O'REILLY®

Networking & Kubernetes

A Layered Approach

Free Chapters

compliments of

LogicMonitor

James Strong
& Vallery Lancey

LogicMonitor

At **LogicMonitor**, we recognize the crucial role of network performance in your organization's operations, whether you're migrating to the cloud or harnessing IT data for insights. Unfortunately, network outages can be challenging to pinpoint, manage, and quantify.

We are excited to sponsor chapters in *Networking and Kubernetes* to help you enhance your networking and performance knowledge. In these chapters, you will explore the latest networking trends, delve into the intricacies of Linux networking, and come to understand the challenges of deploying containerized applications in cloud networks. Afterwards, you can use this newfound knowledge to optimize your tech stack, troubleshoot network issues, and reduce downtime.

What sets LogicMonitor apart? We provide a comprehensive monitoring solution for a single view across your systems, regardless of whether your network is on-premise or cloud-managed.

With LogicMonitor you gain:

- **A unified view of your IT data** across a hybrid, multi-cloud environment
- **Comprehensive visibility** across distributed Kubernetes environments, spanning topology, performance, availability, and log-based insights
- **Ease of use** with automated discovery, data collection, and thresholding
- **A robust ecosystem** with over 2,500 ready-to-use integrations
- **Automated intelligence** with forecasting, in-context logs and views, and AIOps-enabled workflows

For more information, visit logicmonitor.com. Contact LogicMonitor when you're ready to consolidate your infrastructure monitoring tools behind a single pane of glass.

Networking and Kubernetes

A Layered Approach

This excerpt contains Chapters 1, 2, and 6. The complete book is available on the O'Reilly Online Learning Platform and through other retailers.

James Strong and Vallery Lancey

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Networking and Kubernetes
by James Strong and Vallery Lancey

Printed in the United States of America.

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

Acquisitions Editor: John Devins
Development Editor: Melissa Potter
Production Editor: Beth Kelly
Copyeditor: Kim Wimpsett
Proofreader: Piper Editorial Consulting, LLC

Indexer: Sam Arnold-Boyd
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

September 2021: First Edition

Revision History for the First Edition
2021-09-07: First Release

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

978-1-492-08165-4

[LSI]

Table of Contents

CHAPTER 1

Networking Introduction

"Guilty until proven innocent." That's the mantra of networks and the engineers who supervise them. In this opening chapter, we will wade through the development of networking technologies and standards, give a brief overview of the dominant theory of networking, and introduce our Golang web server that will be the basis of the networking examples in Kubernetes and the cloud throughout the book.

Let's begin…at the beginning.

Networking History

The internet we know today is vast, with cables spanning oceans and mountains and connecting cities with lower latency than ever before. Barrett Lyon's "Mapping the Internet," shown in Figure 1-1, shows just how vast it truly is. That image illustrates all the connections between the networks of networks that make up the internet. The purpose of a network is to exchange information from one system to another system. That is an enormous ask of a distributed global system, but the internet was not always global; it started as a conceptual model and slowly was built up over time, to the behemoth in Lyon'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 their internet service provider's network—all the way to scaling up to the geopolitical landscape of the internet. The internet is integrated into the fabric of our society. In this book, we will discuss how networks operate and how Kubernetes abstracts them for us.

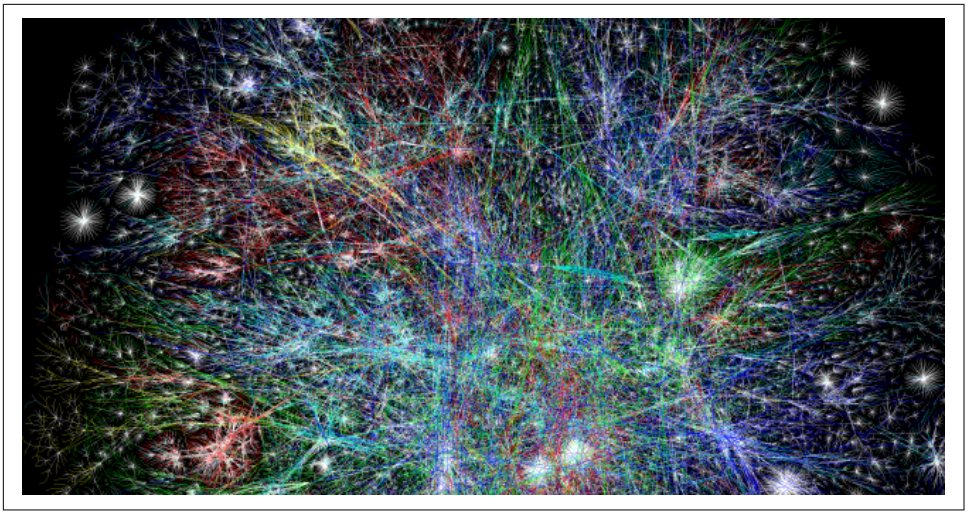

Figure 1-1. Barrett Lyon, "Mapping the Internet," 2003

Table 1-1 briefly outlines the history of networking before we dive into a few of the important details.

Table 1-1. A brief history of networking

Year	Event
1969	ARPANET's first connection test
1969	Telnet 1969 Request for Comments (RFC) 15 drafted
1971	FTP RFC 114 drafted
1973	FTP RFC 354 drafted
1974	TCP RFC 675 by 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 (OSI) model published
1991	National Information Infrastructure (NII) Bill passed with Al Gore's help
1991	First version of Linux released
2015	First version of Kubernetes released

In its earliest forms, networking was government run or sponsored; in the United States, the Department of Defense (DOD) sponsored the Advanced Research Projects Agency Network (ARPANET), well before Al Gore's time in politics, which will be relevant in a moment. In 1969, ARPANET was deployed at the University of California–Los Angeles, the Augmentation Research Center at Stanford Research Institute, the University of California–Santa Barbara, and the University of Utah School of Computing. Communication between these 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 protocols like Telnet and File Transfer Protocol (FTP).

The success of ARPANET and NCP, the first protocol to power ARPANET, led to NCP's downfall. It could not keep up with the demands of the network and the variety of networks connected. In 1974, Vint Cerf, Yogen Dalal, and Carl Sunshine began drafting RFC 675 for Transmission Control Protocol (TCP). (You'll learn more about RFCs in a few paragraphs.) TCP would go on to become the standard for network connectivity. TCP allowed for exchanging packets across different types of networks. In 1981, the Internet Protocol (IP), defined in RFC 791, helped break out the responsibilities of TCP into a separate protocol, increasing the modularity of the network. In the following years, many organizations, including the DOD, adopted TCP as the standard. By January 1983, TCP/IP had become the only approved protocol on ARPANET, replacing the earlier NCP because of its versatility and modularity.

A competing standards organization, the International Organization for Standardization (ISO), developed and published ISO 7498, "Open Systems Interconnection Reference Model," which detailed 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, however.

In 1991, Al Gore invented the internet (well, really he helped pass the National Information Infrastructure [NII] Bill), which helped lead to the creation of the Internet Engineering Task Force (IETF). Nowadays standards for the internet are under the management of the IETF, an open consortium of leading experts and companies in the field of networking, like Cisco and Juniper. RFCs 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 for the internet's functioning.

An IETF RFC has two states:

Proposed Standard
: A protocol specification has reached enough community support to be considered a standard. The designs are stable and 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, enjoys significant public support, and is recognizably useful in some parts of the internet."

Draft standard is a third classification that was discontinued in 2011.

There are thousands of internet standards defining how to implement protocols for all facets of networking, including wireless, encryption, and data formats, among others. Each one is implemented by contributors of open source 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 OSI model.

OSI Model

The 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 relationships 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"

The ISO/IEC 7498-1 describes what the OSI model attempts to convey:

> 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… 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)-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+l)-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 cakes or onions. The OSI model breaks the responsibilities of the network into seven distinct layers, each with different functions to aid in transmitting information from one system to another, as shown in Figure 1-2. The layers encapsulate information from the layer below it; these layers are Application, Presentation, Session, Transport, Network, Data Link, and Physical. Over the next few pages, we will go over each layer's functionality and how it sends data between two systems.

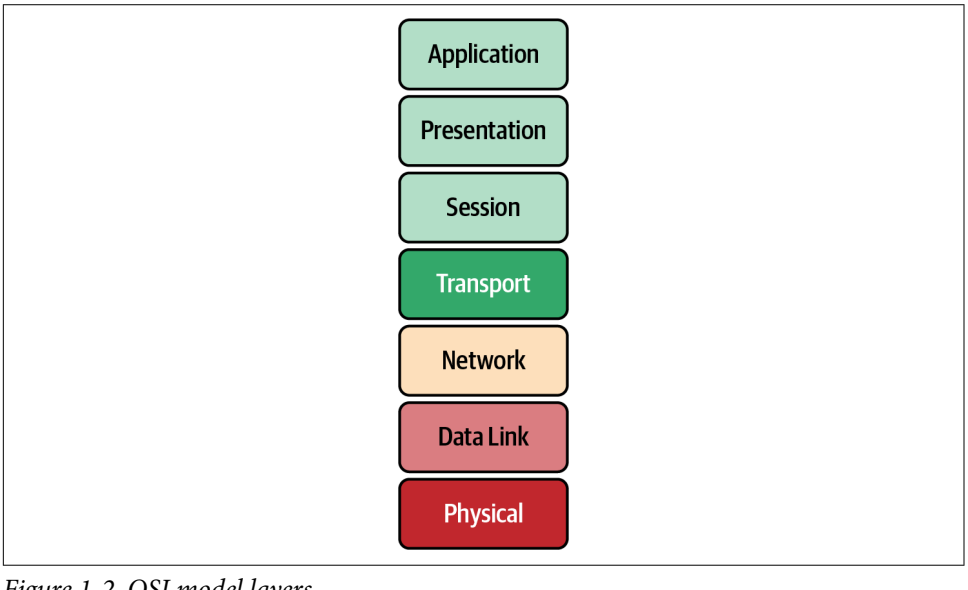

Figure 1-2. OSI model layers

Each layer takes data from the previous layer and encapsulates it to make its Protocol Data Unit (PDU). The PDU is used to describe the data at each layer. PDUs are also part of TCP/IP. The applications of the Session layer are considered "data" for the PDU, preparing the application information for communication. Transport uses ports to distinguish what process on the local system is responsible for the data. The

Network layer PDU is the packet. Packets are distinct pieces of data routed between networks. The Data Link layer is the frame or segment. Each packet is broken up into frames, checked for errors, and sent out on the local network. The Physical layer transmits the frame in bits over the medium. Next we will outline each layer in detail:

Application
: The Application layer is the top layer of the OSI model and is the one the end user interacts with every day. This layer is not where actual applications live, but it provides the interface for applications that use it like a web browser or Office 365. The single biggest interface 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 we'll save for "TLS" on page 25.

Session
: The Session layer is responsible for the duplex of the connection, in other words, 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
: The Transport layer transfers 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 desegmentation, 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 successful data transmission and sends the next data if no errors occurred. TCP/IP has two protocols at this layer: TCP and User Datagram Protocol (UDP).

Network
: The Network layer implements a means of transferring variable-length data flows from a host on one network to a host on another 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 "TCP/IP" on page 8.

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 transfers data between network hosts and provides the means to detect and possibly correct errors from the Physical layer. Data Link frames, the PDU for layer 2, do not cross the boundaries of a local network.

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

There are many mnemonics to remember the layers of the OSI model; our favorite is All People Seem To Need Data Processing.

Table 1-2 summarizes the OSI layers.

Table 1-2. OSI layer details

Layer number	Layer name	Protocol data unit	Function overview
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	Physical	Bit	Sending and receiving of bitstreams over the medium.

The OSI model breaks down all the necessary functions to send a data packet over a network between two hosts. In the late 1980s and early 1990s, it lost out to TCP/IP as the standard adopted by the DOD and all other major players in networking. The standard defined in ISO 7498 gives a brief glimpse into the implementation details that were considered by most at the time to be complicated, inefficient, and to an extent unimplementable. The OSI model at a high level still allows those learning networking to comprehend the basic concepts and challenges in networking. In addition, these terms and functions are used in the TCP/IP model covered in the next section and ultimately in Kubernetes abstractions. Kubernetes services break out each function depending on the layer it is operating at, for example, a layer 3 IP address or a layer 4 port; you will learn more about that in Chapter 4. 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, Linux, or another OS, TCP/IP allows them to communicate; TCP/IP does not care if you are running Apache or Nginx for your web server at the Application layer. The separation of responsibilities similar to the OSI model makes that possible. In Figure 1-3, we compare the OSI model to TCP/IP.

