IBM Cloud Private Network

IBM Cloud

Networking and Kubernetes

Kubernetes and Calico

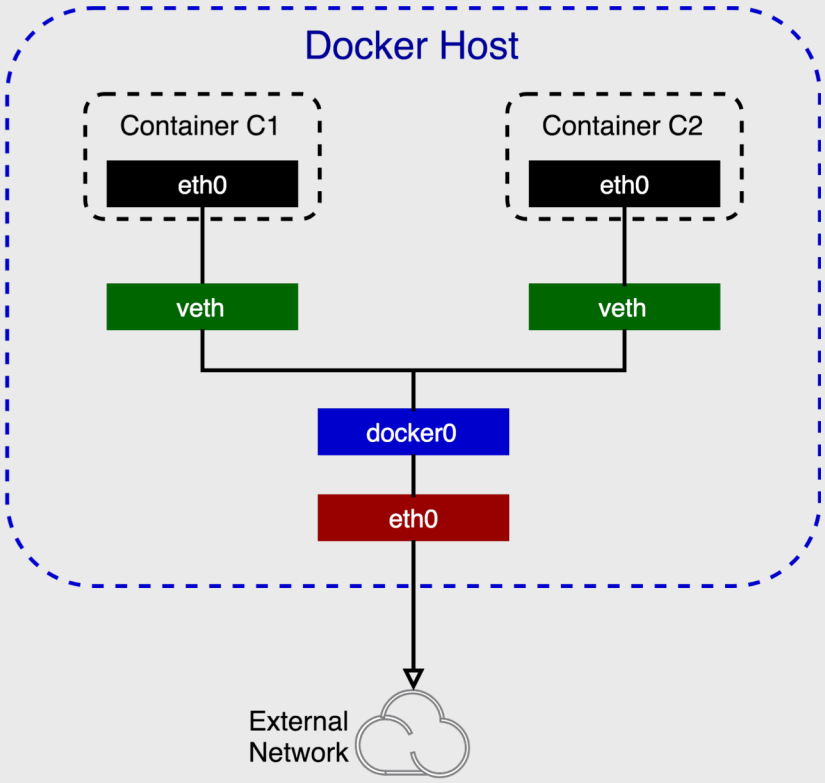
Network Policy

Ingress

Networking within Kubernetes

Pods, Services and Beyond

Docker Network Model



Docker by Default

- Uses host-private
- A virtual bridge is created ***docker0*** allocated with a subnet from one of the private address blocks
- For each Docker container, it allocates a veth (virtual ethernet device) and attaches it to the bridge
- The veth is mapped to appear as eth0 in the container
- This interface is given an IP address from the bridge address range
- Only containers (by default) can communicate with each other
- For Docker containers to communicate across nodes, ports on the host must be allocated (on the host's IP) that are in turn forwarded / proxied to the containers

Kubernetes Network Model

Seeking to cut through the complications of the Docker Network Model

Kubernetes Network Fundamentals

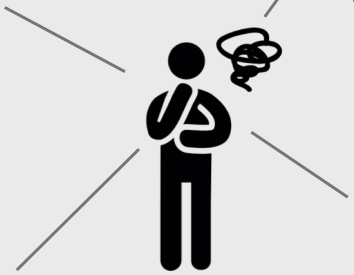
- All pods can communicate with all other pods without NAT
- All pods can communicate with all containers (and vice-versa) without NAT
- The IP the pod sees itself as is the same IP that the others see it as
- K8s applies addresses at the pod, contains within a pod share their network namespaces and IP address and thus can communicate via localhost
- Containers with the pod must coordinate port usage

Implementing the Kubernetes Networking Model with ICP

There are many ways the network model can be implemented

Flannel is one in a list of common and simple overlay networks for the easy configuration of layer 3 network fabric

Project Calico is an policy based open source container networking provider and network policy engine (This is the chosen overlay for ICP)



NSX-T can provide network virtualization for a multi-cloud and multi-hypervisor environment and is focused on emerging application frameworks and architectures that have heterogeneous endpoints and technology stacks

Public platforms such as GCE also offer their own brand of network overlay that can also provide points of integration for the ICP cluster network

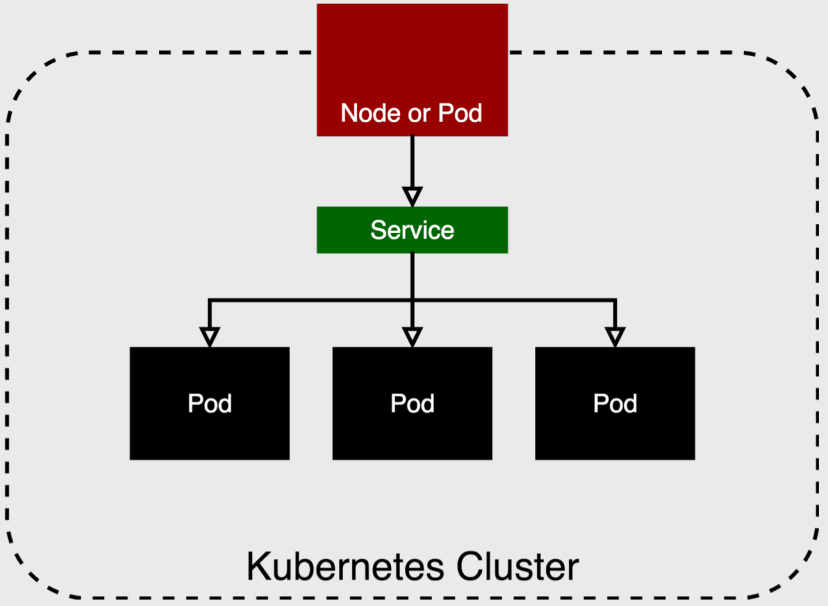
Kubernetes Service

Pods in Kubernetes mortal and transient, they can come and go

Pods need a dynamic way of communicating amongst each other

Services are object abstraction that define a policy to access a pod or set of pods

Consider a replicaset of pod running as a backend. The frontend will access this set of pods using a service. This decouples it from managing which pod (or the complete list of pods) in the backend it is actually interfacing with.



