Diving Deep into Kubernetes Networking

AUTHORS

Adrian Goins

Alena Prokharchyk

Murali Paluru

RANCHER

Introduction

Kubernetes has evolved into a strategic platform for deploying and scaling applications in data centers and the cloud. It provides built-in abstractions for efficiently deploying, scaling, and managing applications. Kubernetes also addresses concerns such as storage, networking, load balancing, and multi-cloud deployments.

Networking is a critical component for the success of a Kubernetes implementation. Network components in a Kubernetes cluster control interaction at multiple layers, from communication between containers running on different hosts to exposing services to clients outside of a cluster. The requirements within each environment are different, so before we choose which solution is the most appropriate, we have to understand how networking works within Kubernetes and what benefits each solution provides.

This book is based on the Networking Master Class online meetup that is available on YouTube.

This eBook covers Kubernetes networking concepts, but we do not intend for it to be a detailed explanation of Kubernetes in its entirety. For more information on Kubernetes, we recommend reading the Kubernetes documentation or enrolling in a training program from a CNCF-certified training provider.

GOALS OF THIS BOOK

This book introduces various networking concepts related to Kubernetes that an operator, developer, or decision maker might find useful. Networking is a complex topic and even more so when it comes to a distributed system like Kubernetes. It is essential to understand the technology, the tooling, and the available choices. These choices affect an organization's ability to scale the infrastructure and the applications running on top of it.

The reader is expected to have a basic understanding of containers, Kubernetes, and operating system fundamentals.

HOW THIS BOOK IS ORGANIZED

In this book, we cover Kubernetes networking from the basics to the advanced topics. We start by explaining Docker container networking, as Docker is a fundamental component of Kubernetes. We then introduce Kubernetes networking, its unique model and how it seamlessly scales. In doing so, we explain the abstractions that enable Kubernetes to communicate effectively between applications. We touch upon the Container Network Interface (CNI) specification and how it relates to Kubernetes, and finally, we do a deep dive into some of the more popular CNI plugins for Kubernetes such as Calico, Flannel and Canal. We discuss load balancing, DNS and how to expose applications to the outside world.

An Introduction to Networking with Docker

Docker follows a unique approach to networking that is very different from the Kubernetes approach. Understanding how Docker works help later in understanding the Kubernetes model, since Docker containers are the fundamental unit of deployment in Kubernetes.

DOCKER NETWORKING TYPES

When a Docker container launches, the Docker engine assigns it a network interface with an IP address, a default gateway, and other components, such as a routing table and DNS services. By default, all addresses come from the same pool, and all containers on the same host can communicate with one another. We can change this by defining the network to which the container should connect, either by creating a custom user-defined network or by using a network provider plugin.

The network providers are pluggable using drivers. We connect a Docker container to a particular network by using the `--net` switch when launching it.

The following command launches a container from the busybox image and joins it to the host network. This container prints its IP address and then exits.

```
docker run --rm --net=host busybox ip addr
```

Docker offers five network types, each with a different capacity for communication with other network entities.

A. Host Networking: The container shares the same IP address and network namespace as that of the host. Services running inside of this container have the same network capabilities as services running directly on the host.

B. Bridge Networking: The container runs in a private network internal to the host. Communication is open to other containers in the same network. Communication with services outside of the host goes through network address translation (NAT) before exiting the host. (This is the default mode of networking when the --net option isn't specified)

C. Custom bridge network: This is the same as Bridge Networking but uses a bridge explicitly created for this (and other) containers. An example of how to use this would be a container that runs on an exclusive "database" bridge network. Another container can have an interface on the default bridge and the database bridge, enabling it to communicate with both networks.

D. Container-defined Networking: A container can share the address and network configuration of another container. This type enables process isolation between containers, where each container runs one service but where services can still communicate with one another on the `localhost` address.

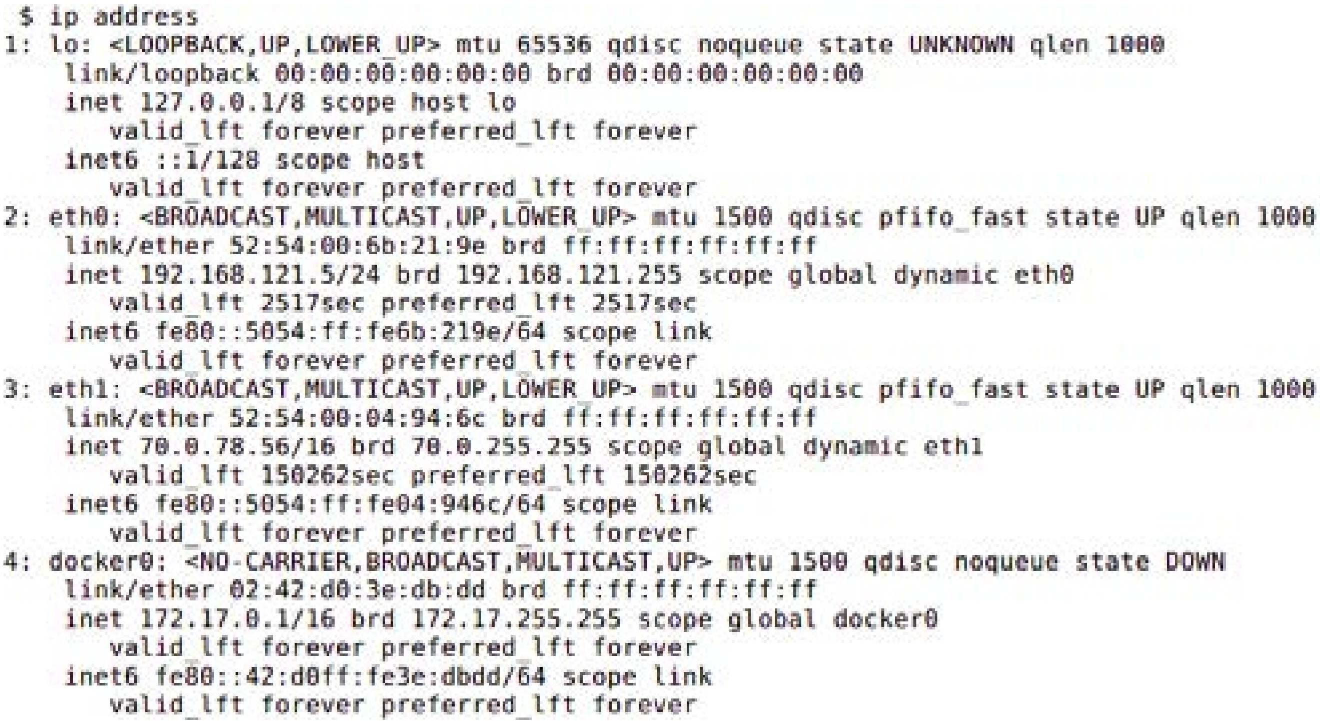
E. No networking: This option disables all networking for the container.

Host Networking

The host mode of networking allows the Docker container to share the same IP address as that of the host and disables the network isolation otherwise provided by network namespaces. The container's network stack is mapped directly to the host's network stack. All interfaces and addresses on the host are visible within the container, and all communication possible to or from the host is possible to or from the container.

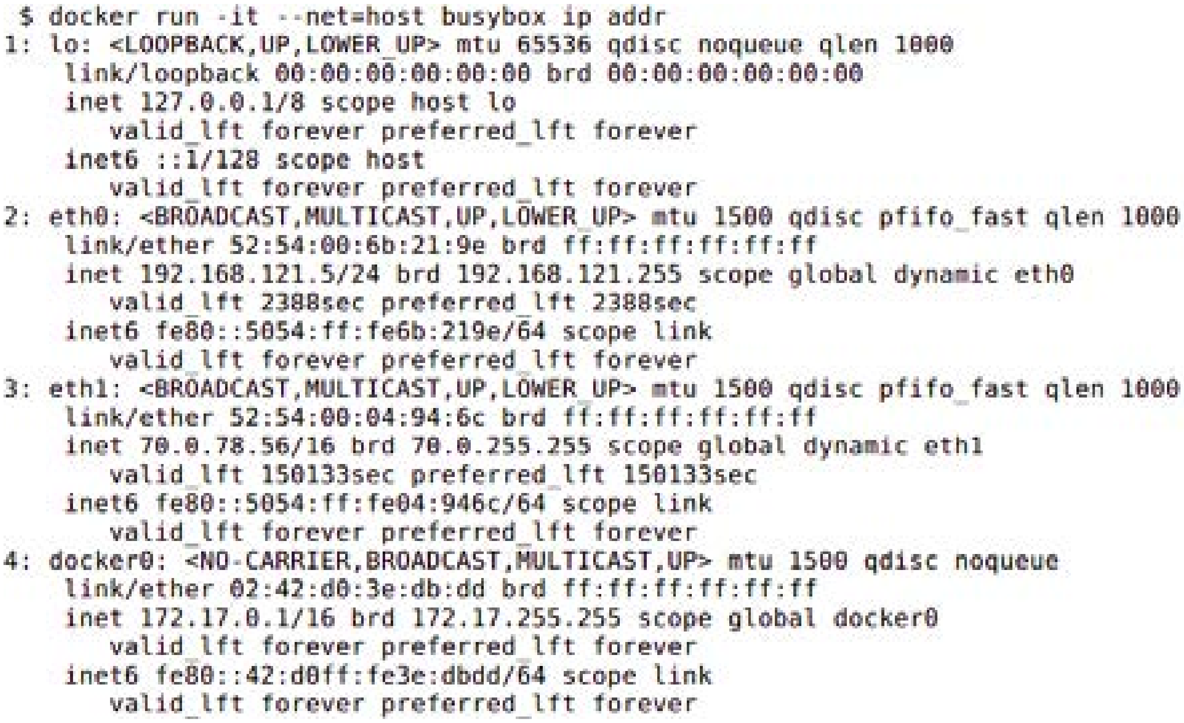
Container
eth0

If you run the command ip addr on a host (or **ifconfig -a** if your host doesn't have the **ip** command available), you will see information about the network interfaces.



```
 $ ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    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
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 52:54:00:6b:21:9e brd ff:ff:ff:ff:ff:ff
    inet 192.168.121.5/24 brd 192.168.121.255 scope global dynamic eth0
       valid_lft 2517sec preferred_lft 2517sec
    inet6 fe80::5054:ff:fe6b:219e/64 scope link
       valid_lft forever preferred_lft forever
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 52:54:00:04:94:6c brd ff:ff:ff:ff:ff:ff
    inet 70.0.78.56/16 brd 70.0.255.255 scope global dynamic eth1
       valid_lft 150262sec preferred_lft 150262sec
    inet6 fe80::5054:ff:fe04:946c/64 scope link
       valid_lft forever preferred_lft forever
4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN
    link/ether 02:42:d0:3e:db:dd brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:d0ff:fe3e:dbdd/64 scope link
       valid_lft forever preferred_lft forever
```

If you run the same command from a container using host networking, you will see the same information.

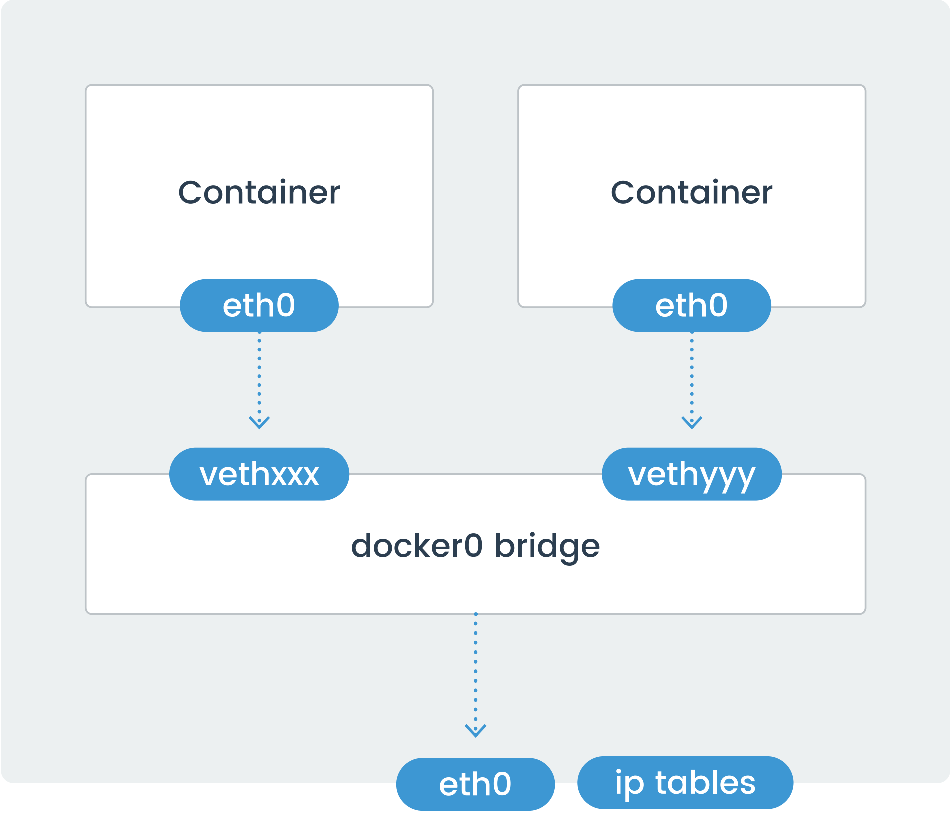


```
 $ docker run -it --net=host busybox ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast qlen 1000
    link/ether 52:54:00:6b:21:9e brd ff:ff:ff:ff:ff:ff
    inet 192.168.121.5/24 brd 192.168.121.255 scope global dynamic eth0
       valid_lft 2388sec preferred_lft 2388sec
    inet6 fe80::5054:ff:fe6b:219e/64 scope link
       valid_lft forever preferred_lft forever
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast qlen 1000
    link/ether 52:54:00:04:94:6c brd ff:ff:ff:ff:ff:ff
    inet 70.0.78.56/16 brd 70.0.255.255 scope global dynamic eth1
       valid_lft 150133sec preferred_lft 150133sec
    inet6 fe80::5054:ff:fe04:946c/64 scope link
       valid_lft forever preferred_lft forever
4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue
    link/ether 02:42:d0:3e:db:dd brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:d0ff:fe3e:dbdd/64 scope link
       valid_lft forever preferred_lft forever
```

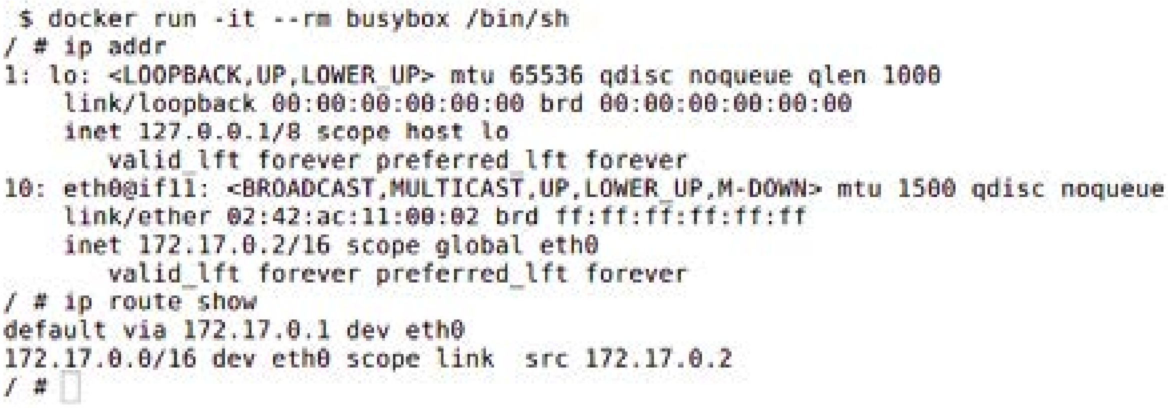
Bridge Networking

In a standard Docker installation, the Docker daemon creates a bridge on the host with the name of `docker0`. When a container launches, Docker then creates a virtual ethernet device for it. This device appears within the container as `eth0` and on the host with a name like `vethxxx` where `xxx` is a unique identifier for the interface. The `vethxxx` interface is added to the `docker0` bridge, and this enables communication with other containers on the same host that also use the default bridge.

To demonstrate using the default bridge, run the following command on a host with Docker installed. Since we are not specifying the network - the container will connect to the default bridge when it launches.

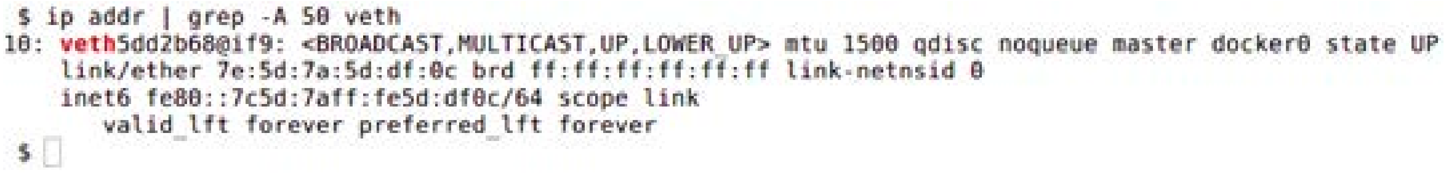


Run the `ip addr` and `ip route` commands inside of the container. You will see the IP address of the container with the `eth0` interface:



```
 $ docker run -it --rm busybox /bin/sh
/ # ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    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,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 scope global eth0
       valid_lft forever preferred_lft forever
/ # ip route show
default via 172.17.0.1 dev eth0
172.17.0.0/16 dev eth0 scope link  src 172.17.0.2
/ # 
```

In another terminal connected to the host, run the `ip addr` command. You will see the corresponding interface created for the container. In the image below it is named `veth5dd2b68@if9`. Yours will be different.