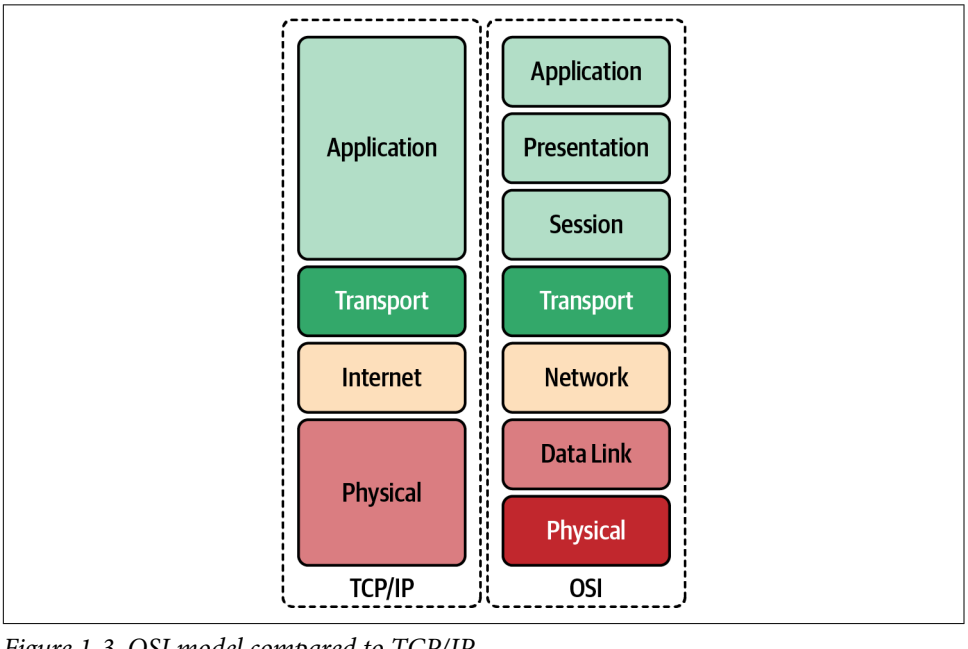

Figure 1-3. OSI model compared to TCP/IP

Here we expand on the differences between the OSI model and the 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 the host-to-host data transfer. The lower Transport layer also manages the data exchange in network communications. Applications at this layer are defined in RFCs; in this book, we will continue to use HTTP, RFC 7231 as our example for the Application layer.

Transport
: TCP and UDP are the 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 and is considered unreliable; you'll learn more about that in "UDP" on page 28. Each port identifies the host process responsible for processing the information from the network communication. HTTP uses the well-known port 80 for nonsecure communication and 443 for secure communication. Each port on the server identifies its traffic, and 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. For an outgoing packet, it selects the next-hop host and transmits it to that host by passing it to the appropriate link-layer. Once the packet is received by the destination, the Internet layer will pass the packet payload up to the appropriate Transport layer protocol.

  IP provides the 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. 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.

Link
: The Link layer in the TCP/IP model comprises networking protocols that operate only on the local network that a host connects to. Packets are not routed to nonlocal networks; that is the Internet layer's 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 9 (ARP), data sent off the local network is processed by the Internet layer. This layer also includes protocols for moving packets between two Internet layer hosts.

Physical layer
: The Physical layer defines the components of the hardware to use for the network. For example, the Physical network layer stipulates the physical characteristics of the communications media. The Physical layer of TCP/IP details hardware standards such as IEEE 802.3, the specification for Ethernet network media. Several interpretations of RFC 1122 for the Physical layer are included with the other layers; we have added this for completeness.

Throughout this book, we will use the minimal Golang web server (also called Go) from Example 1-1 to show various levels of networking components from `tcpdump`, a Linux syscall, to show how Kubernetes abstracts the syscalls. This section will use it to demonstrate what is happening at the Application, Transport, Network, and Data Link layers.

Application

As mentioned, Application is the highest layer in the TCP/IP stack; it is where the user interacts with data before it gets sent over the network. In our example walkthrough, we are going to use Hypertext 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 (HTML) 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, and tweets all use HTTP. A client will make an HTTP request to our minimal Golang web server from Example 1-1, and it will send an HTTP response with "Hello" text. The web server runs locally in an Ubuntu virtual machine to test the full TCP/IP stack.

See the example code repository for full instructions.

Example 1-1. 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)
}
```

In our Ubuntu virtual machine we need to start our minimal web server, or if you have Golang installed locally, you can just run this:

```
go run web-server.go
```

Let's 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 we're using cURL to simplify and show the command line.

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.

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.

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: */* ❻
>
< HTTP/1.1 200 OK ❼
```

```
< Date: Sat, 25 Jul 2020 14:57:46 GMT ❽
< Content-Length: 5 ❾
< Content-Type: text/plain; charset=utf-8 ❿
<
* Connection #0 to host localhost left intact
Hello* Closing connection 0 ⓫
```

❶ `curl localhost:8080 -vvv`: This is the `curl` command that opens a connection to the locally running web server, `localhost` on TCP port 8080. `-vvv` sets the verbosity of the output so we can see everything happening with the request. Also, `TCP_NODELAY` instructs the TCP connection to send the data without delay, one of many options available to the client to set.

❷ `Connected to localhost (::1) port 8080`: It worked! cURL connected to the web server on localhost and over port 8080.

❸ `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" response. The forward slash is the next part, a Uniform Resource Locator (URL), which 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.

❹ `Host: localhost:8080`: HTTP has several options for sending information about the request. In our request, the cURL process has set 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 (:) and then its value.

❺ `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.

❻ `Accept: */*`: This header instructs the web server what content types the client understands. Table 1-3 shows examples of common content types that can be sent.

❼ `HTTP/1.1 200 OK`: This is the server response to our request. The server responds with the HTTP version and the response status code. There are several possible responses from the server. A status code of 200 indicates the response was successful. 1XX means informational, 2XX means successful, 3XX means redirects, 4XX responses indicate there are issues with the requests, and 5XX generally refers to issues from the server.

❽ `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.

❾ `Content-Length: 5`: The `Content-Length` header indicates the size of the message body, in bytes, sent to the recipient; in our case, the message is 5 bytes.

❿ `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 a plain-text file that is UTF-8 encoded.

⓫ `Hello* Closing connection 0`: This prints out the response from our web server and closes out the HTTP connection.

Table 1-3. Common content types for HTTP data

Type	Description
application	Any kind of binary data that doesn't fall explicitly into one of the other types. Common examples include application/json, application/pdf, application/pkcs8, and application/zip.
audio	Audio or music data. Examples include audio/mpeg and audio/vorbis.
font	Font/typeface data. Common examples include font/woff, font/ttf, and font/otf.
image	Image or graphical data including both bitmap and vector such as animated GIF or APNG. Common examples are image/jpg, image/png, and image/svg+xml.
model	Model data for a 3D object or scene. Examples include model/3mf and model/vrml.
text	Text-only data including human-readable content, source code, or text data. Examples include text/plain, text/csv, and text/html.
video	Video data or files, such as video/mp4.

This is a simplistic view that happens with every HTTP request. Today, a single web page makes an exorbitant number of requests with one load of a page, and in just a matter of seconds! This is a brief example for cluster administrators of how HTTP (and for that matter, the other seven layers' applications) operate. We will continue to build our knowledge of how this request is completed at each layer of the TCP/IP stack and then how Kubernetes completes those same requests. All this data is formatted and options are set at layer 7, but the real heavy lifting is done at the lower layers of the TCP/IP stack, which we will go over in the next sections.

Transport

The Transport layer 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 web server is a layer 7 application using HTTP; the Transport layer that HTTP relies on is TCP.

TCP

As already mentioned, TCP is a connection-oriented, reliable protocol, and it provides flow control and multiplexing. TCP is considered connection-oriented because it manages the connection state through the life cycle of the connection. In TCP, the window size manages flow control, unlike UDP, which does not manage the congestion flow. In addition, UDP is unreliable, and data may arrive out of sequence. Each port identifies the host process responsible for processing the information from the network communication. TCP is known as a host-to-host layer protocol. 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 nonsecure communication and 443 for secure communication using Transport Layer Security (TLS). Clients requesting a new connection create a source port local in the range of 0–65534.

To understand how TCP performs multiplexing, let's review a simple HTML page retrieval:

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.

Let's walk through how TCP manages multiplexing with the information provided in the TCP segment headers:

`Source port` *(16 bits)*
: This identifies the sending port.

`Destination port` *(16 bits)*
: This 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 is this sequence number plus 1. It is also used to reassemble data if it arrives out of order.

`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)*
: This specifies the size of the TCP header in 32-bit words.

`Reserved` *(3 bits)*
: This is for future use and should be set to zero.

`Flags` *(9 bits)*
: There are nine 1-bit fields defined for the TCP header:

  - NS–ECN-nonce: Concealment protection.
  - CWR: Congestion Window Reduced; the sender reduced its sending rate.
  - ECE: ECN Echo; the sender received an earlier congestion notification.
  - URG: Urgent; the Urgent Pointer field is valid, but this is rarely used.
  - ACK: Acknowledgment; the Acknowledgment Number field is valid and is always on after a connection is established.
  - PSH: Push; the receiver should pass this data to the application as soon as possible.
  - RST: Reset the connection or connection abort, usually because of an error.
  - SYN: Synchronize sequence numbers to initiate a connection.
  - FIN: The sender of the segment is finished sending data to its peer.

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 ECN improving robustness against malicious or accidental concealment of marked packets.

`Window size` *(16 bits)*
: This is the size of the receive window.

`Checksum` *(16 bits)*
: The checksum field is used for error checking of the TCP header.

`Urgent pointer` *(16 bits)*
: This is 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`
: This is the piece of application data being sent in this segment.

In Figure 1-4, we can see all the TCP segment headers that provide metadata about the TCP streams.

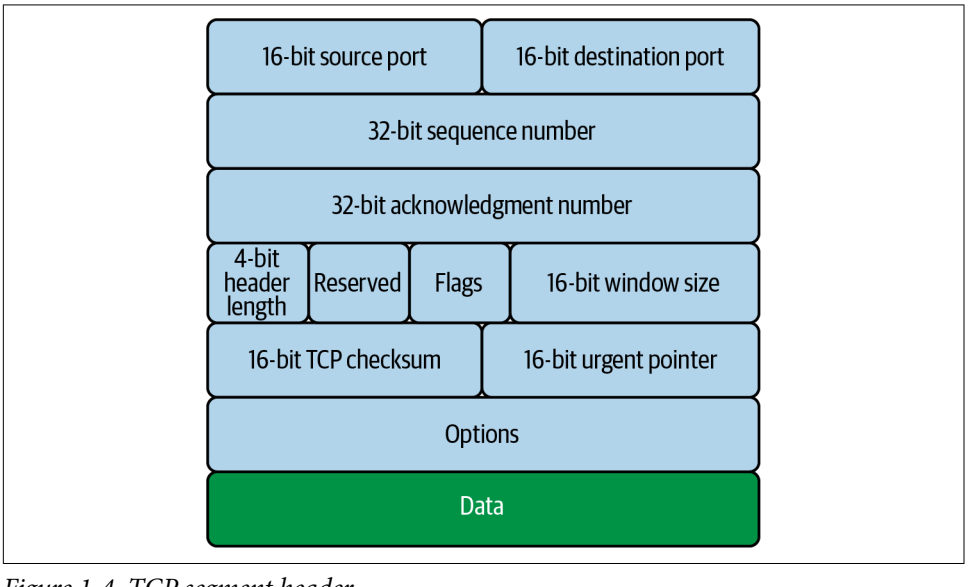

Figure 1-4. TCP segment header

These fields 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 layers 1 and 2, to get data to the destination host.

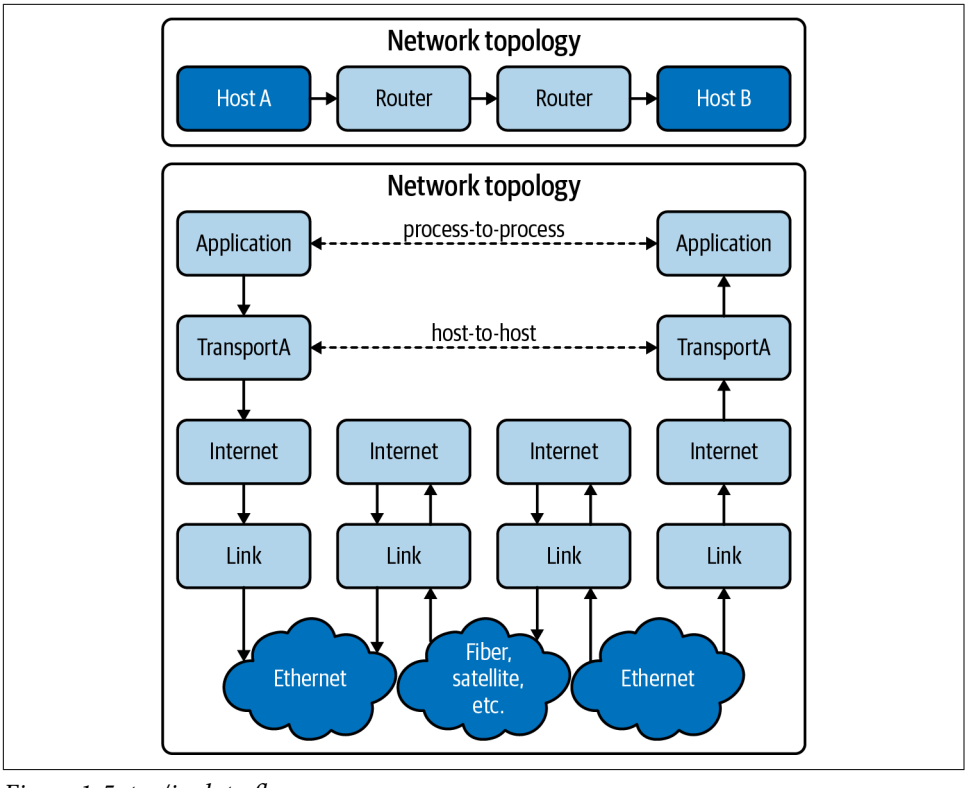

Figure 1-5. tcp/ip data flow

In the next section, we will show how TCP uses these fields to initiate a connection through the three-way handshake.

TCP handshake

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

1. The requesting node sends a connection request via a SYN packet to get the transmission started.
2. If the receiving node is listening on the port the sender requests, the receiving node replies with a SYN-ACK, acknowledging that it has heard the requesting node.
3. The requesting node returns an ACK packet, exchanging information and letting them know the nodes are good to send each other information.

