RX-M Cloud Native Consulting

Advanced Kubernetes

Lab 6 – DNS

DNS is a foundational service of the public Internet and private networks alike. So much so, that it is often overlooked. DNS also plays a critical role in most Kubernetes clusters.

Our current cluster is running:

- etcd: the cluster state store
- api-server: the central API of the cluster
- scheduler: binds pods to nodes
- controller manager: creates pod replicas and manages deployments
- kubelets: to run pods
- kube-proxy: to configure services in iptables
- flannneld: to provide pod/container network integration with AWS VPC

Though the core cluster services are running, we do not have the ability to discover newly created services by name.

To demonstrate this we will run a simple service to illustrate the problem and, after installing DNS, the solution.

Make sure you have all of the components from the previous lab running.

```
ubuntu@nodea:~$ ps -aefo comm

COMMAND
bash
 \_ sudo
     \_ flanneld
bash
 \_ ps
bash
```

```
  \_ sudo
      \_ kube-proxy
 bash
  \_ sudo
      \_ kubelet
 bash
  \_ etcd
 bash
  \_ sudo
      \_ kube-apiserver
 bash
  \_ kube-scheduler
 bash
  \_ kube-controller
 agetty
 agetty
 ubuntu@nodea:~$
```

> N.B. if you want to use Docker instead of flannel and your LAB VMs have been disconnected and reconnected to the external network you will need to recreate your cluster inter-node routes (the network manager removes temporary routes each time DHCP data is reassigned).

1. Run a simple service and try to discover it

As a first step we will run a simple nginx service and then try to reach the web servers from a client pod. To begin run an nginx-based deployment with two pods (same as previous lab):

```
ubuntu@nodea:~$ cat testdep.yaml

apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    name: nginx-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
```

```
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
ubuntu@nodea:~$
```

```
ubuntu@nodea:~$ kubectl create -f testdep.yaml

deployment.apps/nginx-deployment created
ubuntu@nodea:~$
```

Now wait until the deployment is fully running:

```
ubuntu@nodea:~$ kubectl get deploy,rs,po

NAME                       DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/nginx-deployment    2         2         2            2           5s

NAME                               DESIRED   CURRENT   READY     AGE
rs/nginx-deployment-171375908      2         2         2         5s

NAME                                    READY     STATUS    RESTARTS   AGE
po/nginx-deployment-171375908-0bggz     1/1       Running   0          5s
po/nginx-deployment-171375908-g1cms     1/1       Running   0          5s
ubuntu@nodea:~$
```

Now create a service to front end the pods (modify ports information):

```
ubuntu@nodea:~$ vim websvc.yaml
ubuntu@nodea:~$ cat websvc.yaml

apiVersion: v1
```

```
kind: Service
metadata:
  name: websvc
spec:
  ports:
    - port: 80
  selector:
    app: nginx
ubuntu@nodea:~$
```

```
ubuntu@nodea:~$ kubectl create -f websvc.yaml

service/websvc created
ubuntu@nodea:~$
```

```
ubuntu@nodea:~$ kubectl get service --all-namespaces

NAMESPACE   NAME         CLUSTER-IP    EXTERNAL-IP   PORT(S)   AGE
default     kubernetes   10.0.0.1      <none>        443/TCP   1m
default     websvc       10.0.13.26    <none>        80/TCP    4s
user@nodea:~$
```

Now that we have a named service running we can start up a client pod to access it with. Run a busybox container to act as the client:

```
ubuntu@nodea:~$ vim clientpod.yaml
ubuntu@nodea:~$ cat clientpod.yaml

apiVersion: v1
kind: Pod
metadata:
  name: clientpod
spec:
  containers:
  - name: clientpod
    image: busybox
    command: ['tail', '-f', '/dev/null']
ubuntu@nodea:~$
```

```
ubuntu@nodea:~$ kubectl create -f clientpod.yaml

pod/clientpod created
ubuntu@nodea:~$
```

```
ubuntu@nodea:~$ kubectl get po clientpod

NAME        READY     STATUS    RESTARTS   AGE
clientpod   1/1       Running   0          3m
ubuntu@nodea:~$
```

