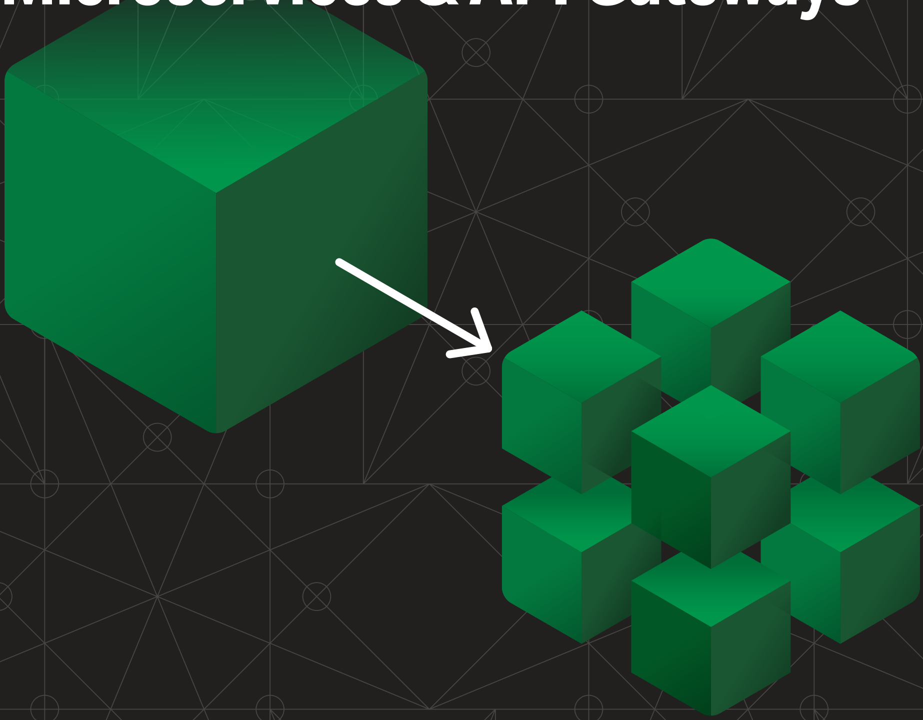
O'REILLY®

Compliments of NGINX

Container Networking

From Docker to Kubernetes

Michael Hausenblas



Container Networking

From Docker to Kubernetes

Michael Hausenblas

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Container Networking

by Michael Hausenblas

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Editor: Nikki McDonald
Production Editors: Melanie Yarbrough and Justin Billing
Copyeditor: Rachel Head
Proofreader: Charles Roumeliotis
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

May 2018: First Edition

Revision History for the First Edition

2018-04-17: First Release

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

This work is part of a collaboration between O'Reilly and NGINX. See our *statement of editorial independence*.

978-1-492-03681-4

[LSI]

Table of Contents

Preface

When you start building your first containerized application, you're excited about the capabilities and opportunities you encounter: it runs the same in dev and in prod, it's straightforward to put together a container image using Docker, and the distribution is taken care of by a container registry.

So, you're satisfied with how quickly you were able to containerize an existing, say, Python app, and now you want to connect it to another container that has a database, such as PostgreSQL. Also, you don't want to have to manually launch the containers and implement your own system that takes care of checking if the containers are still running and, if not, relaunching them.

At this juncture, you might realize there's a challenge you're running into: container networking. Unfortunately, there are still a lot of moving parts in this domain and there are currently few best practice resources available in a central place. Fortunately, there are tons of articles, repos, and recipes available on the wider internet and with this book you have a handy way to get access to many of them in a simple and comprehensive format.

Why I Wrote This Book

I thought to myself: what if someone wrote a book providing basic guidance for the container networking topic, pointing readers in the right direction for each of the involved technologies, such as overlay networks, the Container Network Interface (CNI), and load balancers?

That *someone* turned out to be me. With this book, I want to provide you with an overview of the challenges and available solutions for container networking, container orchestration, and (container) service discovery. I will try to drive home three points throughout this book:

- Without a proper understanding of the networking aspect of (Docker) containers and a sound strategy in place, you will have more than one bad day when adopting containers.
- Service discovery and container orchestration are two sides of the same coin.
- The space of container networking and service discovery is still relatively young: you will likely find yourself starting out with one set of technologies and then changing gears and trying something else. Don't worry, you're in good company.

Who Is This Book For?

My hope is that you'll find the book useful if one or more of the following applies to you:

- You are a software developer who drank the (Docker) container Kool-Aid.
- You work in network operations and want to brace yourself for the upcoming onslaught of your enthusiastic developer colleagues.
- You are an aspiring Site Reliability Engineer (SRE) who wants to get into the container business.
- You are an (enterprise) software architect who is in the process of migrating existing workloads to a containerized setup.

Last but not least, distributed application developers and backend engineers should also be able to extract some value out of it.

Note that this is not a hands-on book. Besides some single-host Docker networking stuff in Chapter 2 and some of the material about Kubernetes in Chapter 7, I don't show a lot of commands or source code; consider this book more like a guide, a heavily annotated bookmark collection. You will also want to use it to make informed decisions when planning and implementing containerized applications.

About Me

I work at Red Hat in the OpenShift team, where I help devops to get the most out of the software. I spend my time mainly upstream—that is, in the Kubernetes community, for example in the Autoscaling, Cluster Lifecycle, and Apps Special Interest Groups (SIGs).

Before joining Red Hat in the beginning of 2017 I spent some two years at Mesosphere, where I also did containers, in the context of (surprise!) Mesos. I also have a data engineering background, having worked as Chief Data Engineer at

MapR Inc. prior to Mesosphere, mainly on distributed query engines and datastores as well as building data pipelines.

Last but not least, I'm a pragmatist and tried my best throughout the book to make sure to be unbiased toward the technologies discussed here.

Acknowledgments

A big thank you to the O'Reilly team, especially Virginia Wilson. Thanks for your guidance and feedback on the first iteration of the book (back then called *Docker Networking and Service Discovery*), which came out in 2015, and for putting up with me again.

A big thank you to Nic (Sheriff) Jackson of HashiCorp for your time around Nomad. You rock, dude!

Thanks a million Bryan Boreham of Weaveworks! You provided super-valuable feedback and I appreciate your suggestions concerning the flow as well as your diligence, paying attention to details and calling me out when I drifted off and/or made mistakes. Bryan, who's a container networking expert and CNI 7th dan, is the main reason this book in its final version turned out to be a pretty good read (I think).

Last but certainly not least, my deepest gratitude to my awesome and supportive family: our two girls Saphira (aka The Real Unicorn—love you hun :) and Ranya (whose talents range from Scratch programming to Irish Rugby), our son Iannis (sigh, told you so, you ain't gonna win the rowing championship with a broken hand, but you're still dope), and my wicked smart and fun wife Anneliese (did I empty the washing machine? Not sure!).

CHAPTER 1
Motivation

In this chapter I'll introduce you to the pets versus cattle approach concerning compute infrastructure as well as what container networking entails. It sets the scene, and if you're familiar with the basics you may want to skip this chapter.

Introducing Pets Versus Cattle

In February 2012, Randy Bias gave an impactful talk on architectures for open and scalable clouds. In his presentation, he established the *pets versus cattle meme*:[1]

- With the *pets approach to infrastructure*, you treat the machines as individuals. You give each (virtual) machine a name, and applications are statically allocated to machines. For example, `db-prod-2` is one of the production servers for a database. The apps are manually deployed, and when a machine gets ill you nurse it back to health and manually redeploy the app it ran onto another machine. This approach is generally considered to be the dominant paradigm of a previous (non–cloud native) era.
- With the *cattle approach to infrastructure*, your machines are anonymous; they are all identical (modulo hardware upgrades), they have numbers rather than names, and apps are automatically deployed onto any and each of the machines. When one of the machines gets ill, you don't worry about it immediately; you replace it—or parts of it, such as a faulty hard disk drive—when you want and not when things break.

[1] In all fairness, Randy did attribute the origins to Bill Baker of Microsoft.

While the original meme was focused on virtual machines, we apply the cattle approach to infrastructure.

Go Cattle!

The beautiful thing about applying the cattle approach to infrastructure is that it allows you to scale out on commodity hardware.[2]

It gives you elasticity with the implication of hybrid cloud capabilities. This is a fancy way of saying that you can have parts of your deployments on premises and burst into the public cloud—using services provided by the likes of Amazon, Microsoft, and Google, or the infrastructure-as-a-service (IaaS) offerings of different provides like VMware—if and when you need to.

Most importantly, from an operator's point of view, the cattle approach allows you to get a decent night's sleep, as you're no longer paged at 3 a.m. just to replace a broken hard disk drive or to relaunch a hanging app on a different server, as you would have done with your pets.

However, the cattle approach poses some challenges that generally fall into one of the following two categories:

Social challenges

I dare say most of the challenges are of a social nature: How do I convince my manager? How do I get buy-in from my CTO? Will my colleagues oppose this new way of doing things? Does this mean we will need fewer people to manage our infrastructure?

I won't pretend to offer ready-made solutions for these issues; instead, go buy a copy of *The Phoenix Project* by Gene Kim, Kevin Behr, and George Spafford (O'Reilly), which should help you find answers.

Technical challenges

This category includes issues dealing with things like base provisioning of the machines—e.g., using Ansible to install Kubernetes components, how to set up the communication links between the containers and to the outside world, and most importantly, how to ensure the containers are automatically deployed and are discoverable.

Now that you know about pets versus cattle, you are ready to have a look at the overall container networking stack.

[2] Typically even heterogeneous hardware. For example, see slide 7 of Thorvald Natvig's talk "Challenging Fundamental Assumptions of Datacenters: Decoupling Infrastructure from Hardware" from Velocity 2015.

The Container Networking Stack

The overall stack we're dealing with here is comprised of the following:

The low-level networking layer
: This includes networking gear, `iptables`, routing, IPVLAN, and Linux namespaces. You usually don't need to know the details of this layer unless you're on the networking team, but you should at least be aware of it. Note that the technologies here have existed and been used for a decade or more.

The container networking layer
: This layer provides some abstractions, such as the single-host bridge networking mode and the multi-host, IP-per-container solution. I cover this layer in Chapters 2 and 3.

The container orchestration layer
: Here, we're marrying the container scheduler's decisions on where to place a container with the primitives provided by lower layers. In Chapter 4 we look at container orchestration systems in general, and in Chapter 5 we focus on the service discovery aspect, including load balancing. Chapter 6 deals with the container networking standard, CNI, and finally in Chapter 7 we look at Kubernetes networking.

Software-Defined Networking (SDN)

SDN is really an umbrella (marketing) term, providing essentially the same advantages to networks that virtual machines (VMs) introduced over bare-metal servers. With this approach, the network administration team becomes more agile and can react faster to changing business requirements. Another way to view it is this: SDN is the configuration of networks using software, whether that is via APIs, complementing network function virtualization, or the construction of networks from software.

Especially if you're a developer or an architect, I suggest taking a quick look at Cisco's nice overview on this topic as well as SDxCentral's article, "What Is Software-Defined Networking (SDN)?"

If you are on the network operations team, you're probably good to go for the next chapter. However, if you're an architect or developer and your networking knowledge might be a bit rusty, I suggest brushing up by studying the Linux Network Administrators Guide before advancing.

Do I Need to Go "All In"?

Oftentimes, when I'm at conferences or user groups, I meet people who are very excited about the opportunities in the container space. At the same time, folks rightfully worry about how deeply they need to commit to containers in order to benefit from them. The following table provides an informal overview of deployments I have seen in the wild, grouped by level of commitment expressed via stages:

Stage	Typical setup	Examples
Traditional	Bare metal or VM, no containers	Majority of today's prod deployments
Simple	Manually launched containers used for app-level dependency management	Development and test environments
Ad hoc	A custom, homegrown scheduler to launch and potentially restart containers	RelateIQ, Uber
Full-blown	An established scheduler from Chapter 4 to manage containers; fault tolerant, self-healing	Google, Zulily, Gutefrage.de

Note that the stage doesn't necessarily correspond with the size of the deployment. For example, Gutefrage.de only has six bare-metal servers under management but uses Apache Mesos to manage them, and you can run a Kubernetes cluster easily on a Raspberry Pi.

One last remark before we move on: by now, you might have realized that we are dealing with distributed systems in general here. Given that we will usually want to deploy containers into a network of computers, may I suggest reading up on the fallacies of distributed computing, in case you are not already familiar with this topic?

And now let's move on to the deep end of container networking.

CHAPTER 2

Introduction to Container Networking

This chapter focuses on networking topics for single-host container networking, with an emphasis on Docker. We'll also have a look at administrative challenges such as IP address management and security considerations. In Chapter 3, we will discuss multi-host scenarios.

Single-Host Container Networking 101

A container needs a host to run on. This can be a physical machine, such as a bare-metal server in your on-premises datacenter, or a virtual machine, either on premises or in the cloud.

In the case of a Docker container the host has a daemon and a client running, as depicted in Figure 2-1, enabling you to interact with a container registry. Further, you can pull/push container images and start, stop, pause, and inspect containers. Note that nowadays most (if not all) containers are compliant with the Open Container Initiative (OCI), and alongside Docker there are interesting alternatives, especially in the context of Kubernetes, available.

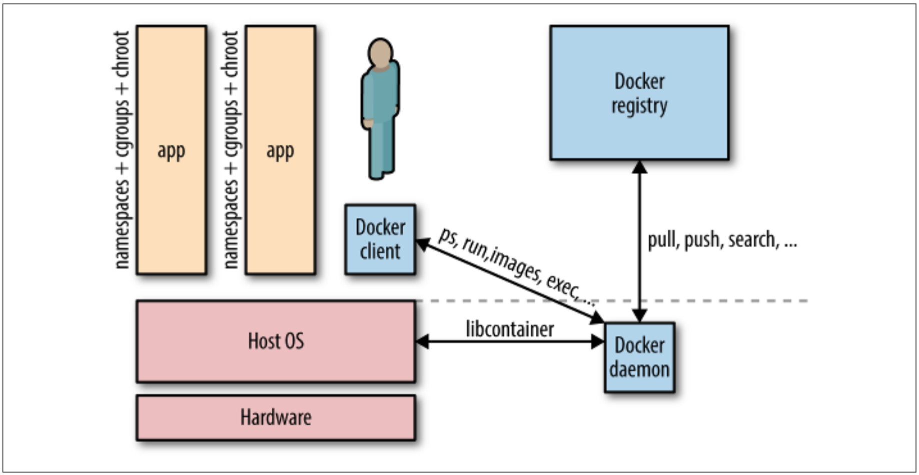


Figure 2-1. Simplified Docker architecture for a single host

The relationship between a host and containers is 1:*N*. This means that one host typically has several containers running on it. For example, Facebook reports that —depending on how beefy the machine is—it sees on average some 10 to 40 containers per host running.

No matter if you have a single-host deployment or use a cluster of machines, you will likely have to deal with networking:

- For *single-host deployments*, you almost always have the need to connect to other containers on the same host; for example, an application server like WildFly might need to connect to a database.
- In *multi-host deployments*, you need to consider two aspects: how containers are communicating within a host and how the communication paths look between different hosts. Both performance considerations and security aspects will likely influence your design decisions. Multi-host deployments usually become necessary either when the capacity of a single host is insufficient, for resilience reasons, or when one wants to employ distributed systems such as Apache Spark or Apache Kafka.

Distributed Systems and Data Locality

The basic idea behind using a distributed system (for computation or storage) is to benefit from parallel processing, usually together with data locality. By data locality I mean the principle of shipping the code to where the data is rather than the (traditional) other way around.

Think about the following for a moment: if your dataset size is in TB scale and your code size is in MB scale, it's more efficient to move the code across the cluster than it would be to transfer all the data to a central processing place. In addition to being able to process things in parallel, you usually gain fault tolerance with distributed systems, as parts of the system can continue to work more or less independently.

Simply put, Docker networking is the native container SDN solution you have at your disposal when working with Docker.

Modes for Docker Networking

In a nutshell, there are four single-host networking modes available for Docker:

Bridge mode
: Usually used for apps running in standalone containers; this is the default network driver. See "Bridge Mode Networking" on page 7 for details.

Host mode
: Also used for standalone containers; removes network isolation to the host. See "Host Mode Networking" on page 8 for details.

Container mode
: Lets you reuse the network namespace of another container. Used in Kubernetes. See "Container Mode Networking" on page 9 for details.

No networking
: Disables support for networking from the Docker side and allows you to, for example, set up custom networking. See "No Networking" on page 10 for details.

We'll take a closer look at each of these modes now, and end this chapter with some administrative considerations, including IP/port management and security.

Bridge Mode Networking

In this mode (see Figure 2-2), the Docker daemon creates `docker0`, a virtual Ethernet bridge that automatically forwards packets between any other network interfaces that are attached to it. By default, the daemon then connects all containers on a host to this internal network by creating a pair of peer interfaces, assigning one of the peers to become the container's `eth0` interface and placing the other peer in the namespace of the host, as well as assigning an IP address/subnet from the private IP range to the bridge. Here's an example of using bridge mode:

```
$ docker run -d -P --net=bridge nginx:1.9.1
$ docker ps
CONTAINER ID   IMAGE                  COMMAND    CREATED
STATUS         PORTS                  NAMES
17d447b7425d   nginx:1.9.1            nginx -g   19 seconds ago
Up 18 seconds  0.0.0.0:49153->443/tcp,
               0.0.0.0:49154->80/tcp  trusting_feynman
```

NOTE

Because bridge mode is the Docker default, you could have used `docker run -d -P nginx:1.9.1` in the previous command instead. If you do not use the `-P` argument, which publishes all exposed ports of the container, or `-p <host_port>:<container_port>`, which publishes a specific port, the IP packets will not be routable to the container outside of the host.

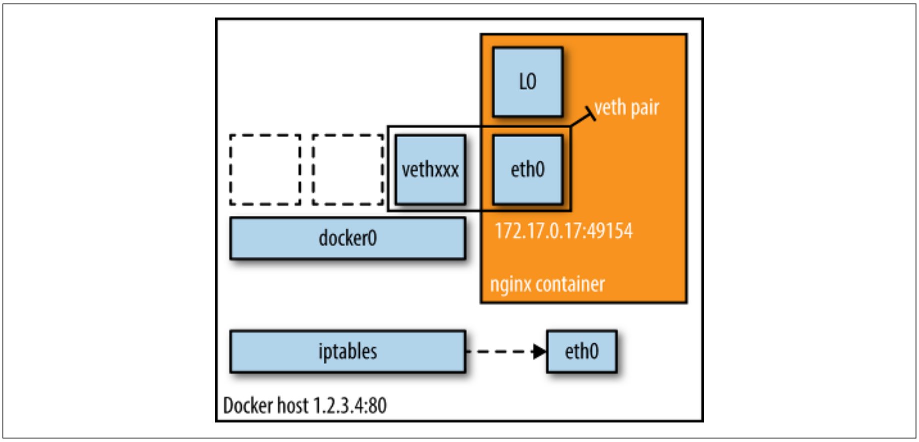


Figure 2-2. Bridge mode networking setup

Host Mode Networking

This mode effectively disables network isolation of a Docker container. Because the container shares the network namespace of the host, it may be directly exposed to the public network if the host network is not firewalled. As a consequence of the shared namespace, you need to manage port allocations somehow. Here's an example of host mode networking in action:

```
$ docker run -d --net=host ubuntu:14.04 tail -f /dev/null
$ ip addr | grep -A 2 eth0:
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc mq state UP group
default qlen 1000
    link/ether 06:58:2b:07:d5:f3 brd ff:ff:ff:ff:ff:ff
    inet **10.0.7.197**/22 brd 10.0.7.255 scope global dynamic eth0

$ docker ps
CONTAINER ID  IMAGE         COMMAND  CREATED
```

```
STATUS          PORTS           NAMES
b44d7d5d3903  ubuntu:14.04  tail -f  2 seconds ago
Up 2 seconds                jovial_blackwell
$ docker exec -it b44d7d5d3903 ip addr
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc mq state UP group
default qlen 1000
    link/ether 06:58:2b:07:d5:f3 brd ff:ff:ff:ff:ff:ff
    inet **10.0.7.197**/22 brd 10.0.7.255 scope global dynamic eth0
```

And there we have it: the container has the same IP address as the host, namely `10.0.7.197`.

In Figure 2-3 we see that when using host mode networking, the container effectively inherits the IP address from its host. This mode is faster than the bridge mode because there is no routing overhead, but it exposes the container directly to the public network, with all its security implications.

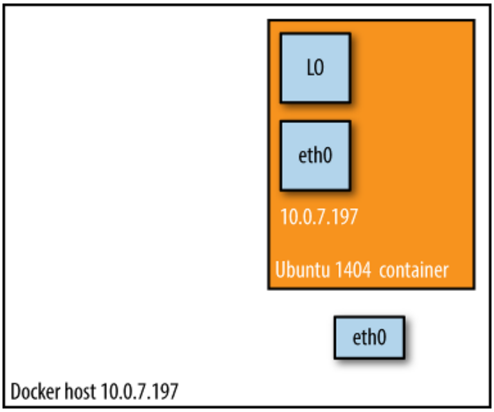


Figure 2-3. Docker host mode networking setup

Container Mode Networking

In this mode, you tell Docker to reuse the network namespace of another container. In general, this mode is useful if you want to have fine-grained control over the network stack and/or to control its lifecycle. In fact, Kubernetes networking uses this mode, and you can read more about it in Chapter 7. Here it is in action:

```
$ docker run -d -P --net=bridge nginx:1.9.1
$ docker ps
CONTAINER ID  IMAGE         COMMAND   CREATED         STATUS
PORTS                       NAMES
eb19088be8a0  nginx:1.9.1   nginx -g  3 minutes ago   Up 3 minutes
0.0.0.0:32769->80/tcp,
0.0.0.0:32768->443/tcp      admiring_engelbart
$ docker exec -it admiring_engelbart ip addr
8: eth0@if9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc noqueue state
```

```
UP group default
    link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff:ff
    inet **172.17.0.3**/16 scope global eth0

$ docker run -it --net=container:admiring_engelbart ubuntu:14.04 ip addr
...
8: eth0@if9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc noqueue state
UP group default
    link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff:ff
    inet **172.17.0.3**/16 scope global eth0
```

The result as shown in this example is what we would have expected: the second container, started with `--net=container`, has the same IP address as the first container (namely `172.17.0.3`), with the glorious autoassigned name `admiring_engelbart`.

No Networking

This mode puts the container inside its own network namespace but doesn't configure it. Effectively, this turns off networking and is useful for two cases: for containers that don't need a network, such as batch jobs writing to a disk volume, or if you want to set up your own custom networking (see Chapter 3 for a number of options that leverage this). Here's an example:

```
$ docker run -d -P --net=none nginx:1.9.1
$ docker ps
CONTAINER ID  IMAGE          COMMAND   CREATED
STATUS        PORTS          NAMES
d8c26d68037c  nginx:1.9.1    nginx -g  2 minutes ago
Up 2 minutes                 grave_perlman
$  docker inspect d8c26d68037c | grep IPAddress
    "IPAddress": "",
    "SecondaryIPAddresses": null,
```

As this example shows, there is no network configured, precisely as we would have expected.

You can read more about networking and learn about configuration options via the Docker docs.

Administrative Considerations

We will now briefly discuss other aspects you should be aware of from an administrative point of view. Most of these issues are equally relevant for multi-host deployments:

Allocating IP addresses

Manually allocating IP addresses when containers come and go frequently and in large numbers is not sustainable.[1] The bridge mode takes care of this issue to a certain extent. To prevent ARP collisions on a local network, the Docker daemon generates a MAC address from the allocated IP address.

Managing ports

There are two approaches to managing ports: fixed port allocation or dynamic allocation of ports. The allocation can be per service (or application) or it can be applied as a global strategy. For bridge mode, Docker can automatically assign (UDP or TCP) ports and consequently make them routable. Systems like Kubernetes that sport a flat, IP-per-container networking model don't suffer from this issue.

Network security

Out of the box, Docker has inter-container communication enabled (meaning the default is `--icc=true`). This means containers on a host can communicate with each other without any restrictions, which can potentially lead to denial-of-service attacks. Further, Docker controls the communication between containers and the wider world through the `--ip_forward` and `--iptables` flags. As a good practice, you should study the defaults of these flags and loop in your security team concerning company policies and how to reflect them in the Docker daemon setup.

Systems like CRI-O, the Container Runtime Interface (CRI) using OCI, offer alternative runtimes that don't have one big daemon like Docker has and potentially expose a smaller attack surface.

Another network security aspect is that of on-the-wire encryption, which usually means TLS/SSL as per RFC 5246.

Wrapping It Up

In this chapter, we had a look at the four basic single-host networking modes and related admin issues. Now that you have a basic understanding of the single-host case, let's have a look at a likely more interesting case: multi-host container networking.

[1] New Relic, for example, found the majority of the overall uptime of the containers in one particular setup in the low minutes; see also the update here.

CHAPTER 3

Multi-Host Networking

As long as you're using containers on a single host, the techniques introduced in Chapter 2 are sufficient. However, if the capacity of a single host is not enough to handle your workload or you want more resilience, you'll want to scale out horizontally.

Multi-Host Container Networking 101

When scaling out horizontally you end up with a network of machines, also known as a cluster of machines, or cluster for short. Now, a number of questions arise: How do containers talk to each other on different hosts? How do you control communication between containers, and with the outside world? How do you keep state, such as IP address assignments, consistent in a cluster? What are the integration points with the existing networking infrastructure? What about security policies?

In order to address these questions, we'll review technologies for multi-host container networking in the remainder of this chapter. Since different use cases and environments have different requirements, I will abstain from providing a recommendation for a particular project or product. You should be aware of the trade-offs and make an informed decision.

Options for Multi-Host Container Networking

In a nutshell, Docker itself offers support for overlay networks (creating a distributed network across hosts on top of the host-specific network) as well as network plug-ins for third-party providers.

There are a number of multi-host container networking options that are often used in practice, especially in the context of Kubernetes. These include:

- Flannel by CoreOS (see "flannel" on page 14)
- Weave Net by Weaveworks (see "Weave Net" on page 14)
- Metaswitch's Project Calico (see "Project Calico" on page 14)
- Open vSwitch from the OpenStack project (see "Open vSwitch" on page 15)
- OpenVPN (see "OpenVPN" on page 15)

In addition, Docker offers multi-host networking natively; see "Docker Networking" on page 15 for details.

flannel

CoreOS's flannel is a virtual network that assigns a subnet to each host for use with container runtimes. Each container—or pod, in the case of Kubernetes—has a unique, routable IP inside the cluster. flannel supports a range of backends, such as VXLAN, AWS VPC, and the default layer 2 UDP network. The advantage of flannel is that it reduces the complexity of doing port mapping. For example, Red Hat's Project Atomic uses flannel.

Weave Net

Weaveworks's WeaveNet creates a virtual network that connects Docker containers deployed across multiple hosts. Applications use the network just as if the containers were all plugged into the same network switch, with no need to configure port mappings and links. Services provided by application containers on the Weave network can be made accessible to the outside world, regardless of where those containers are running.

Similarly, existing internal systems can be exposed to application containers irrespective of their location. Weave can traverse firewalls and operate in partially connected networks. Traffic can be encrypted, allowing hosts to be connected across an untrusted network. You can learn more about Weave's discovery features in the blog post "Automating Weave Deployment on Docker Hosts with Weave Discovery" by Alvaro Saurin.

If you want to give Weave a try, check out its Katacoda scenarios.

Project Calico

Metaswitch's Project Calico uses standard IP routing—to be precise, the venerable Border Gateway Protocol (BGP), as defined in RFC 1105—and networking tools to provide a layer 3 solution. In contrast, most other networking solutions build an overlay network by encapsulating layer 2 traffic into a higher layer.

The primary operating mode requires no encapsulation and is designed for datacenters where the organization has control over the physical network fabric.

See also Canal, which combines Calico's network policy enforcement with the rich superset of Calico and flannel overlay and nonoverlay network connectivities.

Open vSwitch

Open vSwitch is a multilayer virtual switch designed to enable network automation through programmatic extension while supporting standard management interfaces and protocols, such as NetFlow, IPFIX, LACP, and 802.1ag. In addition, it is designed to support distribution across multiple physical servers and is used in Red Hat's Kubernetes distro OpenShift, the default switch in Xen, KVM, Proxmox VE, and VirtualBox. It has also been integrated into many private cloud systems, such as OpenStack and oVirt.

OpenVPN

OpenVPN, another OSS project that has a commercial offering, allows you to create virtual private networks (VPNs) using TLS. These VPNs can also be used to securely connect containers to each other over the public internet. If you want to try out a Docker-based setup, I suggest taking a look at DigitalOcean's "How to Run OpenVPN in a Docker Container on Ubuntu 14.04" walk-through tutorial.

Docker Networking

Docker 1.9 introduced a new `docker network` command. With this, containers can dynamically connect to other networks, with each network potentially backed by a different network driver.

In March 2015, Docker Inc. acquired the SDN startup SocketPlane and rebranded its product as the Overlay Driver. Since Docker 1.9, this is the default for multi-host networking. The Overlay Driver extends the normal bridge mode with peer-to-peer communication and uses a pluggable key-value store backend to distribute cluster state, supporting Consul, etcd, and ZooKeeper.

To learn more, I suggest checking out the following blog posts:

- Aleksandr Tarasov's "Splendors and Miseries of Docker Network"
- Project Calico's "Docker 1.9 Includes Network Plugin Support and Calico Is Ready!"
- Weaveworks's "Life and Docker Networking – One Year On"

Administrative Considerations

In the last section of this chapter we will discuss some administrative aspects you should be aware of:

IPVLAN

Linux kernel version 3.19 introduced an *IP-per-container feature*. This assigns each container on a host a unique and routable IP address. Effectively, IPVLAN takes a single network interface and creates multiple virtual network interfaces with different MAC addresses assigned to them.

This feature, which was contributed by Mahesh Bandewar of Google, is conceptually similar to the macvlan driver but is more flexible because it's operating both on L2 and L3. If your Linux distro already has a kernel > 3.19, you're in luck. Otherwise, you cannot yet benefit from this feature.

IP address management (IPAM)

One of the key challenges of multi-host networking is the allocation of IP addresses to containers in a cluster. There are two strategies one can pursue: either find a way to realize it in your existing (corporate) network or spawn an orthogonal, practically hidden networking layer (that is, an overlay network). Note that with IPv6 this situation is relaxed, since it should be a lot easier to find a free address space.

Orchestration tool compatibility

Many of the multi-host networking solutions discussed in this chapter are effectively coprocesses wrapping the Docker API and configuring the network for you. This means that before you select one, you should make sure to check for any compatibility issues with the container orchestration tool you're using. You'll find more on this topic in Chapter 4.

IPv4 versus IPv6

To date, most Docker deployments use the standard IPv4, but IPv6 is witnessing some uptake. Docker has supported IPv6 since v1.5, released in February 2015; however, the IPv6 support in Kubernetes is not yet complete. The ever-growing address shortage in IPv4-land might encourage more IPv6 deployments down the line, also getting rid of network address translation (NAT), but it is unclear when exactly the tipping point will be reached.

Wrapping It Up

In this chapter, we reviewed multi-host networking options and touched on admin issues such as IPAM and orchestration. At this juncture you should have a good understanding of the low-level single-host and multi-host networking options and their challenges. Let's now move on to container orchestration, looking at how it depends on networking and how it interacts with it.

CHAPTER 4
Orchestration

With the cattle approach to managing infrastructure, you don't manually allocate certain machines for running an application. Instead, you leave it up to an orchestrator to manage the life cycle of your containers. In Figure 4-1, you can see that container orchestration includes a range of functions, including but not limited to:

- Organizational primitives, such as labels in Kubernetes, to query and group containers
- Scheduling of containers to run on a host
- Automated health checks to determine if a container is alive and ready to serve traffic and to relaunch it if necessary
- Autoscaling (that is, increasing or decreasing the number of containers based on utilization or higher-level metrics)
- Upgrade strategies, from rolling updates to more sophisticated techniques such as A/B and canary deployments
- Service discovery to determine which host a scheduled container ended upon, usually including DNS support

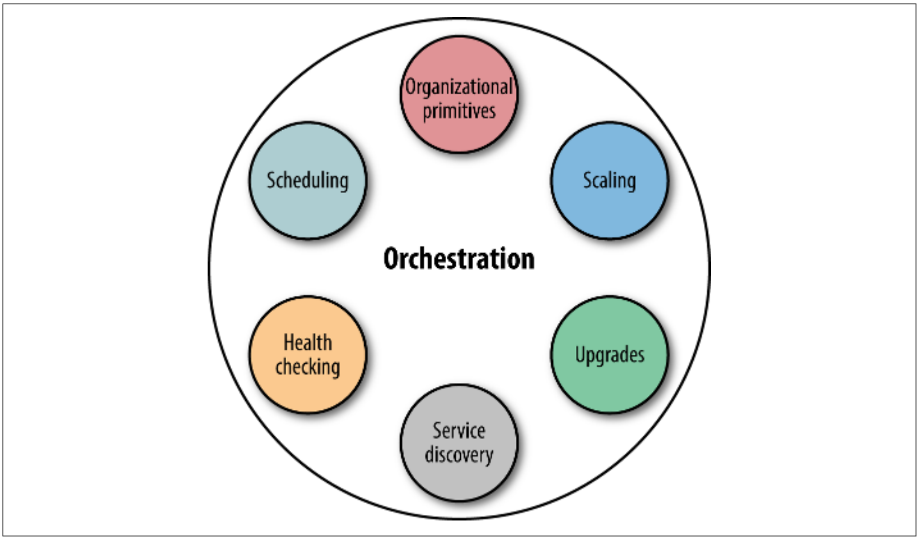


Figure 4-1. Orchestration and its constituents

Sometimes considered part of orchestration but outside the scope of this book is the topic of *base provisioning*—that is, installing or upgrading the local operating system on a node or setting up the container runtime there.

Service discovery (covered in greater detail in Chapter 5) and scheduling are really two sides of the same coin. The scheduler decides where in a cluster a container is placed and supplies other parts with an up-to-date mapping in the form `containers -> locations`. This mapping can then be represented in various ways, be it in a distributed key-value store such as etcd, via DNS, or through environment variables.

In this chapter we will discuss networking and service discovery from the point of view of the following container orchestration solutions: Docker Swarm and swarm mode, Apache Mesos, and HashiCorp Nomad. These three are (along with Kubernetes, which we will cover in detail in Chapter 7) alternatives your organization may already be using, and hence, for the sake of completeness, it's worth exploring them here. To make it clear, though, as of early 2018 the industry has standardized on Kubernetes as the portable way of doing container orchestration.

NOTE

In addition to the three orchestrators discussed in this chapter, there are other (closed source) solutions out there you could have a look at, including Facebook's Bistro or hosted solutions such as Amazon ECS.

Should you want to more fully explore the topic of distributed system scheduling, I suggest reading Google's research papers on Borg and Omega.

Before we dive into container orchestration systems, though, let's step back and review what the scheduler—which is the core component of orchestration—actually does in the context of containerized workloads.

What Does a Scheduler Actually Do?

A scheduler for a distributed system takes an application—binary or container image—as requested by a user and places it on one or more of the available hosts. For example, a user might request to have 100 instances of the app running, so the scheduler needs to find space (CPU, RAM) to run these 100 instances on the available hosts.

In the case of a containerized setup, this means that the respective container image must exist on a host (if not, it must be pulled from a container registry first), and the scheduler must instruct the container runtime on that host to launch a container based on the image.

Let's look at a concrete example. In Figure 4-2, you can see that the user requested three instances of the app running in the cluster. The scheduler decides the placement based on its knowledge of the state of the cluster. The cluster state may include the utilization of the machines, the resources necessary to successfully launch the app, and constraints such as *launch this app only on a machine that is SSD-backed*.

Further, quality of service might be taken into account for the placement decision; see Michael Gasch's great article "QoS, *Node allocatable* and the Kubernetes Scheduler" for more details.

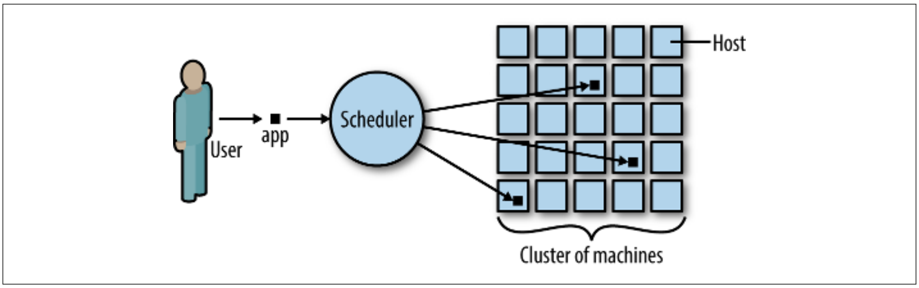


Figure 4-2. Distributed system scheduler in action

If you want to learn more about scheduling in distributed systems I suggest you check out the excellent resource "Cluster Management at Google" by John Wilkes.

Beware of the semantics of constraints that you can place on scheduling containers. For example, I once gave a demo using Marathon that wouldn't work as planned because I screwed up the placement constraints: I used a combination of unique hostname and a certain role, and it wouldn't scale because there was only one node with the specified role in the cluster. The same thing can happen with Kubernetes labels.

Docker

Docker at the time of writing uses the so-called *swarm mode* in a distributed setting, whereas previous to Docker 1.12 the standalone Docker Swarm model was used. We will discuss both here.

Swarm Mode

Since Docker 1.12, swarm mode has been integrated with Docker Engine. The orchestration features embedded in Docker Engine are built using SwarmKit.

A swarm in Docker consists of multiple hosts running in swarm mode and acting as managers and workers—hosts can be managers, workers, or perform both roles at once. A task is a running container that is part of a swarm service and managed by a swarm manager, as opposed to a standalone container. A service in the context of Docker swarm mode is a definition of the tasks to execute on the manager or worker nodes. Docker works to maintain that desired state; for example, if a worker node becomes unavailable, Docker schedules the tasks onto another host.

Docker running in swarm mode doesn't prevent you from running standalone containers on any of the hosts participating in the swarm. The essential difference between standalone containers and swarm services is that only swarm managers can manage a swarm, while standalone containers can be started on any host.

To learn more about Docker's swarm mode, check out the official "Getting Started with Swarm Mode" tutorial or check out the Katacoda "Docker Orchestration – Getting Started with Swarm Mode" scenario.

Docker Swarm

Docker historically had a native clustering tool called Docker Swarm. Docker Swarm builds upon the Docker API[1] and works as follows: there's one Swarm

[1] Essentially, this means that you can simply keep using `docker run` commands and the deployment of your containers in a cluster happens automagically.

manager that's responsible for scheduling, and on each host an agent runs that takes care of the local resource management (Figure 4-3).

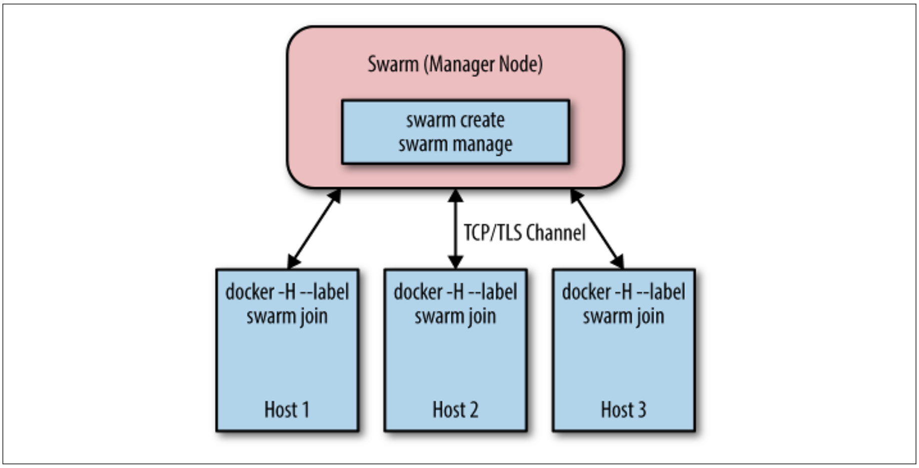


Figure 4-3. Docker Swarm architecture, based on the T-Labs presentation "Swarm – A Docker Clustering System"

Docker Swarm supports different backends: etcd, Consul, and ZooKeeper. You can also use a static file to capture your cluster state with Swarm, and recently a DNS-based service discovery tool for Swarm called wagl has been introduced.

NOTE

Out of the box, Docker provides a basic service discovery mechanism for single-node deployments called Docker links. Linking allows a user to let any container discover both the IP address and exposed ports of other Docker containers on the same host. In order to accomplish this, Docker provides the `--link` flag. But hard-wiring of links between containers is neither fun nor scalable. In fact, it's so bad that this feature has been deprecated.

Apache Mesos

Apache Mesos (Figure 4-4) is a general-purpose cluster resource manager that abstracts the resources of a cluster (CPU, RAM, etc.) in such a way that the cluster appears like one giant computer to the developer. In a sense, Mesos acts like the kernel of a distributed operating system. It is hence never used on its own, but always together with so-called *frameworks* such as Marathon (for long-running stuff like a web server) or Chronos (for batch jobs), or big data and fast data frameworks like Apache Spark or Apache Cassandra.

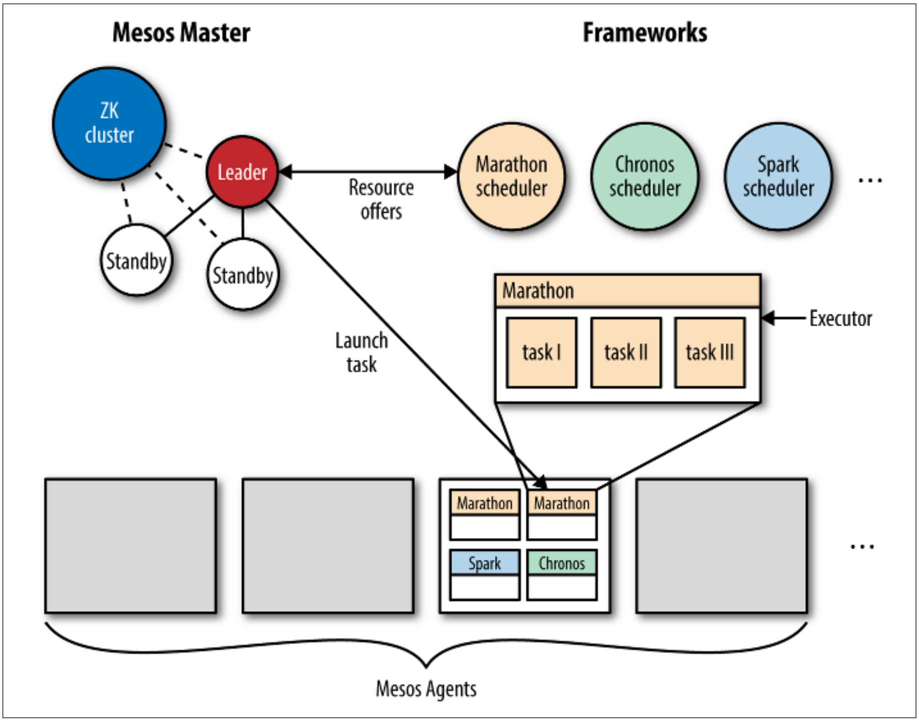


Figure 4-4. Apache Mesos architecture at a glance

Mesos supports both containerized workloads (that is, running Docker containers) and plain executables (for example, bash scripts or Linux ELF format binaries for both stateless and stateful services).

In the following discussion, I'm assuming you're familiar with Mesos and its terminology. If you're new to Mesos, I suggest checking out David Greenberg's wonderful book *Building Applications on Mesos* (O'Reilly), a gentle introduction to this topic that's particularly useful for distributed application developers.

The networking characteristics and capabilities mainly depend on the Mesos containerizer used:

- For the *Mesos containerizer* there are a few prerequisites, such as having a Linux Kernel version > 3.16 and `libnl` installed. You can then build a Mesos agent with network isolator support enabled. At launch, you would use something like the following:

  ```
  $mesos-slave --containerizer=mesos
   --isolation=network/port_mapping
   --resources=ports:[31000-32000];ephemeral_ports:[33000-35000]
  ```

  This would configure the Mesos agent to use nonephemeral ports in the range from 31,000 to 32,000 and ephemeral ports in the range from 33,000 to

35,000. All containers share the host's IP, and the port ranges are spread over the containers (with a 1:1 mapping between destination port and container ID). With the network isolator, you also can define performance limitations such as bandwidth, and it enables you to perform per-container monitoring of the network traffic. See Jie Yu's MesosCon 2015 talk "Per Container Network Monitoring and Isolation in Mesos" for more details on this topic.

- For the *Docker containerizer*, see Chapter 2.

Note that Mesos supports IP-per-container since version 0.23. If you want to learn more about Mesos networking check out Christos Kozyrakis and Spike Curtis's "Mesos Networking" talk from MesosCon 2015.

While Mesos is not opinionated about service discovery, there is a Mesos-specific solution that is often used in practice: Mesos-DNS (see "Pure-Play DNS-Based Solutions" on page 31). There are also a multitude of emerging solutions, such as traefik (see "Wrapping It Up" on page 34) that are integrated with Mesos and gaining traction.

> **NOTE**
>
> Because Mesos-DNS is the recommended default service discovery mechanism with Mesos, it's important to pay attention to how Mesos-DNS represents tasks. For example, a running task might have the (logical) service name `webserver.marathon.mesos`, and you can find out the port allocations via DNS SRV records.

If you want to try out Mesos online for free you can use the Katacoda "Deploying Containers to DC/OS" scenario.

Hashicorp Nomad

Nomad is a cluster scheduler by HashiCorp, the makers of Vagrant. It was introduced in September 2015 and primarily aims at simplicity. The main idea is that Nomad is easy to install and use. Its scheduler design is reportedly inspired by Google's Omega, borrowing concepts such as having a global state of the cluster as well as employing an optimistic, concurrent scheduler.

Nomad has an agent-based architecture with a single binary that can take on different roles, supporting rolling upgrades as well as draining nodes for rebalancing. Nomad makes use of both a consensus protocol (strongly consistent) for all state replication and scheduling and a gossip protocol used to manage the addresses of servers for automatic clustering and multiregion federation. In Figure 4-5, you can see Nomad's architecture:

- Servers are responsible for accepting jobs from users, managing clients, and computing task placements.

- Clients (one per VM instance) are responsible for interacting with the tasks or applications contained within a job. They work in a pull-based manner; that is, they register with the server and then they poll it periodically to watch for pending work.

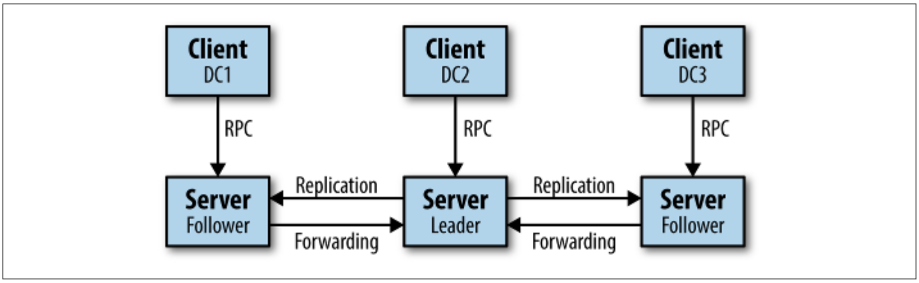


Figure 4-5. Nomad architecture

Jobs in Nomad are defined in a HashiCorp-proprietary format called HCL or in JSON, and Nomad offers a command-line interface as well as an HTTP API to interact with the server process. Nomad models infrastructure as regions and datacenters. Regions may contain multiple datacenters, depending on what scale you are operating at. You can think of a datacenter like a zone in AWS, Azure, or Google Cloud (say, `us-central1-b`), and a region might be something like Iowa (`us-central1`).

I'm assuming you're familiar with Nomad and its terminology. If not, I suggest you watch "Nomad: A Distributed, Optimistically Concurrent Schedule: Armon Dadgar, HashiCorp", a nice introduction to Nomad, and also read the well-done docs.

To try out Nomad, use the UI Demo HashiCorp provides or try it out online for free using the Katacoda "Introduction to Nomad" scenario.

Nomad comes with a couple of so-called *task drivers*, from general-purpose `exec` to `java` to `qemu` and `docker`. For the `docker` driver Nomad requires, at the time of this writing, Docker version 1.10 or greater and uses port binding to expose services running in containers using the port space on the host's interface. It provides automatic and manual mapping schemes for Docker, binding both TCP and UDP protocols to ports used for Docker containers.

For more details on networking options, such as mapping ports and using labels, see the documentation.

With v0.2, Nomad introduced a Consul-based (see "Consul" on page 30) service discovery mechanism. It includes health checks and assumes that tasks running inside Nomad also need to be able to connect to the Consul agent, which can, in the context of containers using bridge mode networking, pose a challenge.

Community Matters

An important aspect you'll want to consider when selecting an orchestration system is the community behind and around it.[2] Here are a few indicators and metrics you can use:

- Is the governance backed by a formal entity or process, such as the Apache Software Foundation (ASF) or the Linux Foundation (LF)?
- How active are the mailing list, the IRC channel, the bug/issue tracker, the Git repo (number of patches or pull requests), and other community initiatives?

  Take a holistic view, but make sure that you actually pay attention to the activity there. Healthy and hopefully growing communities will tend to have high participation in at least one of these areas.
- Is the orchestration tool (implicitly) controlled by a single entity? For example, in the case of Nomad HashiCorp is in control, for Apache Mesos it's mainly Mesosphere (and to some extent Twitter), etc.
- Are multiple independent providers and support channels available? For example, you can run Kubernetes in many different environments and get help from many (commercial) organizations as well as individuals on Slack, mailing lists, or forums.

Wrapping It Up

As of early 2018, Kubernetes (discussed in Chapter 7) can be considered the de facto container orchestration standard. All major providers, including Docker and DC/OS (Mesos), support Kubernetes.

Next, we'll move on to service discovery, a vital part of container orchestration.

[2] Now, you might argue that this is not specific to the container orchestration domain but a general OSS issue, and you'd be right. Still, I believe it is important enough to mention it, as many people are new to this area and can benefit from these insights.

CHAPTER 5

Service Discovery

One challenge arising from adopting the cattle approach to managing infrastructure is service discovery. If you subscribe to the cattle approach, you treat all of your machines equally and you do not manually allocate certain machines for certain applications; instead, you leave it up to a piece of software (the scheduler) to manage the life cycle of the containers.

The question then is, how do you determine which host your container ended up being scheduled on so that you can connect to it? This is called service discovery, and we touched on it already in Chapter 4.

The Challenge

Service discovery has been around for a while—essentially, as long as distributed systems and services have existed. In the context of containers, the challenge boils down to reliably maintaining a mapping between a running container and its location. By location, I mean its IP address and the port on which it is reachable. This mapping has to be done in a timely manner and accurately across relaunches of the container throughout the cluster. Two distinct operations must be supported by a container service discovery solution:

Registration
: Establishes the `container -> location` mapping. Because only the container scheduler knows where containers "live," we can consider it to be the absolute source of truth concerning a container's location.

Lookup
: Enables other services or applications to look up the mapping we stored during registration. Interesting properties include the freshness of the information and the latency of a query (average, p50, p90, etc.).

Let's examine a few slightly orthogonal considerations for the selection process:

- Rather than simply sending a requestor in a certain direction, how about excluding unhealthy hosts or hanging containers from the lookup path? You've guessed it, this is the strongly related topic of load balancing, and because it is of such importance we'll discuss options in the last section of this chapter.
- Some argue it's an implementation detail, others say the position in the CAP triangle matters: the choice of strong consistency versus high availability in the context of the service discovery tool might influence your decision. Be at least aware of it.
- Your choice might also be impacted by scalability considerations. Sure, if you only have a handful of nodes under management then all of the solutions discussed here will be a fit. If your cluster, however, has several hundred or even thousands of nodes, then you will want to make sure you do some proper load testing before you commit to one particular technology.

In this chapter you'll learn about service discovery options and how and where to use them.

If you want to learn more about the requirements and fundamental challenges in this space, read Jeff Lindsay's "Understanding Modern Service Discovery with Docker" and check out what Simon Eskildsen of Shopify shared on this topic at a recent DockerCon.

Technologies

This section briefly introduces a variety of service discovery technologies, listing pros and cons and pointing to further discussions on the web. For a more in-depth treatment, check out Adrian Mouat's excellent book *Using Docker* (O'Reilly).

ZooKeeper

Apache ZooKeeper is an ASF top-level project and a JVM-based, centralized tool for configuration management,[1] providing comparable functionality to what Google's Chubby brings to the table. ZooKeeper (ZK) organizes its payload data somewhat like a filesystem, in a hierarchy of so-called *znodes*. In a cluster, a leader is elected and clients can connect to any of the servers to retrieve data. You want $2n+1$ nodes in a ZK cluster. The most often found configurations in the

[1] ZooKeeper was originally developed at Yahoo! in order to get its ever-growing zoo of software tools, including Hadoop, under control.

wild are three, five, or seven nodes. Beyond that, you'll experience diminishing returns concerning the fault tolerance/throughput trade-off.

ZooKeeper is a battle-proven, mature, and scalable solution, but it has some operational downsides. Some people consider the installation and management of a ZK cluster a not-so-enjoyable experience. Most ZK issues I've seen come from the fact that certain services (Apache Storm comes to mind) misuse it. They either put too much data into the *znodes* or, worse, have an unhealthy read/write ratio, essentially writing too fast. If you plan to use ZK, at least consider using higher-level interfaces such as Apache Curator, which is a wrapper library around ZK implementing a number of recipes, and Netflix's Exhibitor for managing and monitoring ZK clusters.

Looking at Figure 5-1, you see two components: *R/W* (which stands for *registration watcher*, a piece of software you need to provide yourself), and HAProxy, which is controlled by the R/W. Whenever a container is scheduled on a node it registers with ZK, using a znode with a path like */$nodeID/$containerID* and the IP address as its payload. The R/W watches changes on those znodes and configures HAProxy accordingly.

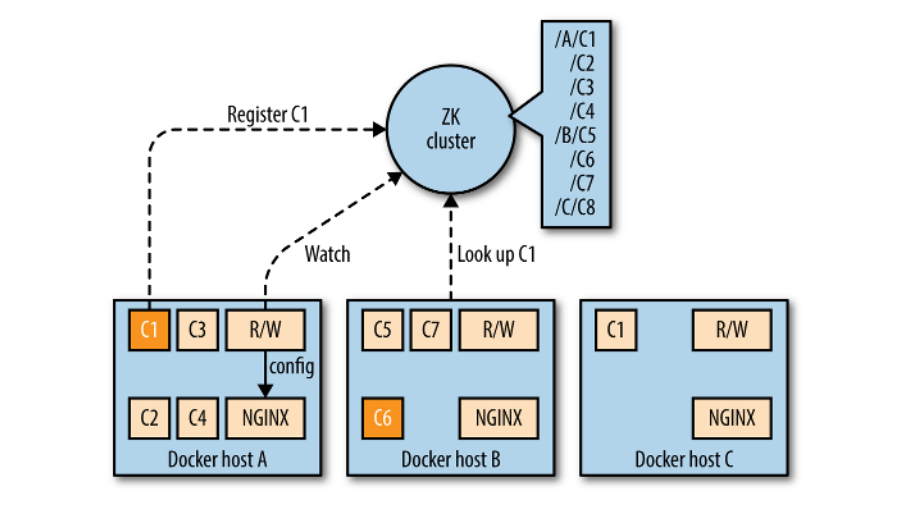


Figure 5-1. Example service discovery setup with ZooKeeper

etcd

Written in the Go language, etcd is a product of the CoreOS team.[2] It is a lightweight, distributed key-value store that uses the Raft algorithm for consensus (a leader–follower model, with leader election) and employs a replicated log across

[2] Did you know that etcd comes from */etc distributed*? What a name!

the cluster to distribute the writes a leader receives to its followers. In a sense, etcd is conceptually quite similar to ZK. While the payload can be arbitrary, etcd's HTTP API is JSON-based,[3] and as with ZK, you can watch for changes in the values etcd makes available to the cluster. A very useful feature of etcd is that of TTLs on keys, which is a great building block for service discovery. In the same manner as ZK, you want $2n+1$ nodes in an etcd cluster, for the same reasons.

The security model etcd provides allows on-the-wire encryption through TLS/SSL as well as client certificate authentication, both between clients and the cluster and between the etcd nodes.

In Figure 5-2, you can see that the etcd service discovery setup is quite similar to the ZK setup. The main difference is the usage of confd, which configures HAProxy, rather than having you write your own script.

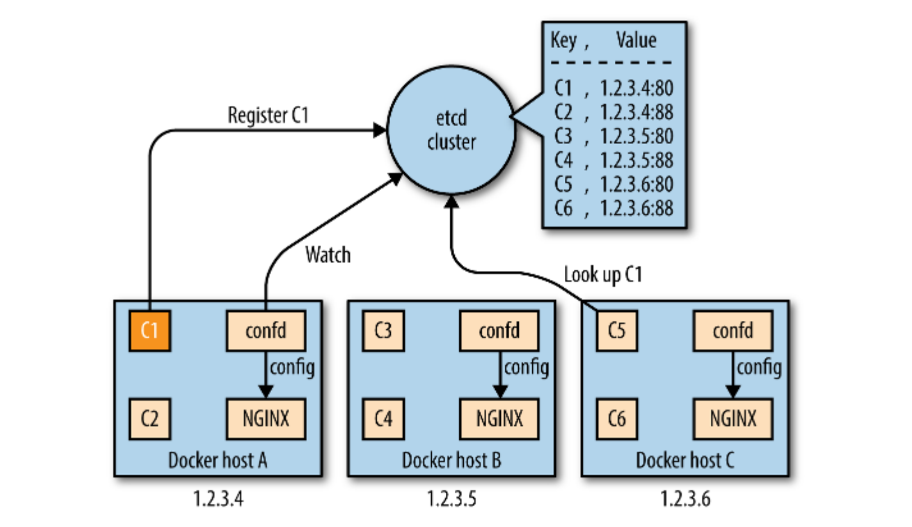


Figure 5-2. Example service discovery setup with etcd

Consul

Consul, a HashiCorp product also written in the Go language, exposes functionality for service registration, discovery, and health checking in an opinionated way. Services can be queried using the HTTP API or through DNS. Consul supports multi-datacenter deployments.

One of Consul's features is a distributed key-value store, akin to etcd. It also uses the Raft consensus algorithm (and again the same observations concerning $2n+1$ nodes as with ZK and etcd apply), but the deployment is different. Consul has the

[3] That is, in contrast to ZK, all you need to interact with etcd is `curl` or the like.

concept of *agents*, which can be run in either of the two available modes—as a server (provides a key-value store and DNS) or as a client (registers services and runs health checks)—and with the membership and node discovery implemented by Serf.

With Consul, you have essentially four options to implement service discovery (ordered from most desirable to least desirable):

- Use a service definition config file, interpreted by the Consul agent.
- Use a tool like traefik that has a Consul backend.
- Write your own sidekick process that registers the service through the HTTP API.
- Bake the registration into the service itself by leveraging the HTTP API.

Want to learn more about using Consul for service discovery? Check out these two great blog posts: "Consul Service Discovery with Docker" by Jeff Lindsay and "Docker DNS & Service Discovery with Consul and Registrator" by Joseph Miller.

Pure-Play DNS-Based Solutions

DNS has been a robust and battle-proven workhorse on the internet for many decades. The eventual consistency of the DNS system, the fact that certain clients aggressively cache DNS lookups,[4] and also the reliance on SRV records make this option something you will want to use when you know that it is the right one.

I've titled this section "Pure-Play DNS-Based Solutions" because Consul technically also has a DNS server, but that's only one option for how you can use it to do service discovery. Here are some popular and widely used pure-play DNS-based service discovery solutions:

Mesos-DNS

This solution is specific for service discovery in Apache Mesos. Written in Go, Mesos-DNS polls the active Mesos master process for any running tasks and exposes the `<ip>:<port>` info via DNS as well as through an HTTP API. For DNS requests for other hostnames or services, Mesos-DNS can either use an external nameserver or leverage your existing DNS server to forward only the requests for Mesos tasks to Mesos-DNS.

SkyDNS

Using etcd, you can announce your services to SkyDNS, which stores service definitions into etcd and updates its DNS records. Your client application

[4] Java, I'm looking at you.

issues DNS queries to discover the services. Thus, functionality-wise it is quite similar to Consul, without the health checks.

WeaveDNS

WeaveDNS was introduced in Weave 0.9 as a simple solution for service discovery on the Weave network, allowing containers to find other containers' IP addresses by their hostnames. In Weave 1.1, a so-called Gossip DNS protocol was introduced, making lookups faster through a cache as well as including timeout functionality. In the new implementation, registrations are broadcast to all participating instances, which subsequently hold all entries in memory and handle lookups locally.

Airbnb's SmartStack and Netflix's Eureka

In this section, we'll take a look at two bespoke systems that were developed to address specific requirements. This doesn't mean you can't or shouldn't use them, just that you should be aware of this heritage.

Airbnb's SmartStack is an automated service discovery and registration framework, transparently handling creation, deletion, failure, and maintenance work. SmartStack uses two separate services that run on the same host as your container: Nerve (writing into ZK) for service registration, and Synapse (dynamically configuring HAProxy) for lookup. It is a well-established solution for non-containerized environments, and time will tell if it will also be as useful with Docker.

Netflix's Eureka is different, mainly because it was born in the AWS environment (where all of Netflix runs). Eureka is a REST-based service used for locating services for the purpose of load balancing and failover of middle-tier servers, and it also comes with a Java-based client component, which makes interactions with the service straightforward. This client has a built-in load balancer that does basic round-robin load balancing. At Netflix, Eureka is used for red/black deployments, for Cassandra and memcached deployments, and for carrying application-specific metadata about services.

Participating nodes in a Eureka cluster replicate their service registries between each other asynchronously. In contrast to ZK, etcd, or Consul, Eureka favors service availability over strong consistency; it leaves it up to the client to deal with stale reads, but has the upside of being more resilient in case of networking partitions. And, you know, the network is reliable. Not.

Load Balancing

An orthogonal but related topic to that of service discovery is *load balancing*. Load balancing enables you to spread the load—that is, the inbound service requests—across a number of containers. In the context of containers and microservices, load balancing achieves a couple of things at the same time:

- It allows throughput to be maximized and response time to be minimized.
- It can avoid hot-spotting—that is, overwhelming a single container with work.
- It can help with overly aggressive DNS caching, such as that found with Java.

The following list outlines some popular load balancing options for containerized setups, in alphabetical order:

Bamboo
: A daemon that automatically configures HAProxy instances, deployed on Apache Mesos and Marathon. See the p24e guide "Service Discovery with Marathon, Bamboo and HAProxy" for a concrete recipe.

Envoy
: A high-performance distributed proxy written in C++, originally built at Lyft. Envoy was designed to be used for single services and applications, and to provide a communication bus and data plane for service meshes. It's the default data plane in Istio.

HAProxy
: A stable, mature, and battle-proven (if not very feature-rich) workhorse. Often used in conjunction with NGINX, HAProxy is reliable and integrations with pretty much everything under the sun exist.

`kube-proxy`
: Runs on each node of a Kubernetes cluster and updates services IPs. It supports simple TCP/UDP forwarding and round-robin load balancing. Note that it's only for cluster-internal load balancing and also serves as a service discovery support component.

MetalLB
: A load-balancer implementation for bare-metal Kubernetes clusters, addressing the fact that Kubernetes does not offer a default implementation for such clusters. In other words, you need to be in a public cloud environment to benefit from this functionality. Note that you may need one or more routers capable of speaking BGP in order for MetalLB to work.

NGINX
: The leading solution in this space. With NGINX you get support for `round-robin`, `least-connected`, and `ip-hash` strategies, as well as on-the-fly configuration, monitoring, and many other vital features.

servicerouter.py
A simple script that gets app configurations from Marathon and updates HAProxy; see also the p24e guide "Service Discovery with Marathon, Bamboo and HAProxy".

traefik
The rising star in this category. Emile Vauge (traefik's lead developer) must be doing something right. I like it a lot, because it's like HAProxy but comes with a bunch of backends, such as Marathon and Consul, out of the box.

Vamp-router
Inspired by Bamboo and Consul–HAProxy, Magnetic.io wrote Vamp-router, which supports updates of the config through a REST API or ZooKeeper, routes and filters for canary releasing and A/B testing, and ACLs, as well as providing statistics.

Vulcand
A reverse proxy for HTTP API management and microservices, inspired by Hystrix.

If you want to learn more about load balancing, check out Kevin Reedy's talk from nginx.conf 2014 on load balancing with NGINX and Consul.

Wrapping It Up

To close out this chapter, I've put together a table that provides an overview of the service discovery solutions we've discussed. I explicitly do not aim at declaring a winner, because I believe the best choice depends on your use case and requirements. So, take the following table as a quick orientation and summary but not as a shootout (also, note that in the context of Kubernetes you don't need to choose one—it's built into the system):

Name	Consistency	Language	Registration	Lookup
ZooKeeper	Strong	Java	Client	Bespoke clients
etcd	Strong	Go	Sidekick + client	HTTP API
Consul	Strong	Go	Automatic and through traefik (Consul backend)	DNS + HTTP/JSON API
Mesos-DNS	Strong	Go	Automatic and through traefik (Marathon backend)	DNS + HTTP/JSON API
SkyDNS	Strong	Go	Client registration	DNS
WeaveDNS	Eventual	Go	Auto	DNS
SmartStack	Strong	Java	Client registration	Automatic through HAProxy config
Eureka	Eventual	Java	Client registration	Bespoke clients

NOTE

Because container service discovery is a moving target, you are well advised to reevaluate your initial choices on an ongoing basis, at least until some consolidation has taken place.

In this chapter you learned about service discovery and how to tackle it, as well as about load balancing options. We will now switch gears and move on to Kubernetes, the de facto container orchestration standard that comes with built-in service discovery (so you don't need to worry about the topics discussed in this chapter) and has its own very interesting approach to container networking across machines.

CHAPTER 6

The Container Network Interface

The Container Network Interface (CNI), as depicted in Figure 6-1, provides a plug-in-oriented networking solution for containers and container orchestrators. It consists of a specification and libraries for writing plug-ins to configure network interfaces in Linux containers.

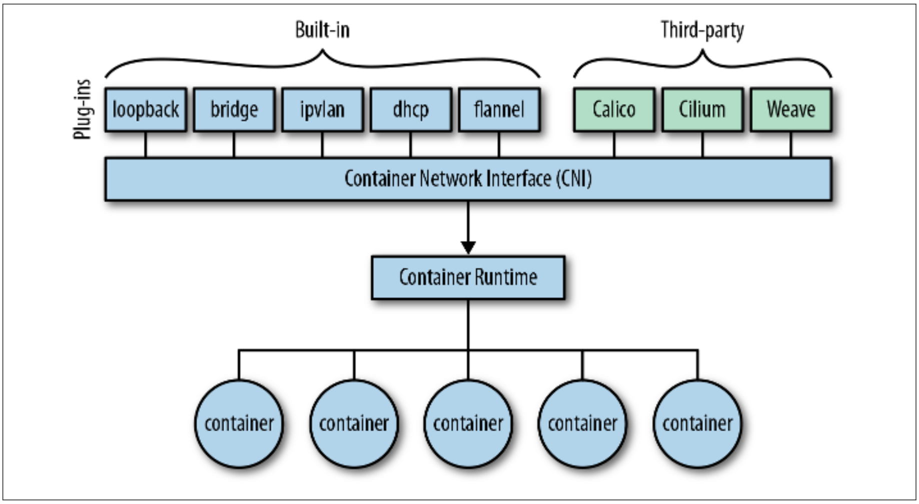


Figure 6-1. 100,000 ft view on CNI

The CNI specification is lightweight; it only deals with the network connectivity of containers, as well as the garbage collection of resources once containers are deleted.

We will focus on CNI in this book since it's the de facto standard for container orchestrators, adopted by all major systems such as Kubernetes, Mesos, and Cloud Foundry. If you're exclusively using Docker Swarm you'll need to use

Docker's libnetwork and might want to read the helpful article by Lee Calcote titled "The Container Networking Landscape: CNI from CoreOS and CNM from Docker", which contrasts CNI with the Docker model and provides you with some guidance.

History

CNI was pioneered by CoreOS in the context of the container runtime `rkt`, to define a common interface between the network plug-ins and container runtimes and orchestrators. Docker initially planned to support it but then came up with the Docker-proprietary libnetwork approach to container networking.

CNI and the libnetwork plug-in interface were developed in parallel from April to June 2015, and after some discussion the Kubernetes community decided not to adopt libnetwork but rather to use CNI. Nowadays pretty much every container orchestrator with the exception of Docker Swarm uses CNI; all runtimes support it and there's a long list of supported plug-ins, as discussed in "Container Runtimes and Plug-ins" on page 40.

In May 2017, the Cloud Native Computing Foundation (CNCF) made CNI a full-blown top-level project.

Specification and Usage

In addition to the CNI specification, at time of writing in version 0.3.1-dev, the repository contains the Go source code of a library for integrating CNI into applications as well as an example command-line tool for executing CNI plug-ins. The plug-ins repository contains reference plug-ins and a template for creating new plug-ins.

Before we dive into the usage, let's look at two central definitions in the context of CNI:

Container
: Synonymous with a Linux network namespace. What unit this corresponds to depends on the container runtime implementation (single container or pod).

Network
: A uniquely addressable group of entities that can communicate with one another. These entities might be an individual container, a machine, or some other network device such as a router.

Let's now have a look at how CNI—on a high level—works, as depicted in Figure 6-2. First, the container runtime takes some configuration and issues a command to a plug-in. The plug-in then goes off and configures the network.

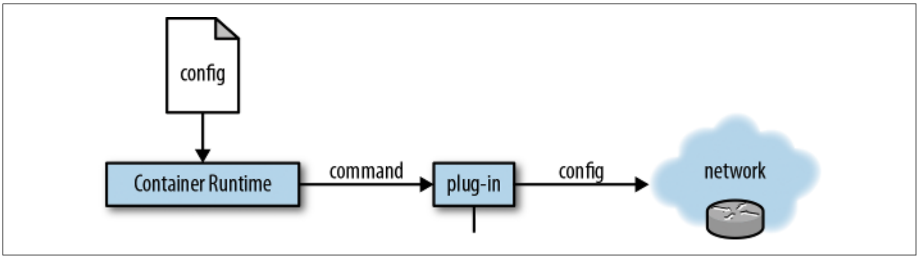


Figure 6-2. CNI high-level architecture

So, what CNI conceptually enables you to do is to add containers to a network as well as remove them. The current version of CNI defines the following operations:

- Add container to one or more networks
- Delete container from network
- Report CNI version

In order for CNI to add a container to a network, the container runtime must first create a new network namespace for the container and then invoke one or more of the defined plug-ins. The network configuration is in JSON format and includes mandatory fields such as `name` and `type` as well as plug-in type–specific fields. The actual command (for example, `ADD`) is passed in as an environment variable aptly named `CNI_COMMAND`.

IP Allocation with CNI

A CNI plug-in is expected to assign an IP address to the interface and set up network routes relevant for it. This gives the CNI plug-in great flexibility but also places a large burden on it. To accommodate this, CNI defines a dedicated IP Address Management (IPAM) plug-in that takes care of the IP range management independently.

Let's have a look at a concrete CNI command in action:

```
$ CNI_COMMAND=ADD \
  CNI_CONTAINERID=875410a4c38d7 \
  CNI_NETNS=/proc/1234/ns/net \
  CNI_IFNAME=eth0 \
  CNI_PATH=/opt/cni/bin \
  someplugin < /etc/cni/net.d/someplugin.conf
```

This example shows how a certain plug-in (`someplugin`) is applied to a given container (`875410a4c38d7`) using a specific configuration (*someplugin.conf*). Note that while initially all configuration parameters were passed in as environment variables, the people behind the spec are moving more and more toward using the (JSON) configuration file.

You can learn more about using CNI in the excellent blog post "Understanding CNI" by Jon Langemak.

Container Runtimes and Plug-ins

In addition to pretty much any container orchestrator and container runtime (Kubernetes, Mesos, Cloud Foundry) supporting CNI, it ships with a number of built-in plug-ins, such as `loopback` and `vlan`. There's also a long list of third-party CNI plug-ins available. Here is a selection of the most important ones, in alphabetical order:

Amazon ECS CNI Plugins
: A collection of CNI plug-ins used by the Amazon ECS Agent to configure the network namespace of containers with Elastic Network Interfaces (ENIs).

Bonding
: A CNI plug-in for failover and high availability of networking in cloud-native environments, by Intel.

Calico
: Project Calico's network plug-in for CNI. Project Calico manages a flat layer 3 network, assigning each workload a fully routable IP address. For environments requiring an overlay network Calico uses IP-in-IP tunneling or can work with other overlay networks, such as flannel.

Cilium
: A BPF-based solution providing connectivity between containers, operating at layer 3/4 to provide networking and security services as well as layer 7 to protect modern protocols such as HTTP and gRPC.

CNI-Genie
: A generic CNI network plug-in by Huawei.

Infoblox
: An IPAM driver for CNI that interfaces with Infoblox to provide IP Address Management services.

Linen
: A CNI plug-in designed for overlay networks with Open vSwitch.

Multus
: A powerful multi–plug-in environment by Intel.

Nuage CNI
: A Nuage Networks SDN plug-in supporting network policies for Kubernetes.

Romana
A layer 3 CNI plug-in supporting network policy for Kubernetes.

Silk
A network fabric for containers, inspired by flannel, designed for Cloud Foundry.

Vhostuser
A plug-in to run with Open vSwitch and OpenStack VPP along with the Multus CNI plug-in in Kubernetes for bare-metal container deployment models.

Weave Net
A multi-host Docker network by Weaveworks.

Wrapping It Up

With this we conclude the CNI chapter and move on to Kubernetes and its networking approach. CNI plays a central role in Kubernetes (networking), and you might want to check the docs there as well.

CHAPTER 7
Kubernetes Networking

This chapter will first quickly bring you up to speed concerning Kubernetes, then introduce you to the networking concepts on a high level. Then we'll jump into the deep end, looking at how container networking is realized in Kubernetes, what traffic types exist and how you can make services talk to each other within the cluster, as well as how you can get traffic into your cluster and to a specific service.

A Gentle Kubernetes Introduction

Kubernetes is an open source container orchestration system. It captures Google's lessons learned from running containerized workloads on Borg for more than a decade. As of early 2018 Kubernetes is considered the de facto industry standard for container orchestration, akin to the Linux kernel for the case of a single machine.

> NOTE
>
> I'd argue that there are at least two significant points in time concerning the birth of Kubernetes. The first was on June 7, 2014, with Joe Beda's initial commit on GitHub that marked the beginning of the open sourcing of the project. The second was almost a year later, on July 20, 2015, when Google launched Kubernetes 1.0 and announced the formation of a dedicated entity to host and govern Kubernetes, the Cloud Native Computing Foundation (CNCF). As someone who was at the launch event (and party), I can tell you, that's certainly one way to celebrate the birth of a project.

Kubernetes's architecture (Figure 7-1) provides support for a number of workloads, allowing you to run stateless as well as stateful containerized applications. You can launch long-running processes, such as low-latency app servers, as well as batch jobs.

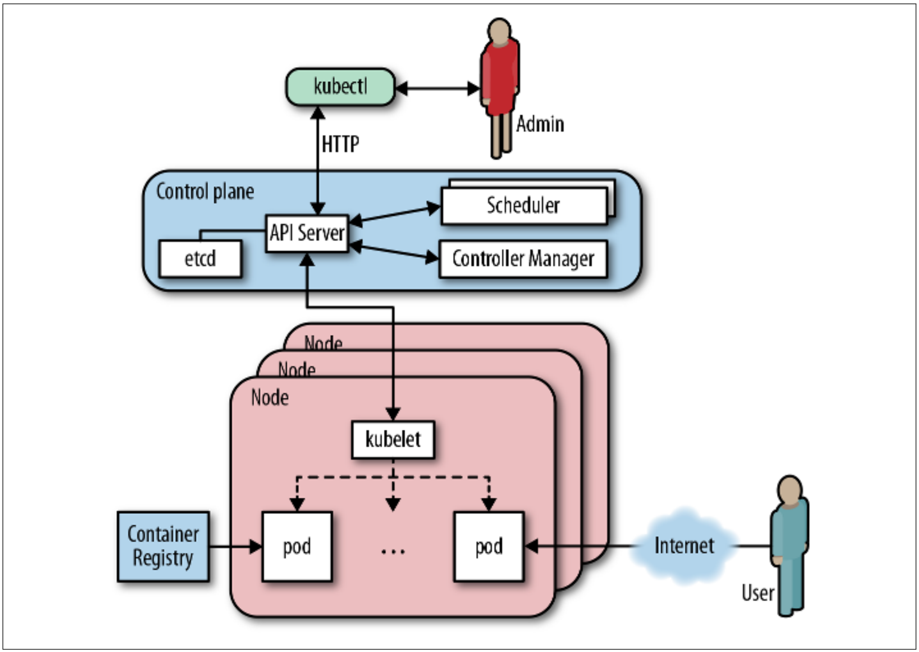


Figure 7-1. An overview of the Kubernetes architecture

The unit of scheduling in Kubernetes is a pod. Essentially, this is a tightly coupled set of one or more containers that are always collocated (that is, scheduled onto a node as a unit); they cannot be spread over nodes. The number of running instances of a pod—called replicas—can be declaratively stated and enforced through controllers. The logical organization of all resources, such as pods, deployments, or services, happens through labels.

With Kubernetes you almost always have the option to swap out the default implementations with some open source or closed source alternative, be it DNS or monitoring. Kubernetes is highly extensible, from defining new workloads and resource types in general to customizing its user-facing parts, such as the CLI tool `kubectl` (pronounced *cube cuddle*).

This chapter assumes you're somewhat familiar with Kubernetes and its terminology. Should you need to brush up your knowledge of how Kubernetes works, I suggest checking out the Concepts section in the official docs or the book *Kubernetes Up and Running* (O'Reilly) by Brendan Burns, Kelsey Hightower, and Joe Beda.

Kubernetes Networking Overview

Rather than prescribing a certain networking solution, Kubernetes only states three fundamental requirements:

- Containers can communicate with all other containers without NAT.
- Nodes can communicate with all containers (and vice versa) without NAT.
- The IP a container sees itself is the same IP as others see it.

How you meet these requirements is up to you. This means you have a lot of freedom to realize networking with and for Kubernetes. It also means, however, that Kubernetes on its own will only provide so much; for example, it supports CNI (Chapter 6) but it doesn't come with a default SDN solution. In the networking area, Kubernetes is at the same time strangely opinionated (see the preceding requirements) and not at all (no batteries included).

From a network traffic perspective we differentiate between three types in Kubernetes, as depicted in Figure 7-2:

Intra-pod networking

All containers within a pod share a network namespace and see each other on `localhost`. Read "Intra-Pod Networking" on page 46 for details.

Inter-pod networking

Two types of east–west traffic are supported: pods can directly communicate with other pods or, preferably, pods can leverage services to communicate with other pods. Read "Inter-Pod Networking" on page 47 for details.

Ingress and egress

Ingress refers to routing traffic from external users or apps to pods, and egress refers to calling external APIs from pods. Read "Ingress and Egress" on page 53 for details.

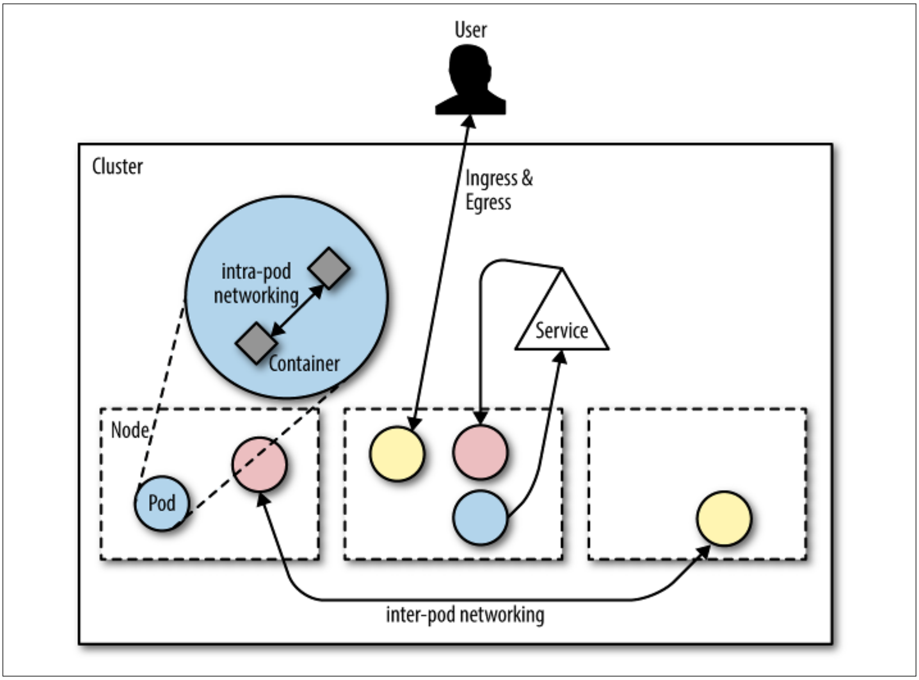


Figure 7-2. Kubernetes network traffic types

NOTE

Kubernetes requires each pod to have an IP in a flat networking namespace with full connectivity to other nodes and pods across the network. This IP-per-pod model yields a backward-compatible way for you to treat a pod almost identically to a VM or a physical host, in the context of naming, service discovery, or port allocations. The model allows for a smoother transition from non–cloud native apps and environments.

Intra-Pod Networking

Within a pod there exists a so-called *infrastructure container*. This is the first container that the kubelet launches, and it acquires the pod's IP and sets up the network namespace. All the other containers in the pod then join the infra container's network and IPC namespace. The infra container has network bridge mode enabled (see "Bridge Mode Networking" on page 7) and all the other containers in the pod join this namespace via container mode (covered in "Container Mode Networking" on page 9). The initial process that runs in the infra container does effectively nothing,[1] as its sole purpose is to act as the home for the name-

[1] See pause.go for details; basically it blocks until it receives a `SIGTERM`.

spaces. If the infra container dies, the kubelet kills all the containers in the pod and then starts the process over.

As a result of above, all containers within a pod can communicate amongst each other using `localhost` (or `127.0.0.1` in IPv4). You are responsible yourself to make sure containers within a pod do not conflict with each other in terms of ports used. Note also that the Kubernetes approach here also means reduced isolation between containers within a pod; however, this is by design and since we consider the tight coupling here a good thing, it is probably not something you need to worry about, though it's good to be aware of it.

If you want to learn more about the infra container, read The Almighty Pause Container by Ian Lewis.

Inter-Pod Networking

In Kubernetes, each pod has a routable IP, allowing pods to communicate across cluster nodes without NAT and no need to manage port allocations. Because every pod gets a real (that is, not machine-local) IP address, pods can communicate without proxies or translations (such as NAT). The pod can use well-known ports and can avoid the use of higher-level service discovery mechanisms such as those we discussed in Chapter 5.

We distinguish between two types of inter-pod communication, sometimes also called East-West traffic:

- Pods can directly communicate with other pods; in this case the caller pod needs to find out the IP address of the callee and risks repeating this operation since pods come and go (cattle behaviour).
- Preferably, pods use services to communicate with other pods. In this case, the service provides a stable (virtual) IP address that can be discovered, for example, via DNS.

Difference to Docker Model

Note that the Kubernetes flat address space model is different from the default Docker model. There, each container gets an IP address in the `172.x.x.x` space and only sees this `172.` address. If this container connects to another container the peer would see the connection coming from a different IP than the container itself sees. This means you can never self-register anything from a container because a container cannot be reached on its private IP.

When a container tries to obtain the address of network interface it sees the same IP that any peer container would see them coming from; each pod has its own IP address that other pods can find and use. By making IP addresses and ports the same both inside and outside the pods, Kubernetes creates a flat address space across the cluster. For more details on this topic see also the article "Understanding Kubernetes Networking: Pods" by Mark Betz.

Let's now focus on the *service*, as depicted in Figure 7-3.

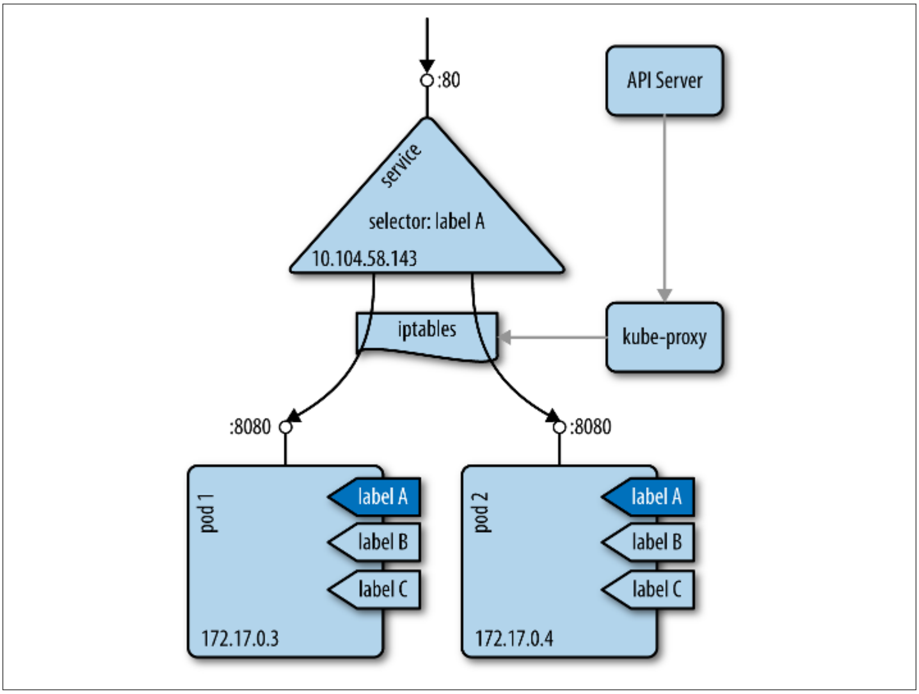


Figure 7-3. The Kubernetes service concept

A service provides a stable virtual IP (VIP) address for a set of pods. While pods may come and go, services allow clients to reliably discover and connect to the containers running in the pods by using the VIP. The "virtual" in VIP means it's not an actual IP address connected to a network interface; its purpose is purely to act as the stable front to forward traffic to one or more pods, with IP addresses that may come and go.

NOTE

What VIPs Really Are

It's essential to realize that VIPs do not exist as such in the networking stack. For example, you can't ping them. They are only Kubernetes-internal administrative entities. Also note that the format is IP:PORT, so the IP address along with the port make up the VIP. Just think of a VIP as a kind of index into a data structure mapping to actual IP addresses.

As you can see in Figure 7-3, the service with the VIP 10.104.58.143 routes the traffic to one of the pods 172.17.0.3 or 172.17.0.4. Note here the different subnets for the service and pods, see Network Ranges for further details on the reason behind that. Now, you might be wondering how this actually works? Let's have a look at it.

You specify the set of pods you want a service to target via a label selector, for example, for spec.selector.app=someapp Kubernetes would create a service that targets all pods with a label app=someapp. Note that if such a selector exists, then for each of the targeted pods a sub-resource of type Endpoint will be created, and if no selector exists then no endpoints are created. For example, see in the following code example the output of the kubectl describe command. Such endpoints are also not created in the case of so-called headless services, which allow you to exercise great control over how the IP management and service discovery takes place.

Keeping the mapping between the VIP and the pods up-to-date is the job of kube-proxy (see also the docs on kube-proxy), a process that runs on every node on the cluster.

This kube-proxy process queries the API server to learn about new services in the cluster and updates the node's iptables rules accordingly, to provide the necessary routing information. To learn more how exactly services work, check out Kubernetes Services By Example.

Let's see how this works in practice: assuming there's an existing deployment called nginx (for example, execute kubectl run webserver --image nginx) you can automatically create a service like so:

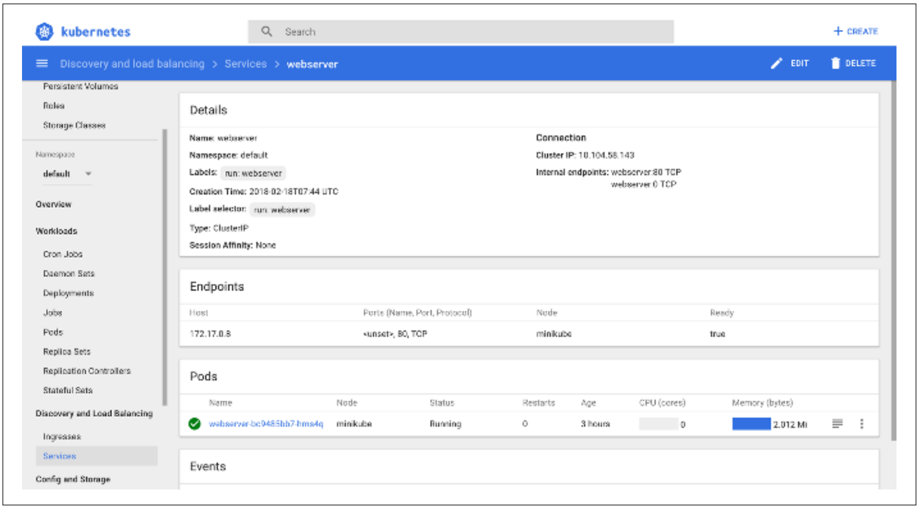
```
$ kubectl expose deployment/webserver --port 80
service "webserver" exposed

$ kubectl describe service/webserver
Name:               webserver
Namespace:          default
Labels:             run=webserver
Annotations:        <none>
Selector:           run=webserver
Type:               ClusterIP
IP:                 10.104.58.143
```

```
Port:              <unset>  80/TCP
TargetPort:        80/TCP
Endpoints:         172.17.0.3:8080,172.17.0.4:8080
Session Affinity:  None
Events:            <none>
```

After executing the above `kubectl expose` command, you will see the service appear:

```
$ kubectl get service -l run=webserver
NAME        TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
webserver   ClusterIP   10.104.58.143   <none>        80/TCP    1m
```

Above, note two things: the service has got itself a cluster-internal IP (`CLUSTER-IP` column) and the `EXTERNAL-IP` column tells you that this service is only available from within the cluster, that is, no traffic from outside of the cluster can reach this service (yet)—see "Ingress and Egress" on page 53 to learn how to change this situation.



Details

Name: webserver
Namespace: default
Labels: run: webserver
Creation Time: 2018-02-18T07:44 UTC
Label selector: run: webserver
Type: ClusterIP
Session Affinity: None

Connection
Cluster IP: 10.104.58.143
Internal endpoints: webserver:80 TCP
webserver:0 TCP

Endpoints

Host	Ports (Name, Port, Protocol)	Node	Ready
172.17.0.8	<unset>, 80, TCP	minikube	true

Pods

	Name	Node	Status	Restarts	Age	CPU (cores)	Memory (bytes)
✓	webserver-bc9485bb7-hms4q	minikube	Running	0	3 hours	0	2.012 Mi

Events

Figure 7-4. Kubernetes service in the dashboard

In Figure 7-4 you can see the representation of the service in the Kubernetes dashboard.

Service Discovery in Kubernetes

Let us now talk about how service discovery works in Kubernetes.

Conceptually, you can use one of the two built-in discovery mechanisms:

- Through environment variables (limited)

- Using DNS, which is available cluster-wide if a respective DNS cluster add-on has been installed

Environment Variables–Based Service Discovery

For the environment variables–based discovery method, a simple example might look like the example code to follow: using a jump pod to get us into the cluster and then running from there the `env` built-in shell command (note that the output has been edited to be easier to digest).

```
$ kubectl run -it --rm jump --restart=Never \
              --image=quay.io/mhausenblas/jump:v0.1 -- sh
If you don't see a command prompt, try pressing enter.
/ # env
HOSTNAME=jump
WEBSERVER_SERVICE_HOST=10.104.58.143
WEBSERVER_PORT=tcp://10.104.58.143:80
WEBSERVER_SERVICE_PORT=80
WEBSERVER_PORT_80_TCP_ADDR=10.104.58.143
WEBSERVER_PORT_80_TCP_PORT=80
WEBSERVER_PORT_80_TCP_PROTO=tcp
WEBSERVER_PORT_80_TCP=tcp://10.104.58.143:80
...
```

Above, you can see the service discovery in action: the environment variables `WEBSERVER_XXX` give you the IP address and port you can use to connect to the service. For example, while still in the jump pod, you could execute `curl 10.104.58.143` and you should see the NGINX welcome page.

While convenient, note that discovery via environment variables has a fundamental drawback: any service that you want to discover must be created before the pod from which you want to discover it as otherwise the environment variables will not be populated by Kubernetes. Luckily there exists a better way: *DNS*.

DNS-Based Service Discovery

Mapping a fully qualified domain name (FQDN) like `example.com` to an IP address such as `123.4.5.66` is what DNS was designed for and has been doing for us on a daily basis on the internet for more than 30 years.

NOTE

Choosing a DNS Solution

When rolling your own Kubernetes distro, that is, putting together all the required components such as SDN or the DNS add-on yourself rather than using an offering from the more than 30 certified Kubernetes offerings, it's worth considering the CNCF project CoreDNS over the older and less feature-rich kube-dns DNS cluster add-on (which is part of Kubernetes proper).

So how can we use DNS to do service discovery in Kubernetes? It's easy, if you have the DNS cluster add-on installed and enabled. This DNS server watches on the Kubernetes API for services being created or removed. It creates a set of DNS records for each service it observes.

In the next example, let's use our `webserver` service from above and assume we have it running in the `default` namespace. For this service, a DNS record `webserver.default` (with a FQDN of `webserver.default.cluster.local`) should be present.

```
$ kubectl run -it --rm jump --restart=Never \
              --image=quay.io/mhausenblas/jump:v0.1 -- sh
If you don't see a command prompt, try pressing enter.
/ # curl webserver.default
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

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

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

Pods in the same namespace can reach the service by its shortname `webserver`, whereas pods in other namespaces must qualify the name as `webserver.default`. Note that the result of these FQDN lookups is the pod's cluster IP. Further, Kubernetes supports DNS service (`SRV`) records for named ports. So if our `webserver` service had a port named, say, `http` with the protocol type TCP, you could issue a DNS `SRV` query for `_http._tcp.webserver` from the same namespace to discover the port number for `http`. Note also that the virtual IP for a service is stable, so the DNS result does not have to be requeried.

Network Ranges

From an administrative perspective, you are conceptually dealing with three networks: the pod network, the service network, and the host network (the machines hosting Kubernetes components such as the `kubelet`). You will need a strategy regarding how to partition the network ranges; one often found strategy is to use networks from the private range as defined in RFC 1918, that is, `10.0.0.0/8`, `172.16.0.0/12`, and `192.168.0.0/16`.

Ingress and Egress

In the following we'll have a look at how traffic flows in and out of a Kubernetes cluster, also called North-South traffic.

Ingress

Up to now we have discussed how to access a pod or service from within the cluster. Accessing a pod from outside the cluster is a bit more challenging. Kubernetes aims to provide highly available, high-performance load balancing for services.

Initially, the only available options for North-South traffic in Kubernetes were `NodePort`, `LoadBalancer`, and `ExternalName`, which are still available to you. For layer 7 traffic (i.e., HTTP) a more portable option is available, however: introduced in Kubernetes 1.2 as a beta feature, you can use Ingress to route traffic from the external world to a service in our cluster.

Ingress in Kubernetes works as shown in Figure 7-5: conceptually, it is split up into two main pieces, an *Ingress resource*, which defines the routing to the backing services, and the *Ingress controller*, which listens to the `/ingresses` endpoint of the API server, learning about services being created or removed. On service status changes, the Ingress controller configures the routes so that external traffic lands at a specific (cluster-internal) service.

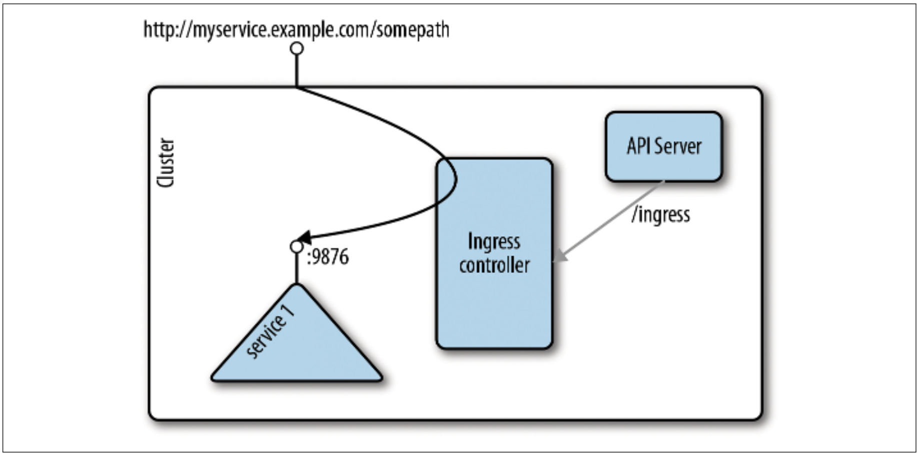


Figure 7-5. Ingress concept

The following example presents a concrete example of an Ingress resource, to route requests for `myservice.example.com/somepath` to a Kubernetes service named `service1` on port 9876.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: example-ingress
spec:
  rules:
  - host: myservice.example.com
    http:
      paths:
      - path: /somepath
        backend:
          serviceName: service1
          servicePort: 9876
```

Now, the Ingress resource definition is nice, but without a controller, nothing happens. So let's deploy an ingress controller, in this case using Minikube.

```
$ minikube addons enable ingress
```

Once you've enabled Ingress on Minikube, you should see it appear as enabled in the list of Minikube add-ons. After a minute or so, two new pods will start in the `kube-system` namespace, the backend and the controller. So now you can use it, using the manifest in the following example, which configures a path to an NGINX webserver.

```
$ cat nginx-ingress.yaml
kind: Ingress
apiVersion: extensions/v1beta1
metadata:
  name: nginx-public
  annotations:
    ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host:
    http:
      paths:
      - path: /web
        backend:
          serviceName: nginx
          servicePort: 80
```

```
$ kubectl create -f nginx-ingress.yaml
```

Now NGINX is available via the IP address `192.168.99.100` (in this case my Minikube IP) and the manifest file defines that it should be exposed via the path `/web`.

Note that Ingress controllers can technically be any system capable of reverse proxying, but NGINX is most commonly used. Further, Ingress can also be implemented by a cloud-provided load balancer, such as Amazon's ALB.

For more details on Ingress, read the excellent article "Understanding Kubernetes Networking: Ingress" by Mark Betz and make sure to check out the results of the survey the Kubernetes SIG Network carried out on this topic.

Egress

While in the case of Ingress we're interested in routing traffic from outside the cluster to a service, in the case of Egress we are dealing with the opposite: how does an app in a pod call out to (cluster-)external APIs?

One may want to control which pods are allowed to have a communication path to outside services and on top of that impose other policies. Note that by default all containers in a pod can perform Egress. These policies can be enforced using network policies as described in "Network Policies" on page 55 or by deploying a service mesh as in "Service Meshes" on page 56.

Advanced Kubernetes Networking Topics

In the following I'll cover two advanced and somewhat related Kubernetes networking topics: network policies and service meshes.

Network Policies

Network policies in Kubernetes are a feature that allow you to specify how groups of pods are allowed to communicate with each other. From Kubernetes version 1.7 and above network policies are considered stable and hence you can use them in production.

Let's take a look at a concrete example of how this works in practice. For example, say you want to suppress all traffic to pods in the namespace `superprivate`. You'd create a default Egress policy for that namespace as in the following example:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: bydefaultnoegress
  namespace: superprivate
spec:
  podSelector: {}
  policyTypes:
  - Egress
```

Note that different Kubernetes distros support network policies to different degrees: for example, in OpenShift they are supported as first-class citizens and a range of examples is available via the *redhat-cop/openshift-toolkit* GitHub repo.

If you want to learn more about how to use network policies, check out Ahmet Alp Balkan's brilliant and detailed hands-on blog post, "Securing Kubernetes Cluster Networking".

Service Meshes

Going forward, you can make use of service meshes such as the two discussed in the following. The idea of a service mesh is that rather than putting the burden of networking communication and control onto you, the developer, you outsource these nonfunctional things to the mesh. So you benefit from traffic control, observability, security, etc. *without any changes* to your source code. Sound fantastic? It is, believe you me.

Istio

Istio is a modern and popular service mesh, available for Kubernetes but not exclusively so. It's using Envoy as the default data plane and mainly focusing on the control-plane aspects. It supports monitoring (Prometheus), tracing (Zipkin/Jaeger), circuit breakers, routing, load balancing, fault injection, retries, timeouts, mirroring, access control, and rate limiting out of the box, to name a few features. Istio takes the battle-tested Envoy proxy (cf. "Load Balancing" on page 32) and packages it up as a sidecar container in your pod. Learn more about Istio via Christian Posta's wonderful resource: Deep Dive Envoy and Istio Workshop.

Buoyant's Conduit

This service mesh is deployed on a Kubernetes cluster as a data plane (written in Rust) made up of proxies deployed as sidecar containers alongside your app and a control plane (written in Go) of processes that manages these proxies, akin to what you've seen in Istio above. After the CNCF project Linkerd this is Buoyant's second iteration on the service mesh idea; they are the pioneers in this space, establishing the service mesh idea in 2016. Learn more via Abhishek Tiwari's excellent blog post, "Getting started with Conduit - lightweight service mesh for Kubernetes".

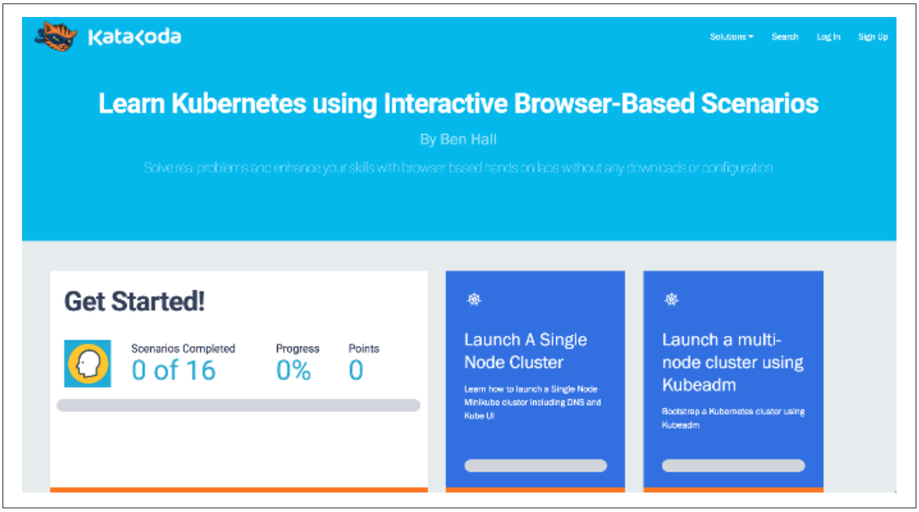
One note before we wrap up this chapter and also the book: service meshes are still pretty new, so you might want to think twice before deploying them in prod —unless you're Lyft or Google or the like ;)

Wrapping It Up

In this chapter we've covered the Kubernetes approach to container networking and showed how to use it in various setups. With this we conclude the book; thanks for reading and if you have feedback, please do reach out via Twitter.

References

Reading stuff is fine, and here I've put together a collection of links that contain either background information on topics covered in this book or advanced material, such as deep dives or teardowns. However, for a more practical approach I suggest you check out Katacoda, a free online learning environment that contains 100+ scenarios from Docker to Kubernetes (see for example the screenshot in Figure A-1).



Learn Kubernetes using Interactive Browser-Based Scenarios

By Ben Hall

Solve real problems and enhance your skills with browser based hands on labs without any downloads or configuration

Get Started!

Scenarios Completed: 0 of 16
Progress: 0%
Points: 0

Launch A Single Node Cluster

Learn how to launch a Single Node Minikube cluster including DNS and Kube UI

Launch a multi-node cluster using Kubeadm

Bootstrap a Kubernetes cluster using Kubeadm

Figure A-1. Katacoda Kubernetes scenarios

You can use Katacoda in any browser; sessions are typically terminated after one hour.

Container Networking References

Networking 101

- “Network Protocols” from the Programmer’s Compendium
- “Demystifying Container Networking” by Michele Bertasi
- “An Empirical Study of Load Balancing Algorithms” by Khalid Lafi

Linux Kernel and Low-Level Components

- “The History of Containers” by thildred
- “A History of Low-Level Linux Container Runtimes” by Daniel J. Walsh
- “Networking in Containers and Container Clusters” by Victor Marmol, Rohit Jnagal, and Tim Hockin
- “Anatomy of a Container: Namespaces, cgroups & Some Filesystem Magic” by Jérôme Petazzoni
- “Network Namespaces” by corbet
- Network classifier cgroup documentation
- “Exploring LXC Networking” by Milos Gajdos

Docker

- Docker networking overview
- “Concerning Containers’ Connections: On Docker Networking” by Federico Kereki
- “Unifying Docker Container and VM Networking” by Filip Verloy
- “The Tale of Two Container Networking Standards: CNM v. CNI” by Harmeet Sahni

Kubernetes Networking References

Kubernetes Proper and Docs

- Kubernetes networking design
- Services
- Ingress

- Cluster Networking
- Provide Load-Balanced Access to an Application in a Cluster
- Create an External Load Balancer
- Kubernetes DNS example
- Kubernetes issue 44063: Implement IPVS-based in-cluster service load balancing
- "Data and analysis of the Kubernetes Ingress survey 2018" by the Kubernetes SIG Network

General Kubernetes Networking

- "Kubernetes Networking 101" by Bryan Boreham of Weaveworks
- "An Illustrated Guide to Kubernetes Networking" by Tim Hockin of Google
- "The Easy—Don't Drive Yourself Crazy—Way to Kubernetes Networking" by Gerard Hickey (KubeCon 2017, Austin)
- "Understanding Kubernetes Networking: Pods", "Understanding Kubernetes Networking: Services", and "Understanding Kubernetes Networking: Ingress" by Mark Betz
- "Understanding CNI (Container Networking Interface)" by Jon Langemak
- "Operating a Kubernetes Network" by Julia Evans
- "nginxinc/kubernetes-ingress" Git repo
- "The Service Mesh: Past, Present, and Future" by William Morgan (KubeCon 2017, Austin)
- "Meet Bandaid, the Dropbox Service Proxy" by Dmitry Kopytkov and Patrick Lee
- "Kubernetes NodePort vs LoadBalancer vs Ingress? When Should I Use What?" by Sandeep Dinesh

About the Author

Michael Hausenblas is a developer advocate for Go, Kubernetes, and OpenShift at Red Hat, where he helps appops to build and operate distributed services. His background is in large-scale data processing and container orchestration and he's experienced in advocacy and standardization at the W3C and IETF. Before Red Hat, Michael worked at Mesosphere and MapR and in two research institutions in Ireland and Austria. He contributes to open source software (mainly using Go), speaks at conferences and user groups, blogs, and hangs out on Twitter too much.