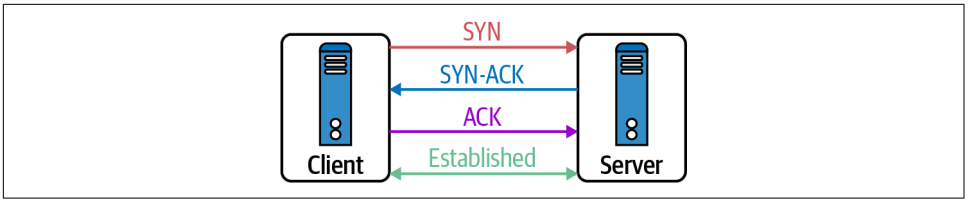

Figure 1-6. TCP three-way handshake

Now the connection is established. Data can be transmitted over the physical medium, routed between networks, to 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 2, we will discuss how a Linux client and server handle sockets.

TCP is a stateful protocol, tracking the connection's state throughout its life cycle. The state of the connection depends on both the sender and the receiver agreeing where they are in the connection flow. The connection state 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.

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 host

`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

`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

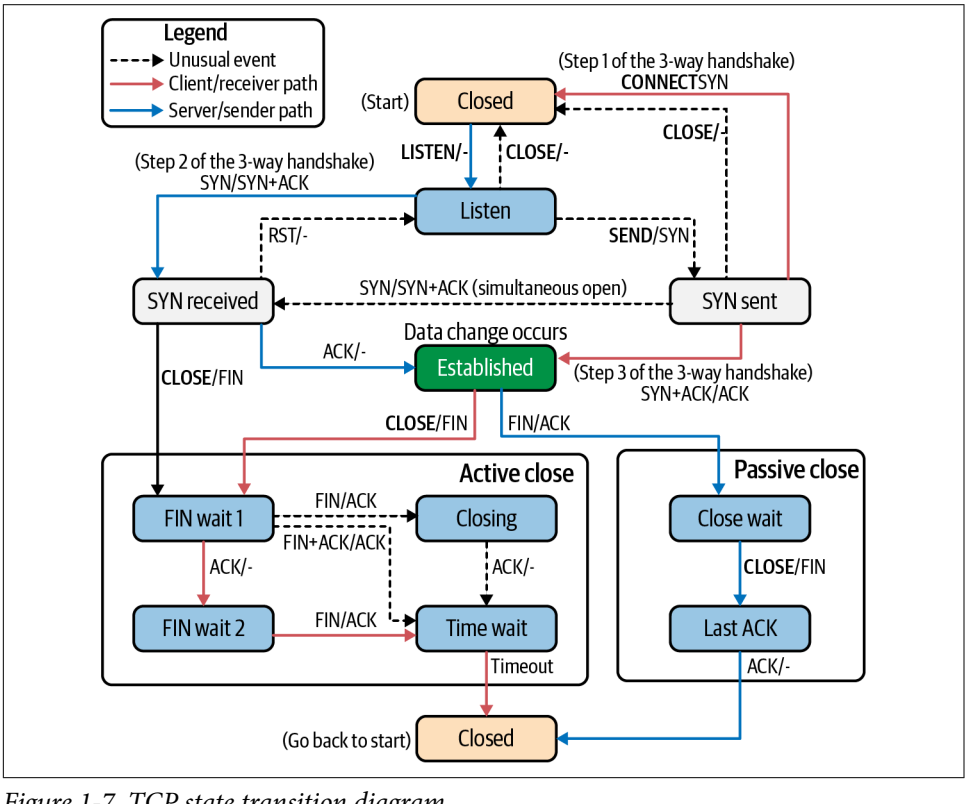

Figure 1-7. TCP state transition diagram

Example 1-3 is a sample of a Mac's TCP connections, their state, and the 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's review the HTTP request for our web server at the Transport layer using TCP. To accomplish this, we use a command-line tool called `tcpdump`.

tcpdump

> `tcpdump` prints out a description of the contents of packets on a network interface that matches the boolean expression.
>
> —tcpdump man page

`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 web server 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 network interface card (NIC) driver and layer 2.

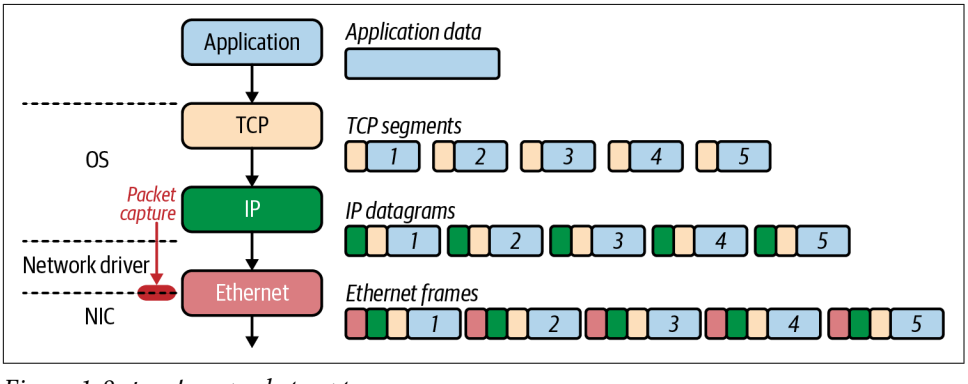

Figure 1-8. `tcpdump` packet capture

A loopback interface is a logical, virtual interface on a device. A loopback interface is not a physical interface like Ethernet interface. Loopback interfaces are always up and running and always available, even if other interfaces are down on the host.

The general format of a `tcpdump` output will contain the following fields: `tos`,`TTL`, `id`, `offset`, `flags`, `proto`, `length`, and `options`. Let's review these:

`tos`
: The type of service field.

`TTL`
: The time to live; it is not reported if it is zero.

`id`
: The IP identification field.

`offset`
: The fragment offset field; it is printed whether this is part of a fragmented datagram or not.

`flags`
: The DF, Don't Fragment, flag, which indicates that the packet cannot be fragmented for transmission. When unset, it indicates that the packet can be fragmented. The MF, More Fragments, flag indicates there are packets that contain more fragments and when unset, it indicates that no more fragments remain.

`proto`
: The protocol ID field.

`length`
: The total length field.

`options`
: The IP options.

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

To produce the output for Example 1-4, open another terminal and start a `tcpdump` trace on the loopback for only TCP and port 8080; otherwise, you will see a lot of other packets not relevant to our example. You'll need to use escalated privileges to trace packets, so that means using `sudo` in this case.

Example 1-4. `tcpdump`

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

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

❸ This is the first packet in the TCP handshake. We can tell it's the SYN because the flags bit is set with `[S]`, and the sequence number is set to `2784345138` by cURL, with the localhost process number being `50399`.

❹ The SYN-ACK packet is the the one filtered by `tcpdump` from the `localhost.http-alt` process, the Golang web server. The flag is to `[S.]`, so it is a 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 the ACK 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 is 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 a 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. The data push flag is set, `[P.]`, and it acknowledges the previous packet with an 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 the ACK flag set, sets the acknowledgment number to 122, and sets the sequence number to 79.

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

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

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

⓮ This is the final ACK from the client, with acknowledgment number 123. The connection is closed now.

⓯ `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 excellent troubleshooting application for network engineers as well as cluster administrators. Being able to verify connectivity at many levels in the cluster and the network are valuable skills to have. You will see in Chapter 6 how useful `tcpdump` can be.

Our example was a simple HTTP application using TCP. All of this data was sent over the network in plain text. While this example was a simple Hello World, other requests like our bank logins need to have some security. The Transport layer does not offer any security protection for data transiting the network. TLS adds additional security on top of TCP. Let's dive into that in our next section.

TLS

TLS adds encryption to TCP. TLS is an add-on to the TCP/IP suite and is not considered to be part of the base operation for TCP. HTTP transactions can be completed without TLS but are not secure from eavesdroppers 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 handshake to establish encryption capabilities and exchange keys for encryption. The following steps detail the TLS handshake between the client and the server, which can also be seen in Figure 1-9:

1. ClientHello: This 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: This contains the server's certificate and 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 premaster 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 premaster secret and exchange 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.

Kubernetes applications and components will manage TLS for developers, so a basic introduction is required; Chapter 5 reviews more about TLS and Kubernetes.

As demonstrated with our web server, cURL, and `tcpdump`, TCP 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, the overhead that comes with UDP offers an alternative. Now that we understand how TCP works as a reliable connection-oriented protocol, let's review how UDP differs from TCP.

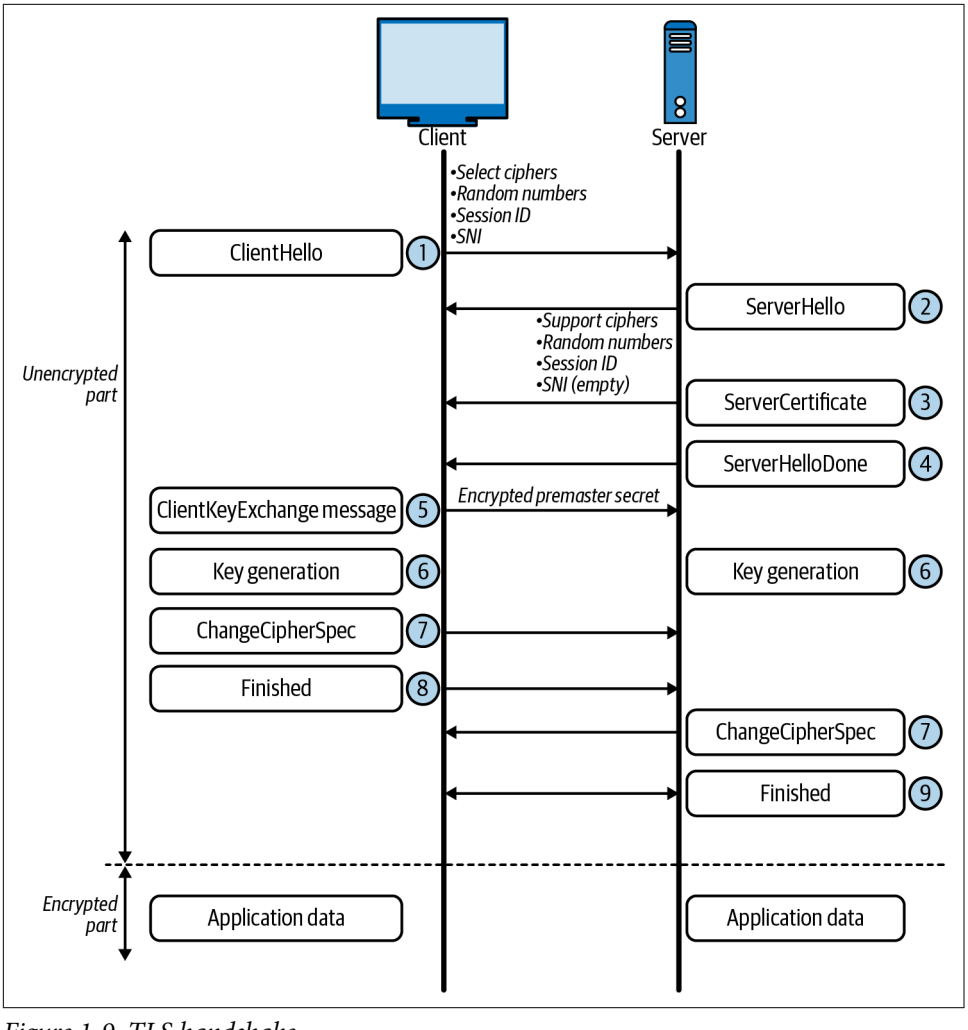

Figure 1-9. TLS handshake

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 the Domain Name System (DNS) and Simple Network Management Protocol (SNMP). UDP slices a request into datagrams, making it capable for use with other protocols for tunneling like a virtual private network (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 makes it an apt choice for streaming video.

Let's look at the small number of headers required in a UDP datagram (see Figure 1-10):

`Source port number` *(2 bytes)*
: 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` *(2 bytes)*
: Identifies the receiver's port and is required.

`Length` *(2 bytes)*
: Specifies the length in bytes of the UDP header and UDP data. The minimum length is 8 bytes, the length of the header.

`Checksum` *(2 bytes)*
: Used for error checking of the header and data. It is optional in IPv4, but mandatory in IPv6, and is all zeros if unused.

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

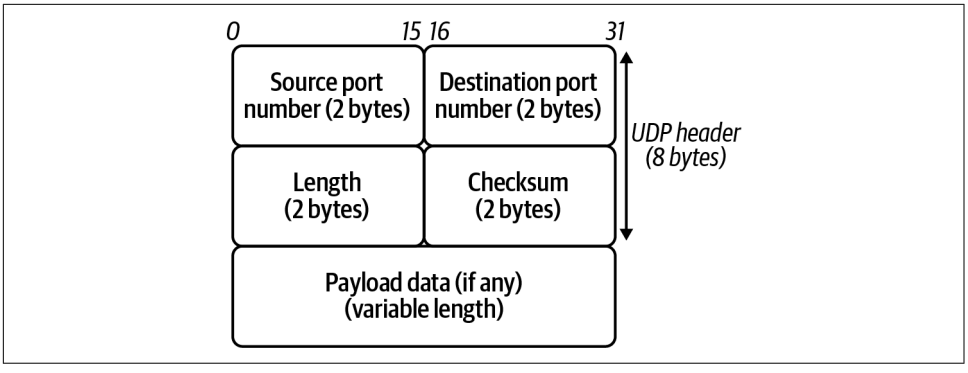

Figure 1-10. UDP header

The next layer in the TCP/IP stack is the Internetworking layer—these are packets that can get sent across the globe on the vast networks that make up the internet. Let's review how that gets completed.