Logical Representation

Kubernetes Service: Example

Our example manifest to deploy our NGINX ReplicaSet

Includes the “app” label “skol-nginx”

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: skol-nginx
  replicas: 3
  template:
    metadata:
      labels:
        app: skol-nginx
    spec:
      containers:
      - name: skol-nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

```
kind:
ServiceapiVersion: v1
metadata:
  name: my-nginx-service
spec:
  selector:
    app: skol-nginx
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```

Creates a Service object named “my-nginx-service” that targets TCP port 80 our Pod with the "app” label of “skol-nginx

By default the Service is assigned the ”cluster IP”

A corresponding Endpoint object named “my-nginx-service” is created and updated constantly

Service Types

In Kubernetes, Service Types change how the services is deployed and how it behaves:

- **ClusterIP:** The service is exposed internally to the cluster on the Cluster IP (this is the default)
- **NodePort:** The service is exposed on each of the cluster Nodes' IP at the port specified by "NodePort". An associated ClusterIP service is automatically created. The service is thus accessible via <any Nodes IP>:<NodePort>, as well as, internally via the ClusterIP and specified or automatically assigned.
- **LoadBalancer:** Exposes the service externally in the case of using a cloud provider's LoadBalancer. Associated NodePort and ClusterIP Services are created.
- **ExternalName:** Maps the service to the "externalName" field by returning a CNAME record with its value. Note: No proxy of any kind is configured.

CNAME: A Canonical Name or **CNAME record** is a type of DNS **record** that maps an alias name to a true or canonical domain name.

Kube-dns

Every Service defined in the cluster gets a CNAME record in the cluster DNS
Only Services can be resolved by the DNS service
Name resolution is based upon namespace

```
web-terminal@webbterm-ibm-webterminal-6bf989c956-nrtfv:~$ kubectl get svc
NAME                        TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)          AGE
csi-attacher-nfsplugin      ClusterIP   10.0.0.104    <none>        12345/TCP        1d
details                     ClusterIP   10.0.0.144    <none>        9080/TCP         5d
dx9-digitalexperience       ClusterIP   10.0.0.218    <none>        30015/TCP        14d
kubernetes                  ClusterIP   10.0.0.1      <none>        443/TCP          54d
my-nginx-service            ClusterIP   10.0.0.248    <none>        80/TCP           32d
my-other-service            ClusterIP   10.0.0.252    <none>        80/TCP           32d
nginx-np-service            NodePort    10.0.0.225    <none>        80:31357/TCP     32d
productpage                 ClusterIP   10.0.0.45     <none>        9080/TCP         5d
ratings                     ClusterIP   10.0.0.126    <none>        9080/TCP         5d
reviews                     ClusterIP   10.0.0.157    <none>        9080/TCP         5d
webbterm-ibm-webterminal    NodePort    10.0.0.204    <none>        3000:31864/TCP   6d
web-terminal@webbterm-ibm-webterminal-6bf989c956-nrtfv:~$ ping reviews
PING reviews.default.svc.cluster.local (10.0.0.157) 56(84) bytes of data.
```

Kube-dns

The service name can also be resolved from outside of the name space by appending the its namespace

```
web-terminal@webbterm-ibm-webterminal-6bf989c956-nrtfv:~$ kubectl get svc -n kube-system
NAME                                TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)             AGE
catalog-ui                          NodePort    10.0.0.76     <none>        4000:32058/TCP      54d
default-backend                     ClusterIP   10.0.0.205    <none>        80/TCP              54d
mongodb                             NodePort    10.0.0.46     <none>        27017:30077/TCP     54d
.
.
.
platform-ui                         ClusterIP   10.0.0.202    <none>        3000/TCP            54d
service-catalog-apiserver           NodePort    10.0.0.251    <none>        443:30443/TCP       54d
tiller-deploy                       ClusterIP   10.0.0.9      <none>        44134/TCP           54d
unified-router                      ClusterIP   10.0.0.177    <none>        9090/TCP            54d
web-terminal@webbterm-ibm-webterminal-6bf989c956-nrtfv:~$ ping mongodb.kube-system
PING mongodb.kube-system.svc.cluster.local (10.0.0.46) 56(84) bytes of data.
```

Kube-dns

The IP address and search domains are automatically added to the containers /etc/resolve.conf
This can be observed by shelling into a running pod

```
web-terminal@webbterm-ibm-webterminal-6bf989c956-nrtfv:~$ cat /etc/resolv.conf
nameserver 10.0.0.10
search default.svc.cluster.local svc.cluster.local cluster.local
options ndots:5
```

Kubernetes and Calico

How it works

Project Calico

A new approach to virtual networking and network security for containers, VMs, and bare metal services, that provides a rich set of security enforcement capabilities running on top of a highly scalable and efficient virtual network

- The calico/node Docker container runs on the Kubernetes master and each Kubernetes node in the cluster
- The calico-cni plugin integrates directly with the Kubernetes kubelet process on each node to discover which pods have been created, and adds them to Calico networking
- The calico/kube-policy-controller container runs as a pod on top of Kubernetes and implements the NetworkPolicy API
- Calico makes use of Layer 3

Calico network policy enforcement ensures that the only packets that flow to and / or from a workload are the ones the developer expects

Calico: etcd

The backend data store for Calico is etcd

With IBM Cloud Private, by default Calico shares the instance of etcd with the K8s cluster

Calico stores all of the configurations it requires in this datastore

You can examine the information that calico provides by using etcdctl

Calico: BIRD

BIRD is a BGP routing daemon running on every node in the ICP K8s Cluster as a DaemonSet

Calico leverages BGP to propagate routes between the nodes

BGP (Border Gateway Protocol) is a scalable, complex and secure routing protocol

Calico: ConfD

A simple configuration management tool that

Runs inside of the Calico node container

Reads values (BIRD configuration for Calico) from etcd and writes them to files on disk

It will loop through pools (networks / subnetworks) apply configuration data (ie. CIDR keys) and assemble it in a way that BIRD can use

Calico: Felix

The calico-felix daemon runs inside the calico/node and brings the entire solution together performing several roles:

- Reads information from the K8s etcd
- Building the routing table
- Configures the IPTables (kube-proxy mode IPTables)
- Configures IPVS (kube-proxy mode IPVS)

Each and every pod gets an IP address

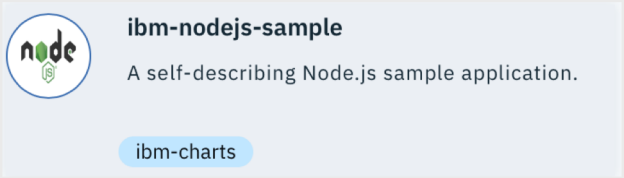
All containers in that pod communicate with each other over localhost

Where does that IP Address come from?

- When the cluster was installed a “network_cidr” was specified in the config.yaml
- By default this is a very large network 10.0.0.1/16 giving a maximum ~65536 hosts
- The range would start at 10.0.0.1 and going through to 10.0.255.255 and subnet mask of 255.255.0.0
- IBM Cloud Private stores these available IPs in etcd

Kubernetes Connectivity

This example uses the below Helm chart from the catalog:



Pods need to communicate
- Need connectivity with the nodes (K8s Hosts)
- Connectivity between Pods
- Connectivity (potentially) with the outside world

(In the beginning) Should be able to get ICMP traffic from any K8s Node to any Pod in the cluster
Example: Deploy an application such as the ibm-nodejs-sample. We should be able to ping the pod from our K8s Nodes

```
$ kubectl get pods -o wide --sort-by="{.spec.nodeName}" -n default
NAME                                      READY STATUS    RESTARTS   AGE   IP             NODE
node-skol-1-ibm-nodejs-s-69f4dd7b79-9wh8l   1/1   Running   0          3d   10.1.196.133   pit-icp-worker-01
$ ssh icpuser@pit-icp-master-01

master-01 $ ping -c 1 10.1.196.133
PING 10.1.196.133 (10.1.196.133) 56(84) bytes of data.
64 bytes from 10.1.196.133: icmp_seq=1 ttl=63 time=0.683 ms
--- 10.1.196.133 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.683/0.683/0.683/0.000 ms
master-01 $
```

kube-proxy

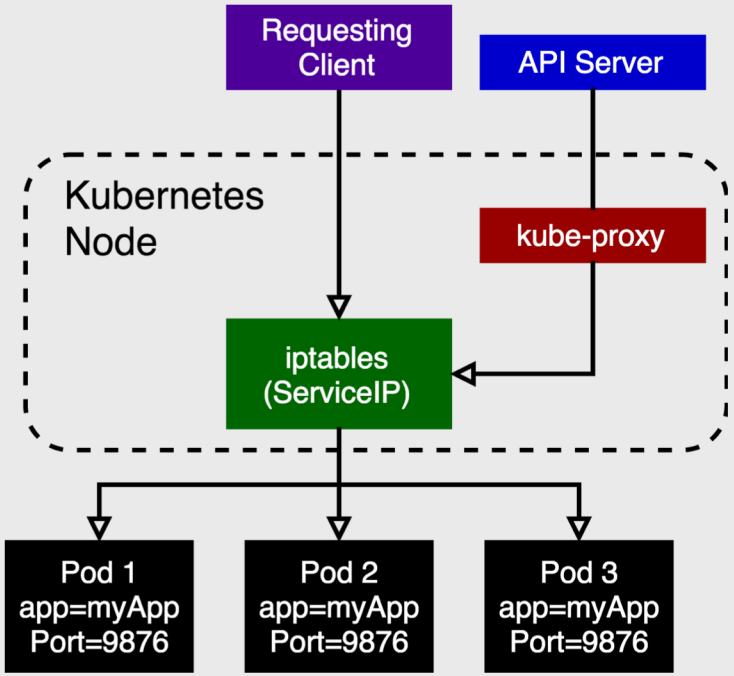
From Wikipedia: A mesh network is a local network topology in which the infrastructure nodes (i.e. bridges, switches and other infrastructure devices) connect directly, dynamically and non-hierarchically to as many other nodes as possible and cooperate with one another to efficiently route data from/to clients.

Every node in the Kubernetes cluster runs a kube-proxy

kube-proxy is responsible for implementing a form of virtual IP for Services (with the exception of the type ExternalName)

A brief history:

- Kubernetes v1.0: Services are a “layer 4” construct (TCP/UDP over IP) and the proxy was purely in userspace
- Kubernetes v1.1: The Ingress API was added (beta) to represent “layer 7” services (HTTP)
- Kubernetes v1.2: IPTables proxy becomes the default operating mode
- Kubernetes v1.8.0-beta.0: IPVS proxy was added



kube-proxy in IPTables mode

Kube-proxy (iptables mode)

kube-proxy in iptables mode writes iptables rules on the nodes for each K8s service

iptables is a policy driven firewall utility that establishes rules for allowing and blocking traffic

The policies are otherwise known as chains

iptables attempts to match each connection attempt to one of these rules, if none match the default action is applied

To view the default behavior run “iptables –L | grep policy” from any node:

```
worker-01 $ iptables -L  | grep policy | grep
Chain
Chain INPUT (policy ACCEPT)
Chain FORWARD (policy ACCEPT)
Chain OUTPUT (policy ACCEPT)
worker-01 $
```

Accept *– Allows the connection*
Drop *– Drops the connection as if the system doesn’t exist*
Reject *– Does not allow a connection but does send back an error*

Kube-proxy (iptables mode)

To view the rules applying to a service you can run iptables-save and grep the output by the service:

```
worker-01 $ iptables-save | grep skol
-A KUBE-NODEPORTS -p tcp -m comment --comment "default/node-skol-1-ibm-nodejs-s:" -m tcp --dport 32626 -
j KUBE-MARK-MASQ
-A KUBE-NODEPORTS -p tcp -m comment --comment "default/node-skol-1-ibm-nodejs-s:" -m tcp --dport 32626 -
j KUBE-SVC-3OW7CSJFSST6HVP2
-A KUBE-SEP-X5SJ3PQGIDTN4W5A -s 10.1.196.133/32 -m comment --comment "default/node-skol-1-ibm-nodejs-s:"
-j KUBE-MARK-MASQ
-A KUBE-SEP-X5SJ3PQGIDTN4W5A -p tcp -m comment --comment "default/node-skol-1-ibm-nodejs-s:" -m tcp -j
DNAT --to-destination 10.1.196.133:3000
-A KUBE-SERVICES ! -s 10.1.0.0/16 -d 10.0.0.19/32 -p tcp -m comment --comment "default/node-skol-1-ibm-
nodejs-s: cluster IP" -m tcp --dport 3000 -j KUBE-MARK-MASQ
-A KUBE-SERVICES -d 10.0.0.19/32 -p tcp -m comment --comment "default/node-skol-1-ibm-nodejs-s: cluster
IP" -m tcp --dport 3000 -j KUBE-SVC-3OW7CSJFSST6HVP2
-A KUBE-SVC-3OW7CSJFSST6HVP2 -m comment --comment "default/node-skol-1-ibm-nodejs-s:" -j KUBE-SEP-
X5SJ3PQGIDTN4W5A
```

For a functioning K8s service you will see (at a minimum) rules for KUBE-SEP, KUBE-SERVICES and KUBE-SVC The above listing also shows the NODEPORTS entries

Routing Tables

Examining the routing table from the node where the pod is scheduled

```
worker-01 $ ip route
default via 172.16.3.1 dev ens192 proto static metric 100
10.1.32.192/26 via 172.16.3.22 dev tunl0 proto bird onlink
10.1.154.192/26 via 172.16.3.25 dev tunl0 proto bird onlink
10.1.181.192/26 via 172.16.3.20 dev tunl0 proto bird onlink
10.1.184.0/26 via 172.16.3.24 dev tunl0 proto bird onlink
blackhole 10.1.196.128/26 proto bird
10.1.196.129 dev cali86e88d8f9b3 scope link
10.1.196.130 dev cali302c48e4faa scope link
10.1.196.131 dev cali94ac78b62f8 scope link
10.1.196.132 dev califefab845e59 scope link
10.1.196.133 dev calibdc76856466 scope link
10.1.222.64/26 via 172.16.3.21 dev tunl0 proto bird onlink
172.16.3.0/26 dev ens192 proto kernel scope link src 172.16.3.23 metric 100
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1
worker-01 $
```

```
calibdc76856466: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
    inet6 fe80::ecee:eeff:feee:eeee  prefixlen 64  scopeid 0x20<link>
    ether ee:ee:ee:ee:ee:ee  txqueuelen 0  (Ethernet)
    RX packets 14458141  bytes 15983357987 (14.8 GiB)
    RX errors 0  dropped 1278  overruns 0  frame 0
    TX packets 6369872  bytes 4396401825 (4.0 GiB)
    TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

The node.js pod deployed previously has IP **10.1.196.133** and a corresponding row highlighted above from the routing table ifconfig shows the associated interface **calibdc76856466**

When the pod pod was instantiated Calico via CNI created an interface and assigned it to the pod

CNI = Container Network Interface

Subnets

From the previous ”ip route” output on one of the our K8s nodes

```
worker-01 $ ip route
default via 172.16.3.1 dev ens192 proto static metric 100
10.1.32.192/26 via 172.16.3.22 dev tunl0 proto bird onlink
10.1.154.192/26 via 172.16.3.25 dev tunl0 proto bird onlink
10.1.181.192/26 via 172.16.3.20 dev tunl0 proto bird onlink
10.1.184.0/26 via 172.16.3.24 dev tunl0 proto bird onlink
blackhole 10.1.196.128/26 proto bird
10.1.196.129 dev cali86e88d8f9b3 scope link
10.1.196.130 dev cali302c48e4faa scope link
10.1.196.131 dev cali94ac78b62f8 scope link
10.1.196.132 dev califefab845e59 scope link
10.1.196.133 dev calibdc76856466 scope link
10.1.222.64/26 via 172.16.3.21 dev tunl0 proto bird onlink
172.16.3.0/26 dev ens192 proto kernel scope link src 172.16.3.23 metric 100
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1
```

- Each of the nodes in the cluster gets a /26 subnet
- Felix found this information for each node (from within etcd)
- Felix then creates the route: ***to get to a pod on another node’s subnet, use the IP address of the hosting node over the tun10 interface***

(for the curious) Running “ifconfig” shows the details for the tun10 interface

This tunnel interface exists because we have IP-in-IP encapsulation configured in Calico

```
tunl0: flags=193<UP,RUNNING,NOARP>  mtu 1430
        inet 10.1.196.128  netmask 255.255.255.255
        tunnel   txqueuelen 1000  (IPIP Tunnel)
        RX packets 1165499  bytes 173178625 (165.1 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 1561747  bytes 1544928258 (1.4 GiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Network Policy and IPTables

Previously it was determined that the node.js pod deployed was assigned the IP **10.1.196.133** and the associated interface **calibdc76856466**

Running an “iptables –L” from the node the corresponding Chain associated with the interface named above is included

```
Chain cali-tw-calibdc76856466 (1 references)
target     prot opt source               destination
ACCEPT     all  --  anywhere             anywhere   /* cali:yTr3HFSbZdVIqTOc */ ctstate RELATED,ESTABLISHED
DROP       all  --  anywhere             anywhere   /* cali:0HqIK9qZ3dkDL7d7 */ ctstate INVALID
MARK       all  --  anywhere             anywhere   /* cali:ROOdt_rB-iT8VKkN */ MARK and 0xfeffffff
cali-pri-kns.default  all -- anywhere  anywhere   /* cali:mzSPjjKDWxlmHBGb */
RETURN     all  --  anywhere             anywhere   /* cali:ofWrcNEg05TT_X2H */ /* Return if profile
accepted */ mark match 0x1000000/0x1000000
DROP       all  --  anywhere             anywhere   /* cali:g90aI6_7hxa9Fz53 */ /* Drop if no profiles
matched */
```

With this Chain in place the associated packets destined for the pod / container are thus identified and passed on their way according to the rules herein

Without this chain the packets are subject to the default policy and would summarily be dropped

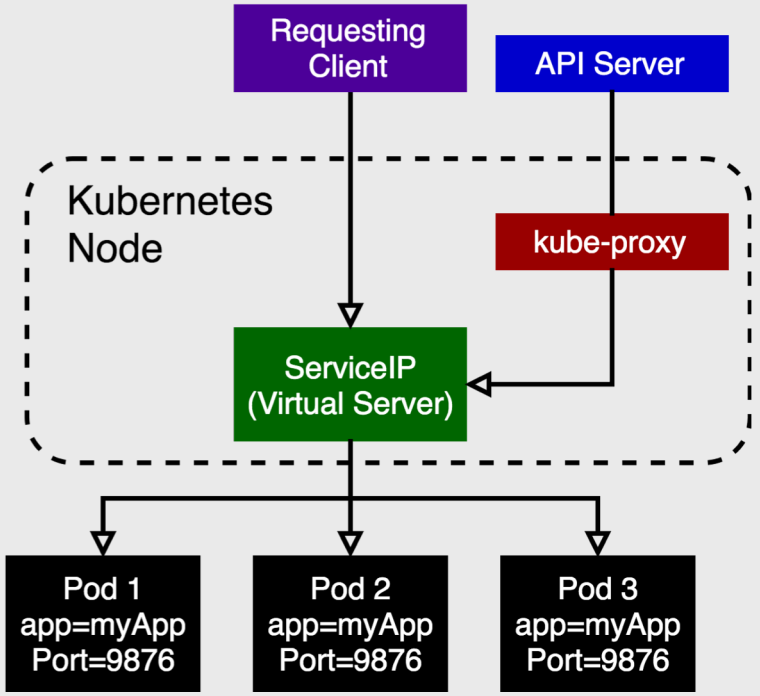
kube-proxy: IPVS Mode

IPVS provides load balancing / switching at layer 4

Available in Kubernetes since 1.9 (GA in K8s 1.11)

Uses Hash tables and operates in the kernel space thus redirects traffic much faster

Provides various load balancing algorithms



kube-proxy in IPVS mode

IPVS vs. IPTables (kube-proxy: proxy mode)

IPTables have lower performance as the number of services increase (+4000) with performance deteriorating by both the processing of packets and the processing of rules

IPTables process rules sequentially while IPVS uses a hash

Service Access Time

- With less than 1000 services IPVS and IPTables perform nearly identically
- As services increase IPVS performance remains flat
- As the number of services increase to a few thousand, IPTables performance becomes unpredictable (and slow)

IPVS is very lean from a resource perspective IPVS uses a fraction of the memory of IPTables (and no CPU)

Network Policy

Policy Based Micro-Segmentation

Policy Driven Network Security

“A micro-firewall for every workload”
–Project Calico

Calico Fundamentals for Kubernetes

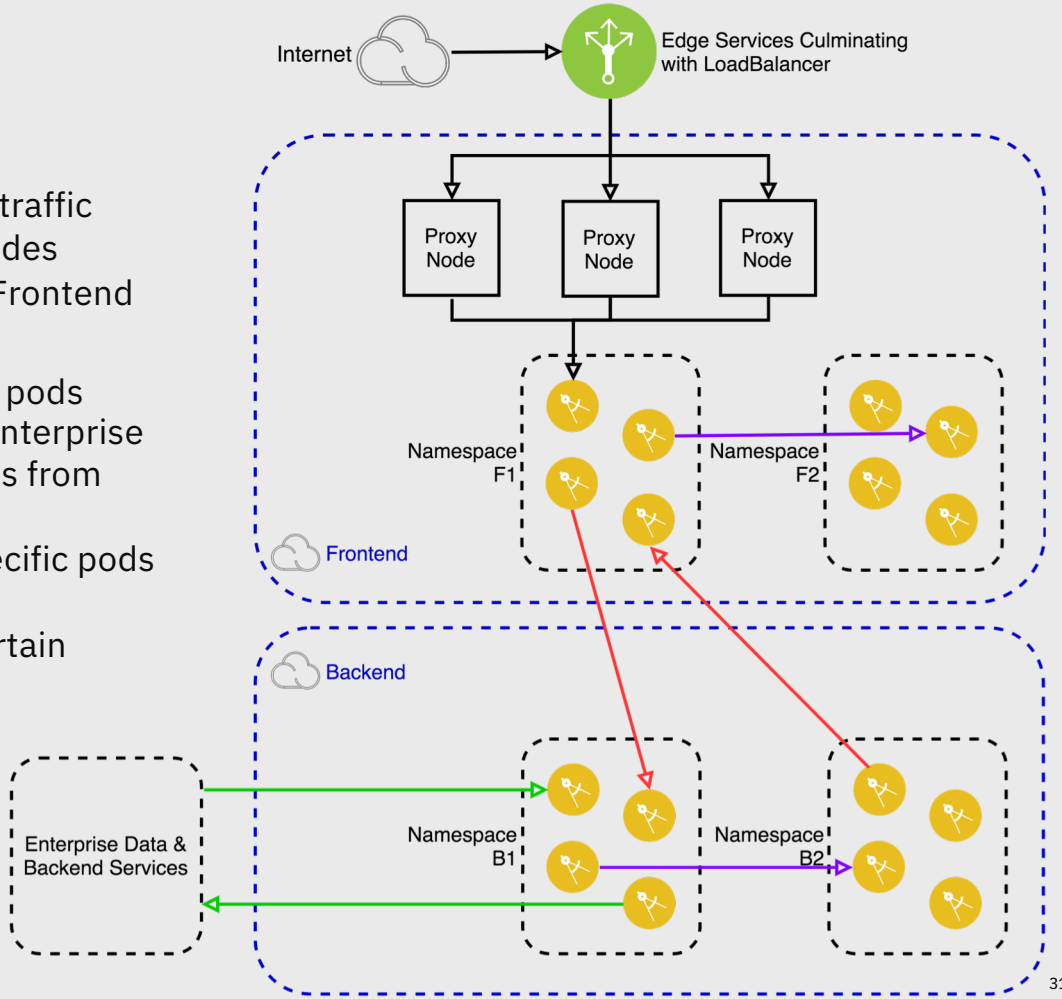
- Perimeter security alone is insufficient
- Calico builds upon and extends the capabilities of the platform
- Calico enables greater granularity in defining allowable network connections passing this role to developer / operator
- Rules are implemented on each node dynamically during the deployment of workload

Network Segmentation

Scenario:

- Enterprise network edge services qualify traffic and load balance across the ICP Proxy nodes
- There are two logical network segments Frontend and Backend
- Ingress allows proxy of specific Frontend pods
- Frontend segment forbids egress to the enterprise
- Specific pods in the Backend allow ingress from specific pods in the Frontend
- Backend segment allows egress from specific pods to specific data services in the enterprise
- Backend segment allows ingress from certain services located in the enterprise

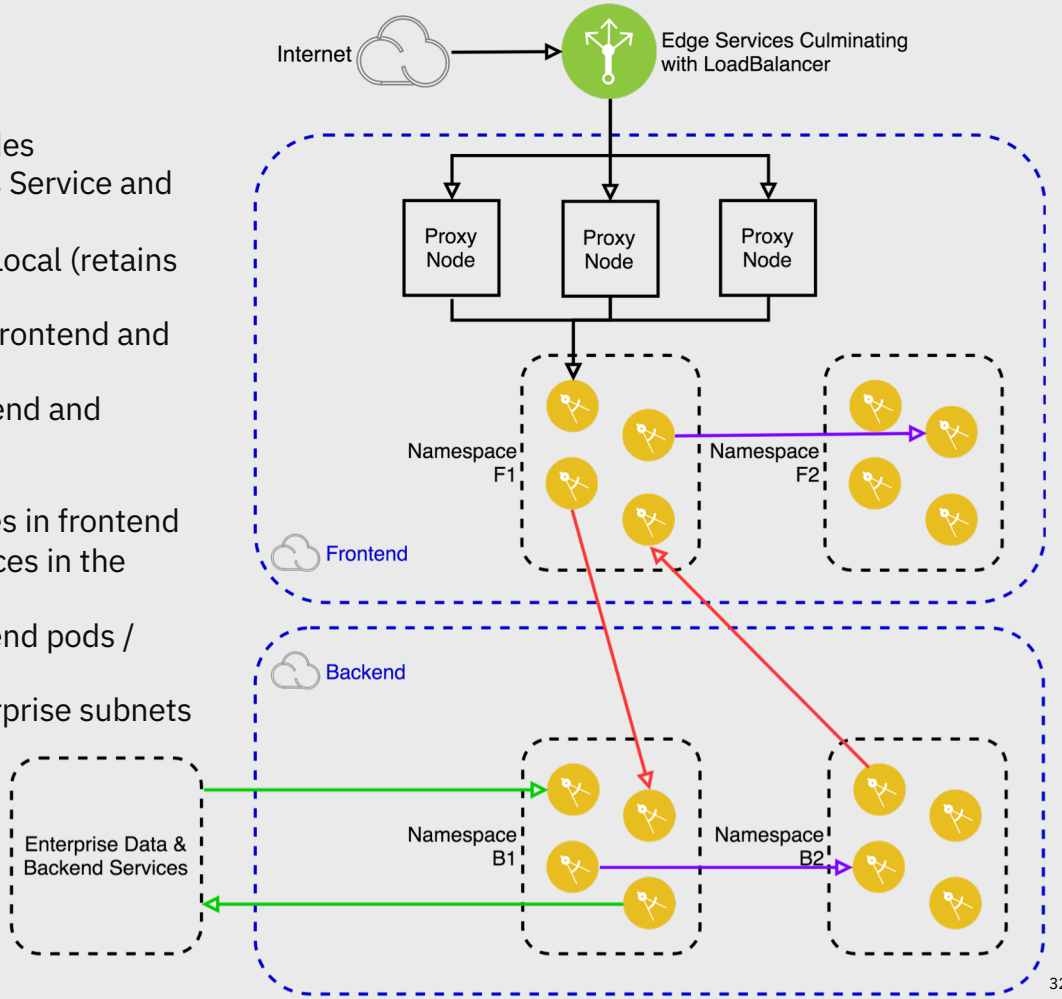
Segmentation in this example does not assume node subnet separation and any specific node affinity for workloads (however these concepts can be applied)



Implementation Details

- Load Balancer targets the IPs of the Proxy Nodes
- Frontend workload is exposed to proxy via K8s Service and Ingress object
- Configure services with externalTrafficPolicy: Local (retains the client IP)
- Use Calico policy to deny all ingress traffic to Frontend and Backend
- Use Calico policy to deny all egress from frontend and backend namespaces
- Calico policy allows DNS egress traffic
- Calico policy allows specific pods / namespaces in frontend to communicate with specific pods / namespaces in the backend
- Calico policy allows egress from specific backend pods / namespaces to specific enterprise subnets
- Calico policy allows ingress from specific enterprise subnets into backend namespaces / pods

Note: Services define communication within the frontend and backend segments



Configuring the Calico CLI

For current configuration information see the IBM Cloud Private Knowledge Center and the docs available at Project Calico. In the below example it is assumed that the Calico CLI will be accessed from outside of the cluster nodes.

The Kubernetes NetworkPolicy object is capable of configuring many of the network connection behaviors, but some detail requires the Calico CLI and the use of the Policy object. These two concepts can be used in concert with each other.

Extract the `calicoctl` executable binary

```
Boot-node $ docker run -v /root:/data --entrypoint=cp ibmcom/calico-ctl:v2.0.2 /calicoctl /data
```

Move this to the system where the CLI will be run and add it to the $PATH

Copy `/etc/cfc/conf/etcd/ca.pem`, `/etc/cfc/conf/etcd/client-key.pem`, `/etc/cfc/conf/etcd/client.pem` to the CLI system

Export the certificate file, CA certificate file, key file and CA domain (domain can be found in the `config.yaml`)

```
$ export ETCD_CERT_FILE=/etc/cfc/conf/etcd/client.pem
$ export ETCD_CA_CERT_FILE=/etc/cfc/conf/etcd/ca.pem
$ export ETCD_KEY_FILE=/etc/cfc/conf/etcd/client-key.pem
$ export ETCD_ENDPOINTS=https://<cluster_CA_domain>:4001
```

Command reference can be found at https://docs.projectcalico.org/v3.0/reference/calicoctl/commands/ or run "`calicoctl --help`" to get the top level help menu

Example NetworkPolicy: Deny All Ingress & Egress

One of the first steps in creating a logical segment or zone within your cluster is to first create isolation by disallowing any traffic flowing into the namespace(s)

This policy can be applied via "`kubectl –f <filename>`"

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all-ingress
  namespace: frontend-ns
spec:
  podSelector:
    matchLabels: {}
  policyTypes:
  - Ingress
```

Further isolate default traffic within the namespace(s) segment by denying all egress

This policy can be applied via "`kubectl –f <filename>`"

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all-egress
  namespace: frontend-ns
spec:
  podSelector:
    matchLabels: {}
  policyTypes:
  - Egress
```

Example Calico Policy: Deny All Ingress & Egress

This Policy object is now sent via the Calico CLI

This policy can be applied via
"`calicoctl apply –f <filename>`"

All egress and ingress traffic will be disallowed until specifically allowed by a subsequent policy (because no ingress or egress defined)

```
apiVersion: v1
kind: policy
metadata:
  name: isolate-frontend
spec:
  order: 1000
  selector: calico/k8s_ns == 'fe1' && 'fe2'
  types:
    - ingress
    - egress
```

The selector in this example selects based upon the namespace name versus a label and multiple namespaces are chained together via the '&&' operator

The concept of order is introduced here as well:

- Order insures priority that the Calico policy will execute prior to the NetworkPolicy in K8s
- Higher priority executes first
- Typically, perform the "denial" style with highest order and then specifically allow egress / ingress with lower priorities
- Best-Practice if using K8s Network Policy use it to provide very specific access for specific workload

Note: List out your Calico Policy "`calicoctl get policy –o wide`"

Example: Allow Traffic to DNS

Building upon the previous example with all egress traffic denial

One way to re-enable this traffic is to label the kube-system namespace (where DNS services run)

```
$ kubectl label namespace kube-system name=kube-system
```

Use namespace selector by label to create the policy and apply via "`kubectl -f <filename>`"

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-dns-access
  namespace: frontend-ns
spec:
  podSelector:
    matchLabels: {}
  policyTypes:
  - Egress
  egress:
  - to:
    - namespaceSelector:
      matchLabels:
        name: kube-system
    ports:
    - protocol: UDP
      port: 53
```

Equivalent Policy for Calico CLI:

```
apiVersion: v1
kind: policy
metadata:
  name: frontend-dns-access
spec:
  egress:
  - action: allow
    destination:
      ports:
      - 53
      selector: has(calico/k8s_ns)
    protocol: tcp
    source: {}
  - action: allow
    destination:
      ports:
      - 53
      selector: has(calico/k8s_ns)
    protocol: udp
    source: {}
  order: 990
  selector: calico/k8s_ns == 'fe1' && 'fe2'
  types:
  - egress
```

Example: Allow Traffic from Specific Subnets

This Policy object will be sent via Calico CLI

The Calico API handles the ability of specifying multiple networks in CIDR format

A selector for namespace is provided that doesn't require the namespace label

This rule uses a selector for the pod label of "backend-app"

This policy can be applied via "`calicoctl apply –f <filename>`"

```
apiVersion: v1
kind: policy
metadata:
  name: backend-ext-ent-traffic
spec:
  egress:
  - action: allow
    destination: {}
    source: {}
  ingress:
  - action: allow
    source:
      nets: [10.40.0.0/24, 10.41.0.0/24]
    destination: {}
  order: 400
  selector: calico/k8s_ns == 'backend-ns' && app == 'backend-app'
```

Ingress

Proxy and Ports

Which ingress options are right for you?

There are many ways to communicate with your workload

Cluster IP

Proxy Nodes

Node ports

Custom Proxy

Services

Load Balancer

VIP

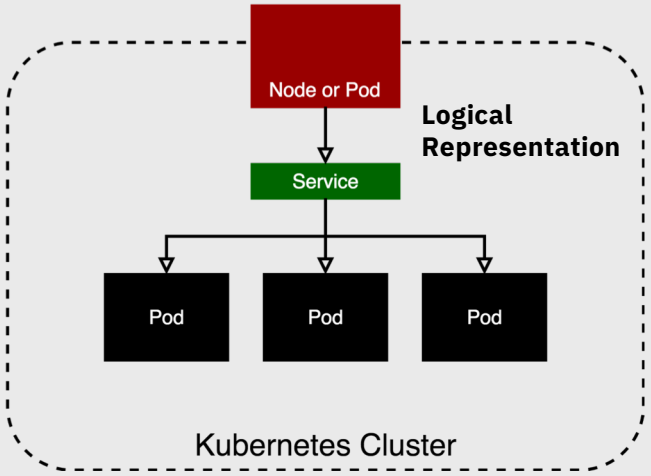
Istio

ClusterIP

ClusterIP is the default access mode when setting up a Kubernetes service

This gives access to the service from within the cluster via its assigned IP

Note: kubectl proxy can be used to access the service via the API from outside of the cluster for testing



Service definition

```
kind:
ServiceapiVersion: v1
metadata:
  name: my-nginx-service
spec:
  selector:
    app: skol-nginx
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```

Implementation

```
$ kubectl get services -o wide
NAME                  TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)         AGE       SELECTOR
My-nginx-service.     ClusterIP   10.0.0.226    <none>        80/TCP          23s       app=skol-nginx
$
```

NodePort

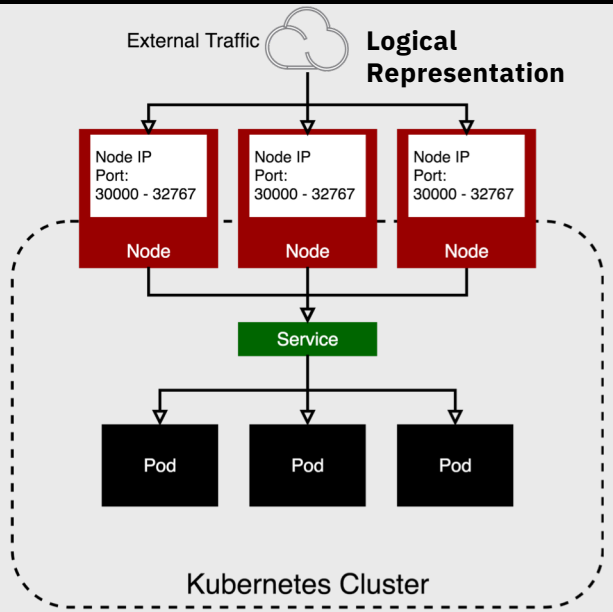
A service of type NodePort may be used to get external traffic directly to the workload

NodePorts are opened for the service on all nodes in the cluster

The port can be user specified, but the best-practice is to allow K8s to assign it

The port will be between 30000–32767

Limit of one service per port



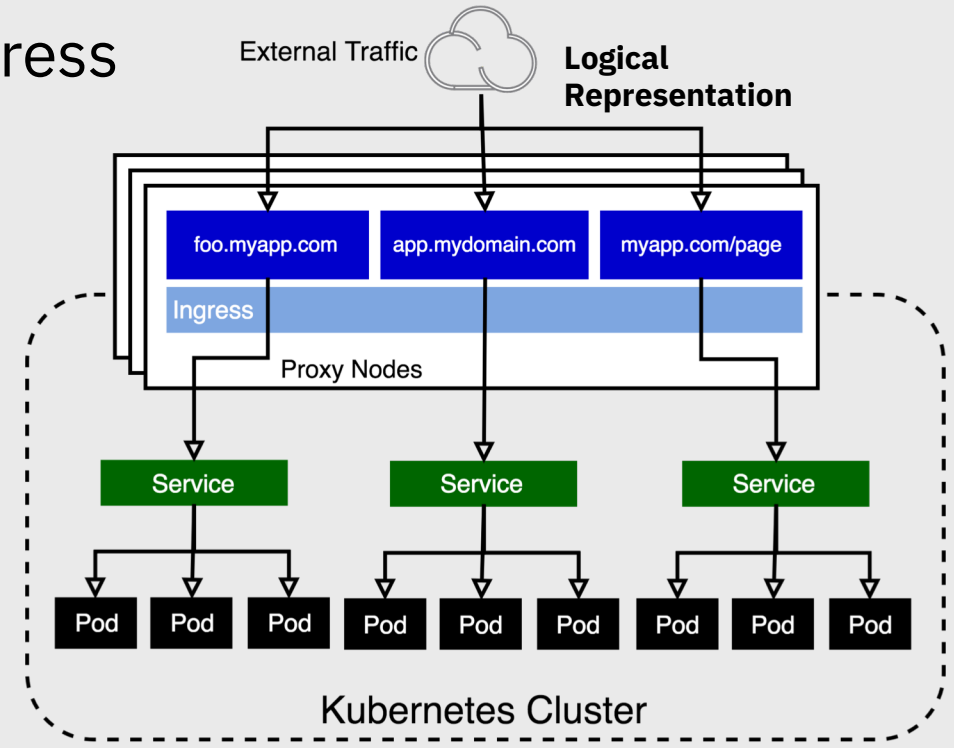
Service definition

```
kind:
ServiceapiVersion: v1
metadata:
  name: my-nginx-
service
spec:
  selector:
    app: skol-nginx
  type: NodePort
  ports:
  - name: http
    protocol: TCP
    port: 80
    targetPort: 80
```

Implementation

```
$ kubectl get services -o wide
NAME                    TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)      AGE       SELECTOR
My-nginx-service.       ClusterIP   10.0.0.226    <none> 80/TCP               23m       app=skol-nginx
nginx-np-service        NodePort    10.0.0.225    <none> 80:31357/TCP         7s        app=skol-nginx
```

Ingress



Ingress definition

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: my-ingress
spec:
  rules:
  - http:
      paths:
      - path: myapp.com/page
        backend:
          serviceName: my-nginx-service
          servicePort: 80
      - path: foo.myapp.com
        backend:
          serviceName: my-other-service
          servicePort: 80
```

The ingress object sits in front of multiple services and proxies / routes traffic into the cluster and to the app

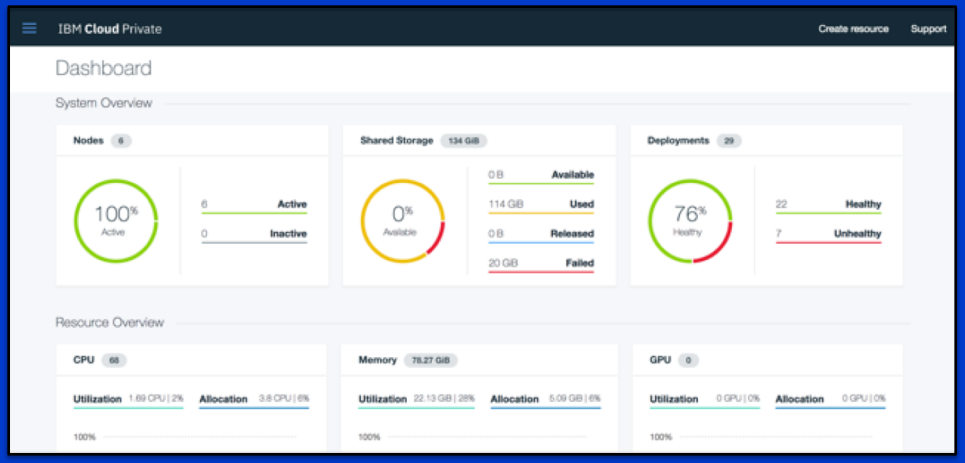
ICP implements this via the proxy node(s) running NGINX

Ingress is defined via the API and can be tuned via the NGINX proxy node configmaps

Try IBM Cloud Private today!

Guided and Proof of Technology demos

Free Community Edition!



http://ibm.biz/ICP-DTE