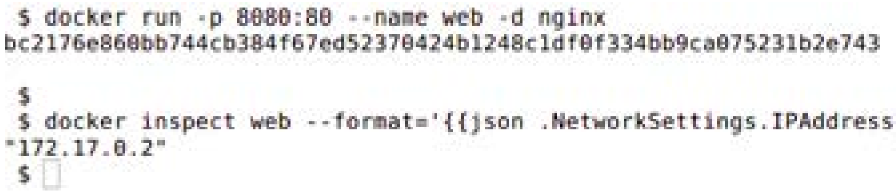
```
 $ ip addr | grep -A 50 veth
10: veth5dd2b68@if9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP
    link/ether 7e:5d:7a:5d:df:0c brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet6 fe80::7c5d:7aff:fe5d:df0c/64 scope link
       valid_lft forever preferred_lft forever
 $ 
```

Although Docker mapped the container IPs on the bridge, network services running inside of the container are not visible outside of the host. To make them visible, the Docker Engine must be told when launching a container to map ports from that container to ports on the host. This process is called publishing. For example, if you want to map port 80 of a container to port 8080 on the host, then you would have to publish the port as shown in the following command:

```
docker run --name nginx -p 8080:80 nginx
```

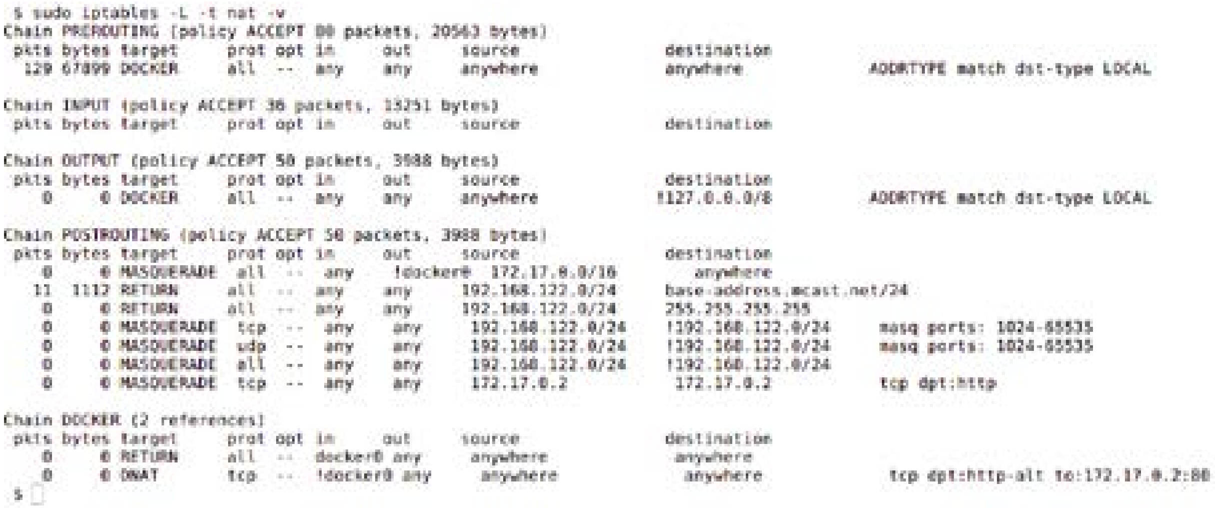
By default, the Docker container can send traffic to any destination. The Docker daemon creates a rule within Netfilter that modifies outbound packets and changes the source address to be the address of the host itself. The Netfilter configuration allows inbound traffic via the rules that Docker creates when initially publishing the container's ports.

The output included below shows the Netfilter rules created by Docker when it publishes a container's ports.



```
 $ docker run -p 8080:80 --name web -d nginx
bc2176e860bb744cb384f67ed52370424b1248c1df0f334bb9ca075231b2e743

 $
 $ docker inspect web --format='{{json .NetworkSettings.IPAddress}}'
"172.17.0.2"
 $ 
```



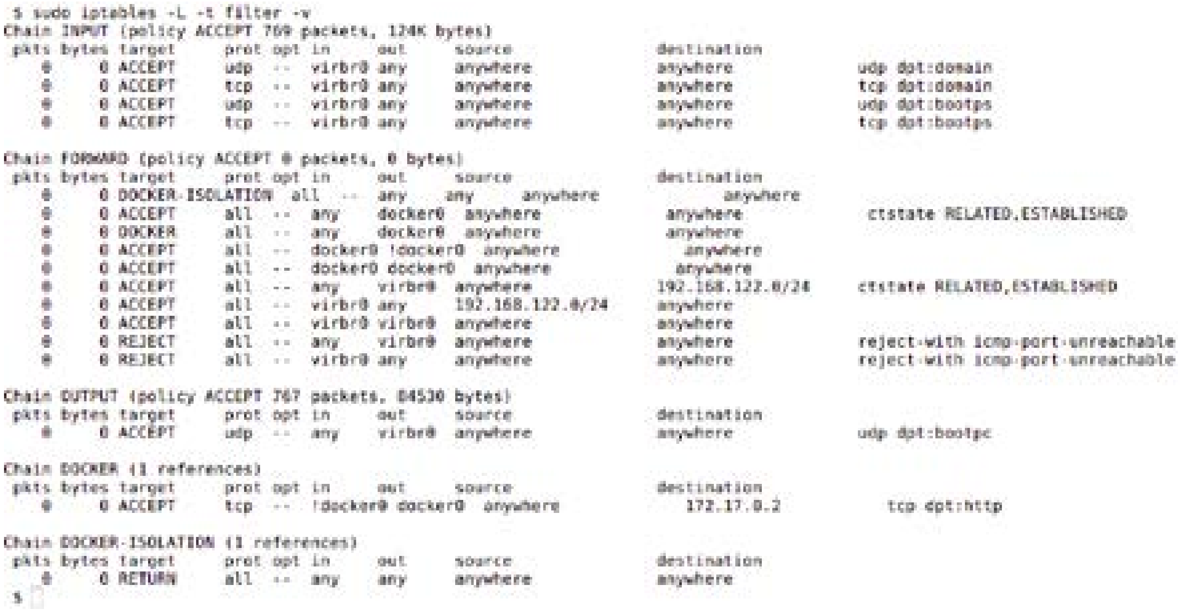
```
 $ sudo iptables -L -t nat -v
Chain PREROUTING (policy ACCEPT 80 packets, 20563 bytes)
 pkts bytes target     prot opt in     out     source               destination
  129 67899 DOCKER     all  --  any    any     anywhere             anywhere             ADDRTYPE match dst-type LOCAL

Chain INPUT (policy ACCEPT 36 packets, 13251 bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain OUTPUT (policy ACCEPT 50 packets, 3988 bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 DOCKER     all  --  any    any     anywhere            !127.0.0.0/8          ADDRTYPE match dst-type LOCAL

Chain POSTROUTING (policy ACCEPT 50 packets, 3988 bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 MASQUERADE  all  --  any    !docker0  172.17.0.0/16        anywhere
   11  1112 RETURN     all  --  any    any     192.168.122.0/24     base-address.mcast.net/24
    0     0 RETURN     all  --  any    any     192.168.122.0/24     255.255.255.255
    0     0 MASQUERADE  tcp  --  any    any     192.168.122.0/24    !192.168.122.0/24     masq ports: 1024-65535
    0     0 MASQUERADE  udp  --  any    any     192.168.122.0/24    !192.168.122.0/24     masq ports: 1024-65535
    0     0 MASQUERADE  all  --  any    any     192.168.122.0/24    !192.168.122.0/24
    0     0 MASQUERADE  tcp  --  any    any     172.17.0.2           172.17.0.2           tcp dpt:http

Chain DOCKER (2 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 RETURN     all  --  docker0 any     anywhere             anywhere
    0     0 DNAT       tcp  --  !docker0 any     anywhere             anywhere              tcp dpt:http-alt to:172.17.0.2:80
 $ 
```



```
 $ sudo iptables -L -t filter -v
Chain INPUT (policy ACCEPT 769 packets, 124K bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 ACCEPT     udp  --  virbr0 any     anywhere             anywhere             udp dpt:domain
    0     0 ACCEPT     tcp  --  virbr0 any     anywhere             anywhere             tcp dpt:domain
    0     0 ACCEPT     udp  --  virbr0 any     anywhere             anywhere             udp dpt:bootps
    0     0 ACCEPT     tcp  --  virbr0 any     anywhere             anywhere             tcp dpt:bootps

Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 DOCKER-ISOLATION  all  --  any    any     anywhere             anywhere
    0     0 ACCEPT     all  --  any    docker0  anywhere             anywhere             ctstate RELATED,ESTABLISHED
    0     0 DOCKER     all  --  any    docker0  anywhere             anywhere
    0     0 ACCEPT     all  --  docker0 !docker0  anywhere             anywhere
    0     0 ACCEPT     all  --  docker0 docker0  anywhere             anywhere
    0     0 ACCEPT     all  --  any    virbr0  anywhere             192.168.122.0/24     ctstate RELATED,ESTABLISHED
    0     0 ACCEPT     all  --  virbr0 any     192.168.122.0/24     anywhere
    0     0 ACCEPT     all  --  virbr0 virbr0  anywhere             anywhere
    0     0 REJECT     all  --  any    virbr0  anywhere             anywhere             reject-with icmp-port-unreachable
    0     0 REJECT     all  --  virbr0 any     anywhere             anywhere             reject-with icmp-port-unreachable

Chain OUTPUT (policy ACCEPT 767 packets, 84530 bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 ACCEPT     udp  --  any    virbr0  anywhere             anywhere             udp dpt:bootpc

Chain DOCKER (1 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 ACCEPT     tcp  --  !docker0 docker0  anywhere             172.17.0.2            tcp dpt:http

Chain DOCKER-ISOLATION (1 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 RETURN     all  --  any    any     anywhere             anywhere
 $ 
```

NAT table within Netfilter

Custom Bridge Network

There is no requirement to use the default bridge on the host; it's easy to create a new bridge network and attach containers to it. This provides better isolation and interoperability between containers, and custom bridge networks have better security and features than the default bridge.

- All containers in a custom bridge can communicate with the ports of other containers on that bridge. This means that you do not need to publish the ports explicitly. It also ensures that the communication between them is secure. Imagine an application in which a backend container and a database container need to communicate and where we also want to make sure that no external entity can talk to the database. We do this with a custom bridge network in which only the database container and the backend containers reside. You can explicitly expose the backend API to the rest of the world using port publishing.
- The same is true with environment variables - environment variables in a bridge network are shared by all containers on that bridge.
- Network configuration options such as MTU can differ between applications. By creating a bridge, you can configure the network to best suit the applications connected to it.

To create a custom bridge network and two containers that use it, run the following commands:

```
$ docker network create mynetwork
$ docker run -it --rm --name=container-a --network=mynetwork busybox /bin/sh
$ docker run -it --rm --name=container-b --network=mynetwork busybox /bin/sh
```

Container-Defined Network

A specialized case of custom networking is when a container joins the network of another container. This is similar to how a Pod works in Kubernetes.

The following commands launch two containers that share the same network namespace and thus share the same IP address. Services running on one container can talk to services running on the other via the `localhost` address.

```
$ docker run -it --rm --name=container-a busybox /bin/sh
$ docker run -it --rm --name=container-b --network=container:container-a busybox /bin/sh
```

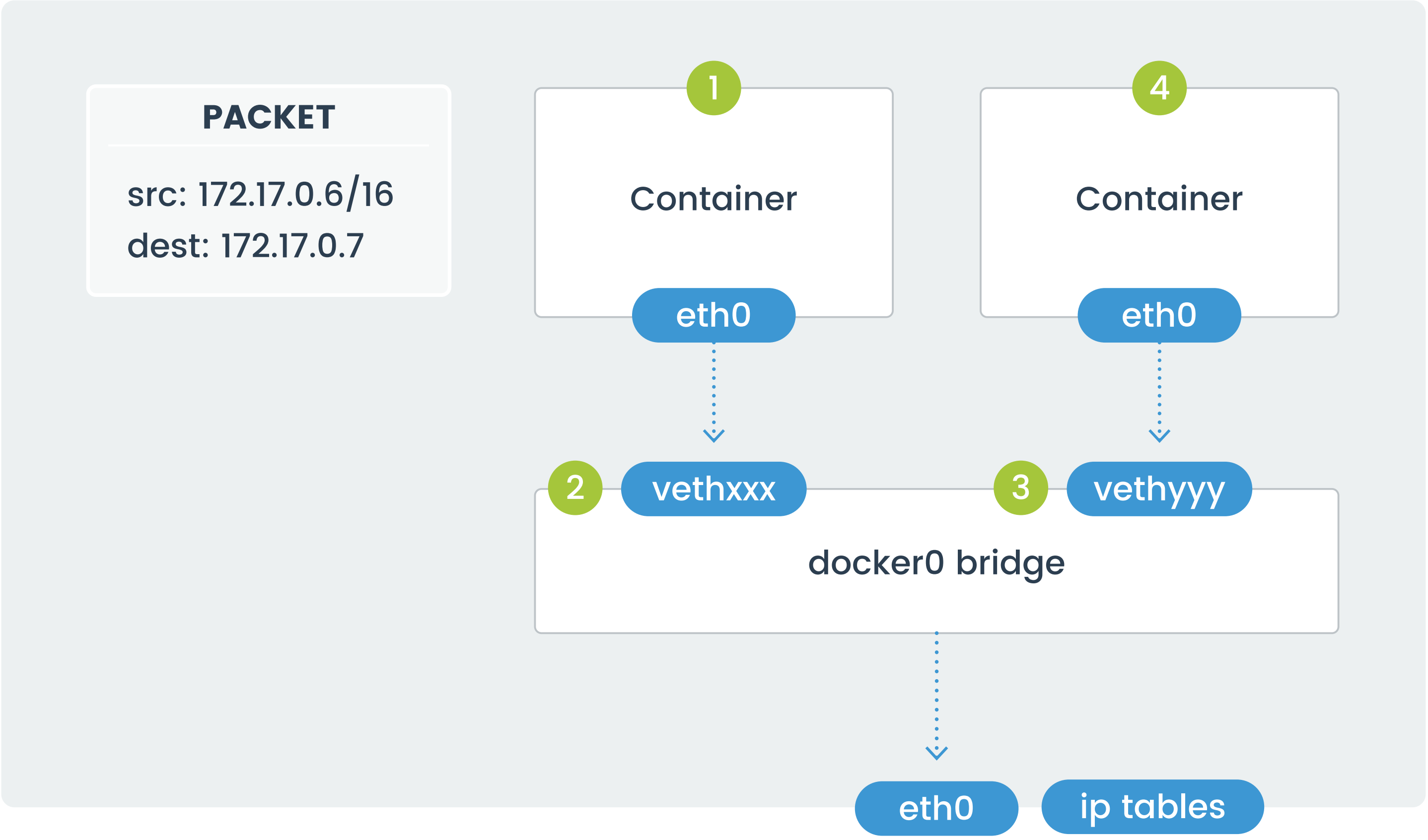
No Networking

This mode is useful when the container does not need to communicate with other containers or with the outside world. It is not assigned an IP address, and it cannot publish any ports.

```
$ docker run --net=none --name busybox busybox ip a
```

CONTAINER-TO-CONTAINER COMMUNICATION

How do two containers on the same bridge network talk to one another?

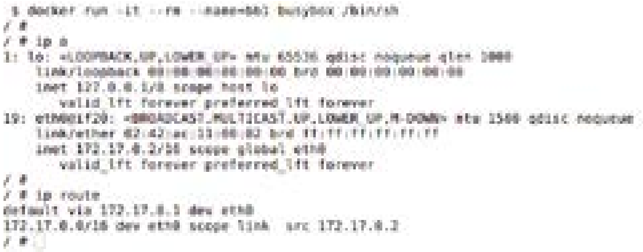


In the above diagram, two containers running on the same host connect via the `docker0` bridge. If `172.17.0.6` (on the left-hand side) wants to send a request to `172.17.0.7` (the one on the right-hand side), the packets move as follows:

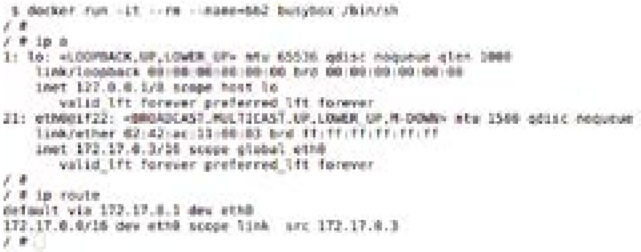
1. A packet leaves the container via `eth0` and lands on the corresponding `vethxxx` interface.
2. The `vethxxx` interface connects to the `vethyyy` interface via the `docker0` bridge.
3. The `docker0` bridge forwards the packet to the `vethyyy` interface.
4. The packet moves to the `eth0` interface within the destination container.