Network

All TCP and UDP data gets 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 the data to that host by passing it 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 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. Source and destination IP addresses identify packets on the network, which we'll address next.

Internet Protocol

This almighty packet is defined in RFC 791 and is used for sending data across networks. Figure 1-11 shows the IPv4 header format.

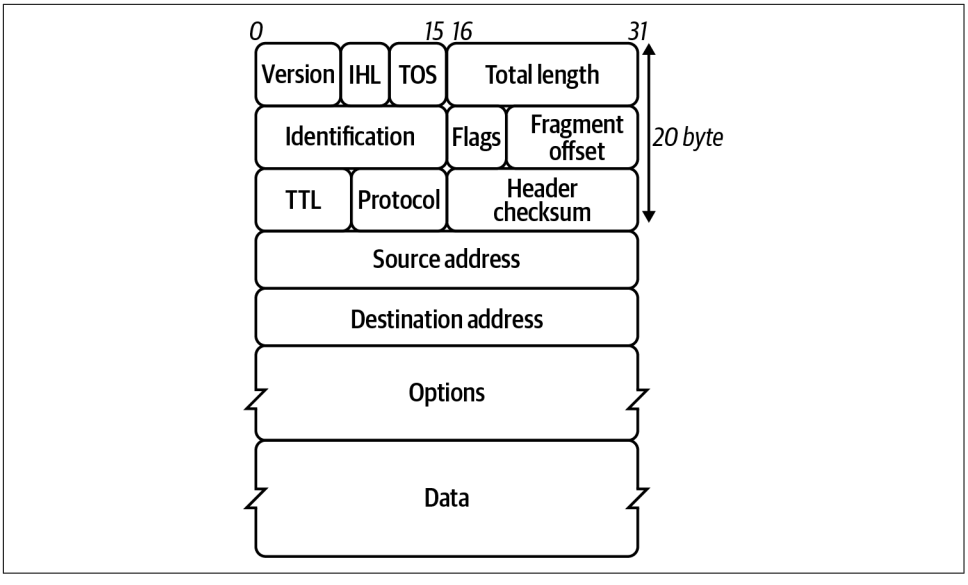

Figure 1-11. IPv4 header format

Let's look at the header fields in more detail:

`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 option.

`Type of Service`
: Originally defined as the type of service (ToS), now Differentiated Services Code Point (DSCP), this field specifies differentiated services. DSC Pallows 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`
: This is the entire packet size in bytes.

`Identification`
: This is the identification field and is used for uniquely identifying the group of fragments of a single IP datagram.

`Flags`
: This is used to control or identify fragments. In order from most significant to least:

  - bit 0: Reserved, set to zero
  - bit 1: Do not Fragment
  - bit 2: More Fragments

`Fragment Offset`
: This 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 8-bit time to live field helps prevent datagrams from going in circles on a network.

`Protocol`
: This is 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 1-4.

Table 1-4. 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)*
: The 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.

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`
: This is the IPv4 address of the sender of the packet.

The source address may be changed in transit by a network address translation device; NAT will be discussed later in this chapter and extensively in Chapter 3.

`Destination address`
: This is the 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, and 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 "Getting round the network" on page 35). Understanding how to identify an IP address is critical for an engineer. First, we will review IPv4 and then understand the drastic changes in IPv6.

IPv4 addresses are in the dotted-decimal notation for us humans; computers read them out as binary strings. Figure 1-12 details the dotted-decimal notation and binary. Each section is 8 bits in length, with 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.

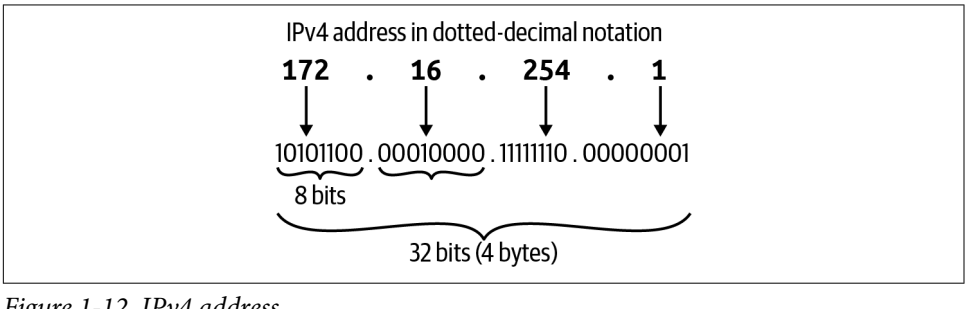

Figure 1-12. IPv4 address

In Example 1-5, we have the output of a computer's IP address for its network interface card and we can see its IPv4 address is `192.168.1.2`. The IP address also has a subnet mask or netmask associated with it 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 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 had 65,536; and Class C had 256. Each class had a host address, the first one in its boundary, and the last one was designated as the broadcast address. Figure 1-13 demonstrates this for us.

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.

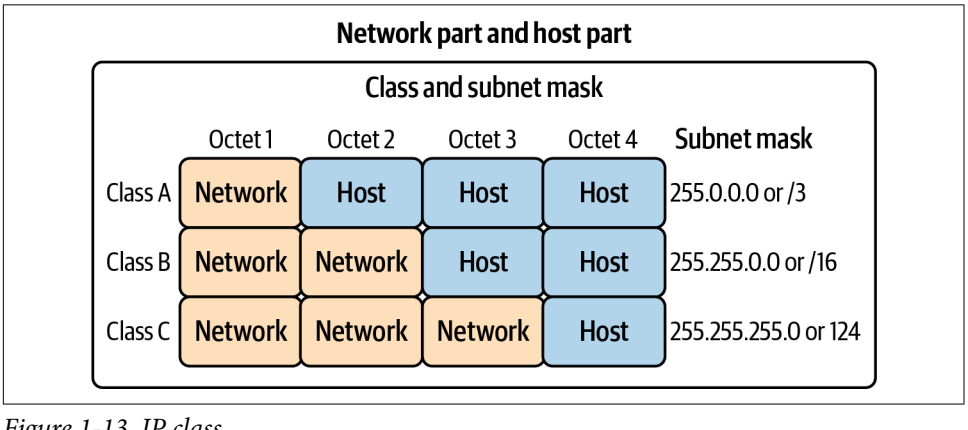

Figure 1-13. IP class

Classful addressing was not scalable on the internet, so to help alleviate that scale issue, we began breaking up the class boundaries using Classless Inter-Domain Routing (CIDR) ranges. Instead of having the full 16 million-plus addresses in a class address range, an internet entity gives only a subsection of that range. This effectively allows network engineers to move the subnet boundary to anywhere inside the class

range, giving them more flexibility with CIDR ranges, and helping to scale IP address ranges.

In Figure 1-14, 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 ARIN from IANA. ARIN 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`.

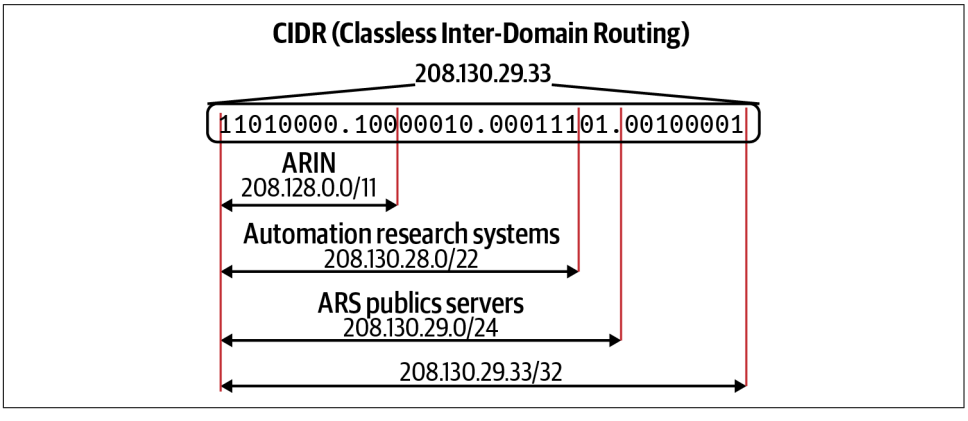

Figure 1-14. CIDR example

The global coordination of the DNS root, IP addressing, and other internet protocol resources is performed by IANA.

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

In Figure 1-15, we can see that IPv6, unlike IPv4, uses hexadecimal to shorten addresses for writing purposes. It has similar characteristics to IPv4 in that it has a host and network prefix.

The most significant difference between IPv4 and IPv6 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.

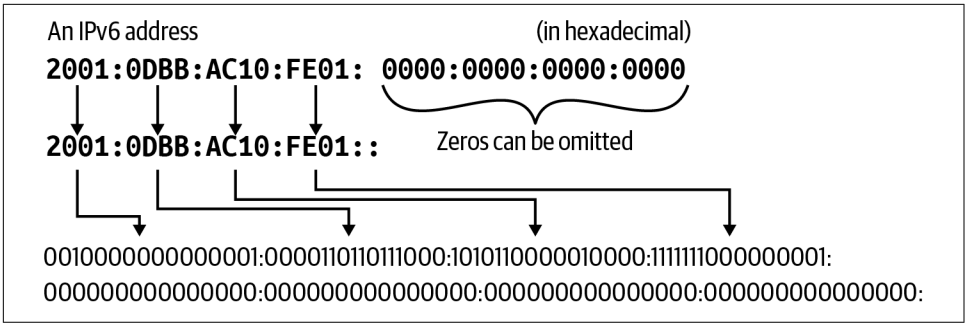

Figure 1-15. IPv6 address

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, and data is ready to be sent, but how do our packets get from our host on our network to the intended hosted on another network halfway around the world? That is the job of routing. There are several routing protocols, but the internet relies on Border Gateway Protocol (BGP), 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 autonomous system number (ASN) to designate a single administrative entity or corporation that represents a common and clearly defined routing policy on the internet. BGP and ASNs allows network administrators to maintain control of their internal network routing while announcing and summarizing their routes on the internet. Table 1-5 lists the available ASNs managed by IANA and other regional entities.[1]

1 "Autonomous System (AS) Numbers". IANA.org. 2018-12-07. Retrieved 2018-12-31.

Table 1-5. ASNs available

Number	Bits	Description	Reference
0	16	Reserved	RFC 1930, RFC 7607
1–23455	16	Public ASNs	
23456	16	Reserved for AS Pool Transition	RFC 6793
23457–64495	16	Public ASNs	
64496–64511	16	Reserved for use in documentation/sample code	RFC 5398
64512–65534	16	Reserved for private use	RFC 1930, RFC 6996
65535	16	Reserved	RFC 7300
65536–65551	32	Reserved for use in documentation and sample code	RFC 4893, RFC 5398
65552–131071	32	Reserved	
131072–4199999999	32	Public 32-bit ASNs	
4200000000–4294967294	32	Reserved for private use	RFC 6996
4294967295	32	Reserved	RFC 7300

In Figure 1-16 ,we have five ASNs, 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.

