CKA Curriculum Part 9 - Networking

Understand the networking configuration on cluster nodes

Figure 11.16. For pods on different nodes to communicate, the bridges need to be connected somehow.

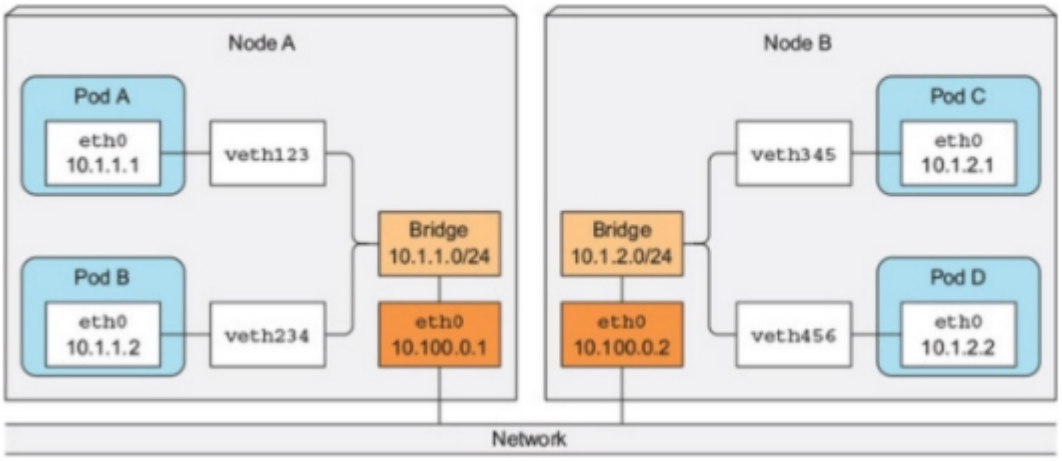


Image source: http://www.kubernet.io/networking_cka_1.9.0.html

If configuring nodes from scratch, typically the only interface that the user creates is eth0. The installation of Kubernetes, Docker and a CNI will create additional interfaces

Each host is responsible for one subnet of the the CNI range. In this example, the left host is responsible for 10.1.1.0/24, and the right host 10.1.2.0/24.

Understand pod networking concepts

Every Pod gets its own IP address. This means you do not need to explicitly create links between Pods and you almost never need to deal with mapping container ports to host ports. This creates a clean, backwards-compatible model where Pods can be treated much like VMs or physical hosts from the perspectives of port allocation, naming, service discovery, load balancing, application configuration, and migration.

Kubernetes imposes the following fundamental requirements on any networking implementation (barring any intentional network segmentation policies):

- Pods on a node can communicate with all pods on all nodes without NAT
- Agents on a node (e.g. system daemons, kubelet) can communicate with all pods on that node

Note: For those platforms that support Pods running in the host network (e.g. Linux):

- Pods in the host network of a node can communicate with all pods on all nodes without NAT

Understand service networking

Pods are ephemeral. Therefore placing these behind a service which provides a stable, static entrypoint is a fundamental use of the kubernetes service object. To reiterate, services take the form of the following:

- ClusterIP** - **Internal only
- LoadBalancer - External, requires cloud provider to provide one
- NodePort - External, requires access the nodes directly
- Ingress resource - L7 An Ingress can be configured to give services externally-reachable URLs, load balance traffic, terminate SSL, and offer name based virtual hosting. An Ingress controller is responsible for fulfilling the

Ingress, usually with a loadbalancer, though it may also configure your edge router or additional frontends to help handle the traffic.

Deploy and Configure a Network load balancer

Load balancer is a type of service, and is configured as such. An example being:

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
 run: my-nginx
spec:
  type: LoadBalancer
  ports:
  - port: 80
 protocol: TCP
 name: http
 targetPort: 80
  selector:
 run: my-nginx
```

**Port **- The port the LB itself will listen on

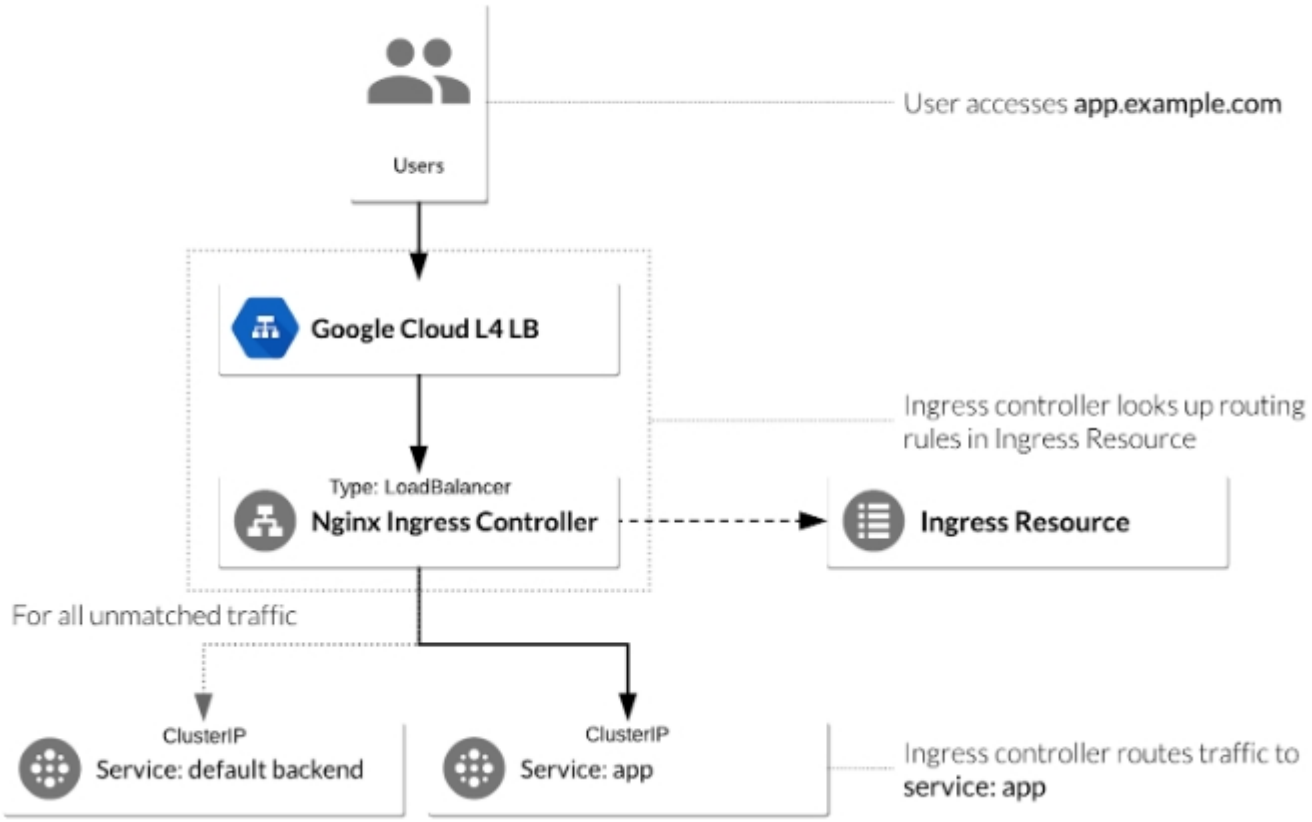
**Target port **- The port on the pods to forward connections to

Selector - The pods that we want to load balance to. In the above example pods matching the “run: my-nginx” label.

Know how to use Ingress rules

Ingress exposes HTTP and HTTPS routes from outside the cluster to services within a cluster. Ingress consists of two components. Ingress Resource is a collection of rules for the inbound traffic to reach Services. These are Layer 7 (L7) rules that allow hostnames (and optionally paths) to be directed to specific Services in Kubernetes. The second component is the Ingress Controller which acts upon the rules set by the Ingress Resource, typically via an HTTP or L7 load balancer. It is vital that both pieces are properly configured to route traffic from an outside client to a Kubernetes Service.

>>>>> gd2md-html alert: inline image link here (to images/CKA-Curriculum1.png). Store image on your image server and adjust path/filename if necessary.
(Back to top)(Next alert)
>>>>>



The following yaml creates two ingress rules for the website foo.bar.com

The default path will direct traffic to the service “default-service” which listens on port 80

Paths ending in /foo will direct traffic to the service “service1” which listens on port 4200

Paths ending in /bar will direct traffic to the service “service2” which listens on port 8080

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: simple-fanout-example
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /
        backend:
          serviceName: default-service
          servicePort: 80
      - path: /foo
        backend:
          serviceName: service1
          servicePort: 4200
      - path: /bar
        backend:
          serviceName: service2
          servicePort: 8080
```

Know how to configure and use the cluster DNS.

As of 1.13, coredns has replace kube-dns as the facilitator of cluster DNS and runs as pods.

```
kubectl get pods -n kube-system
NAME                                   READY   STATUS    RESTARTS   AGE
coredns-fb8b8dccf-jks6g                1/1     Running   4          8d
```

```
coredns-fb8b8dccf-hxbhn                  1/1     Running   9          10d
```

To view the DNS configuration of a pod, spin one up and inspect the /etc/resolv.conf:

```
kubectl run busybox --image=busybox -- sleep 9000
kubectl exec -it busybox-f9db74d47-jkfp5 sh
/ # cat /etc/resolv.conf
nameserver 10.96.0.10
search default.svc.cluster.local svc.cluster.local cluster.local virtualthoughts.co.uk
options ndots:5
```

“10.96.0.10” references the “Kube-DNS” service

“Default.svc.cluster.local” References the namespace with the suffic svc.cluster.local.

All pods are provisioned a DNS record and are in the format of

[Pod IP separated by dashes].[Namespace].[Base Domain Name]

For example:

```
10-244-2-42.default.pod.cluster.local
```

Services follow a similar pattern

[Service Name].[Namespace].[Base Domain Name]

For example:

```
Kube-dns.kube-system.svc.cluster.local
```

Headless services are those without a cluster ip, but will respond with a list of IP’s of pods that are applicable at that particular moment in time.

```
apiVersion: v1
kind: Service
metadata:
  name: test-headless
spec:
  clusterIP: None
  ports:
  - port: 80
    targetPort: 80
  selector:
    app: web-headless
```

We can modify the default behavior of the pod dns configuration in the yaml file:

```
apiVersion: v1
kind: Pod
metadata:
  namespace: default
  name: dns-example
spec:
  containers:
    - name: test
      image: nginx
  dnsPolicy: "None"
  dnsConfig:
    nameservers:
      - 8.8.8.8
    searches:
      - ns1.svc.cluster.local
      - my.dns.search.suffix
    options:
      - name: ndots
        value: "2"
      - name: edns0
```

Understand CNI

A Kubernetes cluster cannot function without a chosen CNI. Which CNI is chosen will be dependent on your available options, infrastructure, and other factors.

https://kubernetes.io/docs/concepts/cluster-administration/addons/#networking-and-network-policy

As a generalisation, CNI's provide some kind of network overlay. But each have their own features, limitations and considerations.