We can see this in action by using `ping` and `tcpdump`. Create two containers and inspect their network configuration with `ip addr` and `ip route`. The default route for each container is via the `eth0` interface.

Ping one container from the other, and let the command run so that we can inspect the traffic. Run `tcpdump` on the `docker0` bridge on the host machine. You will see in the output that the traffic moves between the two containers via the `docker0` bridge.



```
 $ docker run -it --rm --name=bb1 busybox /bin/sh
/ #
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    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
19: eth0@if20: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 scope global eth0
       valid_lft forever preferred_lft forever
/ #
/ # ip route
default via 172.17.0.1 dev eth0
172.17.0.0/16 dev eth0 scope link  src 172.17.0.2
/ #
```

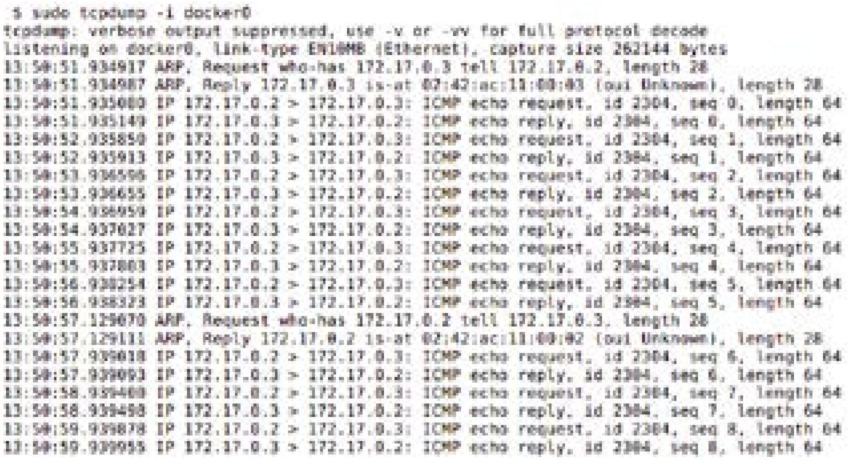


```
 $ docker run -it --rm --name=bb2 busybox /bin/sh
/ #
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
21: eth0@if22: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.3/16 scope global eth0
       valid_lft forever preferred_lft forever
/ #
/ # ip route
default via 172.17.0.1 dev eth0
172.17.0.0/16 dev eth0 scope link  src 172.17.0.3
/ #
```

CONTAINER COMMUNICATION BETWEEN HOSTS

So far we've discussed scenarios in which containers communicate within a single host. While interesting, real-world applications require communication between containers running on different hosts.

Cross-host networking usually uses an overlay network, which builds a mesh between hosts and employs a large block of IP addresses within that mesh. The network driver tracks which addresses are on which host and shuttles packets between the hosts as necessary for inter-container communication.



```
 $ sudo tcpdump -i docker0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on docker0, link-type EN10MB (Ethernet), capture size 262144 bytes
13:50:51.934917 ARP, Request who-has 172.17.0.3 tell 172.17.0.2, length 28
13:50:51.934987 ARP, Reply 172.17.0.3 is-at 02:42:ac:11:00:03 (oui Unknown), length 28
13:50:51.935000 IP 172.17.0.2 > 172.17.0.3: ICMP echo request, id 2304, seq 0, length 64
13:50:51.935149 IP 172.17.0.3 > 172.17.0.2: ICMP echo reply, id 2304, seq 0, length 64
13:50:52.935850 IP 172.17.0.2 > 172.17.0.3: ICMP echo request, id 2304, seq 1, length 64
13:50:52.935913 IP 172.17.0.3 > 172.17.0.2: ICMP echo reply, id 2304, seq 1, length 64
13:50:53.936596 IP 172.17.0.2 > 172.17.0.3: ICMP echo request, id 2304, seq 2, length 64
13:50:53.936655 IP 172.17.0.3 > 172.17.0.2: ICMP echo reply, id 2304, seq 2, length 64
13:50:54.936959 IP 172.17.0.2 > 172.17.0.3: ICMP echo request, id 2304, seq 3, length 64
13:50:54.937027 IP 172.17.0.3 > 172.17.0.2: ICMP echo reply, id 2304, seq 3, length 64
13:50:55.937725 IP 172.17.0.2 > 172.17.0.3: ICMP echo request, id 2304, seq 4, length 64
13:50:55.937803 IP 172.17.0.3 > 172.17.0.2: ICMP echo reply, id 2304, seq 4, length 64
13:50:56.938254 IP 172.17.0.2 > 172.17.0.3: ICMP echo request, id 2304, seq 5, length 64
13:50:56.938323 IP 172.17.0.3 > 172.17.0.2: ICMP echo reply, id 2304, seq 5, length 64
13:50:57.129070 ARP, Request who-has 172.17.0.2 tell 172.17.0.3, length 28
13:50:57.129111 ARP, Reply 172.17.0.2 is-at 02:42:ac:11:00:02 (oui Unknown), length 28
13:50:57.939018 IP 172.17.0.2 > 172.17.0.3: ICMP echo request, id 2304, seq 6, length 64
13:50:57.939093 IP 172.17.0.3 > 172.17.0.2: ICMP echo reply, id 2304, seq 6, length 64
13:50:58.939400 IP 172.17.0.2 > 172.17.0.3: ICMP echo request, id 2304, seq 7, length 64
13:50:58.939458 IP 172.17.0.3 > 172.17.0.2: ICMP echo reply, id 2304, seq 7, length 64
13:50:59.939878 IP 172.17.0.2 > 172.17.0.3: ICMP echo request, id 2304, seq 8, length 64
13:50:59.939955 IP 172.17.0.3 > 172.17.0.2: ICMP echo reply, id 2304, seq 8, length 64
```

Overlay networks can be encrypted or unencrypted. Unencrypted networks are acceptable for environments in which all of the hosts are within the same LAN, but because overlay networks enable communication between hosts across the Internet, consider the security requirements when choosing a network driver. If the packets traverse a network that you don't control, encryption is a better choice.

The overlay network functionality built into Docker is called Swarm. When you connect a host to a swarm, the Docker engine on each host handles communication and routing between the hosts.

Other overlay networks exist, such as IPVLAN, VxLAN, and MACVLAN. More solutions are available for Kubernetes.

For more information on pure-Docker networking implementations for cross-host networking (including Swarm mode and libnetwork), please refer to the documentation available at the Docker website.

Interlude: Netfilter and iptables rules

In the earlier section on Docker networking, we looked at how Docker handles communication between containers. On a Linux host, the component which handles this is called Netfilter, or more commonly by the command used to configure it: `iptables`.

Netfilter manages the rules that define network communication for the Linux kernel. These rules permit, deny, route, modify, and forward packets. It organizes these rules into tables according to their purpose.

The Filter Table

Rules in the Filter table control if a packet is allowed or denied. Packets which are allowed are forwarded whereas packets which are denied are either rejected or silently dropped.

The NAT Table

These rules control network address translation. They modify the source or destination address for the packet, changing how the kernel routes the packet.

The Mangle Table

The headers of packets which go through this table are altered, changing the way the packet behaves. Netfilter might shorten the TTL, redirect it to a different address, or change the number of network hops.

Raw Table

This table marks packets to bypass the iptables stateful connection tracking.

Security Table

This table sets the SELinux security context marks on packets. Setting the marks affects how SELinux (or systems that can interpret SELinux security contexts) handle the packets. The rules in this table set marks on a per-packet or per-connection basis.

Netfilter organizes the rules in a table into chains. Chains are the means by which Netfilter hooks in the kernel intercept packets as they move through processing. Packets flow through one or more chains and exit when they match a rule.

A rule defines a set of conditions, and if the packet matches those conditions, an action is taken. The universe of actions is diverse, but examples include:

- Block all connections originating from a specific IP address.
- Block connections to a network interface.
- Allow all HTTP/HTTPS connections.
- Block connections to specific ports.

The action that a rule takes is called a target, and represents the decision to accept, drop, or forward the packet.

The system comes with five default chains that match different phases of a packet's journey through processing: PREROUTING, INPUT, FORWARD, OUTPUT, and POSTROUTING. Users and programs may create additional chains and inject rules into the system chains to forward packets to a custom chain for continued processing. This architecture allows the Netfilter configuration to follow a logical structure, with chains representing groups of related rules.

Docker creates several chains, and it is the actions of these chains that handle communication between containers, the host, and the outside world.

An Introduction to Kubernetes Networking

Kubernetes networking builds on top of the Docker and Netfilter constructs to tie multiple components together into applications. Kubernetes resources have specific names and capabilities, and we want to understand those before exploring their inner workings.

Pods

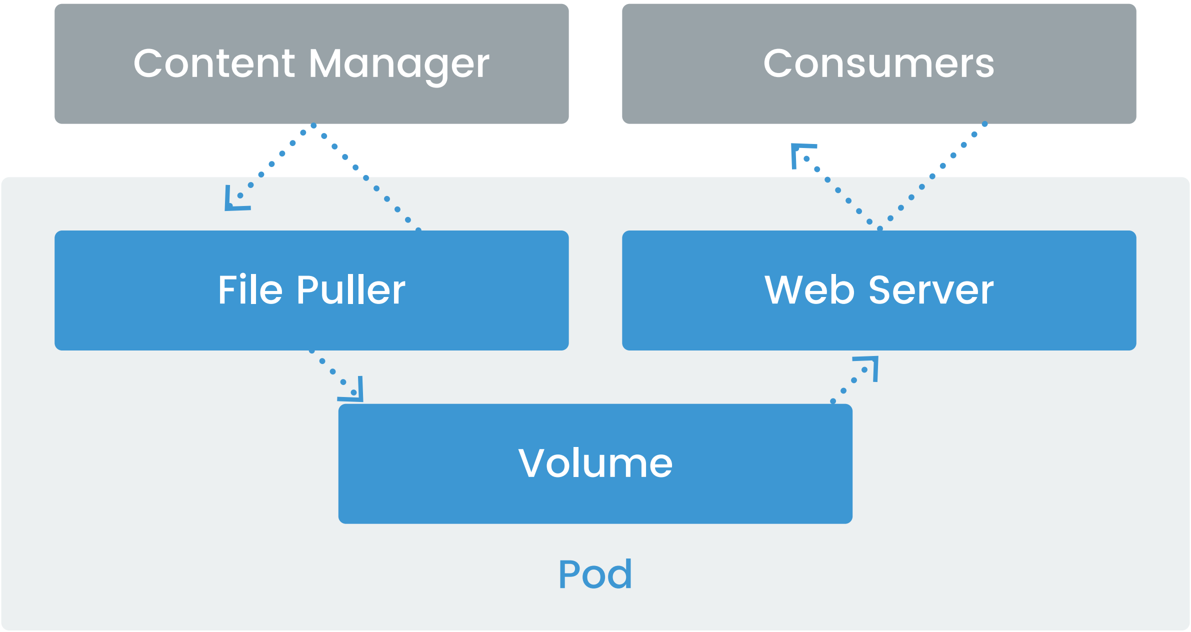
The smallest unit of deployment in a Kubernetes cluster is the Pod, and all of the constructs related to scheduling and orchestration assist in the deployment and management of Pods.

In the simplest definition, a Pod encapsulates one or more containers. Containers in the same Pod always run on the same host. They share resources such as the network namespace and storage.

Each Pod has a routable IP address assigned to it, not to the containers running within it. Having a shared network space for all containers means that the containers inside can communicate with one another over the `localhost` address, a feature not present in traditional Docker networking.

The most common use of a Pod is to run a single container. Situations where different processes work on the same shared resource, such as content in a storage volume, benefit from having multiple containers in a single Pod. Some projects inject containers into running Pods to deliver a service. An example of this is the Istio service mesh, which uses this injected container as a proxy for all communication.

Because a Pod is the basic unit of deployment, we can map it to a single instance of an application. For example, a three-tier application that runs a user interface (UI), a backend, and a database would model the deployment of the application on Kubernetes with three Pods. If one tier of the application needed to scale, the number of Pods in that tier could scale accordingly.



Workloads

Production applications with users run more than one instance of the application. This enables fault tolerance, where if one instance goes down, another handles the traffic so that users don't experience a disruption to the service. In a traditional model that doesn't use Kubernetes, these types of deployments require that an external person or software monitors the application and acts accordingly.

Kubernetes recognizes that an application might have unique requirements. Does it need to run on every host? Does it need to handle state to avoid data corruption? Can all of its pieces run anywhere, or do they need special scheduling consideration? To accommodate those situations where a default structure won't give the best results, Kubernetes provides abstractions for different workload types.

REPLICASET

The ReplicaSet maintains the desired number of copies of a Pod running within the cluster. If a Pod or the host on which it's running fails, Kubernetes launches a replacement. In all cases, Kubernetes works to maintain the desired state of the ReplicaSet.

DEPLOYMENT

A Deployment manages a ReplicaSet. Although it's possible to launch a ReplicaSet directly or to use a ReplicationController, the use of a Deployment gives more control over the rollout strategies of the Pods that the ReplicaSet controller manages. By defining the desired states of Pods through a Deployment, users can perform updates to the image running within the containers and maintain the ability to perform rollbacks.

DAEMONSET

A DaemonSet runs one copy of the Pod on each node in the Kubernetes cluster. This workload model provides the flexibility to run daemon processes such as log management, monitoring, storage providers, or network providers that handle Pod networking for the cluster.

STATEFULSET

A StatefulSet controller ensures that the Pods it manages have durable storage and persistent identity. StatefulSets are appropriate for situations where Pods have a similar definition but need a unique identity, ordered deployment and scaling, and storage that persists across Pod rescheduling.

POD NETWORKING

The Pod is the smallest unit in Kubernetes, so it is essential to first understand Kubernetes networking in the context of communication between Pods. Because a Pod can hold more than one container, we can start with a look at how communication happens between containers in a Pod. Although Kubernetes can use Docker for the underlying container runtime, its approach to networking differs slightly and imposes some basic principles:

- Any Pod can communicate with any other Pod without the use of network address translation (NAT). To facilitate this, Kubernetes assigns each Pod an IP address that is routable within the cluster.
- A node can communicate with a Pod without the use of NAT.
- A Pod's awareness of its address is the same as how other resources see the address. The host's address doesn't mask it.

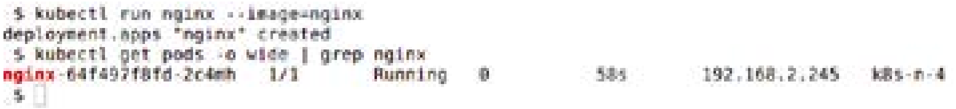
These principles give a unique and first-class identity to every Pod in the cluster. Because of this, the networking model is more straightforward and does not need to include port mapping for the running container workloads. By keeping the model simple, migrations into a Kubernetes cluster require fewer changes to the container and how it communicates.

The Pause Container

A piece of infrastructure that enables many networking features in Kubernetes is known as the pause container. This container runs alongside the containers defined in a Pod and is responsible for providing the network namespace that the other containers share. It is analogous to joining the network of another container that we described in the User Defined Network section above.

The pause container was initially designed to act as the `init` process within a PID namespace shared by all containers in the Pod. It performed the function of reaping zombie processes when a container died. PID namespace sharing is now disabled by default, so unless it has been explicitly enabled in the kubelet, all containers run their process as PID 1.

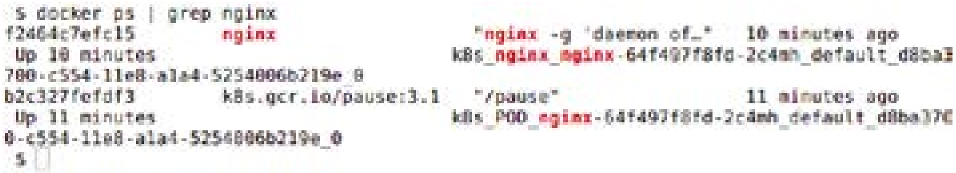
If we launch a Pod running Nginx, we can inspect the Docker container running within the Pod.



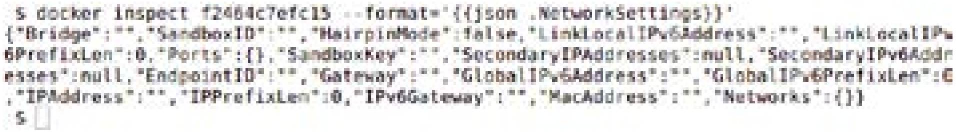
```
 $ kubectl run nginx --image=nginx
deployment.apps "nginx" created
 $ kubectl get pods -o wide | grep nginx
nginx-64f497f8fd-2c4mh   1/1       Running   0          58s       192.168.2.245   k8s-n-4
 $
```

When we do so, we see that the container does not have the network settings provided to it. The pause container which runs as part of the Pod is the one which gives the networking constructs to the Pod.

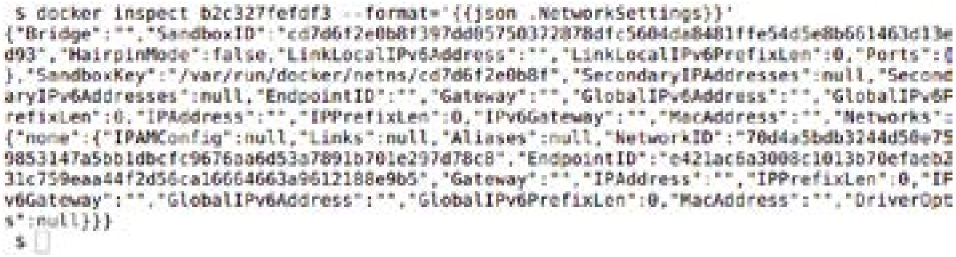
Note: Run the commands below on the host where the nginx Pod is scheduled.



```
 $ docker ps | grep nginx
f2464c7efc15        nginx                      "nginx -g 'daemon of…"   10 minutes ago
 Up 10 minutes                               k8s_nginx_nginx-64f497f8fd-2c4mh_default_d8ba3
700-c554-11e8-a1a4-5254006b219e_0
b2c327fefdf3        k8s.gcr.io/pause:3.1   "/pause"                 11 minutes ago
 Up 11 minutes                               k8s_POD_nginx-64f497f8fd-2c4mh_default_d8ba37C
0-c554-11e8-a1a4-5254006b219e_0
 $
```



```
 $ docker inspect f2464c7efc15 --format='{{json .NetworkSettings}}'
{"Bridge":"","SandboxID":"","HairpinMode":false,"LinkLocalIPv6Address":"","LinkLocalIPv
6PrefixLen":0,"Ports":{},"SandboxKey":"","SecondaryIPAddresses":null,"SecondaryIPv6Addr
esses":null,"EndpointID":"","Gateway":"","GlobalIPv6Address":"","GlobalIPv6PrefixLen":0
,"IPAddress":"","IPPrefixLen":0,"IPv6Gateway":"","MacAddress":"","Networks":{}}
 $
```



```
 $ docker inspect b2c327fefdf3 --format='{{json .NetworkSettings}}'
{"Bridge":"","SandboxID":"cd7d6f2e0b8f397dd05750372878df c5604da8481ffe54d5e8b661463d13e
d93","HairpinMode":false,"LinkLocalIPv6Address":"","LinkLocalIPv6PrefixLen":0,"Ports":{
},"SandboxKey":"/var/run/docker/netns/cd7d6f2e0b8f","SecondaryIPAddresses":null,"Second
aryIPv6Addresses":null,"EndpointID":"","Gateway":"","GlobalIPv6Address":"","GlobalIPv6P
refixLen":0,"IPAddress":"","IPPrefixLen":0,"IPv6Gateway":"","MacAddress":"","Networks":
{"none":{"IPAMConfig":null,"Links":null,"Aliases":null,"NetworkID":"70d4a5bdb3244d50e75
9853147a5bb1dbcfc9676aa6d53a7891b701e297d78c8","EndpointID":"e421ac6a3008c1013b70efacb2
31c759eaa44f2d56ca16664663a9612188e9b5","Gateway":"","IPAddress":"","IPPrefixLen":0,"IP
v6Gateway":"","GlobalIPv6Address":"","GlobalIPv6PrefixLen":0,"MacAddress":"","DriverOpt
s":null}}}
 $
```

Intra-Pod Communication

Kubernetes follows the IP-per-Pod model where it assigns a routable IP address to the Pod. The containers within the Pod share the same network space and communicate with one another over `localhost`. Like processes running on a host, two containers cannot each use the same network port, but we can work around this by changing the manifest.

Inter-Pod Communication

Because it assigns routable IP addresses to each Pod, and because it requires that all resources see the address of a Pod the same way, Kubernetes assumes that all Pods communicate with one another via their assigned addresses. Doing so removes the need for an external service discovery mechanism.

Kubernetes Service

Pods are ephemeral. The services that they provide may be critical, but because Kubernetes can terminate Pods at any time, they are unreliable endpoints for direct communication. For example, the number of Pods in a ReplicaSet might change as the Deployment scales it up or down to accommodate changes in load on the application, and it is unrealistic to expect every client to track these changes while communicating with the Pods. Instead, Kubernetes offers the Service resource, which provides a stable IP address and balances traffic across all of the Pods behind it. This abstraction brings stability and a reliable mechanism for communication between microservices.

Services which sit in front of Pods use a selector and labels to find the Pods they manage. All Pods with a label that matches the selector receive traffic through the Service. Like a traditional load balancer, the service can expose the Pod functionality at any port, irrespective of the port in use by the Pods themselves.

KUBE-PROXY

The kube-proxy daemon that runs on all nodes of the cluster allows the Service to map traffic from one port to another.

This component configures the Netfilter rules on all of the nodes according to the Service's definition in the API server. From Kubernetes 1.9 onward it uses the netlink interface to create IPVS rules. These rules direct traffic to the appropriate Pod.

KUBERNETES SERVICE TYPES

A service definition specifies the type of Service to deploy, with each type of Service having a different set of capabilities.

ClusterIP

This type of Service is the default and exists on an IP that is only visible within the cluster. It enables cluster resources to reach one another via a known address while maintaining the security boundaries of the cluster itself. For example, a database used by a backend application does not need to be visible outside of the cluster, so using a service of type ClusterIP is appropriate. The backend application would expose an API for interacting with records in the database, and a frontend application or remote clients would consume that API.

NodePort

A Service of type NodePort exposes the same port on every node of the cluster. The range of available ports is a cluster-level configuration item, and the Service can either choose one of the ports at random or have one designated in its configuration. This type of Service automatically creates a ClusterIP Service as its target, and the ClusterIP Service routes traffic to the Pods.

External load balancers frequently use NodePort services. They receive traffic for a specific site or address and forward it to the cluster on that specific port.

LoadBalancer

When working with a cloud provider for whom support exists within Kubernetes, a Service of type LoadBalancer creates a load balancer in that provider's infrastructure. The exact details of how this happens differ between providers, but all create the load balancer asynchronously and configure it to proxy the request to the corresponding Pods via NodePort and ClusterIP Services that it also creates.

In a later section, we explore Ingress Controllers and how to use them to deliver a load balancing solution for a cluster.

DNS

As we stated above, Pods are ephemeral, and because of this, their IP addresses are not reliable endpoints for communication. Although Services solve this by providing a stable address in front of a group of Pods, consumers of the Service still want to avoid using an IP address. Kubernetes solves this by using DNS for service discovery.

The default internal domain name for a cluster is `cluster.local`. When you create a Service, it assembles a subdomain of `namespace.svc.cluster.local` (where namespace is the namespace in which the service is running) and sets its name as the hostname. For example, if the service was named nginx and ran in the default namespace, consumers of the service would be able to reach it as `nginx.default.svc.cluster.local`. If the service's IP changes, the hostname remains the same. There is no interruption of service.

The default DNS provider for Kubernetes is KubeDNS, but it's a pluggable component. Beginning with Kubernetes 1.11 CoreDNS is available as an alternative. In addition to providing the same basic DNS functionality within the cluster, CoreDNS supports a wide range of plugins to activate additional functionality.

NETWORK POLICY

In an enterprise deployment of Kubernetes the cluster often supports multiple projects with different goals. Each of these projects has different workloads, and each of these might require a different security policy.

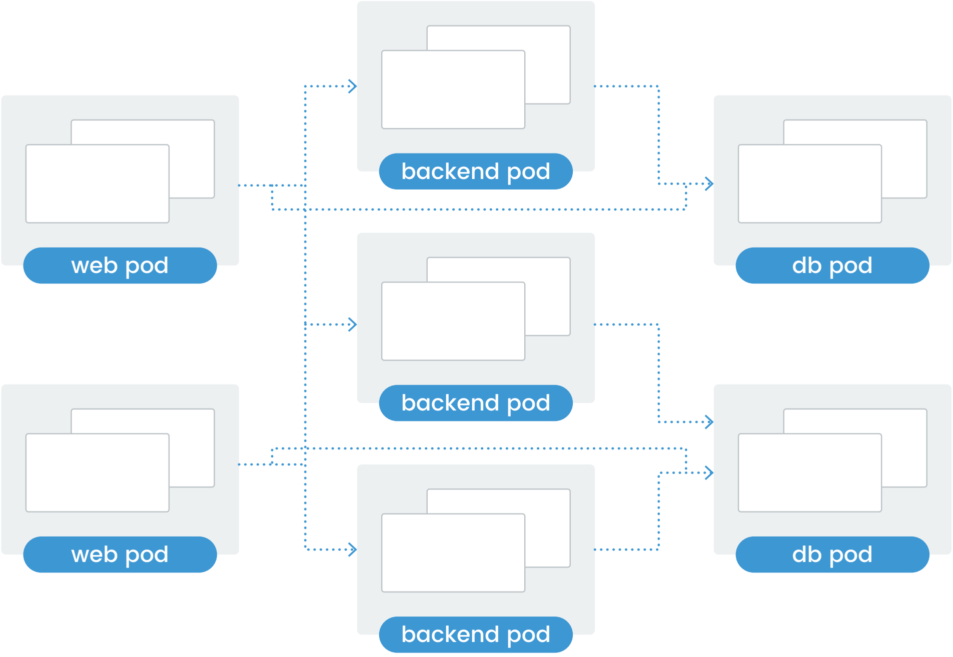
Pods, by default, do not filter incoming traffic. There are no firewall rules for inter-Pod communication. Instead, this responsibility falls to the NetworkPolicy resource, which uses a specification to define the network rules applied to a set of Pods.

The network policies are defined in Kubernetes, but the CNI plugins that support network policy implementation do the actual configuration and processing. In a later section, we look at CNI plugins and how they work.

The image to the right shows a standard three-tier application with a UI, a backend service, and a database, all deployed within a Kubernetes cluster.

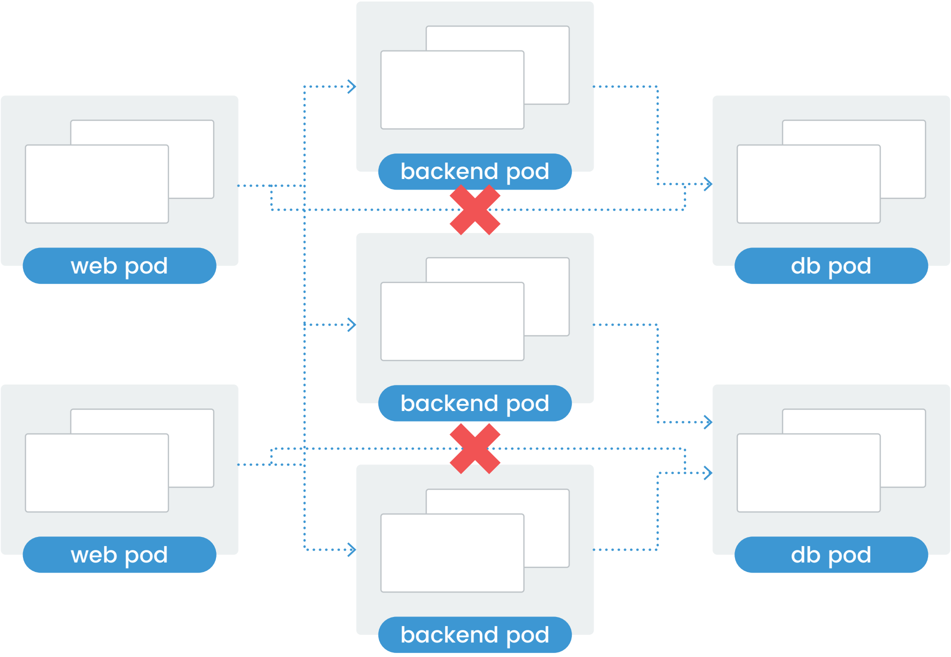
Requests to the application arrive at the web Pods, which then initiate a request to the backend Pods for data. The backend Pods process the request and perform CRUD operations against the database Pods.

If the cluster is not using a network policy, any Pod can talk to any other Pod. Nothing prevents the web Pods from communicating directly with the database Pods. If the security requirements of the cluster dictate a need for clear separation between tiers, a network policy enforces it.



The policy defined below states that the database Pods can only receive traffic from the Pods with the labels `app=myapp` and `role=backend`. It also defines that the backend Pods can only receive traffic from Pods with the labels `app=myapp` and `role=web`.

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
 name: backend-access-ingress
spec:
 podSelector:
   matchLabels:
     app: myapp
     role: backend
 ingress:
 - from:
   - podSelector:
       matchLabels:
         app: myapp
         role: web
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
 name: db-access-ingress
spec:
 podSelector:
   matchLabels:
     app: myapp
     role: db
 ingress:
 - from:
   - podSelector:
       matchLabels:
         app: myapp
         role: backend
```



With this network policy in place, Kubernetes blocks communication between the web and database tiers.

How a Network Policy Works

In addition to the fields used by all Kubernetes manifests, the specification of the NetworkPolicy resource requires some extra fields.

PODSELECTOR

This field tells Kubernetes how to find the Pods to which this policy applies. Multiple network policies can select the same set of Pods, and the ingress rules are applied sequentially. The field is not optional, but if the manifest defines a key with no value, it applies to all Pods in the namespace.

POLICYTYPES

This field defines the direction of network traffic to which the rules apply. If missing, Kubernetes interprets the rules and only applies them to ingress traffic unless egress rules also appear in the rules list. This default interpretation simplifies the manifest's definition by having it adapt to the rules defined later.

Because Kubernetes always defines an ingress policy if this field is unset, a network policy for egress-only rules must explicitly define the `policyType` of `Egress`.

EGRESS

Rules defined under this field apply to egress traffic from the selected Pods to destinations defined in the rule. Destinations can be an IP block (`ipBlock`), one or more Pods (`podSelector`), one or more namespaces (`namespaceSelector`), or a combination of both `podSelector` and `nameSpaceSelector`.

The following rule permits traffic from the Pods to any address in `10.0.0.0/24` and only on TCP port 5978:

```
egress:
- to:
  - ipBlock:
      cidr: 10.0.0.0/24
  ports:
  - protocol: TCP
    port: 5978
```

The next rule permits outbound traffic for Pods with the labels `app=myapp` and `role=backend` to any host on TCP or UDP port 53:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
 name: db-egress-denyall
spec:
 podSelector:
   matchLabels:
     app: myapp
     role: backend
 policyTypes:
 - Egress
 egress:
 - ports:
   - port: 53
     protocol: UDP
   - port: 53
     protocol: TCP
```

Egress rules work best to limit a resource's communication to the other resources on which it relies. If those resources are in a specific block of IP addresses, use the `ipBlock` selector to target them, specifying the appropriate ports:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
 name: db-egress-denyall
spec:
 podSelector:
   matchLabels:
     app: myapp
     role: backend
 policyTypes:
 - Egress
 egress:
 - ports:
   - port: 53
     protocol: UDP
   - port: 53
     protocol: TCP
 - to:
   - ipBlock:
       cidr: 10.0.0.0/24
   ports:
   - protocol: TCP
     port: 3306
```

INGRESS

Rules listed in this field apply to traffic that is inbound to the selected Pods. If the field is empty, all inbound traffic will be blocked. The example below permits inbound access from any address in `172.17.0.0/16` unless it's within `172.17.1.0/24`. It also permits traffic from any Pod in the namespace `myproject`.

(Note the subtle distinction in how the rules are listed. Because `namespaceSelector` is a separate item in the list, it matches with an or value. Had `namespaceSelector` been listed as an additional key in the first list item, it would permit traffic that came from the specified `ipBlock` and was also from the namespace `myproject`.)

```
ingress:
- from:
  - ipBlock:
      cidr: 172.17.0.0/16
      except:
      - 172.17.1.0/24
  - namespaceSelector:
      matchLabels:
        project: myproject
  - podSelector:
      matchLabels:
        role: frontend
  ports:
  - protocol: TCP
    port: 6379
```

The next policy permits access to the Pods labeled `app=myapp` and `role=web` from all sources, external or internal.

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
 name: web-allow-all-access
spec:
 podSelector:
   matchLabels:
     app: myapp
     role: web
 ingress:
 - from: []
```

Consider, however, that this allows traffic to any port on those Pods. Even if no other ports are listening, the principle of least privilege states that we only want to expose what we need to expose for the services to work. The following modifications to the NetworkPolicy take this rule into account by only allowing inbound traffic to the ports where our Service is running.

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
 name: web-allow-all-access-specific-port
spec:
 podSelector:
   matchLabels:
     app: myapp
     role: web
 ingress:
 - ports:
   - port: 8080
   from: []
```

Apart from opening incoming traffic on certain ports, you can also enable all traffic from a set of Pods inside the cluster. This enables a few trusted applications to reach out from the application on all ports and is especially useful when workloads in a cluster communicate with each other over many random ports. The opening of traffic from certain Pods is achieved by using labels as described in the policy below:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
 name: web-allow-internal-port80
spec:
 podSelector:
   matchLabels:
     app: "myapp"
     role: "web"
 ingress:
 - ports:
   - port: 8080
   from:
   - podSelector:
       matchLabels:
         app: "mytestapp"
         role: "web-test-client"
```

Even if a Service listens on a different port than where the Pod's containers listen, use the container ports in the network policy. Ingress rules affect inter-Pod communication, and the policy does not know about the abstraction of the service.

CONTAINER NETWORKING INTERFACE

The Container Networking Interface (CNI) project is also under the governance of the CNCF. It provides a specification and a series of libraries for writing plugins to configure network interfaces in Linux containers.

The specification requires that providers implement their plugin as a binary executable that the container engine invokes. Kubernetes does this via the Kubelet process running on each node of the cluster.

The CNI specification expects the container runtime to create a new network namespace before invoking the CNI plugin. The plugin is then responsible for connecting the container's network with that of the host. It does this by creating the virtual Ethernet devices that we discussed earlier.

Kubernetes and CNI

Kubernetes natively supports the CNI model. It gives its users the freedom to choose the network provider or product best suited for their needs.

To use the CNI plugin, pass `--network-plugin=cni` to the Kubelet when launching it. If your environment is not using the default configuration directory (`/etc/cni/net.d`), pass the correct configuration directory as a value to `--cni-conf-dir`. The Kubelet looks for the CNI plugin binary at `/opt/cni/bin`, but you can specify an alternative location with `--cni-bin-dir`.

The CNI plugin provides IP address management for the Pods and builds routes for the virtual interfaces. To do this, the plugin interfaces with an IPAM plugin that is also part of the CNI specification. The IPAM plugin must also be a single executable that the CNI plugin consumes. The role of the IPAM plugin is to provide to the CNI plugin the gateway, IP subnet, and routes for the Pod.

Networking with Flannel

Flannel is one of the most straightforward network providers for Kubernetes. It operates at Layer 3 and offloads the actual packet forwarding to a backend such as VxLAN or IPSec. It assigns a large network to all hosts in the cluster and then assigns a portion of that network to each host. Routing between containers on a host happens via the usual channels, and Flannel handles routing between hosts using one of its available options.

Flannel uses etcd to store the map of what network is assigned to which host. The target can be an external deployment of etcd or the one that Kubernetes itself uses.

Flannel does not provide an implementation of the NetworkPolicy resource.

RUNNING FLANNEL WITH KUBERNETES

Flannel Pods roll out as a DaemonSet, with one Pod assigned to each host. To deploy it within Kubernetes, use the `kube-flannel.yaml` manifest from the Flannel repository on Github.

Once Flannel is running, it is not possible to change the network address space or the backend communication format without cluster downtime.

Network Type	Backend	Key features
Overlay	VxLAN	• Fast, but with no interhost encryption • Suitable for private/secure networks
Overlay	IPSec	• Encrypts traffic between hosts • Suitable when traffic traverses the Internet
Non Overlay	Host-gw	• Good performance • Cloud agnostic
Non Overlay	AWS VPC	• Good performance • Limited to Amazon's cloud

FLANNEL BACKENDS

VxLAN

VxLAN is the simplest of the officially supported backends for Flannel. Encapsulation happens within the kernel, so there is no additional overhead caused by moving data between the kernel and user space.

The VxLAN backend creates a Flannel interface on every host. When a container on one node wishes to send traffic to a different node, the packet goes from the container to the bridge interface in the host's network namespace. From there the bridge forwards it to the Flannel interface because the kernel route table designates that this interface is the target for the non-local portion of the overlay network. The Flannel network driver wraps the packet in a UDP packet and sends it to the target host.

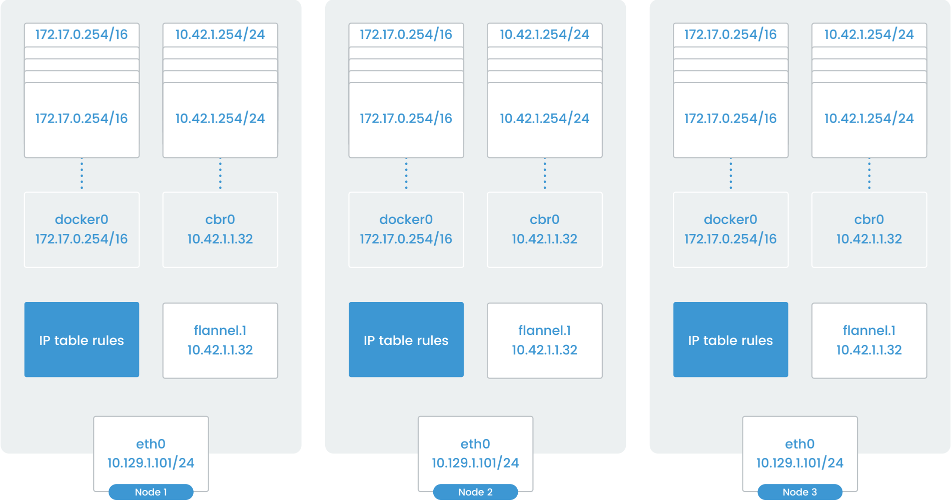
Once it arrives at its destination, the process flows in reverse, with the Flannel driver on the destination host unwrapping the packet, sending it to the bridge interface, and from there the packet finds its way into the overlay network and to the destination Pod.

Host-gw

The Host-gw backend provides better performance than VxLAN but requires Layer 2 connectivity between hosts. It operates by creating IP routes to subnets via remote machine addresses.

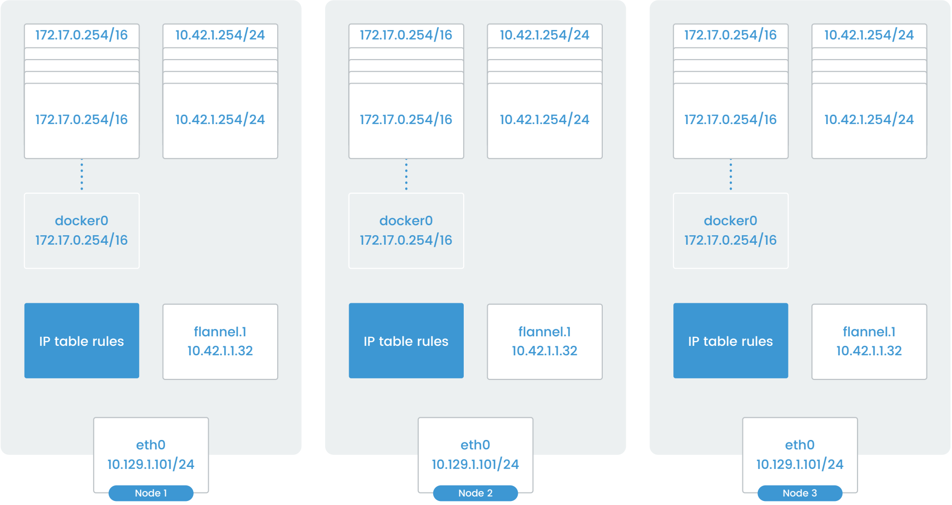
Unlike VxLAN, no Flannel interface is created when using this backend. Instead, each node sends traffic directly to the destination node where the remote network is located.

This backend may require additional network configuration if used in a cloud provider where inter-host communication uses virtual switches.



UDP

The UDP backend is insecure and should only be used for debugging or if the kernel does not support VxLAN.



Networking with Calico

ARCHITECTURE

Calico operates at Layer 3 and assigns every workload a routable IP address. It prefers to operate by using BGP without an overlay network for the highest speed and efficiency, but in scenarios where hosts cannot directly communicate with one another, it can utilize an overlay solution such as VxLAN or IP-in-IP.

Calico supports network policies for protecting workloads and nodes from malicious activity or aberrant applications.

The Calico networking Pod contains a CNI container, a container that runs an agent that tracks Pod deployments and registers addresses and routes, and a daemon that announces the IP and route information to the network via the Border Gateway Protocol (BGP). The BGP daemons build a map of the network that enables cross-host communication.

Calico requires a distributed and fault-tolerant key/value datastore, and deployments often choose etcd to deliver this component. Calico uses it to store metadata about routes, virtual interfaces, and network policy objects. The Felix agent in the `calico-node` Pod communicates with etcd to publish this information. Calico can use a dedicated HA deployment of etcd, or it can use the Kubernetes etcd datastore via the Kubernetes API. Please see the Calico deployment documentation to understand the functional restrictions that are present when using the Kubernetes API for storing Calico data.

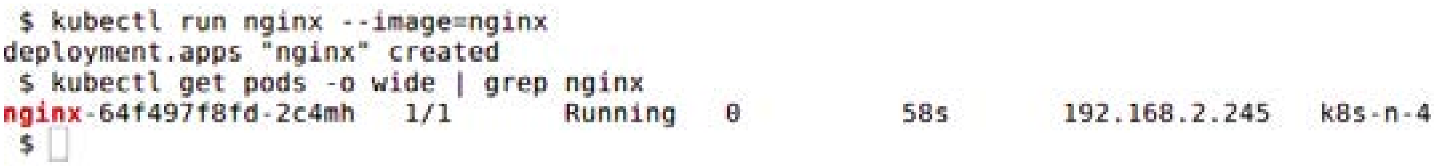
The final piece of a Calico deployment is the controller. Although presented as a single object, it is a set of controllers that run as a control loop within Kubernetes to manage policy, workload endpoints, and node changes.

- The Policy Controller watches for changes in the defined network policies and translates them into Calico network policies.
- The Profile Controller watches for the addition or removal of namespaces and programs Calico objects called Profiles.
- Calico stores Pod information as workload endpoints. The Workload Endpoint Controller watches for updates to labels on the Pod and updates the workload endpoints.
- The Node Controller loop watches for the addition or removal of Kubernetes nodes and updates the kvdb with the corresponding data.

INSTALL CALICO WITH KUBERNETES

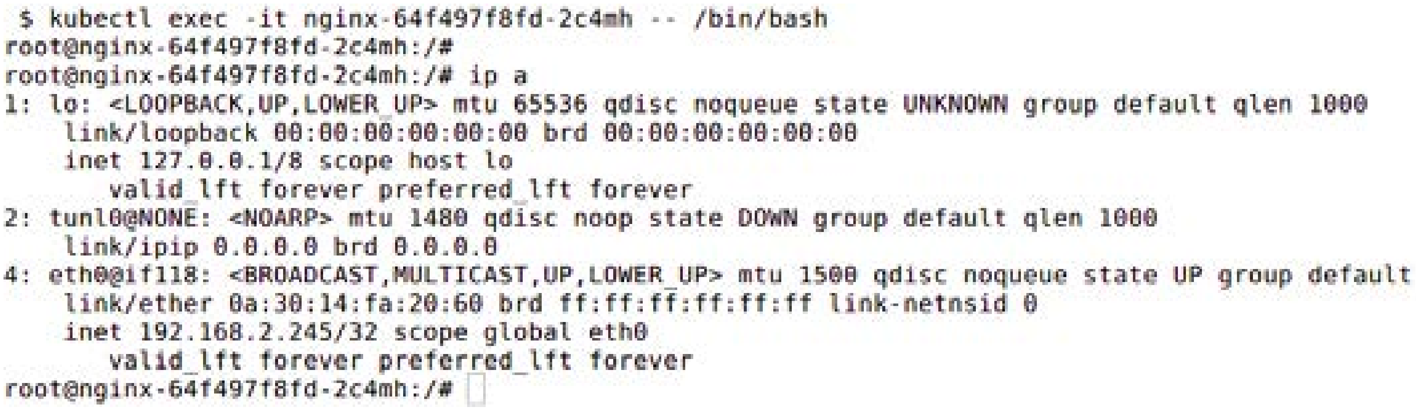
The latest instructions for installing Calico are present on the Calico Project website at https://docs.projectcalico.org. For this section, you need a Kubernetes cluster running the Calico network backend.

When the cluster is ready, deploy a Pod running Nginx:



```
 $ kubectl run nginx --image=nginx
deployment.apps "nginx" created
 $ kubectl get pods -o wide | grep nginx
nginx-64f497f8fd-2c4mh   1/1       Running   0          58s       192.168.2.245   k8s-n-4
 $ 
```

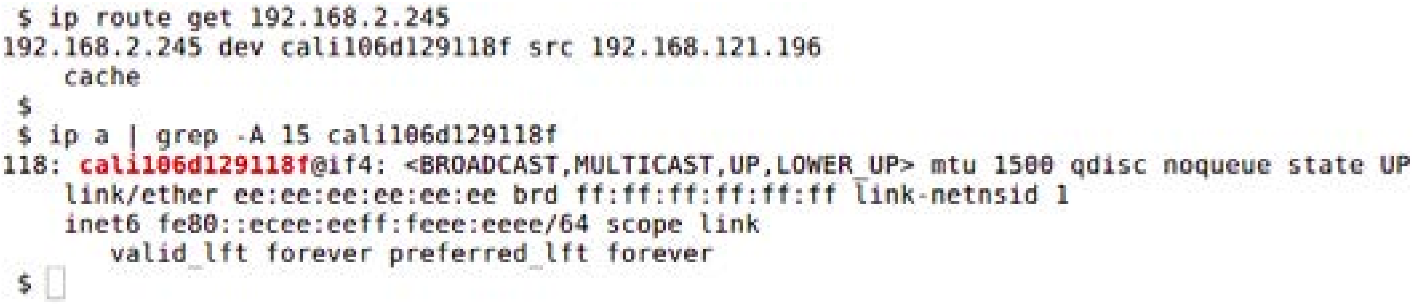
Users can manage Calico objects within the Kubernetes cluster via the command-line tool `calicoctl`*. The tool's only requirement is that it can reach the Calico datastore.*



```
 $ kubectl exec -it nginx-64f497f8fd-2c4mh -- /bin/bash
root@nginx-64f497f8fd-2c4mh:/#
root@nginx-64f497f8fd-2c4mh:/# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    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
2: tunl0@NONE: <NOARP> mtu 1480 qdisc noop state DOWN group default qlen 1000
    link/ipip 0.0.0.0 brd 0.0.0.0
4: eth0@if118: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 0a:30:14:fa:20:60 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.2.245/32 scope global eth0
       valid_lft forever preferred_lft forever
root@nginx-64f497f8fd-2c4mh:/#
```

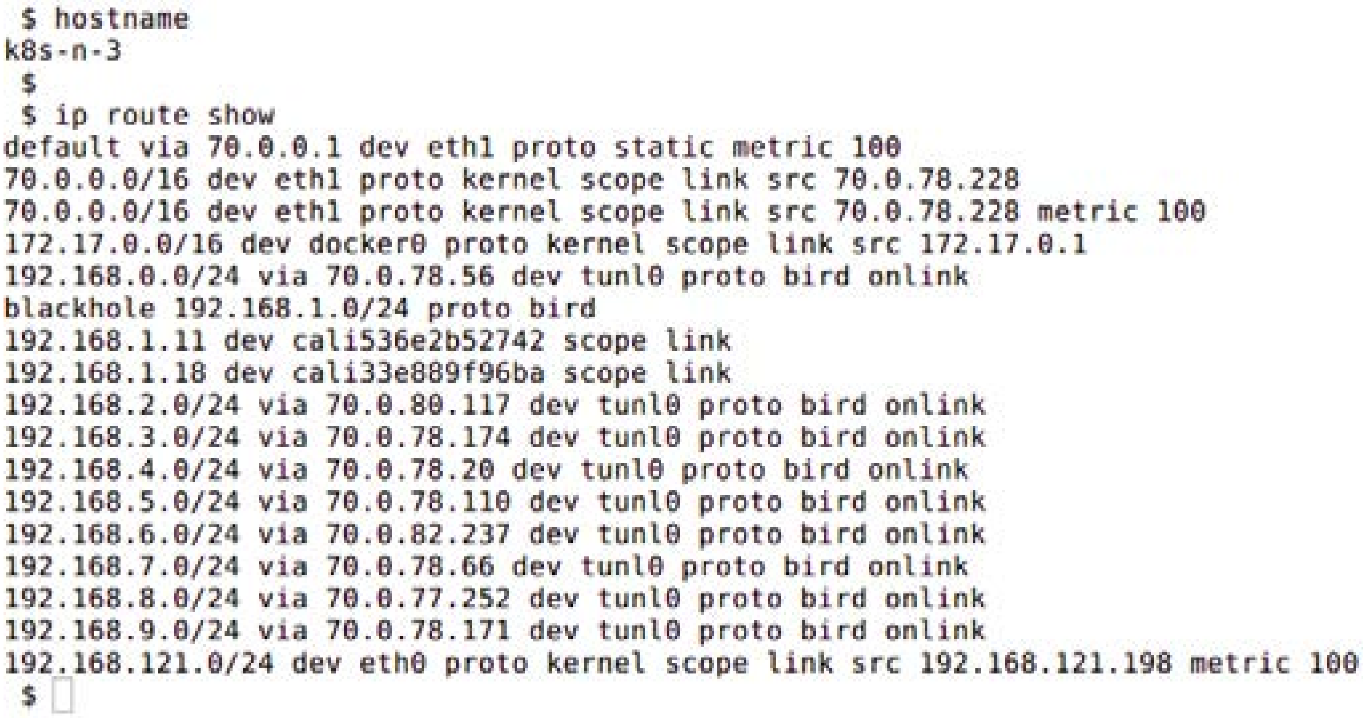
Note the IP address and the `eth0` interface within the Pod:

In the output below, note that the routing table indicates that a local interface (`cali106d129118f`) handles traffic for the IP address of the Pod. The `calico-node` Pod creates this interface and propagates the routes to other nodes in the cluster.

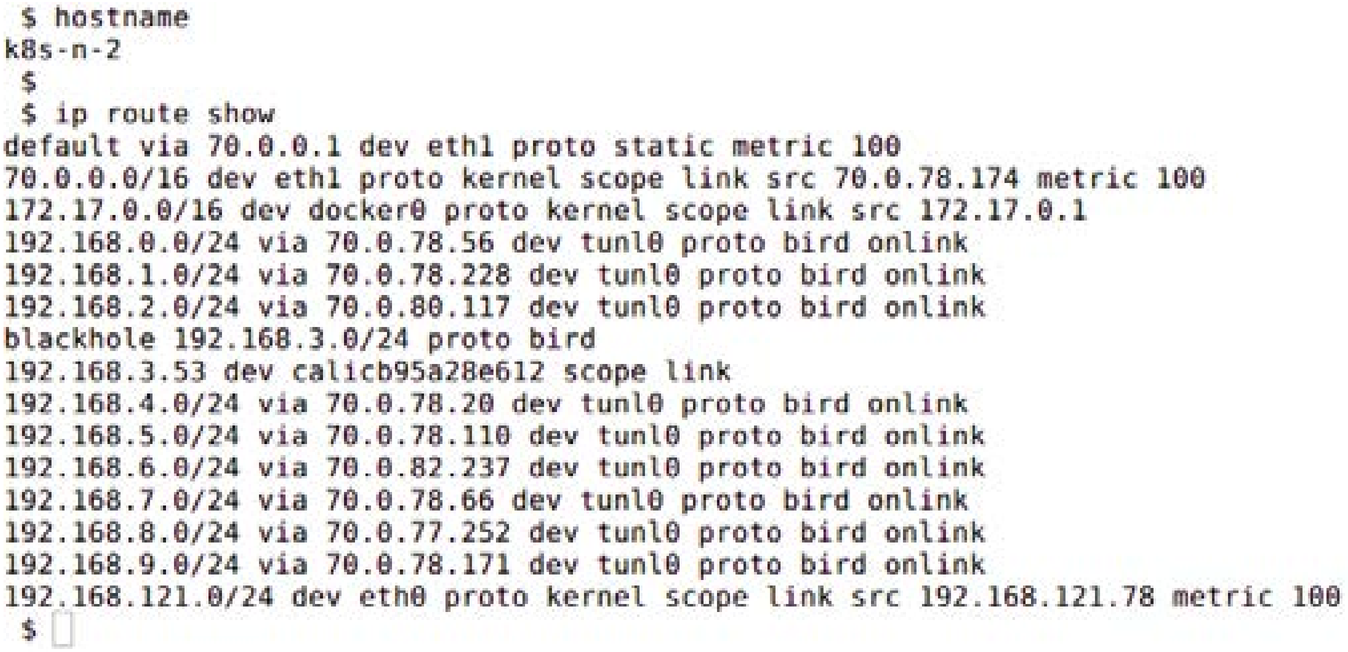


```
 $ ip route get 192.168.2.245
192.168.2.245 dev cali106d129118f src 192.168.121.196
    cache
 $
 $ ip a | grep -A 15 cali106d129118f
118: cali106d129118f@if4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    link/ether ee:ee:ee:ee:ee:ee brd ff:ff:ff:ff:ff:ff link-netnsid 1
    inet6 fe80::ecee:eeff:feee:eeee/64 scope link
       valid_lft forever preferred_lft forever
 $
```

Kubernetes scheduled our Pod to run on `k8s-n-1`. If we look at the route table on the other two nodes, we see that each directs 192.168.2.0/24 to 70.0.80.117, which is the address of `k8s-n-1`.



```
 $ hostname
k8s-n-3
 $
 $ ip route show
default via 70.0.0.1 dev eth1 proto static metric 100
70.0.0.0/16 dev eth1 proto kernel scope link src 70.0.78.228
70.0.0.0/16 dev eth1 proto kernel scope link src 70.0.78.228 metric 100
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1
192.168.0.0/24 via 70.0.78.56 dev tunl0 proto bird onlink
blackhole 192.168.1.0/24 proto bird
192.168.1.11 dev cali536e2b52742 scope link
192.168.1.18 dev cali33e889f96ba scope link
192.168.2.0/24 via 70.0.80.117 dev tunl0 proto bird onlink
192.168.3.0/24 via 70.0.78.174 dev tunl0 proto bird onlink
192.168.4.0/24 via 70.0.78.20 dev tunl0 proto bird onlink
192.168.5.0/24 via 70.0.78.110 dev tunl0 proto bird onlink
192.168.6.0/24 via 70.0.82.237 dev tunl0 proto bird onlink
192.168.7.0/24 via 70.0.78.66 dev tunl0 proto bird onlink
192.168.8.0/24 via 70.0.77.252 dev tunl0 proto bird onlink
192.168.9.0/24 via 70.0.78.171 dev tunl0 proto bird onlink
192.168.121.0/24 dev eth0 proto kernel scope link src 192.168.121.198 metric 100
 $
```



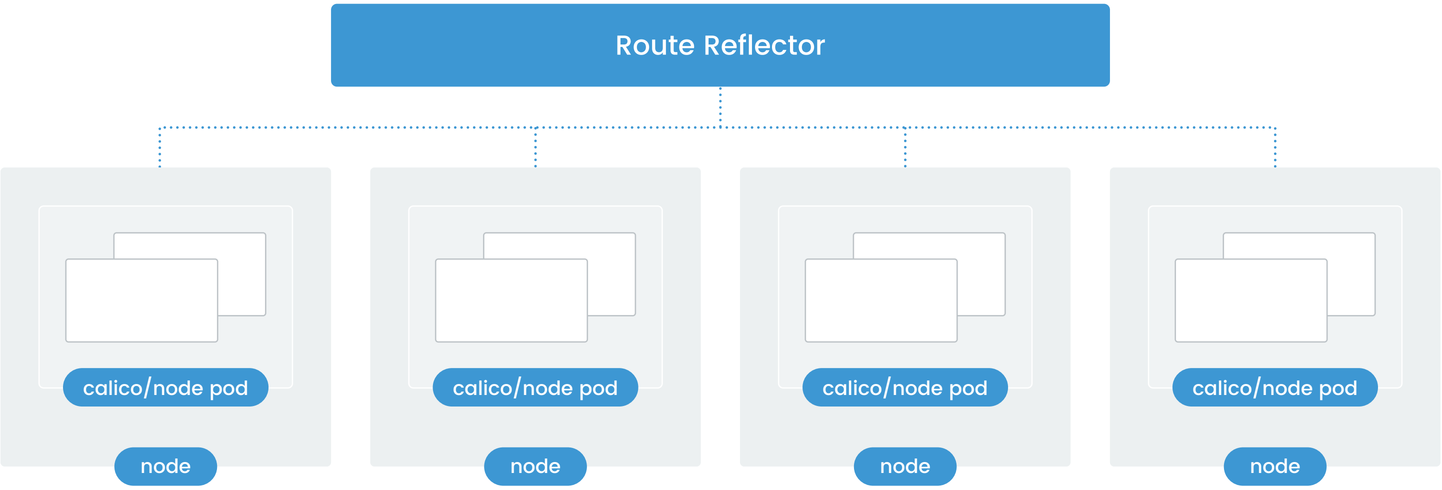
```
 $ hostname
k8s-n-2
 $
 $ ip route show
default via 70.0.0.1 dev eth1 proto static metric 100
70.0.0.0/16 dev eth1 proto kernel scope link src 70.0.78.174 metric 100
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1
192.168.0.0/24 via 70.0.78.56 dev tunl0 proto bird onlink
192.168.1.0/24 via 70.0.78.228 dev tunl0 proto bird onlink
192.168.2.0/24 via 70.0.80.117 dev tunl0 proto bird onlink
blackhole 192.168.3.0/24 proto bird
192.168.3.53 dev calicb95a28e612 scope link
192.168.4.0/24 via 70.0.78.20 dev tunl0 proto bird onlink
192.168.5.0/24 via 70.0.78.110 dev tunl0 proto bird onlink
192.168.6.0/24 via 70.0.82.237 dev tunl0 proto bird onlink
192.168.7.0/24 via 70.0.78.66 dev tunl0 proto bird onlink
192.168.8.0/24 via 70.0.77.252 dev tunl0 proto bird onlink
192.168.9.0/24 via 70.0.78.171 dev tunl0 proto bird onlink
192.168.121.0/24 dev eth0 proto kernel scope link src 192.168.121.78 metric 100
 $
```

```
apiVersion: projectcalico.org/v3
kind: BGPConfiguration
metadata:
  name: default
spec:
  logSeverityScreen: Info
  nodeToNodeMeshEnabled: false
  asNumber: 63400
```

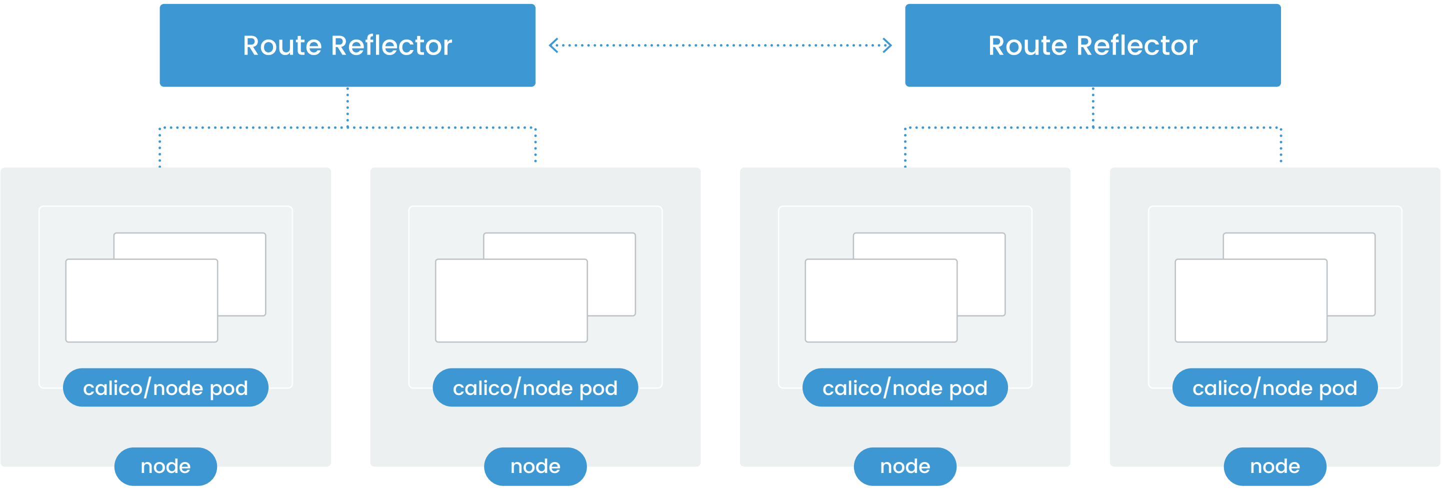
Next, use `calicoctl` to show the autonomous system number (ASN) for each node in the Kubernetes cluster.

```
calicoctl get nodes --output=wide
```

The `calico-node` Pods use one of two methods to build the peering relationship with external peers: global peering or per-node peering.



A single route reflector.



Multiple route reflectors configured within a Kubernetes cluster.

GLOBAL BGP PEERING

If the network has a device that we want to have all of the nodes peer with, we can create a global `BGPPeer` resource within the cluster. Doing it this way assures that we only have to create the configuration once for it to be applied correctly everywhere.

```
$ calicoctl create -f - << EOF
apiVersion: projectcalico.org/v3
kind: BGPPeer
metadata:
  name: bgppeer-global
  peerIP: <IP>
  scope: global
spec:
  asNumber: <ASN>
EOF
```

Use the ASN retrieved above and the IP of the external peer.

To remove a global BGP peer, use the `calicoctl` command:

```
$ calicoctl delete bgpPeer <IP> --scope=global
```

You can view the current list of BGP Peers with the following:

```
$ calicoctl get bgpPeer --scope=global
```

PER NODE BGP PEERING

To create a network topology where only a subset of nodes peers with certain external devices, we create a per-node `BGPPeer` resource within the cluster.

```
$ cat << EOF | calicoctl create -f -
apiVersion: projectcalico.org/v3
kind: BGPPeer
metadata:
  name: bgppeer-2
  peerIP: <IP>
  Node: <NODENAME>
spec:
  asNumber: <ASN>
EOF
```

As before, use the ASN for the Calico network and the IP of the BGP peer. Specify the node to which this configuration applies.

You can remove a per-node BGP peer or view the current per-node configuration with `calicoctl`:

```
$ calicoctl delete bgpPeer <IP> --scope=node --node=<NODENAME>
$ calicoctl get bgpPeer --node=<NODENAME>
```

USING IP-IN-IP

If we're unable to use BGP, perhaps because we're using a cloud provider or another environment where we have limited control over the network or no permission to peer with other routers, Calico's IP-in-IP mode encapsulates packets before sending them to other nodes.

To enable this mode, define the `ipipMode` field on the `IPPool` resource:

```
apiVersion: projectcalico.org/v3
kind: IPPool
metadata:
  name: project1IPPool
spec:
  cidr: 10.11.12.0/16
  ipipMode: CrossSubnet
  natOutgoing: true
```

After activating IP-in-IP, Calico wraps inter-Pod packets in a new packet with headers that indicate the source of the packet is the host with the originating Pod, and the target of the packet is the host with the destination Pod. The Linux kernel performs this encapsulation and then forwards the packet to the destination host where it is unwrapped and delivered to the destination Pod.

IP-in-IP has two modes of operation:

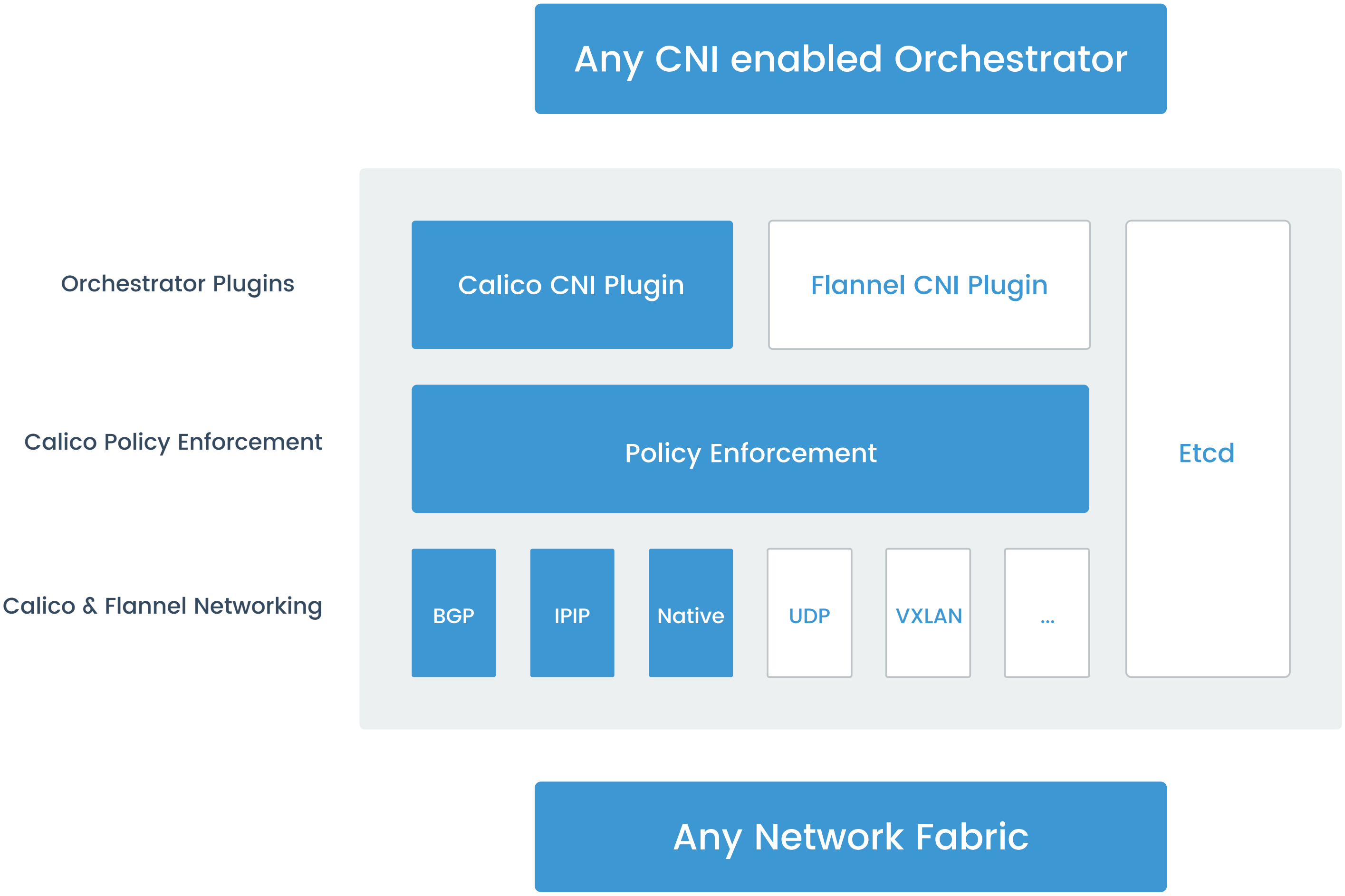
1. `Always`: This is the default mode if an `IPPool` resource is defined.
2. `CrossSubnet`: This only performs IP encapsulation for traffic which crosses subnet boundaries. Doing this provides a performance benefit on networks where cluster members within separate Layer 2 boundaries have routers between them because it performs encapsulation intelligently, only using it for the cross-subnet traffic.

For the `CrossSubnet` mode to work, each Calico node must use the IP address and subnet mask for the host. For more information on this, see the Calico documentation for IP-in-IP.

Combining Flannel and Calico (Canal)

For some time an effort to integrate Flannel's easy overlay networking engine and Calico's network policy enforcement ran under the project name Canal. The maintainers deprecated it as a separate project, and instead, the Calico documentation contains instructions on deploying Flannel and Calico together.

They only abandoned the name and status; the result remains the same. Flannel provides an overlay network using one of its backends, and Calico provides granular access control to the running workloads with its network policy implementation.



Load Balancers and Ingress Controllers

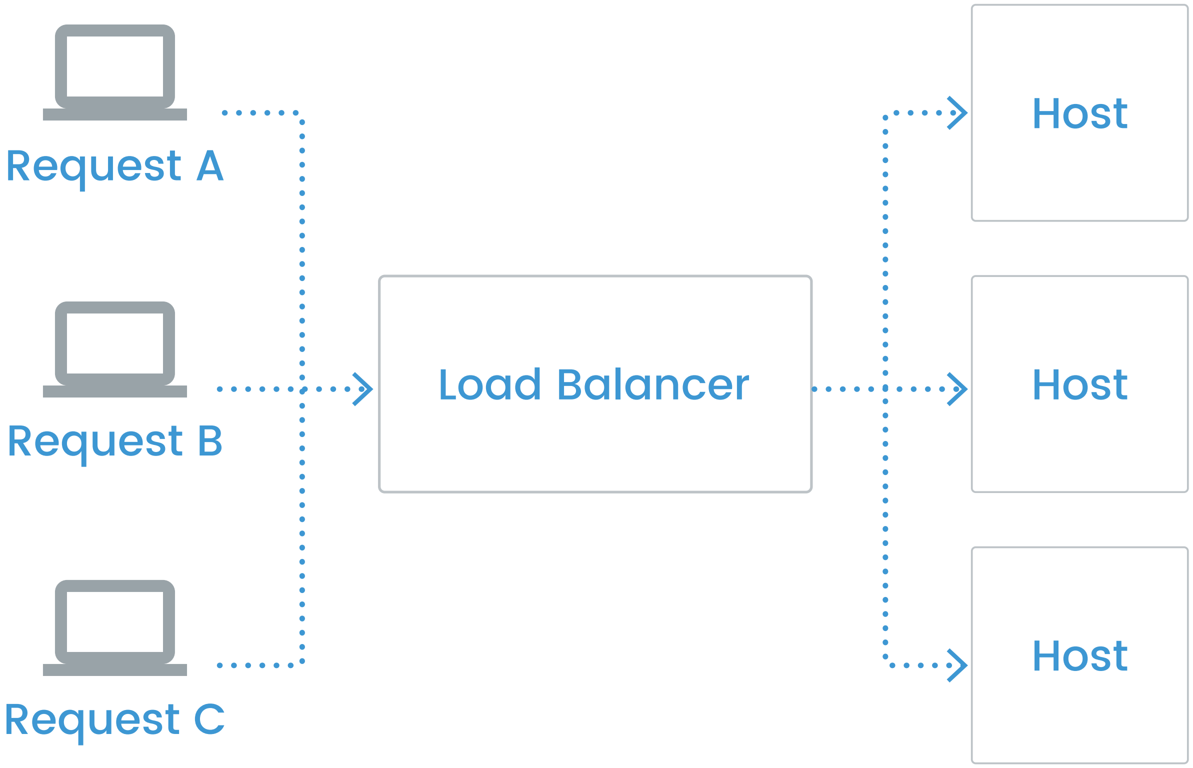
Up until now, we've focused on how to configure networking and how the various providers work in a Kubernetes cluster. While these systems define and control communication within the cluster and between its nodes, they do not, on their own, address how traffic from outside of the cluster finds its way to a destination or what part DNS plays in that process. To understand the full picture, we need to explore how Kubernetes approaches load balancing and DNS.

THE BENEFITS OF LOAD BALANCERS

A load balancer provides valuable features for any deployment, whether it's running inside or outside of the Kubernetes cluster. In addition to distributing load across multiple backends, a load balancer can also move TLS processing to a central location, route traffic based on the requester's hardware or browser, the requested site, or a path within the URL, or it can enable canary deployments and zero-downtime upgrades.

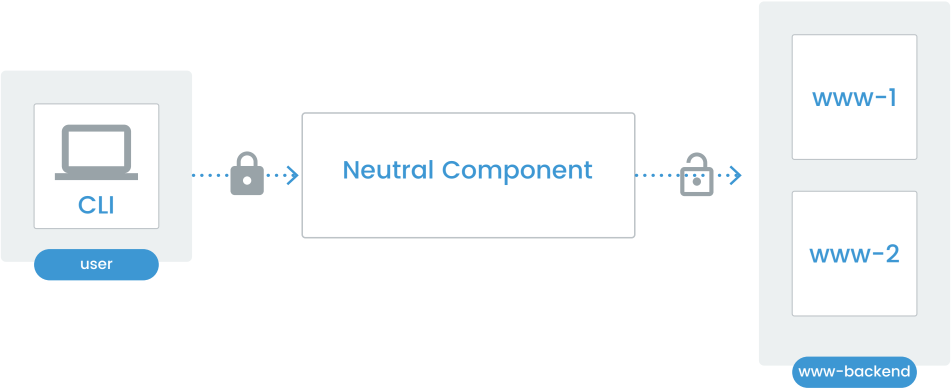
Load Distribution

When client requests arrive, the load balancer directs them across a pool of worker nodes commonly referred to as backends. Because the load balancer presents itself as the endpoint for the site, the clients don't know anything about these backends. The load balancer tracks the health and number of connections to each backend, and it works according to its configured policy to evenly distribute the traffic. If a backend fails or becomes overloaded, the load balancer stops sending traffic to it until it returns to a healthy state. This scenario enables horizontal scaling, where a site can scale capacity by adding and removing backends.



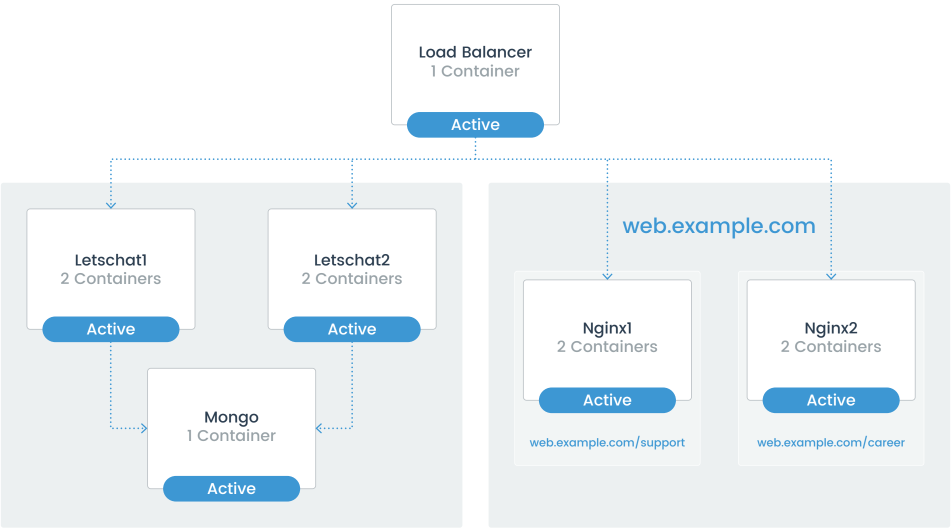
SSL/TLS Termination

The overhead of encrypting and decrypting data can impact the performance of a backend, so deployments frequently move this work to the load balancer. Encrypted traffic lands on the load balancer, which decrypts it and forwards it to a backend. By operating with a decrypted data stream, the load balancer can make informed decisions about how to route the data because it's now able to see more than the basic metadata present in the flow.



Routing By HTTP Host or Path

Organizations who run multiple applications frequently group them under the same logical namespace: their domain name. In this scenario, a load balancer routes traffic based on parameters such as the requested host or site (the Host header), or by the path requested in the URL.

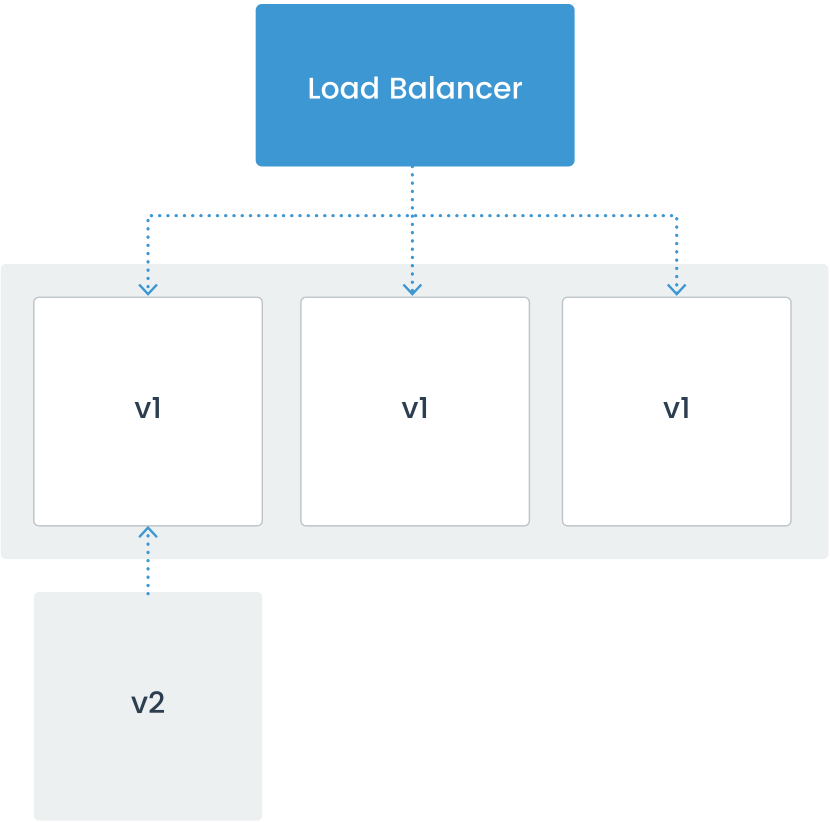


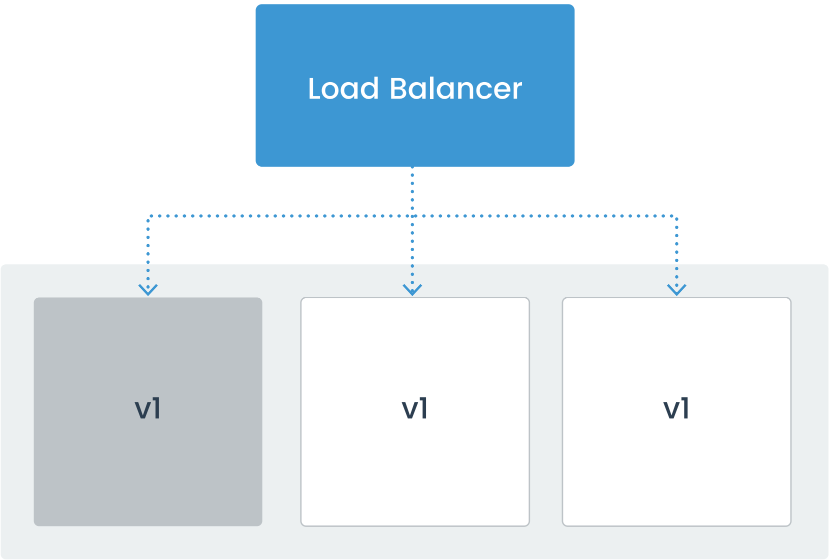
Upgrades and Feature Flags

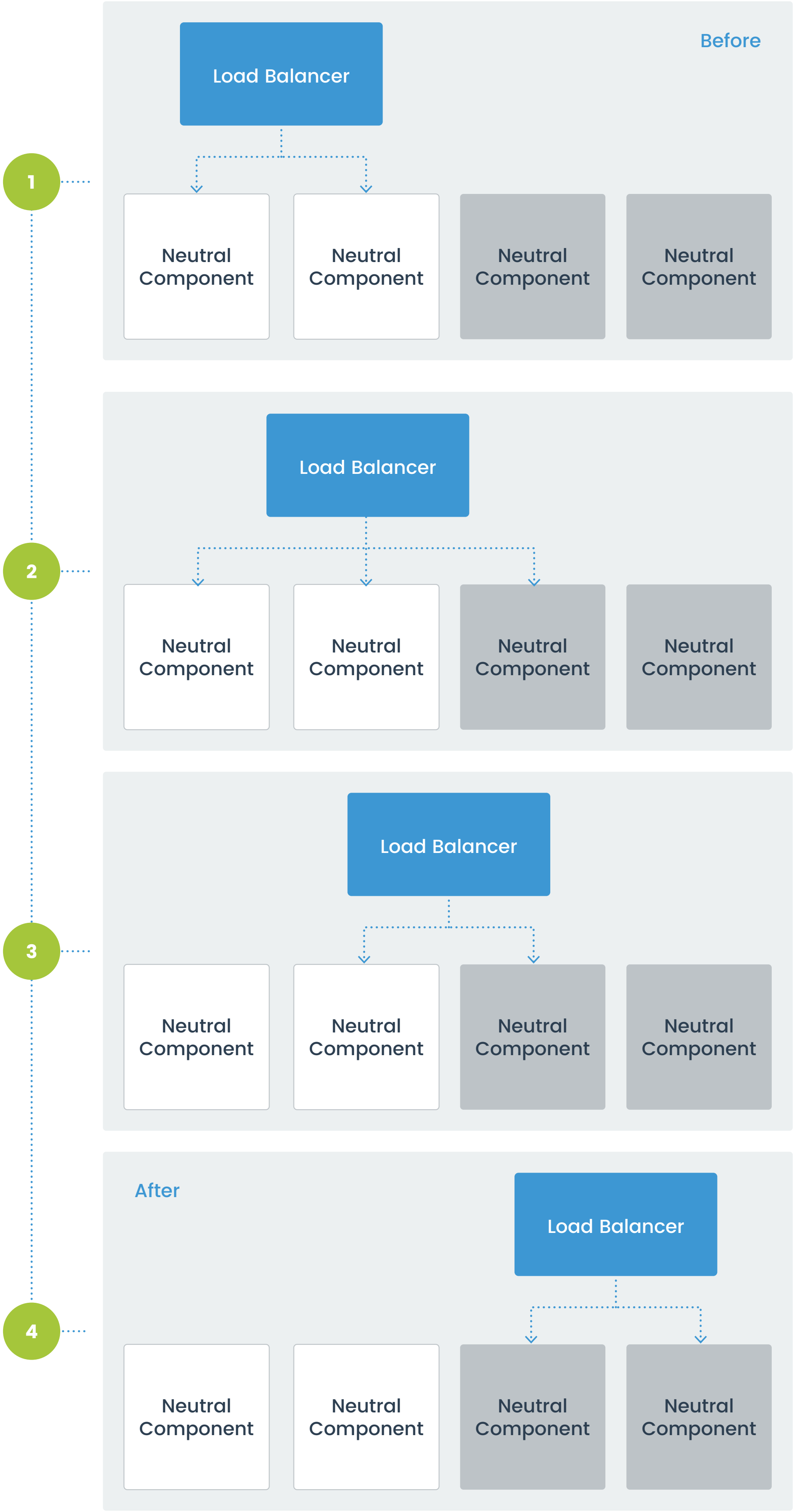
When a load balancer receives an HTTP request, the headers contain a wealth of extra information such as the browser, the device, the operating system, and more. Site maintainers can use this information to route a subset of the traffic to a different destination, perhaps to give an optimized experience to a particular class of mobile device, to test a new feature before rolling it out everywhere, or to see the effect of different changes to the content and determine which one has the more significant impact.

Load balancers also provide a way to roll out upgrades safely. Site administrators first deploy the new version of the website or application to a new set of backends and test it outside of the standard rotation. When ready, they incrementally add the new backends to the pool and rotate the old backends out. The load balancers keep existing traffic on the old backends and direct new traffic to the new backends. Over time the sessions connected to the old backends close, and only new sessions remain. The old backends are then terminated.

In the event of an unforeseen issue, the admins can quickly rotate the old backends into the pool and remove the new ones, returning the site to its previous, working state.






Before
Load Balancer
1
Neutral Component
Neutral Component
Neutral Component
Neutral Component
2
Load Balancer
Neutral Component
Neutral Component
Neutral Component
Neutral Component
3
Load Balancer
Neutral Component
Neutral Component
Neutral Component
Neutral Component
After
4
Load Balancer
Neutral Component
Neutral Component
Neutral Component
Neutral Component

LOAD BALANCING IN KUBERNETES

Kubernetes either can create internal load balancers using Kubernetes resources such as Services and Ingresses, or it can deploy and manage external load balancers such as those provided by AWS, GCP, F5, and others by deploying a service of type LoadBalancer.

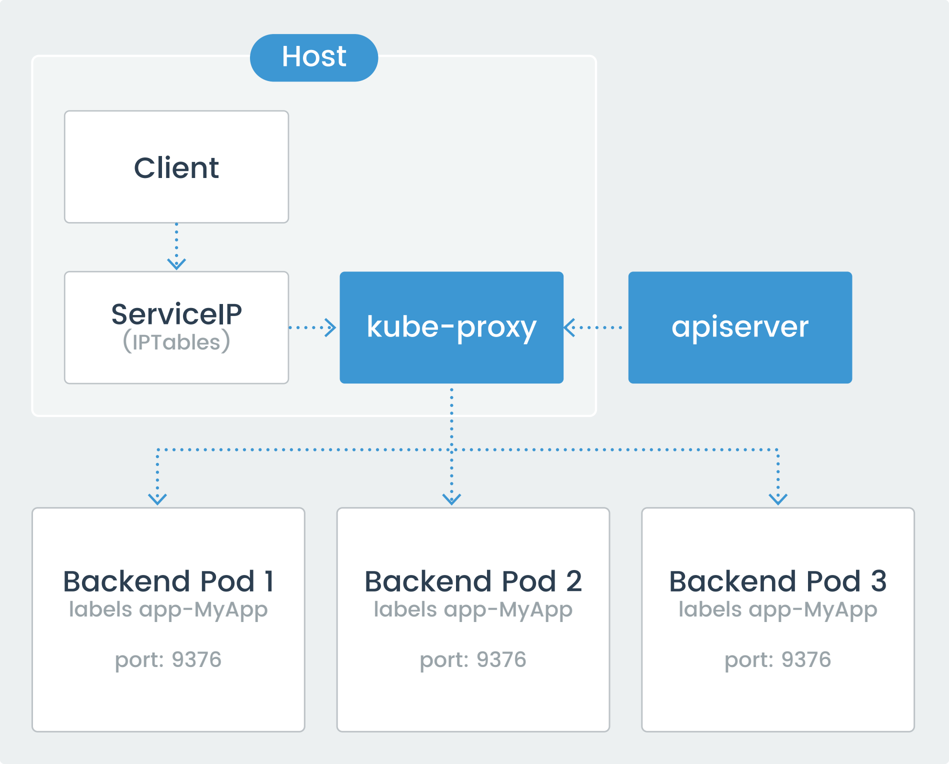
Internal Load Balancing

The easiest and simplest load balancer in Kubernetes is the Service. A Service routes traffic via round-robin to one or more replicas running within the cluster. The Service finds the replicas via a selector, which is a key/value pair that it looks for in the Pod labels. Any Pod that matches the selector is a candidate for traffic, and the Service sends each subsequent request to the next Pod in the list.

Services receive a stable IP address within the cluster, and if the cluster runs a DNS component like KubeDNS or CoreDNS, it also receives a DNS name in the format of {`name}.{namespace}.svc.cluster.local`. For example, applications within the cluster that want to communicate with a Service named my-service in the default namespace would send traffic to `my-service.default.svc.cluster.local`.

The following manifest creates a simple load balancer:

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
```



When traffic arrives at the Service, `kube-proxy` *forwards it to the appropriate backend.*

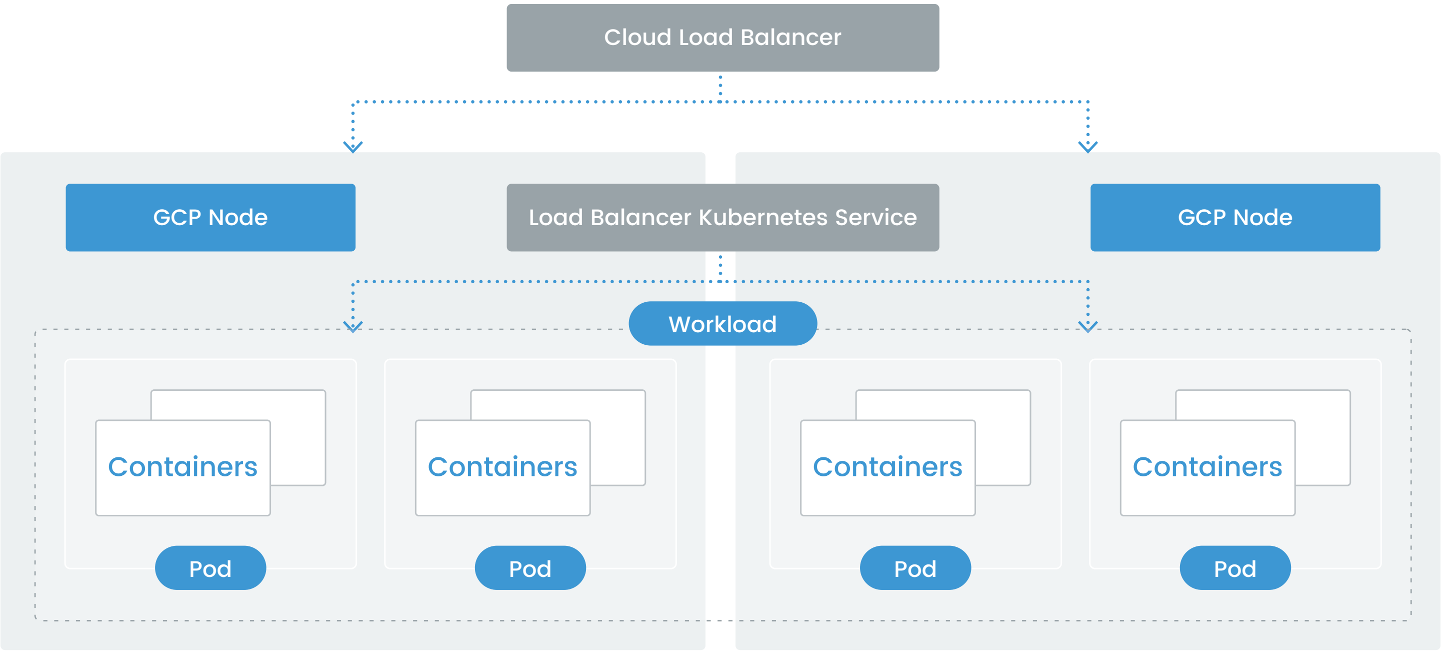
External Load Balancing

LAYER 4

A load balancer that works at Layer 4 only routes traffic based on the TCP or UDP port. It does not look inside the packets or the data stream to make any decisions.

A Kubernetes Service of the type `LoadBalancer` creates a Layer 4 load balancer outside of the cluster, but it only does this if the cluster knows how. External load balancers require that the cluster use a supported cloud provider in its configuration and that the configuration for the cloud provider includes the relevant access credentials when required.

Once created, the Status field of the service shows the address of the external load balancer.



The following manifest creates an external Layer 4 load balancer:

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
  clusterIP: 10.0.171.239
  loadBalancerIP: 78.11.24.19
  type: LoadBalancer
status:
  loadBalancer:
    ingress:
    - ip: 146.148.47.155
```

Because a Layer 4 load balancer does not look into the packet stream, it only has basic capabilities. If a site runs multiple applications, every one of them requires an external load balancer. Escalating costs make that scenario inefficient.

Furthermore, because the `LoadBalancer` Service type requires a supported external cloud provider, and because Kubernetes only supports a small number of providers, many sites instead choose to run a Layer 7 load balancer inside of the cluster.

LAYER 7

The Kubernetes resource that handles load balancing at Layer 7 is called an Ingress, and the component that creates Ingresses is known as an Ingress Controller.

The Ingress Resource

The Ingress resource defines the rules and routing for a particular application. Any number of Ingresses can exist within a cluster, each using a combination of host, path, or other rules to send traffic to a Service and then on to the Pods.

The following manifest defines an Ingress for the site `foo.bar.com`, sending `/foo` to the `s1` Service and `/bar` to the `s2` Service:

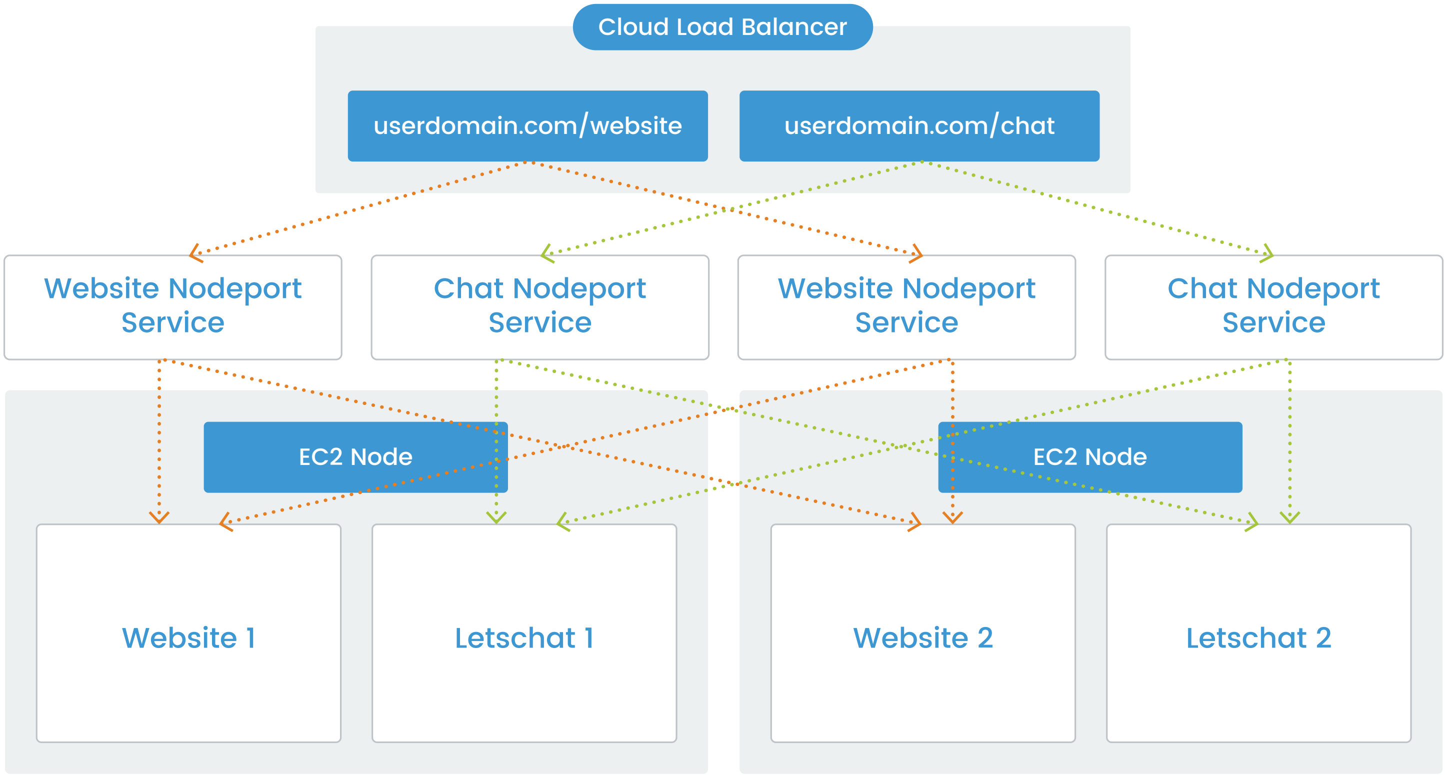
```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test
  annotations:
    nginx.ingress.Kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /foo
        backend:
          serviceName: s1
          servicePort: 80
      - path: /bar
        backend:
          serviceName: s2
          servicePort: 80
```

The Ingress Controller

An Ingress Controller is a daemon, deployed as a Kubernetes pod, that listens for requests to create or modify Ingresses within the cluster and converts the rules in the manifests into configuration directives for a load balancing component. That component is either a software load balancer such as Nginx, HAProxy, or Traefik, or it's an external load balancer such as an Amazon ALB or an F5 Big/IP.

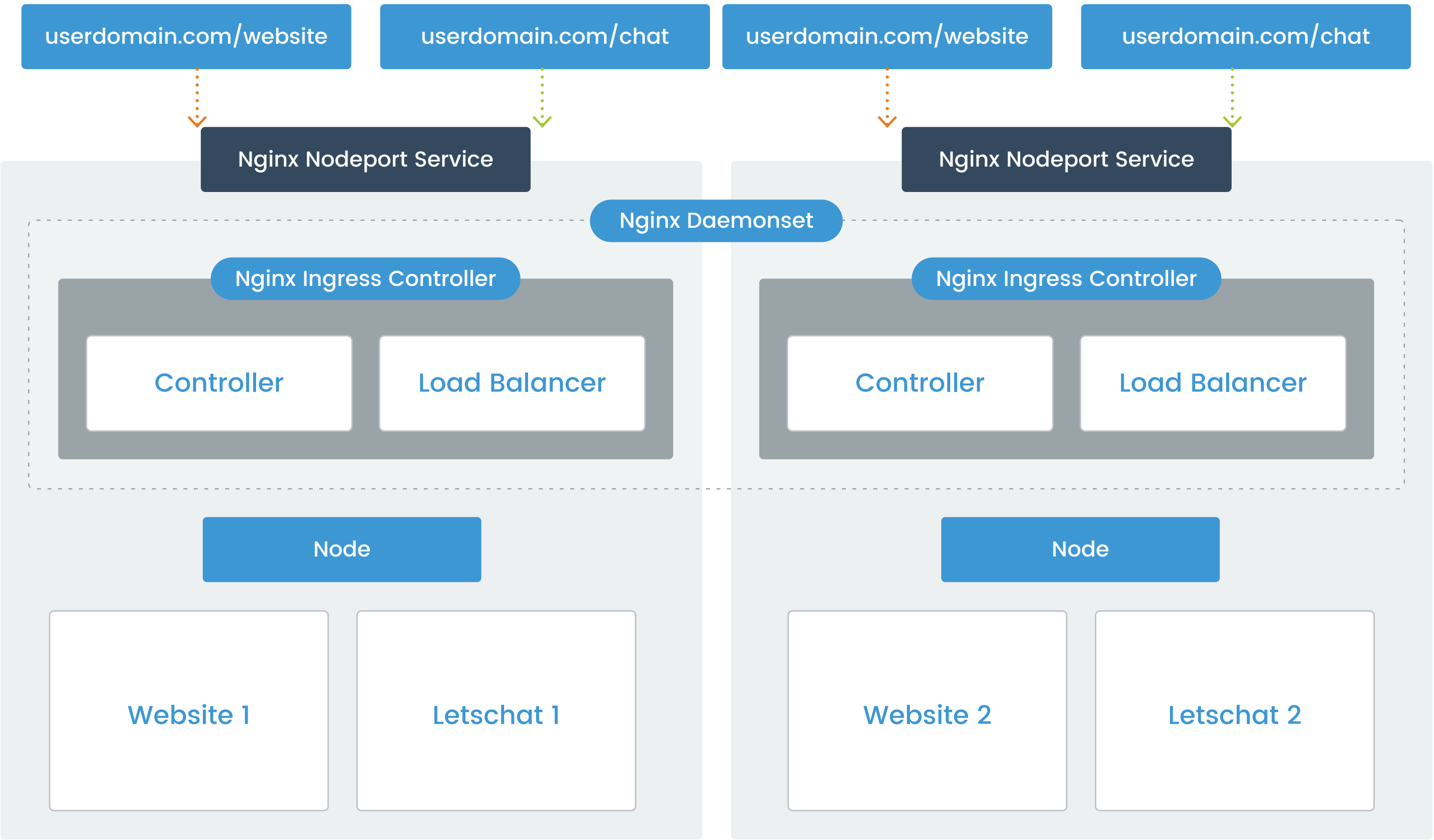
When working with an external load balancer the Ingress Controller is a lightweight component that translates the Ingress resource definitions from the cluster into API calls that configure the external piece.

The following diagram shows an Ingress Controller managing an Amazon ALB.



In the case of internal software load balancers, the Ingress Controller combines the management and load balancing components into one piece. It uses the instructions in the Ingress resource to reconfigure itself.

The following diagram shows a Nginx Ingress Controller working within a cluster.



Kubernetes uses annotations to control the behavior of the Ingress Controller. Each controller has a list of accepted annotations, and their use activates advanced features such as canary deployments, default backends, timeouts, redirects, CORS configuration, and more.

Conclusion

Kubernetes takes a simple container engine like Docker and elevates it to a level of usability appropriate for production environments. What starts as a series of Netfilter rules on a single host grows with Kubernetes to span multiple hosts or even multiple disparate networks separated by geographical boundaries. Kubernetes networking is powerful, and after reading this book, you're ready to make informed decisions about which provider to use, their capabilities, and how to leverage Kubernetes resources to connect the outside world to the applications running inside the cluster.