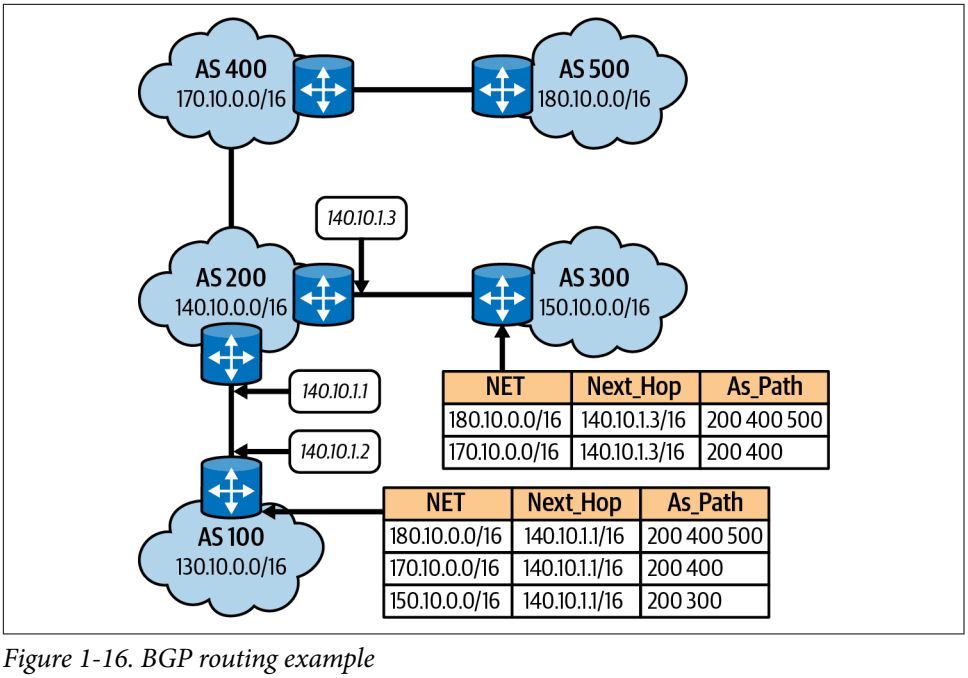

Figure 1-16. BGP routing example

Based on the routing table in 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 until the local router for `150.10.2.300` on AS 300 receives that packet. The flow here is described in Figure 1-6, which shows the TCP/IP data flow between networks. A basic understanding of BGP is needed because some container networking projects, like Calico, use it for routing between nodes; you'll learn more about this in Chapter 3.

```
o → 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 1-17. Local routing table

Figure 1-17 displays a local route table. In the route table, we can see the interface that a packet will be sent out is based on the destination IP address. For example, a packet destined for `192.168.1.153` will be sent out the `link#11` gateway, which is local to the network, and no routing is needed. `192.168.1.254` is the router on the network attached to our internet connection. If the destination network is unknown, it is sent out the default route.

Like all Linux and BSD OSs, you can find more information on `netstat`'s man page (`man netstat`). Apple's `netstat` is derived from the BSD version. More information can be found in the FreeBSD Handbook.

Routers continuously communicate on the internet, exchanging route information and informing each other of changes on their respective networks. BGP takes care of a lot of that data exchange, but network engineers and system administrators can use

the ICMP protocol and `ping` command line 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 `ping` for testing connections, and it is useful in testing container connectivity as well. You'll learn more about this in Chapters 2 and 3 as we deploy our minimal Golang web server into a container and a pod.

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 ICMP packets; they are reviewed here and shown in Figure 1-18:

`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` *(4-byte 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.

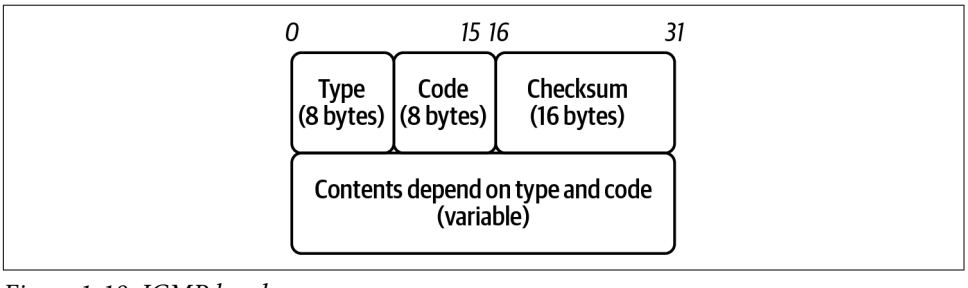

Figure 1-18. ICMP header

Some consider ICMP a Transport layer protocol since it does not use TCP or UDP. 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. ICMP is IP protocol 1, while TCP is 6, and UDP is 17.

The value identifies control messages in the `Type` field. The `code` field gives additional context information for the message. You can find some standard ICMP type numbers in Table 1-6.

Table 1-6. Common ICMP type numbers

Number	Name	Reference
0	Echo reply	RFC 792
3	Destination unreachable	RFC 792
5	Redirect	RFC 792
8	Echo	RFC 792

Now that our packets know which 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 to send the data across the wire. The Link layer of the TCP/IP stack comprises two sublayers: the Media Access Control (MAC) sublayer and the Logical Link Control (LLC) sublayer. Together, they perform OSI layers 1 and 2, Data Link and Physical. 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 layer has the privilege of managing flow control and multiplexing protocols over the MAC layer to transmit and demultiplexing when receiving, as shown in Figure 1-19. IEEE standard 802.3, Ethernet, 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).

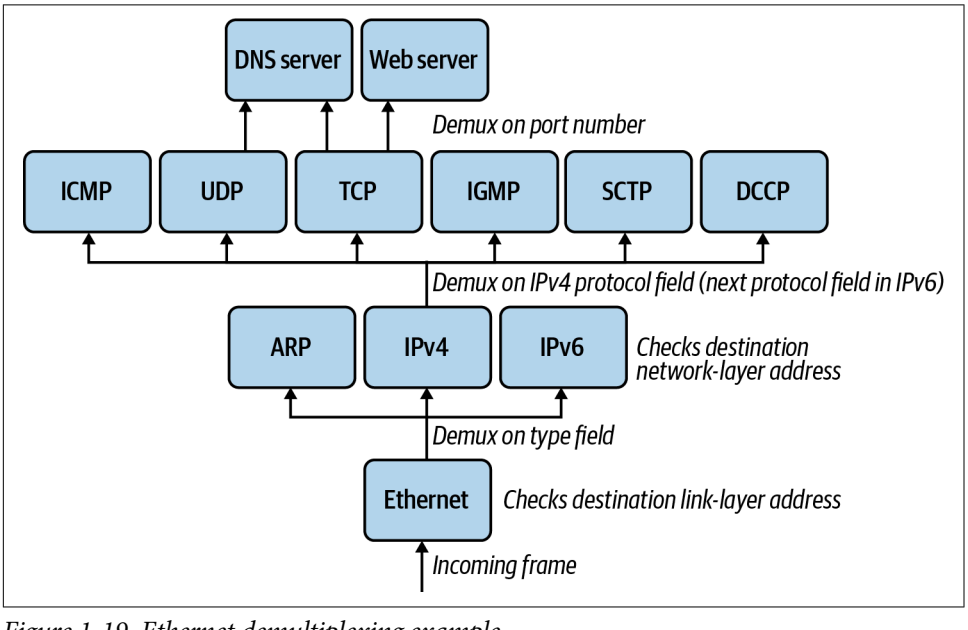

Figure 1-19. Ethernet demultiplexing example

As with the other PDUs, Ethernet has a header and footers, as shown in Figure 1-20.

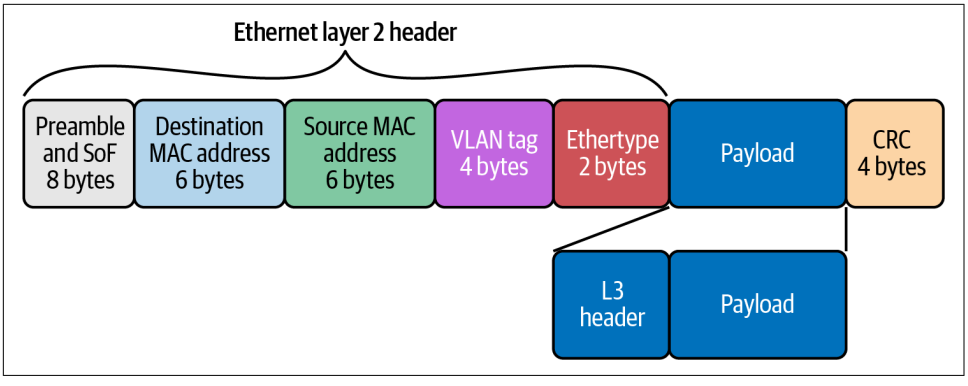

Figure 1-20. Ethernet header and footer

Let's review these in detail:

`Preamble` *(8 bytes)*
: Alternating string of ones and zeros indicate to the receiving host that a frame is incoming.

`Destination MAC Address` *(6 bytes)*
: MAC destination address; the Ethernet frame recipient.

`Source MAC Address` *(6 bytes)*
: MAC source address; the Ethernet frame source.

`VLAN tag` *(4 bytes)*
: Optional 802.1Q tag to differentiate traffic on the network segments.

`Ether-type` *(2 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) that 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.

Figure 1-21 shows 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 NIC-specific parts.

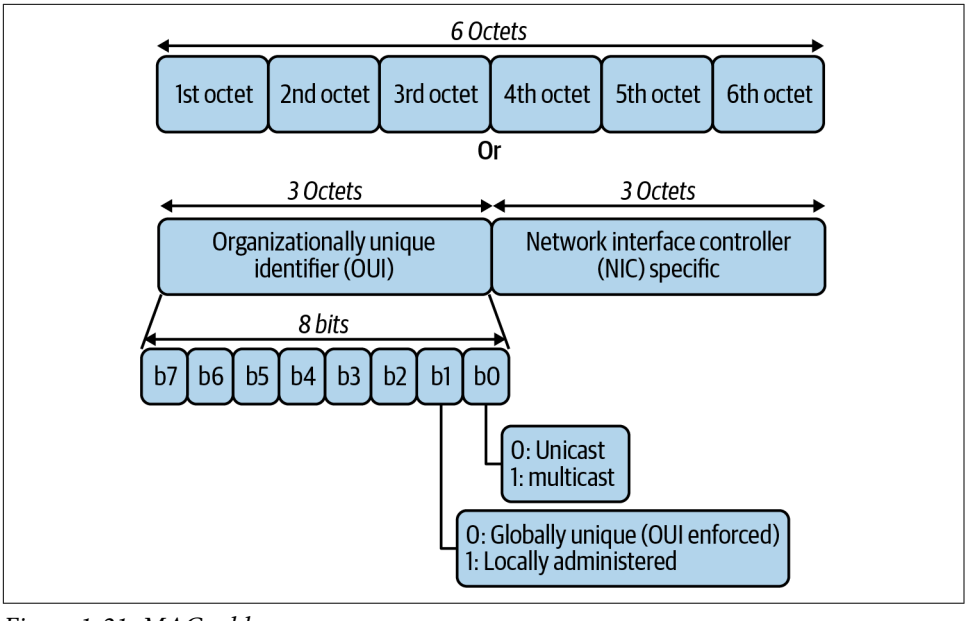

Figure 1-21. MAC address

The frame indicates to the recipient of the Network layer packet type. Table 1-7 details the common protocols handled. In Kubernetes, we are mostly interested in IPv4 and ARP packets. IPv6 has recently been introduced to Kubernetes in the 1.19 release.

Table 1-7. 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 addresses for this purpose.

Example 1-8. 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 shows the exchange between hosts on the local network. The browser makes an HTTP request for a website hosted by the target server. Through DNS, it determines that the server has the IP address `10.0.0.1`. To continue to send the HTTP request, it also requires the server's MAC address. First, the requesting computer consults a cached ARP table to look up `10.0.0.1` for any existing records of the server's MAC address. If the MAC address is found, it sends an Ethernet frame with the destination address of the server's MAC address, containing the IP packet addressed to `10.0.0.1` onto the link. If the cache did not produce a hit for `10.0.0.2`, the requesting computer must send a broadcast ARP request message with a destination MAC address of `FF:FF:FF:FF:FF:FF`, which is accepted by all hosts on the local network, requesting an answer for `10.0.0.1`. The server responds with an ARP response message containing its MAC and IP address. As part of answering the request, the server may insert an entry for requesting the computer's MAC address into its ARP table for future use. The requesting computer receives and caches the response information in its ARP table and can now send the HTTP packets.

This also brings up a crucial concept on the local networks, broadcast domains. All packets on the broadcast domain receive all the ARP messages from hosts. In addition, all frames are sent all nodes on the broadcast, and the host 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.

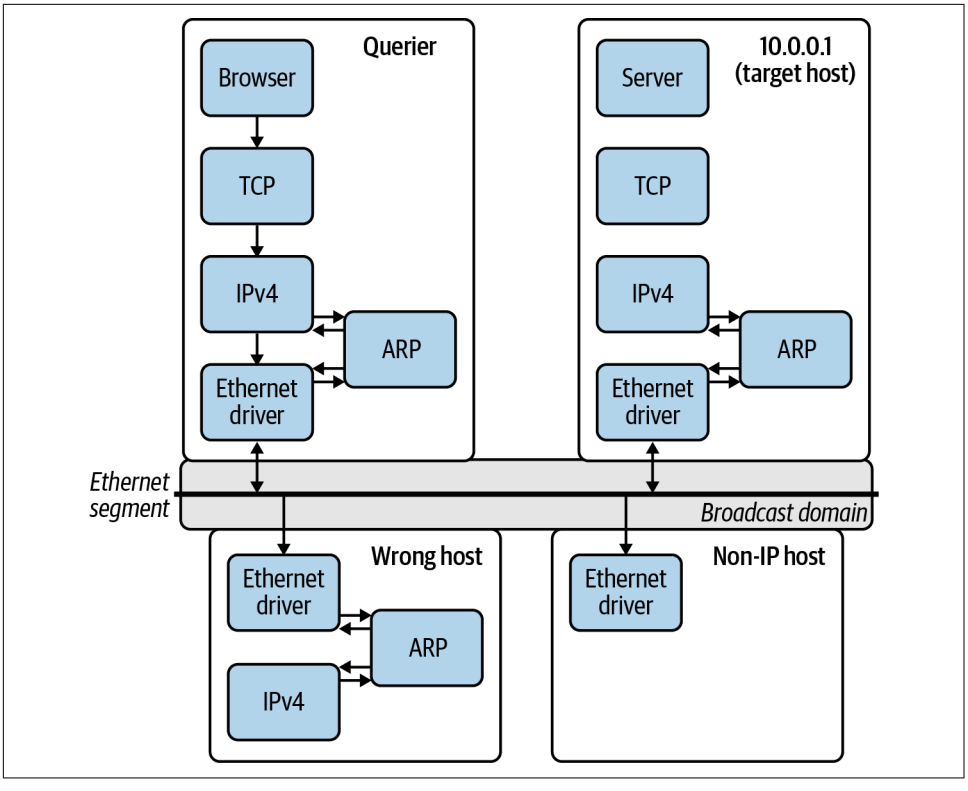

Figure 1-22. ARP request

We can use `tcpdump` to view all the ARP requests happening on the local network as in Example 1-9. The packet capture details the ARP packets; the Ethernet type used, `Ethernet (len 6)`; and the higher-level protocol, `IPv4`. It also includes who is requesting the MAC address of the IP address, `Request who-has 192.168.0.1 tell 192.168.0.12`.

Example 1-9. 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
```

To further segment the layer 2 network, network engineers can use virtual local area network (VLAN) 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 the VLAN technology known as a *virtual extensible LAN* (VXLAN).