With our dummy client pod running we can exec a shell and try to ping the nginx service. But first, install some necessary tools in our container:

```
ubuntu@nodea:~$ kubectl exec -it clientpod sh

/ #
```

Try to ping the service IP:

```
/ # ping -c 1 10.0.13.26

PING 10.0.13.26 (10.0.13.26) 56(84) bytes of data.

--- 10.0.13.26 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms

/ #
```

That never works. The service IP is a virtual IP and no one is listening for ICMP (ping) traffic there. Try to GET the the root route on port 80:

```
/ # wget -S 10.0.13.26 -O /dev/null

Connecting to 10.0.243.154 (10.0.243.154:80)
```

```
  HTTP/1.1 200 OK
  Server: nginx/1.7.9
  Date: Sun, 31 Mar 2019 03:39:46 GMT
  Content-Type: text/html
  Content-Length: 612
  Last-Modified: Tue, 23 Dec 2014 16:25:09 GMT
  Connection: close
  ETag: "54999765-264"
  Accept-Ranges: bytes

null                100% |*******************************************|   612  0:00:00 ETA

/ #
```

When we connect to TCP port 80 (as defined in the service) we get forwarded to one of the nginx pods and receive the resulting HTML.

Now try making use of the service by its name:

```
/ # wget -S websvc -O /dev/null

wget: bad address 'websvc'
/ #
```

```
/ # nslookup websvc

Server:         172.31.0.2
Address:        172.31.0.2:53

** server can't find websvc.us-west-2.compute.internal: NXDOMAIN

*** Can't find websvc.us-west-2.compute.internal: No answe

/ #
```

```
/ # exit
```

```
ubuntu@nodea:~$
```

We can not reach the service by name. When you try to connect to a name rather than an IP address the Linux resolver tries to turn that name into an IP address by looking in the `/etc/hosts` file (inside the container in this case) and then it tries to use DNS. In the example above the DNS server is listed as 172.31.7.235. This is the bogus setting we have been using when starting our kubelets. The kubelets then tell the Docker engine to populate the `/etc/resolv.conf` file with this DNS server address. This is problematic because clients are usually better off parameterizing connections by service name rather than by the more fragile IP address.

What we need is a real DNS service that can supply Kubernetes service VIPs in response to service name lookups.

2. Enter CoreDNS

Prior to Kubernetes 1.11, an integrated DNS server called KubeDNS was used, derived from the older SkyDNS2. The author of SkyDNS2 is the lead developer of CoreDNS--a flexible, extensible DNS server that can serve as the Kubernetes cluster DNS. In Kubernetes 1.11, CoreDNS graduated to General Availability (GA) and is installed by default as a standard system service, launched automatically by installers like kubeadm.

CoreDNS is a distributed service for announcement and discovery of services. On Kubernetes clusters, CoreDNS is generally run as a ConfigMap, Deployment, and Service on the cluster. The kubelets are configured to tell individual containers to use the CoreDNS Service's IP to resolve DNS names.

The CoreDNS Kubernetes plugin supports:

- A, SRV, and PTR records for regular and headless services
- A records for named endpoints that are part of a service
- A records for pods as described in a spec (disabled by default)
- TXT record for discovering the DNS schema version in use

The official coredns repo contains a convenience script and yaml templates that can be used for semi-auto deployment:
https://github.com/coredns/deployment/tree/master/kubernetes

However, the CoreDNS manifests will need to be customized with our local cluster information so we will forgo using the scripted install.

2.a. Corefile

To configure CoreDNS to provide Kubernetes service discovery, you have to set up a Corefile, the CoreDNS configuration file. The file consists of one or more Server Blocks. Each Server Block lists one or more Plugins.

Server Blocks

Each Server Block starts with the zones the Server should be authoritative for and an optional port number to listen on. A Server Block that is responsible for all zones below the root zone on the default port would be:

```
.:53 {
    # Plugins defined here.
}
```

This server should handle every possible query.

