kubernetes.io

Debug Services

An issue that comes up rather frequently for new installations of Kubernetes is that a Service is not working properly. You've run your Pods through a Deployment (or other workload controller) and created a Service, but you get no response when you try to access it. This document will hopefully help you to figure out what's going wrong.

Running commands in a Pod

For many steps here you will want to see what a Pod running in the cluster sees. The simplest way to do this is to run an interactive busybox Pod:

```
kubectl run -it --rm --restart=Never busybox --
image=gcr.io/google-containers/busybox sh
```

If you already have a running Pod that you prefer to use, you can run a command in it using:

```
kubectl exec <POD-NAME> -c <CONTAINER-NAME> --
<COMMAND>
```

Setup

For the purposes of this walk-through, let's run some Pods. Since you're probably debugging your own Service you can substitute your own details, or you can follow along and get a second data point.

```
kubectl create deployment hostnames --
image=registry.k8s.io/serve_hostname
```

```
deployment.apps/hostnames created
```

`kubectl` commands will print the type and name of the resource created or mutated, which can then be used in subsequent commands.

Let's scale the deployment to 3 replicas.

```
kubectl scale deployment hostnames --replicas=3
```

```
deployment.apps/hostnames scaled
```

Note that this is the same as if you had started the Deployment with the following YAML:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: hostnames
  name: hostnames
spec:
  selector:
    matchLabels:
      app: hostnames
  replicas: 3
  template:
    metadata:
      labels:
        app: hostnames
    spec:
      containers:
      - name: hostnames
        image: registry.k8s.io/serve_hostname
```

The label "app" is automatically set by `kubectl create deployment` to the name of the Deployment.

You can confirm your Pods are running:

```
kubectl get pods -l app=hostnames
```

```
NAME                         READY     STATUS
```

```
RESTARTS   AGE
hostnames-632524106-bbpiw   1/1       Running   0
2m
hostnames-632524106-ly40y   1/1       Running   0
2m
hostnames-632524106-tlaok   1/1       Running   0
2m
```

You can also confirm that your Pods are serving. You can get the list of Pod IP addresses and test them directly.

```
kubectl get pods -l app=hostnames \
    -o go-template='{{range .items}}
{{.status.podIP}}{{"\n"}}{{end}}'
```

```
10.244.0.5
10.244.0.6
10.244.0.7
```

The example container used for this walk-through serves its own hostname via HTTP on port 9376, but if you are debugging your own app, you'll want to use whatever port number your Pods are listening on.

From within a pod:

```
for ep in 10.244.0.5:9376 10.244.0.6:9376
10.244.0.7:9376; do
    wget -qO- $ep
done
```

This should produce something like:

```
hostnames-632524106-bbpiw
hostnames-632524106-ly40y
hostnames-632524106-tlaok
```

If you are not getting the responses you expect at this point, your Pods might not be healthy or might not be listening on the port you think they are. You might find `kubectl logs` to be useful for seeing what is happening, or perhaps you need to `kubectl exec`

directly into your Pods and debug from there.

Assuming everything has gone to plan so far, you can start to investigate why your Service doesn't work.

Does the Service exist?

The astute reader will have noticed that you did not actually create a Service yet - that is intentional. This is a step that sometimes gets forgotten, and is the first thing to check.

What would happen if you tried to access a non-existent Service? If you have another Pod that consumes this Service by name you would get something like:

```
Resolving hostnames (hostnames)... failed: Name
or service not known.
wget: unable to resolve host address 'hostnames'
```

The first thing to check is whether that Service actually exists:

```
kubectl get svc hostnames
```

```
No resources found.
Error from server (NotFound): services
"hostnames" not found
```

Let's create the Service. As before, this is for the walk-through - you can use your own Service's details here.

```
kubectl expose deployment hostnames --port=80 --
target-port=9376
```

```
service/hostnames exposed
```

And read it back:

```
kubectl get svc hostnames
```

```
NAME        TYPE        CLUSTER-IP   EXTERNAL-IP
PORT(S)   AGE
hostnames   ClusterIP   10.0.1.175   <none>
80/TCP    5s
```

Now you know that the Service exists.

As before, this is the same as if you had started the Service with YAML:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: hostnames
  name: hostnames
spec:
  selector:
    app: hostnames
  ports:
  - name: default
    protocol: TCP
    port: 80
    targetPort: 9376
```

In order to highlight the full range of configuration, the Service you created here uses a different port number than the Pods. For many real-world Services, these values might be the same.

Any Network Policy Ingress rules affecting the target Pods?

If you have deployed any Network Policy Ingress rules which may affect incoming traffic to `hostnames-*` Pods, these need to be reviewed.

Please refer to Network Policies for more details.

Does the Service work by DNS name?

One of the most common ways that clients consume a Service is through a DNS name.

From a Pod in the same Namespace:

```
Address 1: 10.0.0.10 kube-dns.kube-
system.svc.cluster.local

Name:      hostnames
Address 1: 10.0.1.175
hostnames.default.svc.cluster.local
```

If this fails, perhaps your Pod and Service are in different Namespaces, try a namespace-qualified name (again, from within a Pod):

```
nslookup hostnames.default
```

```
Address 1: 10.0.0.10 kube-dns.kube-
system.svc.cluster.local

Name:      hostnames.default
Address 1: 10.0.1.175
hostnames.default.svc.cluster.local
```

If this works, you'll need to adjust your app to use a cross-namespace name, or run your app and Service in the same Namespace. If this still fails, try a fully-qualified name:

```
nslookup hostnames.default.svc.cluster.local
```

```
Address 1: 10.0.0.10 kube-dns.kube-
system.svc.cluster.local

Name:      hostnames.default.svc.cluster.local
Address 1: 10.0.1.175
hostnames.default.svc.cluster.local
```

Note the suffix here: "default.svc.cluster.local". The "default" is the Namespace you're operating in. The "svc" denotes that this is a Service. The "cluster.local" is your cluster domain, which COULD be different in your own cluster.

You can also try this from a Node in the cluster:

```
nslookup hostnames.default.svc.cluster.local
```

```
10.0.0.10
```

```
Server:         10.0.0.10
Address:        10.0.0.10#53

Name:   hostnames.default.svc.cluster.local
Address: 10.0.1.175
```

If you are able to do a fully-qualified name lookup but not a relative one, you need to check that your `/etc/resolv.conf` file in your Pod is correct. From within a Pod:

You should see something like:

```
nameserver 10.0.0.10
search default.svc.cluster.local
svc.cluster.local cluster.local example.com
options ndots:5
```

The `nameserver` line must indicate your cluster's DNS Service. This is passed into `kubelet` with the `--cluster-dns` flag.

The `search` line must include an appropriate suffix for you to find the Service name. In this case it is looking for Services in the local Namespace ("default.svc.cluster.local"), Services in all Namespaces ("svc.cluster.local"), and lastly for names in the cluster ("cluster.local"). Depending on your own install you might have additional records after that (up to 6 total). The cluster suffix is passed into `kubelet` with the `--cluster-domain` flag. Throughout this document, the cluster suffix is assumed to be "cluster.local". Your own clusters might be configured differently, in which case you should change that in all of the previous commands.

The `options` line must set `ndots` high enough that your DNS client library considers search paths at all. Kubernetes sets this to 5 by default, which is high enough to cover all of the DNS names it generates.

Does any Service work by DNS name?

If the above still fails, DNS lookups are not working for your Service. You can take a step back and see what else is not working. The Kubernetes master Service should always work. From within a Pod:

```
nslookup kubernetes.default
```

```
Server:    10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-
system.svc.cluster.local

Name:      kubernetes.default
Address 1: 10.0.0.1
kubernetes.default.svc.cluster.local
```

If this fails, please see the kube-proxy section of this document, or even go back to the top of this document and start over, but instead of debugging your own Service, debug the DNS Service.

Does the Service work by IP?

Assuming you have confirmed that DNS works, the next thing to test is whether your Service works by its IP address. From a Pod in your cluster, access the Service's IP (from `kubectl get` above).

```
for i in $(seq 1 3); do
    wget -qO- 10.0.1.175:80
done
```

This should produce something like:

```
hostnames-632524106-bbpiw
hostnames-632524106-ly40y
hostnames-632524106-tlaok
```

If your Service is working, you should get correct responses. If not, there are a number of things that could be going wrong. Read on.

Is the Service defined correctly?

It might sound silly, but you should really double and triple check

that your Service is correct and matches your Pod's port. Read back your Service and verify it:

```
kubectl get service hostnames -o json
```


```
{
    "kind": "Service",
    "apiVersion": "v1",
    "metadata": {
        "name": "hostnames",
        "namespace": "default",
        "uid":
"428c8b6c-24bc-11e5-936d-42010af0a9bc",
        "resourceVersion": "347189",
        "creationTimestamp":
"2015-07-07T15:24:29Z",
        "labels": {
            "app": "hostnames"
        }
    },
    "spec": {
        "ports": [
            {
                "name": "default",
                "protocol": "TCP",
                "port": 80,
                "targetPort": 9376,
                "nodePort": 0
            }
        ],
        "selector": {
            "app": "hostnames"
        },
        "clusterIP": "10.0.1.175",
        "type": "ClusterIP",
        "sessionAffinity": "None"
```

```
    },
    "status": {
        "loadBalancer": {}
    }
}
```

- Is the Service port you are trying to access listed in `spec.ports[]`?
- Is the `targetPort` correct for your Pods (some Pods use a different port than the Service)?
- If you meant to use a numeric port, is it a number (9376) or a string "9376"?
- If you meant to use a named port, do your Pods expose a port with the same name?
- Is the port's `protocol` correct for your Pods?

Does the Service have any Endpoints?

If you got this far, you have confirmed that your Service is correctly defined and is resolved by DNS. Now let's check that the Pods you ran are actually being selected by the Service.

Earlier you saw that the Pods were running. You can re-check that:

```
kubectl get pods -l app=hostnames
```

```
NAME                         READY     STATUS
RESTARTS   AGE
hostnames-632524106-bbpiw    1/1       Running   0
1h
hostnames-632524106-ly40y    1/1       Running   0
1h
hostnames-632524106-tlaok    1/1       Running   0
1h
```

The `-l app=hostnames` argument is a label selector configured on the Service.

The "AGE" column says that these Pods are about an hour old, which implies that they are running fine and not crashing.

The "RESTARTS" column says that these pods are not crashing frequently or being restarted. Frequent restarts could lead to intermittent connectivity issues. If the restart count is high, read more about how to debug pods.

Inside the Kubernetes system is a control loop which evaluates the selector of every Service and saves the results into a corresponding Endpoints object.

```
kubectl get endpoints hostnames

NAME        ENDPOINTS
hostnames
10.244.0.5:9376,10.244.0.6:9376,10.244.0.7:9376
```

This confirms that the endpoints controller has found the correct Pods for your Service. If the `ENDPOINTS` column is `<none>`, you should check that the `spec.selector` field of your Service actually selects for `metadata.labels` values on your Pods. A common mistake is to have a typo or other error, such as the Service selecting for `app=hostnames`, but the Deployment specifying `run=hostnames`, as in versions previous to 1.18, where the `kubectl run` command could have been also used to create a Deployment.

Are the Pods working?

At this point, you know that your Service exists and has selected your Pods. At the beginning of this walk-through, you verified the Pods themselves. Let's check again that the Pods are actually working - you can bypass the Service mechanism and go straight to the Pods, as listed by the Endpoints above.

From within a Pod:

```
for ep in 10.244.0.5:9376 10.244.0.6:9376
```

```
10.244.0.7:9376; do
    wget -qO- $ep
done
```

This should produce something like:

```
hostnames-632524106-bbpiw
hostnames-632524106-ly40y
hostnames-632524106-tlaok
```

You expect each Pod in the Endpoints list to return its own hostname. If this is not what happens (or whatever the correct behavior is for your own Pods), you should investigate what's happening there.

Is the kube-proxy working?

If you get here, your Service is running, has Endpoints, and your Pods are actually serving. At this point, the whole Service proxy mechanism is suspect. Let's confirm it, piece by piece.

The default implementation of Services, and the one used on most clusters, is kube-proxy. This is a program that runs on every node and configures one of a small set of mechanisms for providing the Service abstraction. If your cluster does not use kube-proxy, the following sections will not apply, and you will have to investigate whatever implementation of Services you are using.

Is kube-proxy running?

Confirm that `kube-proxy` is running on your Nodes. Running directly on a Node, you should get something like the below:

```
ps auxw | grep kube-proxy
```

```
root  4194  0.4  0.1 101864 17696 ?    Sl Jul04
25:43 /usr/local/bin/kube-proxy --master=https://
kubernetes-master --kubeconfig=/var/lib/kube-
proxy/kubeconfig --v=2
```

Next, confirm that it is not failing something obvious, like contacting

the master. To do this, you'll have to look at the logs. Accessing the logs depends on your Node OS. On some OSes it is a file, such as /var/log/kube-proxy.log, while other OSes use `journalctl` to access logs. You should see something like:

```
I1027 22:14:53.995134    5063 server.go:200]
Running in resource-only container "/kube-proxy"
I1027 22:14:53.998163    5063 server.go:247]
Using iptables Proxier.
I1027 22:14:54.038140    5063 proxier.go:352]
Setting endpoints for "kube-system/kube-dns:dns-
tcp" to [10.244.1.3:53]
I1027 22:14:54.038164    5063 proxier.go:352]
Setting endpoints for "kube-system/kube-dns:dns"
to [10.244.1.3:53]
I1027 22:14:54.038209    5063 proxier.go:352]
Setting endpoints for "default/kubernetes:https"
to [10.240.0.2:443]
I1027 22:14:54.038238    5063 proxier.go:429] Not
syncing iptables until Services and Endpoints
have been received from master
I1027 22:14:54.040048    5063 proxier.go:294]
Adding new service "default/kubernetes:https" at
10.0.0.1:443/TCP
I1027 22:14:54.040154    5063 proxier.go:294]
Adding new service "kube-system/kube-dns:dns" at
10.0.0.10:53/UDP
I1027 22:14:54.040223    5063 proxier.go:294]
Adding new service "kube-system/kube-dns:dns-tcp"
at 10.0.0.10:53/TCP
```

If you see error messages about not being able to contact the master, you should double-check your Node configuration and installation steps.

One of the possible reasons that `kube-proxy` cannot run correctly is that the required `conntrack` binary cannot be found. This may

happen on some Linux systems, depending on how you are installing the cluster, for example, you are installing Kubernetes from scratch. If this is the case, you need to manually install the `conntrack` package (e.g. `sudo apt install conntrack` on Ubuntu) and then retry.

Kube-proxy can run in one of a few modes. In the log listed above, the line `Using iptables Proxier` indicates that kube-proxy is running in "iptables" mode. The most common other mode is "ipvs".

Iptables mode

In "iptables" mode, you should see something like the following on a Node:

```
iptables-save | grep hostnames
```

```
-A KUBE-SEP-57KPRZ3JQVENLNBR -s 10.244.3.6/32 -m
comment --comment "default/hostnames:" -j MARK --
set-xmark 0x00004000/0x00004000
-A KUBE-SEP-57KPRZ3JQVENLNBR -p tcp -m comment --
comment "default/hostnames:" -m tcp -j DNAT --to-
destination 10.244.3.6:9376
-A KUBE-SEP-WNBA2IHDGP2BOBGZ -s 10.244.1.7/32 -m
comment --comment "default/hostnames:" -j MARK --
set-xmark 0x00004000/0x00004000
-A KUBE-SEP-WNBA2IHDGP2BOBGZ -p tcp -m comment --
comment "default/hostnames:" -m tcp -j DNAT --to-
destination 10.244.1.7:9376
-A KUBE-SEP-X3P2623AGDH6CDF3 -s 10.244.2.3/32 -m
comment --comment "default/hostnames:" -j MARK --
set-xmark 0x00004000/0x00004000
-A KUBE-SEP-X3P2623AGDH6CDF3 -p tcp -m comment --
comment "default/hostnames:" -m tcp -j DNAT --to-
destination 10.244.2.3:9376
-A KUBE-SERVICES -d 10.0.1.175/32 -p tcp -m
comment --comment "default/hostnames: cluster IP"
```

```
-m tcp --dport 80 -j KUBE-SVC-NWV5X2332I4OT4T3
-A KUBE-SVC-NWV5X2332I4OT4T3 -m comment --comment
"default/hostnames:" -m statistic --mode random
--probability 0.33332999982 -j KUBE-SEP-
WNBA2IHDGP2BOBGZ
-A KUBE-SVC-NWV5X2332I4OT4T3 -m comment --comment
"default/hostnames:" -m statistic --mode random
--probability 0.50000000000 -j KUBE-SEP-
X3P2623AGDH6CDF3
-A KUBE-SVC-NWV5X2332I4OT4T3 -m comment --comment
"default/hostnames:" -j KUBE-SEP-57KPRZ3JQVENLNBR
```

For each port of each Service, there should be 1 rule in KUBE-SERVICES and one KUBE-SVC-<hash> chain. For each Pod endpoint, there should be a small number of rules in that KUBE-SVC-<hash> and one KUBE-SEP-<hash> chain with a small number of rules in it. The exact rules will vary based on your exact config (including node-ports and load-balancers).