Figure 1-23 shows how a VXLAN is an extension of a VLAN that allows network engineers to encapsulate layer 2 frames into layer 4 UDP packets. A VXLAN increases scalability up to 16 million logical networks and allows for layer 2 adjacency across IP networks. This technology is used in Kubernetes networks to produce overlay networks, which you'll learn more about in later chapters.

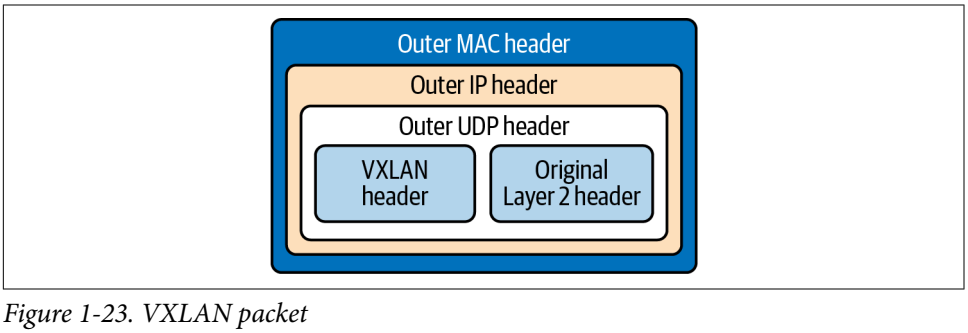

Figure 1-23. VXLAN packet

Ethernet also details the 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, such as the gamma-ray network that powers the Philotic Parallax Instantaneous Communicator.[2] 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 discussed previously, we have only touched 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 of TCP/IP is complete. Figure 1-24 outlines all the headers and footers each layer of the TCP/IP model produces to send data across the internet.

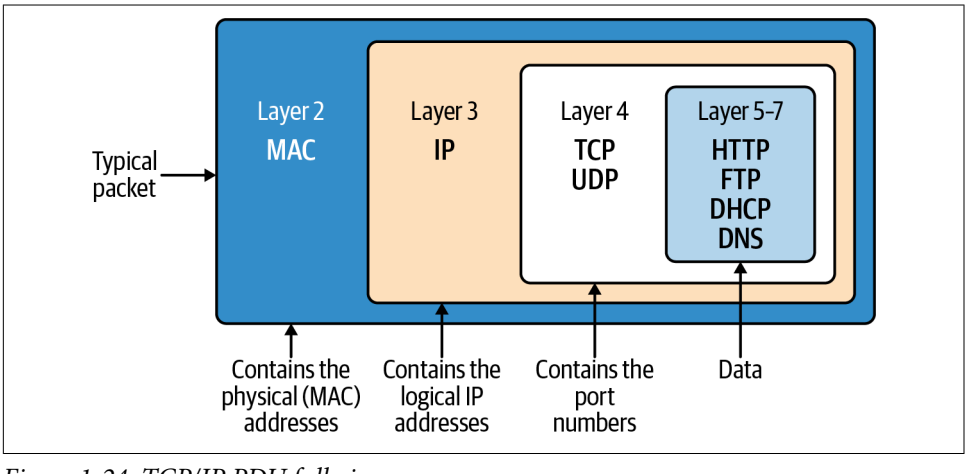

Figure 1-24. TCP/IP PDU full view

Let's review the journey and remind ourselves again what is going on now that we understand each layer in detail. Example 1-10 shows our web server again, and Example 1-11 shows the cURL request for it from earlier in the chapter.

2 In the movie *Ender's Game*, they use the Ansible network to communicate across the galaxy instantly. Philotic Parallax Instantaneous Communicator is the official name of the Ansible network.

Example 1-10. 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)
}
```

Example 1-11. Client request

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

We begin with the web server waiting for a connection in Example 1-10. cURL requests the HTTP server at `0.0.0.0` on port 8080. cURL determines the IP address and port number from the URL and proceeds to establish a TCP connection to the server. Once the connection is set up, via a TCP handshake, cURL sends the HTTP request. When the web server starts up, a socket of 8080 is created on the HTTP server, which matches TCP port 8080; the same is done on the cURL client side with a random port number. Next, this information is sent to the Network layer, where the source and destination IP addresses are attached to the packet's IP header. At the 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 web server.

When the web server receives the request, it creates packets of data that contain the HTTP response. The packets are sent back to the cURL process by routing them 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 Data Link layer, the MAC address is removed. At the Network Protocol layer, the IP address is verified and then removed from the packet. For this reason, if an application requires access to the client IP, it needs to be stored at the Application layer; the best example here is in HTTP requests and the X-Forwarded-For header. Now 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, 50 pages and 50 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 data center network. This is the type of scale that the Kubernetes networks' APIs help developers abstract away into simple YAML. Understanding the scale of the problem is our first in step in mastering the management of a Kubernetes network. By taking our simple example of the Golang web server and learning the first principles of networking, you can begin to wrangle the packets flowing into and out of your clusters.

So far, we have covered the following:

- History of networking
- OSI model
- TCP/IP

Throughout this chapter, we discussed many things related to networks but only those needed to learn about using the Kubernetes abstractions. There are several O'Reilly books about TCP/IP; *TCP/IP Network Administration* by Craig Hunt (O'Reilly) is a great in-depth read on all aspects of TCP.

We 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 will walk through how this is implemented for the client and server with Linux networking.

CHAPTER 2
Linux Networking

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 of 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.

This chapter introduces many Linux programs. Manual, or *man*, pages, accessible with `man <program>`, will provide more detail.

Basics

Let's revisit our Go web server, which we used in Chapter 1. This web server listens on port 8080 and returns "Hello" for HTTP requests to / (see Example 2-1).

Example 2-1. 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)
}
```

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 nonprivileged port, and use infrastructure redirects (load balancer forwarding, Kubernetes services, etc.) to forward an externally visible privileged port to a program listening on a nonprivileged port.

This way, an attacker exploiting a possible vulnerability in your service will not have overly broad permissions available to them.

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 - `8080` with any IP address in our case.

`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 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 this 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 Golang web server, 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 the system calls made by our server, there is a *lot* of output. Let's reduce it somewhat 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.

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 the 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
```

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 do 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) translates the request into packets and transmits those packets back to our cURL client.

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

1. Epoll returns and causes the program to resume.
2. The server sees a connection from `::ffff:10.0.0.57`, the client IP address in this example.
3. The server inspects the socket.
4. The server changes `KEEPALIVE` options: it turns `KEEPALIVE` on, and sets a 180-second interval between `KEEPALIVE` probes.

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 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; they are abstract interfaces provided by the host or hypervisor.

IP addresses are assigned to network interfaces. A typical interface may have one IPv4 address and one IPv6 address, but multiple addresses 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).

The *loopback interface* is a special interface for same-host communication. `127.0.0.1` is the standard IP address for the loopback interface. Packets sent to the loopback interface will not leave the host, and processes listening on `127.0.0.1` will be

accessible only to other processes on the same host. Note that making a process listen on `127.0.0.1` is not a security boundary. CVE-2020-8558 was a past Kubernetes vulnerability, in which `kube-proxy` rules allowed some remote systems to reach `127.0.0.1`. The loopback interface is commonly abbreviated as `lo`.

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

Let's look at a typical `ifconfig` output; see Example 2-2.

Example 2-2. Output from `ifconfig` on a machine with one pysical network interface (ens4), and the loopback 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, so the list would be much longer on a typical Kubernetes node. We'll cover container networking in more detail in Chapter 3.

The Bridge Interface

The bridge interface (shown in Figure 2-1) allows system administrators to create multiple layer 2 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.

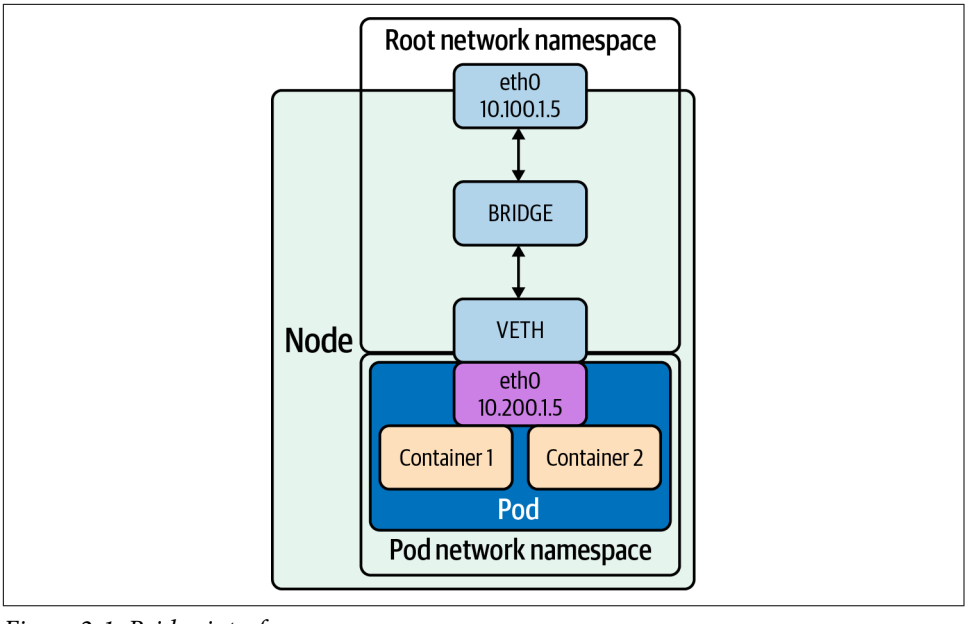

Figure 2-1. Bridge interface

You can read more about Linux bridging in the documentation.

In Example 2-3, we demonstrate how to create a bridge device named `br0` and attach a virtual Ethernet (veth) device, `veth`, and a physical device, `eth0`, using `ip`.

Example 2-3. Creating bridge interface and connecting veth pair

```
# # Add a new bridge interface named br0.
# ip link add br0 type bridge
# # Attach eth0 to our bridge.
# ip link set eth0 master br0
# # Attach veth to our bridge.
# 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` 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 device is a local Ethernet tunnel. Veth devices are created in pairs, as shown in Figure 2-1, where the pod sees an `eth0` interface from the veth. 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.

Example 2-5 shows 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 the steps to create two network namespaces (not to be confused with Kubernetes namespaces), `net1` and `net2`, and a pair of veth devices, with `veth1` assigned to namespace `net1` and `veth2` assigned 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 because 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, and 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.

netfilter.org contains some excellent documentation on the design and use of both Netfilter and `iptables`.

Netfilter has five hooks, shown in Table 2-1.

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 matches 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.

Netfilter triggers each hook during a specific phase of packet handling, and under specific conditions, we can visualize Netfilter hooks with a flow diagram, as shown in Figure 2-2.

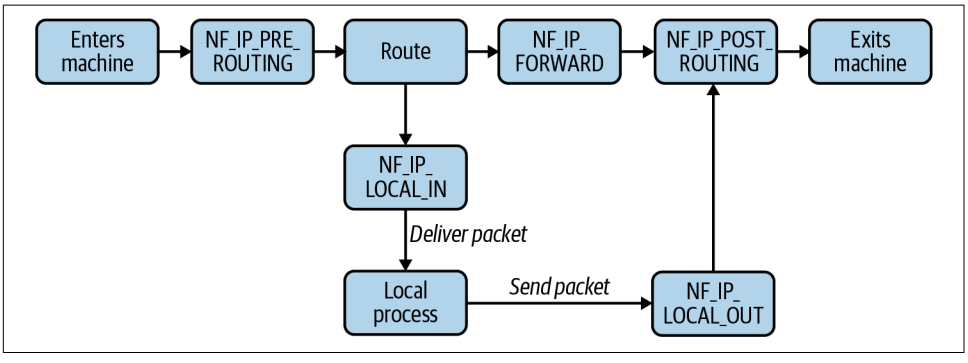

Figure 2-2. The possible flows of a packet through Netfilter hooks

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 and 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` and then the `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 when exposing an API or database to the host without strong authentication.

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 be accessible only locally. Among other things, this 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 API server, 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 on GitHub.

Table 2-2 shows the Netfilter hook order for various packet sources and destinations.

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` and then "leave" the network interface. They will "reenter" and be treated like packets from any other source.

Network address translation (NAT) 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 be performed only 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 a userspace program.

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

Repeat
: Make the packet "reenter" the hook and be reprocessed.

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 connection 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* remapped 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 five 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, the L4 protocol, is present because a program will bind to a port 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, shown in Figure 2-3, 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, and 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 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).

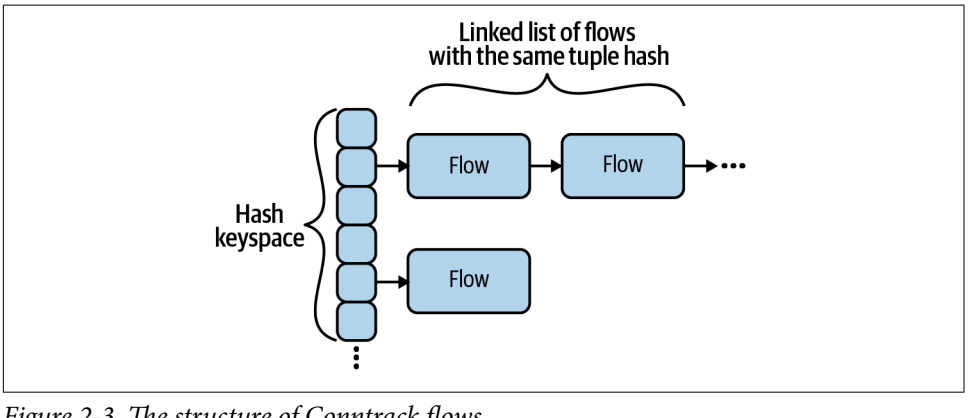

Figure 2-3. The structure of Conntrack flows

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

Conntrack entries contain a connection state, which is one of four 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 details the four 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.	An 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` command-line interface (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 here:

```
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 previous 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:

```
grep ESTABLISHED /proc/net/ip_conntrack | 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` (or `route -n` to show raw IP addresses instead of hostnames). 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`).