Plugins & Directives

Each Server Block specifies plugins that should be chained for the Server. Plugins provide functionality such as the ability to log queries to standrd output, proxying DNS messages upstream, or enabling a health check endpoint.

Most plugins allow more configuration with Directives and/or nested Plugin Blocks. The *hosts* plugin, for example, enables serving zone data from a /etc/hosts style file and uses both Directives and a Plugin Block:

```
...

    hosts [FILE [ZONES...]] {
        [INLINE]
        fallthrough [ZONES...]
    }

...
```

A full list of plugins and their associated Directives/Plugin Blocks can be found here: https://coredns.io/plugins/

Corefile ConfigMap

To begin, create a working directory where we can create our Kubernetes configuration manifests for the CoreDNS service:

```
ubuntu@nodea:~$ mkdir -p coredns

ubuntu@nodea:~$
```

```
ubuntu@nodea:~$ cd coredns/

ubuntu@nodea:~/coredns$
```

For this course, a minimally-configured ConfigMap containing the Corefile is provided from an AWS S3 bucket; curl the file and examine it:

```
ubuntu@nodea:~/coredns$ curl https://s3.us-east-2.amazonaws.com/rx-m-kubernetes/coredns-cm.yaml

apiVersion: v1
kind: ConfigMap
metadata:
  name: coredns
  namespace: kube-system
data:
  Corefile: |
    .:53 {
        errors
        health
        kubernetes cluster.local in-addr.arpa ip6.arpa {
          endpoint http://nodea:8080
          pods insecure
          upstream
          fallthrough in-addr.arpa ip6.arpa
        }
        proxy . /etc/resolv.conf
        cache 30
        reload
    }
ubuntu@nodea:~/coredns$
```

The Corefile defines a single Server Block using the default port: `.:53` and uses several plugins:

- errors - any errors encountered during the query processing will be printed to standard output
- health - enables a health check endpoint
- kubernetes - enables the reading zone data from a Kubernetes cluster
  - `cluster.local in-addr.arpa ip6.arpa` ([ZONES...] Directive) - handle all queries in the specified zones, also handle all in-addr.arpa PTR requests
  - endpoint - specifies the URL for a K8s API endpoint (required because we aren't using a Service Account)
  - `pods POD-MODE` - sets the mode for handling IP-based pod A records
    - `insecure` - always return an A record with IP from request

- proxy - it the current form, a simple reverse proxy in the format `proxy FROM TO`
  - `FROM` is the base domain to match for the request to be proxied
  - `TO` is the destination endpoint to proxy to
- cache - all records except zone transfers and metadata records will be cached for the TTL value (30s)
- reload - plugin periodically checks if the Corefile has changed by reading it and calculating its MD5 checksum
  - If the file has changed, it reloads CoreDNS with the new Corefile

The CoreDNS ConfigMap, Service and Deployment are typically created and run in a separate namespace from the normal applications running on the cluster. In particular the "kube-system" namespace is the standard namespace for cluster services. Verify kube-system namespace exits and create it if does not:

```
ubuntu@nodea:~/coredns$ kubectl get namespace

NAME          STATUS    AGE
default       Active    18m
kube-public   Active    18m
kube-system   Active    18m
ubuntu@nodea:~/coredns$
```

OPTIONAL

ONLY if you do not see `kube-system` in the output above: add the kube-system namespace using a config:

```
ubuntu@nodea:~/coredns$ vim ksns.yaml
ubuntu@nodea:~/coredns$ cat ksns.yaml

apiVersion: v1
kind: Namespace
metadata:
  name: kube-system
ubuntu@nodea:~/coredns$
```

```
ubuntu@nodea:~/coredns$ kubectl create -f ksns.yaml

namespace "kube-system" created
ubuntu@nodea:~/coredns$
```

```
ubuntu@nodea:~/coredns$ kubectl get namespace

NAME          STATUS    AGE
default       Active    18m
kube-public   Active    18m
kube-system   Active    18m
ubuntu@nodea:~/coredns$
```

END OPTIONAL

Save the ConfigMap appending the previous curl command with: `-o coredns-cm.yaml`; **then edit the spec replacing "nodea" in the endpoint directive:** `endpoint http://nodea:8080` **with the IP of your master node**. Without this edit, the CoreDNS pod will fail as the only reference to "nodea" is in our host's /etc/hosts file and the CoreDNS pod has its own /etc/hosts file which *does not* include our edits!

```
ubuntu@nodea:~/coredns$ curl https://s3-us-east-2.amazonaws.com/rx-m-kubernetes/coredns-cm.yaml -o coredns-cm.yaml

  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   295  100   295    0     0    420      0 --:--:-- --:--:-- --:--:--   420

ubuntu@nodea:~/coredns$ vim coredns-cm.yaml
ubuntu@nodea:~/coredns$ cat coredns-cm.yaml
```

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: coredns
  namespace: kube-system
data:
  Corefile: |
    .:53 {
        errors
        health
        kubernetes cluster.local in-addr.arpa ip6.arpa {
          endpoint http://172.31.28.198:8080               # use the IP of your nodea not the example IP!
          pods insecure
          upstream
```

```
          fallthrough in-addr.arpa ip6.arpa
        }
        proxy . /etc/resolv.conf
        cache 30
        reload
    }
ubuntu@nodea:~/coredns$
```

Create the ConfigMap:

```
ubuntu@nodea:~/coredns$ kubectl create -f coredns-cm.yaml

configmap/coredns created

ubuntu@nodea:~/coredns$ kubectl -n kube-system get cm

NAME                                 DATA   AGE
coredns                              1      13s
extension-apiserver-authentication   0      68m
ubuntu@nodea:~/coredns$
```

2.b. CoreDNS Deployment

Inspect the CoreDNS deployment, also available on S3:

```
ubuntu@nodea:~/coredns$ curl https://s3.us-east-2.amazonaws.com/rx-m-kubernetes/coredns-dep.yaml

apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: coredns
  namespace: kube-system
  labels:
    k8s-app: kube-dns
    kubernetes.io/name: "CoreDNS"
spec:
  replicas: 1
  selector:
    matchLabels:
      k8s-app: kube-dns
```

```
  template:
    metadata:
      labels:
        k8s-app: kube-dns
    spec:
      containers:
      - name: coredns
        image: coredns/coredns
        imagePullPolicy: IfNotPresent
        args: [ "-conf", "/etc/coredns/Corefile" ]
        volumeMounts:
        - name: config-volume
          mountPath: /etc/coredns
          readOnly: true
        ports:
        - containerPort: 53
          name: dns
          protocol: UDP
        - containerPort: 53
          name: dns-tcp
          protocol: TCP
        - containerPort: 9153
          name: metrics
          protocol: TCP
        securityContext:
          allowPrivilegeEscalation: false
          capabilities:
            add:
            - NET_BIND_SERVICE
            drop:
            - all
          readOnlyRootFilesystem: true
        livenessProbe:
          httpGet:
            path: /health
            port: 8080
            scheme: HTTP
          initialDelaySeconds: 60
          timeoutSeconds: 5
          successThreshold: 1
          failureThreshold: 5
      dnsPolicy: Default
      volumes:
```

```
        - name: config-volume
          configMap:
            name: coredns
            items:
            - key: Corefile
              path: Corefile
ubuntu@nodea:~/coredns$
```

This is an interesting deployment leveraging many useful features.

No edits are required, you can simply apply the spec from the remote source:

```
ubuntu@nodea:~/coredns$ kubectl apply -f https://s3.us-east-2.amazonaws.com/rx-m-kubernetes/coredns-dep.yaml

deployment.extensions/coredns created
ubuntu@nodea:~/coredns$
```

```
ubuntu@nodea:~/coredns$ kubectl -n kube-system get deploy,po

NAME                            READY   UP-TO-DATE   AVAILABLE   AGE
deployment.extensions/coredns   1/1     1            1           16s

NAME                           READY   STATUS    RESTARTS   AGE
pod/coredns-786dc59d56-hcq8l   1/1     Running   0          16s
ubuntu@nodea:~/coredns$
```

2.c. CoreDNS Service

Now we can create the Service spec for the CoreDNS service:

```
ubuntu@nodea:~/coredns$ curl https://s3.us-east-2.amazonaws.com/rx-m-kubernetes/coredns-svc.yaml

apiVersion: v1
kind: Service
metadata:
  name: kube-dns
  namespace: kube-system
```

```
  labels:
    k8s-app: kube-dns
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: "CoreDNS"
spec:
  selector:
    k8s-app: kube-dns
  clusterIP: 10.0.0.10
  ports:
  - name: dns
    port: 53
    protocol: UDP
  - name: dns-tcp
    port: 53
    protocol: TCP
ubuntu@nodea:~/coredns$
```

Note the reserved IP of `10.0.0.10` for our cluster DNS.

Now create the CoreDNS service:

```
ubuntu@nodea:~/coredns$ kubectl apply -f https://s3.us-east-2.amazonaws.com/rx-m-kubernetes/coredns-svc.yaml

service/kube-dns created
ubuntu@nodea:~/coredns$
```

```
ubuntu@nodea:~/coredns$ kubectl get service --all-namespaces

NAMESPACE     NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)         AGE
default       kubernetes   ClusterIP   10.0.0.1     <none>        443/TCP         2h
default       websvc       ClusterIP   10.0.13.26   <none>        80/TCP          1h
kube-system   kube-dns     ClusterIP   10.0.0.10    <none>        53/UDP,53/TCP   57s
ubuntu@nodea:~/coredns$
```

Note that the service is called "kube-dns" *not* "coredns"; this is not by mistake.

```
ubuntu@nodea:~/coredns$ kubectl describe service kube-dns --namespace=kube-system
```

```
Name:              kube-dns
Namespace:         kube-system
Labels:            k8s-app=kube-dns
                   kubernetes.io/cluster-service=true
                   kubernetes.io/name=CoreDNS
Annotations:       kubectl.kubernetes.io/last-applied-configuration=
{"apiVersion":"v1","kind":"Service","metadata":{"annotations":{},"labels":{"k8s-app":"kube-
dns","kubernetes.io/cluster-service":"true","kubernetes.io/n...
Selector:          k8s-app=kube-dns
Type:              ClusterIP
IP:                10.0.0.10
Port:              dns  53/UDP
TargetPort:        53/UDP
Endpoints:         10.1.22.4:53
Port:              dns-tcp  53/TCP
TargetPort:        53/TCP
Endpoints:         10.1.22.4:53
Session Affinity:  None
Events:            <none>
ubuntu@nodea:~/coredns$
```

Note the "Endpoints" IP above, as we will use it in a subsequent command.

Check with the cluster to see if it knows about DNS:

```
ubuntu@nodea:~/coredns$ kubectl cluster-info

Kubernetes master is running at http://nodea:8080
CoreDNS is running at http://nodea:8080/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
ubuntu@nodea:~/coredns$
```

Perfect, DNS is up and running!

Take a look at the Docker container running in the DNS pod node (you may need to run this command on nodeb, depending on where kubernetes decided to place your coredns pod):

```
ubuntu@nodea:~/coredns$ docker container ls --filter=ancestor=coredns/coredns

CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
NAMES
24ab8f5b3c7e        da1adafc0e78        "/coredns -conf /etc…"   13 minutes ago      Up 13 minutes
k8s_coredns_coredns-7b664f6d66-lmwqv_kube-system_a4ce97a6-87a8-11e8-bcb6-000c2927de8a_0
ubuntu@nodea:~/coredns$
```

Dump the logs of your coredns pod:

```
ubuntu@nodea:~/coredns$ kubectl -n kube-system get pod
NAME                       READY     STATUS    RESTARTS   AGE
coredns-7b664f6d66-cx7vr   1/1       Running   0          5m

ubuntu@nodea:~/coredns$ kubectl -n kube-system logs $(kubectl -n kube-system get pod -o name)

.:53
2019-03-31T03:52:01.950Z [INFO] plugin/reload: Running configuration MD5 = da2297bf3efd33e10b06e29f75eff43b
2019-03-31T03:52:01.950Z [INFO] CoreDNS-1.4.0
2019-03-31T03:52:01.950Z [INFO] linux/amd64, go1.12, 8dcc7fc
CoreDNS-1.4.0
linux/amd64, go1.12, 8dcc7fc
2019-03-31T03:52:01.951Z [INFO] plugin/reload: Running configuration MD5 = da2297bf3efd33e10b06e29f75eff43b
ubuntu@nodea:~/coredns$
```

Not much here, but then again, we didn't include the "log" plugin which would make coredns more verbose; from the docs:

> Description
>
> By just using log you dump all queries (and parts for the reply) on standard output. Options exist to tweak the output a little.
>
> Note that for busy servers this will incur a performance hit.

Now lets try doing some lookups with the new DNS server using its pod ip (endpoint from above) as the DNS host target:

> If you forgot to note the endpoint above, which is used in this query, you can use "kubectl -n kube-system get endpoints kube-dns" to retrive it.

```
ubuntu@nodea:~/coredns$ nslookup websvc.default.svc.cluster.local. 10.1.22.4

Server:         10.1.22.4
Address:        10.1.22.4#53

Name:   websvc.default.svc.cluster.local
Address: 10.0.13.26
ubuntu@nodea:~/coredns$
```

nslookup returned the service IP, 10.0.13.26 in the above case, nice.

Service names are fully qualified in Kubernetes by appending their namespace, the "svc" subdomain, and the domain name of the cluster. The cluster domain name is set through the kubelet (e.g. `--cluster_domain=cluster.local`) and defaults to cluster.local.

Try using dig to look up some other service IPs:

```
ubuntu@nodea:~/coredns$ dig +short websvc.default.svc.cluster.local. @10.1.22.4

10.0.13.26
ubuntu@nodea:~/coredns$
```

```
ubuntu@nodea:~/coredns$ dig +short kubernetes.default.svc.cluster.local. @10.1.22.4

10.0.0.1
ubuntu@nodea:~/coredns$
```

```
ubuntu@nodea:~/coredns$ dig +short kube-dns.kube-system.svc.cluster.local. @10.1.22.4

10.0.0.10
ubuntu@nodea:~/coredns$
```

3. Test lookups from inside a pod

So our DNS server is up and running but our overall cluster configuration is not yet optimal. Let's try accessing the DNS server from inside a pod again. Shell into your clientpod:

```
ubuntu@nodea:~/coredns$ cd ~

ubuntu@nodea:~$ kubectl exec -it clientpod bash
/ #
```

Now rerun the nslookup command you ran on the host by try using not only the DNS pod IP but also the DNS service IP:

```
/ # nslookup websvc.default.svc.cluster.local. 10.1.22.4

Server:         10.1.22.4
Address:        10.1.22.4#53

Name:   websvc.default.svc.cluster.local
Address: 10.0.13.26
/ #
```

```
/ # nslookup websvc.default.svc.cluster.local. 10.0.0.10
Server:         10.0.0.10
Address:        10.0.0.10#53

Name:   websvc.default.svc.cluster.local
Address: 10.0.13.26
/ #
```

They both work, great. Now try a lookup with no DNS target:

```
/ # nslookup websvc.default.svc.cluster.local.

Server:         172.31.0.2
Address:        172.31.0.2#53
```

```
** server can't find websvc.default.svc.cluster.local: NXDOMAIN

/ #
```

This fails because our default DNS server IP is bogus. Examine the container's resolv.conf:

```
/ # cat /etc/resolv.conf

nameserver 172.31.0.2
search us-east-2.compute.internal
/ #
```

The old IP was just a place holder.

Update the `resolv.conf` to use the correct DNS service VIP and test some lookups:

```
/ # vi /etc/resolv.conf
/ # cat /etc/resolv.conf

nameserver 10.0.0.10
nameserver 172.31.0.2
search us-east-2.compute.internal
/ #
```

```
/ # nslookup websvc.default.svc.cluster.local.

Server:         10.0.0.10
Address:        10.0.0.10#53

Name:   websvc.default.svc.cluster.local
Address: 10.0.13.26
/ #
```

An improvement, now we don't need to specify the DNS server address. Now try looking up a service without the qualifying suffix:

```
/ # nslookup websvc

Server:         10.0.0.10
Address:        10.0.0.10#53

** server can't find websvc: NXDOMAIN
/ #
```

This fails because the default search suffix is not set correctly. Update the `resolv.conf` and try again:

```
/ # vi /etc/resolv.conf
/ # cat /etc/resolv.conf

nameserver 10.0.0.10
nameserver 172.31.0.2
search default.svc.cluster.local.
/ #
```

```
/ # nslookup websvc

Server:         10.0.0.10
Address:        10.0.0.10#53

Name:   websvc.default.svc.cluster.local
Address: 10.0.13.26
/ #
```

Now we can achieve our original goal, having one pod just reach another using a service name:

```
/ # wget -qO- websvc
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
```

```
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
/ #
```

```
/ # exit

ubuntu@nodea:~$
```

Everything is working perfectly in this pod but all of the new pods we create will still get a broken `resolv.conf`.

Let's update the kubelet config file to fix the search suffix and the DNS server IP.

4. Update the kubelet configuration

To update the kubelet, stop the currently running kubelet and give it the new `clusterDNS:` and `clusterDomain:` parameters:

```
W0322 17:35:23.613487  106540 container_manager_linux.go:731] MemoryAccounting not enabled for pid: 106540

^C
```

```
ubuntu@nodea:~$
```

```
ubuntu@nodea:~$ vim nodea.yaml
ubuntu@nodea:~$ cat nodea.yaml

apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
authentication:
  anonymous:
    enabled: true
cgroupDriver: cgroupfs
failSwapOn: true
clusterDNS:
- 10.0.0.10
clusterDomain: cluster.local
ubuntu@nodea:~$
```

```
ubuntu@nodea:~$ sudo $HOME/k8s/_output/bin/kubelet \
--kubeconfig=nodea.conf \
--config=nodea.yaml \
--allow-privileged=true \
--runtime-cgroups=/systemd/machine.slice \
--kubelet-cgroups=/systemd/machine.slice \
--pod-infra-container-image=k8s.gcr.io/pause:3.1

...
```

Perform the same operation on the other node:

```
ubuntu@nodeb:~$ vim nodeb.yaml
ubuntu@nodeb:~$ cat nodeb.yaml

apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
authentication:
  anonymous:
```

```
    enabled: true
cgroupDriver: cgroupfs
failSwapOn: true
clusterDNS:
- 10.0.0.10
clusterDomain: cluster.local
ubuntu@nodeb:~$
```

```
ubuntu@nodeb:~$ sudo $HOME/kube-bin/kubelet \
--kubeconfig=nodeb.conf \
--config=nodeb.yaml \
--allow-privileged=true \
--runtime-cgroups=/systemd/machine.slice \
--kubelet-cgroups=/systemd/machine.slice \
--pod-infra-container-image=k8s.gcr.io/pause:3.1

...
```

Delete your clientpod, recreate it and try out DNS:

```
ubuntu@nodea:~$ kubectl delete pod clientpod --now

pod "clientpod" deleted

ubuntu@nodea:~$ kubectl create -f clientpod.yaml

pod/clientpod created

ubuntu@nodea:~$ kubectl get pod

NAME                              READY     STATUS    RESTARTS   AGE
clientpod                         1/1       Running   0          33m
nginx-deployment-171375908-0bggz   1/1       Running   0          49m
nginx-deployment-171375908-g1cms   1/1       Running   0          49m
ubuntu@nodea:~$
```

```
ubuntu@nodea:~$ kubectl exec -it clientpod sh

/ # wget -qO- websvc

<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>

/ # cat /etc/resolv.conf

nameserver 10.0.0.10
search default.svc.cluster.local svc.cluster.local cluster.local us-west-2.compute.internal
options ndots:5

/ #  exit

ubuntu@nodea:~$
```

Congratulations, you have built a full-featured Kubernetes cluster by hand! Now when you have problems in k8s clusters installed by a tool, you'll know who's responsible for what; how to discover problems and how to get the configuration working again.