IPVS mode

In "ipvs" mode, you should see something like the following on a Node:

```
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight
ActiveConn InActConn
...
TCP  10.0.1.175:80 rr
  -> 10.244.0.5:9376                Masq    1
0          0
  -> 10.244.0.6:9376                Masq    1
0          0
  -> 10.244.0.7:9376                Masq    1
0          0
...
```

For each port of each Service, plus any NodePorts, external IPs, and load-balancer IPs, kube-proxy will create a virtual server. For each Pod endpoint, it will create corresponding real servers. In this example, service hostnames(`10.0.1.175:80`) has 3 endpoints(`10.244.0.5:9376`, `10.244.0.6:9376`, `10.244.0.7:9376`).

Is kube-proxy proxying?

Assuming you do see one the above cases, try again to access your Service by IP from one of your Nodes:

```
hostnames-632524106-bbpiw
```

If this still fails, look at the `kube-proxy` logs for specific lines like:

```
Setting endpoints for default/hostnames:default
to [10.244.0.5:9376 10.244.0.6:9376
10.244.0.7:9376]
```

If you don't see those, try restarting `kube-proxy` with the `-v` flag set to 4, and then look at the logs again.

Edge case: A Pod fails to reach itself via the Service IP

This might sound unlikely, but it does happen and it is supposed to work.

This can happen when the network is not properly configured for "hairpin" traffic, usually when `kube-proxy` is running in `iptables` mode and Pods are connected with bridge network. The `Kubelet` exposes a `hairpin-mode` flag that allows endpoints of a Service to loadbalance back to themselves if they try to access their own Service VIP. The `hairpin-mode` flag must either be set to `hairpin-veth` or `promiscuous-bridge`.

The common steps to trouble shoot this are as follows:

- Confirm `hairpin-mode` is set to `hairpin-veth` or `promiscuous-bridge`. You should see something like the below.

hairpin-mode is set to promiscuous-bridge in the following example.

```
root      3392  1.1  0.8 186804 65208 ?        Sl
00:51  11:11 /usr/local/bin/kubelet --enable-
debugging-handlers=true --config=/etc/kubernetes/
manifests --allow-privileged=True --v=4 --
cluster-dns=10.0.0.10 --cluster-
domain=cluster.local --configure-cbr0=true --
cgroup-root=/ --system-cgroups=/system --hairpin-
mode=promiscuous-bridge --runtime-cgroups=/
docker-daemon --kubelet-cgroups=/kubelet --
babysit-daemons=true --max-pods=110 --serialize-
image-pulls=false --outofdisk-transition-
frequency=0
```

- Confirm the effective hairpin-mode. To do this, you'll have to look at kubelet log. Accessing the logs depends on your Node OS. On some OSes it is a file, such as /var/log/kubelet.log, while other OSes use journalctl to access logs. Please be noted that the effective hairpin mode may not match --hairpin-mode flag due to compatibility. Check if there is any log lines with key word hairpin in kubelet.log. There should be log lines indicating the effective hairpin mode, like something below.

```
I0629 00:51:43.648698    3252 kubelet.go:380]
Hairpin mode set to "promiscuous-bridge"
```

- If the effective hairpin mode is hairpin-veth, ensure the Kubelet has the permission to operate in /sys on node. If everything works properly, you should see something like:

```
for intf in /sys/devices/virtual/net/cbr0/brif/*;
do cat $intf/hairpin_mode; done
```

```
1
1
1
```

```
1
```

- If the effective hairpin mode is `promiscuous-bridge`, ensure `Kubelet` has the permission to manipulate linux bridge on node. If `cbr0` bridge is used and configured properly, you should see:

```
ifconfig cbr0 |grep PROMISC
```

```
UP BROADCAST RUNNING PROMISC MULTICAST  MTU:1460  Metric:1
```