This is a typical routing table for a machine on a local network with access to the internet:

```
# route
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 previous example, a request to `1.2.3.4` would be sent to `10.0.0.1`, on the `eth0` interface, because `1.2.3.4` is in the subnet described by the first rule (`0.0.0.0/0`) and not in the subnet described by the second rule (`10.0.0.0/24`). Subnets are specified by the destination and `genmask` values.

Linux prefers to route packets by *specificity* (how "small" a matching subnet is) and then by weight ("metric" in `route` output). Given our example, a packet addressed to `10.0.0.1` will always be sent to gateway `0.0.0.0` because that route matches a smaller set of addresses. If we had two routes with the same specificity, then the route with a lower metric wiould be preferred.

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 use 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 reference 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 reroute packets, and 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, `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.

Most Linux distributions are replacing `iptables` with `nftables`, a similar but more performant tool built atop Netfilter. Some distros already ship with 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 three key concepts in `iptables`: tables, chains, and rules. They are considered hierarchical in nature: a table contains chains, and a chain contains rules.

Tables organize rules according to the type of effect they have. `iptables` has a broad range of functionality, which tables group together. The three most commonly applicable 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, which we'll cover later.

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 five 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 a rule will be evaluated against.

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

It may help to refer 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 contain only specific target types, and many target types can be used only in specific tables. `iptables` has five tables, which are listed in Table 2-4.

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 is 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 this 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 built-in, "top-level" chains are `PREROUTING`, `INPUT`, `NAT`, `OUTPUT`, and `POSTROUTING`. These are powered by Netfilter hooks. Each chain corresponds to a hook. Table 2-5 shows the chain and hook pairs. There are also user-defined subchains that exist to help organize rules.

Table 2-5. `iptables` chains and corresponding Netfilter hooks

iptables chain	Netfilter hook
PREROUTIN	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 (see Figure 2-4).

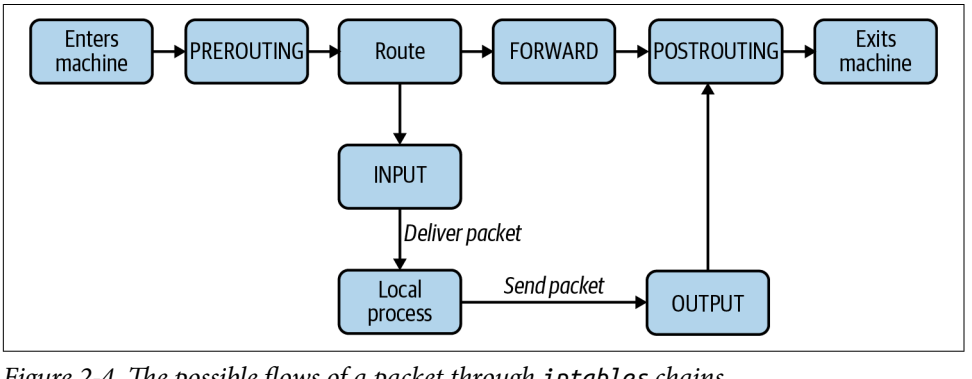

Figure 2-4. The possible flows of a packet through `iptables` chains

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 three 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`). We examine them in Table 2-6.

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, 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)

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 that log size can quickly become unwieldy. Log on important hosts with care.

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 has only `PREROUTING` and `OUTPUT` chains, in accordance with Netfilter's packet flow. The tables that contain each chain are laid out in Table 2-7.

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 be performed in `PREROUTING` or `OUTPUT`, and SNAT can be performed in only `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
2. `INPUT`
   a. Mangle
   b. NAT
   c. 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`, shown in Figure 2-5.

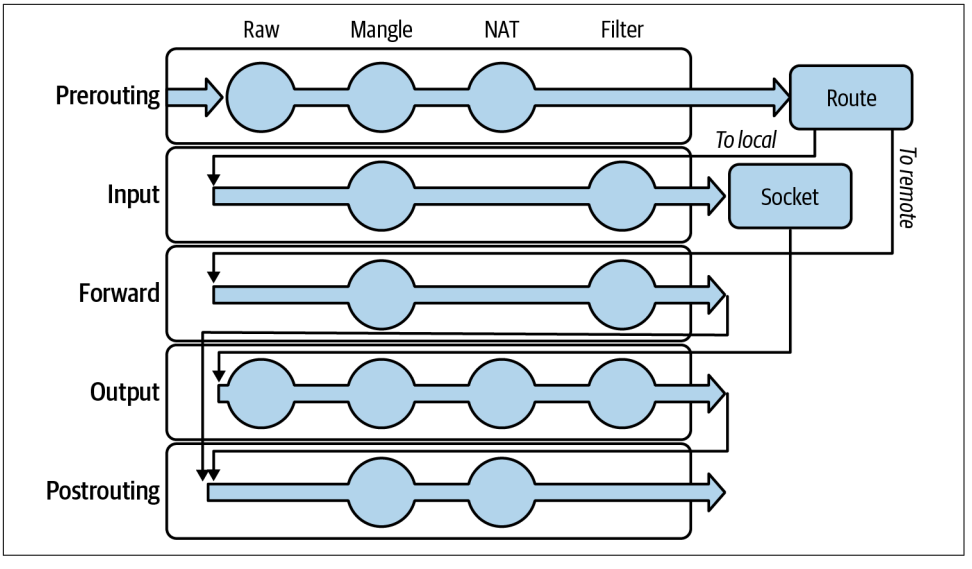

Figure 2-5. The flow of a packet through `iptables` tables and chains. A circle denotes a table/hook combination that exists in `iptables`.

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 entry-point, chains. However, users can define their own subchains 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 useful for logical separation or reusing a series of targets that can be executed in more than one context (i.e., a 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, `iptables`'s 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 creating new chains in Example 2-6.

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 two specific IP addresses, and packets from other addresses are logged and dropped.

Filter chains end in a default action, such as dropping 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 two parts: a match 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. The 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. Table 2-8 shows 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).

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 nonterminating. A terminating target will stop `iptables` from checking subsequent targets in the chain, essentially acting as a final decision. A nonterminating 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 terminating only *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. Example 2-7 shows a set of rules that would reject packets to port 80, 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 2-9 summarizes 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.
`RETURN`	All	Stops processing the current chain (or subchain). 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. Table 2-10 shows some example commands and explanations.

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

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
```

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

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

`--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` 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.

`iptables` can masquerade connections, making it appear as if the packets came from their 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 third 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 fan-out. 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 previous 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. 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 three 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 as shown previously. If you look closely, you can see rounding errors in one of the probability numbers.

Using DNAT fan-out 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 two replicas and 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 two replicas, skewing the load profile among 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.

Although `iptables` is widely used in Linux, it can become slow in the presence of a huge number of rules and offers limited load balancing functionality. Next we'll look at IPVS, an alternative that is more purpose-built for load balancing.

IPVS

IP Virtual Server (IPVS) is a Linux connection (L4) load balancer. Figure 2-6 shows a simple diagram of IPVS's role in routing packets.

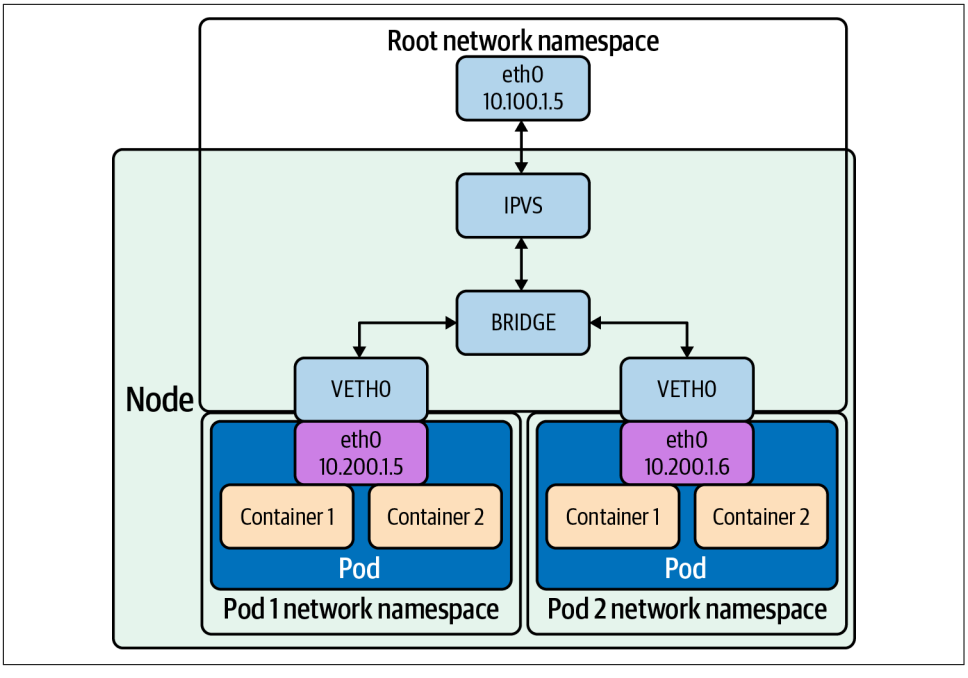

Figure 2-6. IPVS

`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 the `iptables` one), which are outlined in Table 2-11. This allows IPVS to spread load more effectively than `iptables`, depending on IPVS configuration and traffic patterns.

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 connections' 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.

IPVS supports packet forwarding modes:

- NAT rewrites source and destination addresses.
- DR encapsulates IP datagrams within IP datagrams.
- IP tunneling directly routes packets to the backend server by rewriting the MAC address of the data frame with the MAC address of the selected backend server.

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

Number of nodes in the cluster
: Even though Kubernetes already supports 5,000 nodes in release v1.6, `kube-proxy` with `iptables` is a bottleneck to scale the cluster to 5,000 nodes. One example is that with a NodePort service in a 5,000-node cluster, if we have 2,000 services and each service has 10 pods, this will cause at least 20,000 `iptables` records on each worker node, which can make the kernel pretty busy.

Time
: The time spent to add one rule when there are 5,000 services (40,000 rules) is 11 minutes. For 20,000 services (160,000 rules), it's 5 hours.

Latency
: There is latency to access a service (routing latency); each packet must traverse the `iptables` list until a match is made. 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

eBPF is a programming system that allows special sandboxed programs to run in the kernel without passing back and forth between kernel and user space, like we saw with Netfilter and `iptables`.

Before eBPF, there was the Berkeley Packet Filter (BPF). BPF is a technology used in the kernel, among other things, to analyze network traffic. BPF supports filtering packets, which allows a userspace process to supply a filter that specifies which packets it wants to inspect. One of BPF's use cases is `tcpdump`, shown in Figure 2-7. 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. `tcpdump`

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 a userspace program. Because of its performance characteristics, it is well suited for writing networking software.

You can learn more about eBPF on its website.

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

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.

Let's return to `tcpdump` as an example. Figure 2-8 shows a simplified rendition of `tcpdump`'s interactions with eBPF.

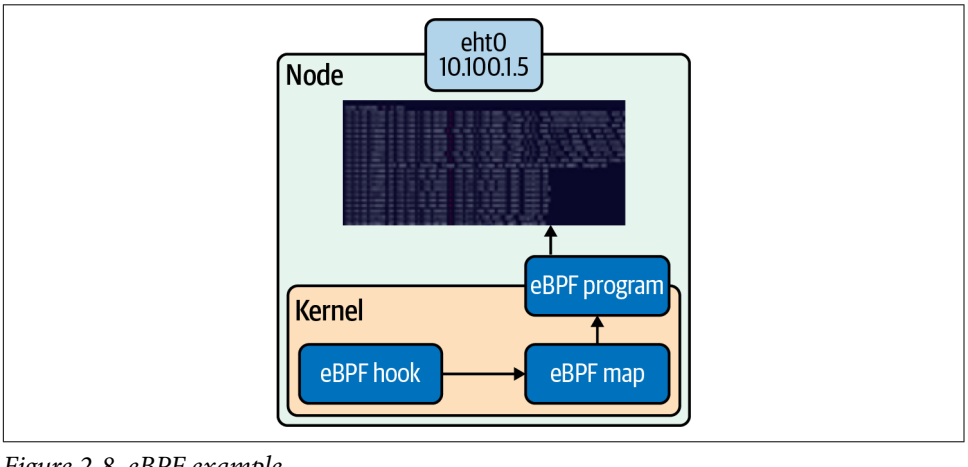

Figure 2-8. eBPF example

Suppose we run `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 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 structure 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 that 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 160,000 `iptables` rules representing 20,000 Kubernetes services.

Tracing
: Using BPF, we can gather pod and container-level network statistics. The BPF socket filter is nothing new, but the 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 will record any commands executed in the `kubectl exec` session and pass those commands to a userspace program that logs those events.

Security

Seccomp
: Secured computing that restricts 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, CNI and service implementation. Cilium replaces `kube-proxy`, which writes `iptables` rules to map a service's IP address 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 (layer 7) 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 (Table 2-12 is provided as a simple cheat sheet of tools and applicable use cases). 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. Cheat sheet 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 preinstalled in your distro of choice, but all should be available through your distro's package manager. We will sometimes use `# Truncated` in command output where we have 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 we consider it best practice to leave the fewest possible networking tools installed on a given machine.

An attacker may still be able to download tools themselves (e.g., by downloading a binary from the internet) or use 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 preinstalling networking tools.

Linux file permissions include something called the *setuid bit* that is sometimes used by 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 /etc/passwd
-rwsr-xr-x 1 root root 68208 May 28  2020 /usr/bin/passwd
```

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.) may use the setuid bit on some systems 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 installed by the system with the setuid bit set.

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 is a layer 4 protocol, like TCP and UDP. Kubernetes services support TCP and UDP, but not ICMP. This means that pings to a Kubernetes service will always fail. Instead, you will need to use `telnet` or a higher-level tool such as cURL to check connectivity to a service. Individual pods may still be reachable by `ping`, depending on your network configuration.

Firewalls and routing software are aware of ICMP packets and can be configured to filter or route specific ICMP packets. It is common, but not guaranteed (or necessarily advisable), to have permissive rules for ICMP packets. Some network administrators, network software, or cloud providers will allow ICMP packets by default.

The basic use of `ping` is simply `ping <address>`. The address can be an IP address 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 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 shows common `ping` options.

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 toward 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 shows common `traceroute` options.

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:e0ff: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:e0ff:fe08:6a7b)
;; WHEN: Sat Aug 08 18:11:48 PDT 2020
;; MSG SIZE  rcvd: 171
```

Table 2-15 shows common `dig` options.

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. The 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. Pressing Enter twice will send a command, which easily allows multiline commands to be written. Press Ctrl-J 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 previous example, `nmap` detects three open ports and guesses which service is running on each port.

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 shows common `nmap` options.

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 three TCP sockets, one 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 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 shows common `netstat` options.

Table 2-17. Useful `netstat` commands

Option	Syntax	Description
Show all sockets	`netstat -a`	Shows 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 many more things.

`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`. It's 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

The OpenSSL technology 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 subcommands (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. Here 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` is a similar tool to the command `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., use `curl -X DELETE foo/bar` to make a `DELETE` request.

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

- URL encoded: `-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 be of some help when debugging TLS issues, but more specialized tools such as `openssl` may be more helpful.

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:

```
$ 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.

Conclusion

This chapter has provided you with a whirlwind tour of networking in Linux. We focused primarily on concepts that are required to understand Kubernetes' implementation, 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 6

Kubernetes and Cloud Networking

The use of the cloud and its service offerings has grown tremendously: 77% of enterprises are using the public cloud in some capacity, and 81% can innovate more quickly with the public cloud than on-premise. With the popularity and innovation available in the cloud, it follows that running Kubernetes in the cloud is a logical step. Each major cloud provider has its own managed service offering for Kubernetes using its cloud network services.

In this chapter, we'll explore the network services offered by the major cloud providers AWS, Azure, and GCP with a focus on how they affect the networking needed to run a Kubernetes cluster inside that specific cloud. All the providers also have a CNI project that makes running a Kubernetes cluster smoother from an integration perspective with their cloud network APIs, so an exploration of the CNIs is warranted. After reading this chapter, administrators will understand how cloud providers implement their managed Kubernetes on top of their cloud network services.

Amazon Web Services

Amazon Web Services (AWS) has grown its cloud service offerings from Simple Queue Service (SQS) and Simple Storage Service (S3) to well over 200 services. Gartner Research positions AWS in the Leaders quadrant of its 2020 Magic Quadrant for Cloud Infrastructure & Platform Services. Many services are built atop of other foundational services. For example, Lambda uses S3 for code storage and DynamoDB for metadata. AWS CodeCommit uses S3 for code storage. EC2, S3, and CloudWatch are integrated into the Amazon Elastic MapReduce service, creating a managed data platform. The AWS networking services are no different. Advanced services such as peering and endpoints use building blocks from core networking fundamentals. Understanding those fundamentals, which enable AWS to build a comprehensive Kubernetes service, is needed for administrators and developers.

AWS Network Services

AWS has many services that allow users to extend and secure their cloud networks. Amazon Elastic Kubernetes Service (EKS) makes extensive use of those network components available in the AWS cloud. We will discuss the basics of AWS networking components and how they are related to deploying an EKS cluster network. This section will also discuss several other open source tools that make managing a cluster and application deployments simple. The first is `eksctl`, a CLI tool that deploys and manages EKS clusters. As we have seen from previous chapters, there are many components needed to run a cluster, and that is also true on the AWS network. `eksctl` will deploy all the components in AWS for cluster and network administrators. Then, we will discuss the AWS VPC CNI, which allows the cluster to use native AWS services to scale pods and manage their IP address space. Finally, we will examine the AWS Application Load Balancer ingress controller, which automates, manages, and simplifies deployments of application load balancers and ingresses for developers running applications on the AWS network.

Virtual private cloud

The basis of the AWS network is the virtual private cloud (VPC). A majority of AWS resources will work inside the VPC. VPC networking is an isolated virtual network defined by administrators for only their account and its resources. In Figure 6-1, we can see a VPC defined with a single CIDR of `192.168.0.0/16`. All resources inside the VPC will use that range for private IP addresses. AWS is constantly enhancing its service offerings; now, network administrators can use multiple nonoverlapping CIDRs in a VPC. The pod IP addresses will also come from VPC CIDR and host IP addressing; more on that in "AWS VPC CNI" on page 113. A VPC is set up per AWS region; you can have multiple VPCs per region, but a VPC is defined in only one.

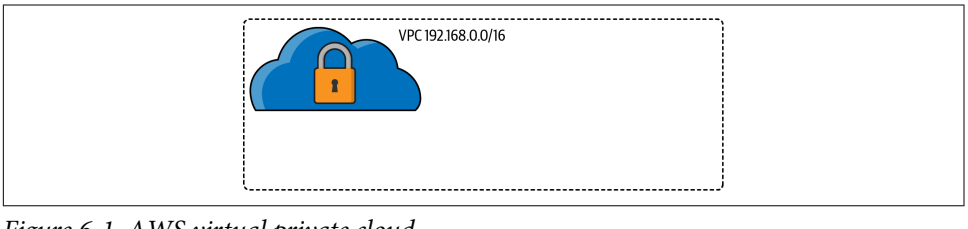

Figure 6-1. AWS virtual private cloud

Region and availability zones

Resources are defined by boundaries in AWS, such as global, region, or availability zone. AWS networking comprises multiple regions; each AWS region consists of multiple isolated and physically separate availability zones (AZs) within a geographic area. An AZ can contain multiple data centers, as shown in Figure 6-2. Some regions can contain six AZs, while newer regions could contain only two. Each AZ is directly

connected to the others but is isolated from the failures of another AZ. This design is important to understand for multiple reasons: high availability, load balancing, and subnets are all affected. In one region a load balancer will route traffic over multiple AZs, which have separate subnets and thus enable HA for applications.

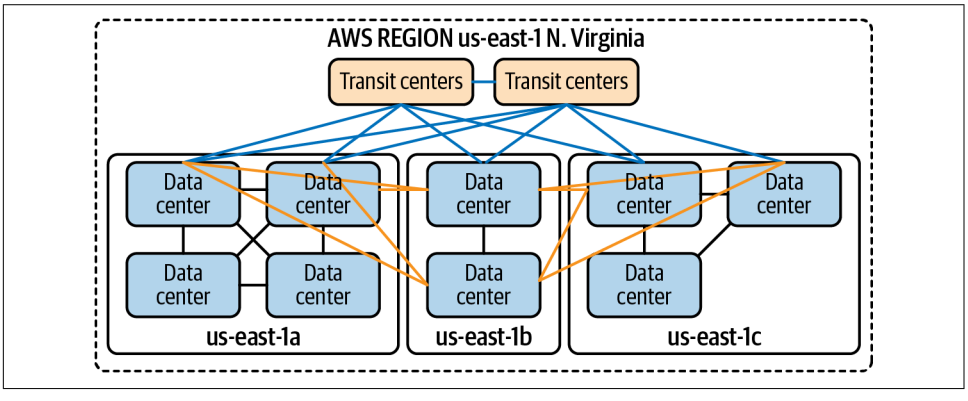

Figure 6-2. AWS region network layout

An up-to-date list of AWS regions and AZs is available in the documentation.

Subnet

A VPC is compromised of multiple subnets from the CIDR range and deployed to a single AZ. Applications that require high availability should run in multiple AZs and be load balanced with any one of the load balancers available, as discussed in "Region and availability zones" on page 94.

A subnet is public if the routing table has a route to an internet gateway. In Figure 6-3, there are three public and private subnets. Private subnets have no direct route to the internet. These subnets are for internal network traffic, such as databases. The size of your VPC CIDR range and the number of public and private subnets are a design consideration when deploying your network architecture. Recent improvements to VPC like allowing multiple CIDR ranges help lessen the ramification of poor design choices, since now network engineers can simply add another CIDR range to a provisioned VPC.

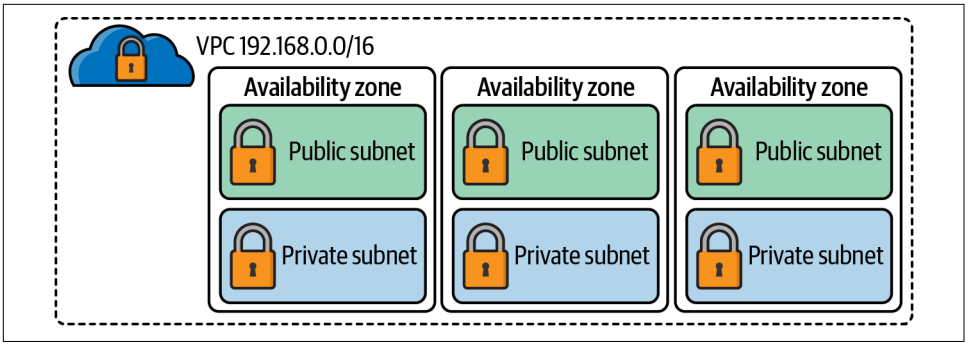

Figure 6-3. VPC subnets

Let's discuss those components that help define if a subnet is public or private.

Routing tables

Each subnet has exactly one route table associated with it. If one is not explicitly associated with it, the main route table is the default one. Network connectivity issues can manifest here; developers deploying applications inside a VPC must know to manipulate route tables to ensure traffic flows where it's intended.

The following are rules for the main route table:

- The main route table cannot be deleted.
- A gateway route table cannot be set as the main.
- The main route table can be replaced with a custom route table.
- Admins can add, remove, and modify routes in the main route table.
- The local route is the most specific.
- Subnets can explicitly associate with the main route table.

There are route tables with specific goals in mind; here is a list of them and a description of how they are different:

Main route table
: This route table automatically controls routing for all subnets that are not explicitly associated with any other route table.

Custom route table
: A route table network engineers create and customize for specific application traffic flow.

Edge association
: A routing table to route inbound VPC traffic to an edge appliance.

Subnet route table
: A route table that's associated with a subnet.

Gateway route table
: A route table that's associated with an internet gateway or virtual private gateway.

Each route table has several components that determine its responsibilities:

Route table association
: The association between a route table and a subnet, internet gateway, or virtual private gateway.

Rules
: A list of routing entries that define the table; each rule has a destination, target, status, and propagated flag.

Destination
: The range of IP addresses where you want traffic to go (destination CIDR).

Target
: The gateway, network interface, or connection through which to send the destination traffic; for example, an internet gateway.

Status
: The state of a route in the route table: active or blackhole. The blackhole state indicates that the route's target isn't available.

Propagation
: Route propagation allows a virtual private gateway to automatically propagate routes to the route tables. This flag lets you know if it was added via propagation.

Local route
: A default route for communication within the VPC.

In Figure 6-4, there are two routes in the route table. Any traffic destined for `11.0.0.0/16` stays on the local network inside the VPC. All other traffic, `0.0.0.0/0`, goes to the internet gateway, `igw-f43c4690`, making it a public subnet.

Figure 6-4. Route table

Elastic network interface

An elastic network interface (ENI) is a logical networking component in a VPC that is equivalent to a virtual network card. ENIs contain an IP address, for the instance, and they are elastic in the sense that they can be associated and disassociated to an instance while retaining its properties.

ENIs have these properties:

- Primary private IPv4 address
- Secondary private IPv4 addresses
- One elastic IP (EIP) address per private IPv4 address
- One public IPv4 address, which can be auto-assigned to the network interface for `eth0` when you launch an instance
- One or more IPv6 addresses
- One or more security groups
- MAC address
- Source/destination check flag
- Description

A common use case for ENIs is the creation of management networks that are accessible only from a corporate network. AWS services like Amazon WorkSpaces use ENIs to allow access to the customer VPC and the AWS-managed VPC. Lambda can reach resources, like databases, inside a VPC by provisioning and attaching to an ENI.

Later in the section we will see how the AWS VPC CNI uses and manages ENIs along with IP addresses for pods.

Elastic IP address

An EIP address is a static public IPv4 address used for dynamic network addressing in the AWS cloud. An EIP is associated with any instance or network interface in any VPC. With an EIP, application developers can mask an instance's failures by remapping the address to another instance.

An EIP address is a property of an ENI and is associated with an instance by updating the ENI attached to the instance. The advantage of associating an EIP with the ENI rather than directly to the instance is that all the network interface attributes move from one instance to another in a single step.

The following rules apply:

- An EIP address can be associated with either a single instance or a network interface at a time.
- An EIP address can migrate from one instance or network interface to another.
- There is a (soft) limit of five EIP addresses.
- IPv6 is not supported.

Services like NAT and internet gateway use EIPs for consistency between the AZ. