates.

If you want to learn more about Kubernetes in general, our guide *An Introduction to Kubernetes* covers the basics. For a networking-specific overview of Kubernetes, please read *Kubernetes Networking Under the Hood*.

Getting Started

This tutorial will assume that you have a Kubernetes cluster, with `kubectl` installed locally and configured to connect to the cluster.

The following sections contain many commands that are intended to be run on a Kubernetes node. They will look like this:

```
# echo 'this is a node command'
```

Commands that should be run on your local machine will have the following appearance:

```
# echo 'this is a local command'
```

> **Note:** Most of the commands in this tutorial will need to be run as the **root** user. If you instead use a sudo-enabled user on your Kubernetes nodes, please add `sudo` to run the commands when necessary.

Finding a Pod's Cluster IP

To find the cluster IP address of a Kubernetes pod, use the `kubectl get pod` command on your local machine, with the option `-o wide`. This option will list more information, including the node the pod resides on, and the pod's cluster IP.

```
$ kubectl get pod -o wide
```

```
Output
NAME                           READY     STATUS    RESTARTS   AGE       IP            NODE
hello-world-5b446dd74b-7c7pk   1/1       Running   0          22m       10.244.18.4   node-one
hello-world-5b446dd74b-pxtzt   1/1       Running   0          22m       10.244.3.4    node-two
```

The **IP** column will contain the internal cluster IP address for each pod.

If you don't see the pod you're looking for, make sure you're in the right namespace. You can list all pods in all namespaces by adding the flag `--all-namespaces`.

Finding a Service's IP

We can find a Service IP using `kubectl` as well. In this case we will list all services in all namespaces:

```
$ kubectl get service --all-namespaces
```

```
Output
NAMESPACE     NAME                       TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)         AG
default       kubernetes                 ClusterIP  10.32.0.1       <none>        443/TCP         6c
kube-system   csi-attacher-doplugin      ClusterIP  10.32.159.128   <none>        12345/TCP       6c
kube-system   csi-provisioner-doplugin   ClusterIP  10.32.61.61     <none>        12345/TCP       6c
kube-system   kube-dns                   ClusterIP  10.32.0.10      <none>        53/UDP,53/TCP   6c
kube-system   kubernetes-dashboard       ClusterIP  10.32.226.209   <none>        443/TCP         6c
```

The service IP can be found in the **CLUSTER-IP** column.

Finding and Entering Pod Network Namespaces

Each Kubernetes pod gets assigned its own network namespace. Network namespaces (or netns) are a Linux networking primitive that provide isolation between network devices.

It can be useful to run commands from within a pod's netns, to check DNS resolution or general network connectivity. To do so, we first need to look up the process ID of one of the containers in a pod. For Docker, we can do that with a series of two commands. First, list the containers running on a node:

```
# docker ps
```

```
Output
CONTAINER ID     IMAGE                                COMMAND                   CREATED
173ee46a3926     gcr.io/google-samples/node-hello     "/bin/sh -c 'node se…"    9 days ago
11ad51cb72df     k8s.gcr.io/pause-amd64:3.1           "/pause"                  9 days ago
. . .
```

Find the **container ID** or **name** of any container in the pod you're interested in. In the above output we're showing two containers:

- The first container is the `hello-world` app running in the `hello-world` pod
- The second is a *pause* container running in the `hello-world` pod. This container exists solely to hold onto the pod's network namespace

To get the process ID of either container, take note of the container ID or name, and use it in the following `docker` command:

```
# docker inspect --format '{{ .State.Pid }}' container-id-or-name
```

```
Output
14552
```

A process ID (or PID) will be output. Now we can use the `nsenter` program to run a command in that process's network namespace:

```
# nsenter -t your-container-pid -n ip addr
```

Be sure to use your own PID, and replace `ip addr` with the command you'd like to run inside the pod's network namespace.

> **Note:** One advantage of using `nsenter` to run commands in a pod's namespace – versus using something like `docker exec` – is that you have access to all of the commands available on the node, instead of the typically limited set of commands installed in containers.

Finding a Pod's Virtual Ethernet Interface

Each pod's network namespace communicates with the node's root netns through a virtual ethernet pipe. On the node side, this pipe appears as a device that typically begins with `veth` and ends in a unique identifier, such as `veth77f2275` or `veth01`. Inside the pod this pipe appears as `eth0`.

It can be useful to correlate which `veth` device is paired with a particular pod. To do so, we will list all network devices on the node, then list the devices in the pod's network namespace. We can then correlate device numbers between the two listings to make the connection.

First, run `ip addr` in the pod's network namespace using `nsenter`. Refer to the previous section *Finding and Entering Pod Network Namespaces*
for details on how to do this:

```
# nsenter -t your-container-pid -n ip addr
```

```
Output
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
10: eth0@if11: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default
    link/ether 02:42:0a:f4:03:04 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.244.3.4/24 brd 10.244.3.255 scope global eth0
       valid_lft forever preferred_lft forever
```

The command will output a list of the pod's interfaces. Note the `if11` number after `eth0@` in the example output. This means this pod's `eth0` is linked to the node's 11th interface. Now run `ip addr` in the node's default namespace to list out its interfaces:

```
# ip addr
```

```
Output
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever

. . .

7: veth77f2275@if6: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue master docker0 state UP
    link/ether 26:05:99:58:0d:b9 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet6 fe80::2405:99ff:fe58:db9/64 scope link
       valid_lft forever preferred_lft forever
9: vethd36cef3@if8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue master docker0 state UP
    link/ether ae:05:21:a2:9a:2b brd ff:ff:ff:ff:ff:ff link-netnsid 1
    inet6 fe80::ac05:21ff:fea2:9a2b/64 scope link
       valid_lft forever preferred_lft forever
11: veth4f7342d@if10: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue master docker0 state U
    link/ether e6:4d:7b:6f:56:4c brd ff:ff:ff:ff:ff:ff link-netnsid 2
    inet6 fe80::e44d:7bff:fe6f:564c/64 scope link
       valid_lft forever preferred_lft forever
```

The 11th interface is `veth4f7342d` in this example output. This is the virtual ethernet pipe to the pod we're investigating.

Inspecting Conntrack Connection Tracking

Prior to version 1.11, Kubernetes used iptables NAT and the conntrack kernel module to track connections. To list all the connections currently being tracked, use the `conntrack` command:

```
# conntrack -L
```

To watch continuously for new connections, use the `-E` flag:

```
# conntrack -E
```

To list conntrack-tracked connections to a particular destination address, use the `-d` flag:

```
# conntrack -L -d 10.32.0.1
```

If your nodes are having issues making reliable connections to services, it's possible your connection tracking table is full and new connections are being dropped. If that's the case you may see messages like the following in your system logs:

/var/log/syslog

```
Jul 12 15:32:11 worker-528 kernel: nf_conntrack: table full, dropping packet.
```

There is a sysctl setting for the maximum number of connections to track. You can list out your current value with the following command:

```
# sysctl net.netfilter.nf_conntrack_max
```

```
Output
net.netfilter.nf_conntrack_max = 131072
```

To set a new value, use the `-w` flag:

```
# sysctl -w net.netfilter.nf_conntrack_max=198000
```

To make this setting permanent, add it to the `sysctl.conf` file:

/etc/sysctl.conf

```
. . .
net.ipv4.netfilter.ip_conntrack_max = 198000
```

Inspecting Iptables Rules

Prior to version 1.11, Kubernetes used iptables NAT to implement virtual IP translation and load balancing for Service IPs.

To dump all iptables rules on a node, use the `iptables-save` command:

```
# iptables-save
```

Because the output can be lengthy, you may want to pipe to a file (`iptables-save > output.txt`) or a pager (`iptables-save | less`) to more easily review the rules.

To list just the Kubernetes Service NAT rules, use the `iptables` command and the `-L` flag to specify the correct chain:

```
# iptables -t nat -L KUBE-SERVICES
```

```
Output
Chain KUBE-SERVICES (2 references)
target     prot opt source               destination
KUBE-SVC-TCOU7JCQXEZGVUNU  udp  --  anywhere             10.32.0.10           /* kube-system/kube-dns
KUBE-SVC-ERIFXISQEP7F7OF4  tcp  --  anywhere             10.32.0.10           /* kube-system/kube-dns
KUBE-SVC-XGLOHA7QRQ3V22RZ  tcp  --  anywhere             10.32.226.209        /* kube-system/kubernet
. . .
```

Querying Cluster DNS

One way to debug your cluster DNS resolution is to deploy a debug container with all the tools you need, then use `kubectl` to exec `nslookup` on it. This is described in the official Kubernetes documentation.

Another way to query the cluster DNS is using `dig` and `nsenter` from a node. If `dig` is not installed, it can be installed with `apt` on Debian-based Linux distributions:

```
# apt install dnsutils
```

First, find the cluster IP of the **kube-dns** service:

```
# kubectl get service -n kube-system kube-dns
```

```
Output
NAME       TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)         AGE
kube-dns   ClusterIP   10.32.0.10   <none>        53/UDP,53/TCP   15d
```

The cluster IP is highlighted above. Next we'll use `nsenter` to run `dig` in the a container namespace. Look at the section *Finding and Entering Pod Network Namespaces* for more information on this:

```
# nsenter -t 14346 -n dig kubernetes.default.svc.cluster.local @10.32.0.10
```

This `dig` command looks up the Service's full domain name of **service-name.namespace.svc.cluster.local** and specifics the IP of the cluster DNS service IP (`@10.32.0.10`).

Looking at IPVS Details

As of Kubernetes 1.11, `kube-proxy` can configure IPVS to handle the translation of virtual Service IPs to pod IPs. You can list the translation table of IPs with `ipvsadm`:

```
# ipvsadm -Ln
```

```
Output
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
TCP  100.64.0.1:443 rr
  -> 178.128.226.86:443           Masq    1      0          0
TCP  100.64.0.10:53 rr
  -> 100.96.1.3:53                Masq    1      0          0
  -> 100.96.1.4:53                Masq    1      0          0
UDP  100.64.0.10:53 rr
  -> 100.96.1.3:53                Masq    1      0          0
  -> 100.96.1.4:53                Masq    1      0          0
```

To show a single Service IP, use the `-t` option and specify the desired IP:

```
# ipvsadm -Ln -t 100.64.0.10:53
```

```
Output
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
TCP  100.64.0.10:53 rr
  -> 100.96.1.3:53                Masq    1      0          0
  -> 100.96.1.4:53                Masq    1      0          0
```

Conclusion

In this article we've reviewed some commands and techniques for exploring and inspecting the details of your Kubernetes cluster's networking. For more information about Kubernetes, take a look at our Kubernetes tutorials tag and the official Kubernetes documentation.

By: Brian Boucheron

Related Tutorials

An Introduction to Service Meshes

How To Deploy a PHP Application with Kubernetes on Ubuntu 16.04

How To Set Up Multi-Node Deployments With Rancher 2.1, Kubernetes, and Docker Machine on Ubu...

How to Set Up an Nginx Ingress with Cert-Manager on DigitalOcean Kubernetes

How To Set Up an Elasticsearch, Fluentd and Kibana (EFK) Logging Stack on Kubernetes

1 Comment

Leave a comment...

Log In to Comment

Liyong *November 1, 2018*

0 Good post! Thanks for sharing these.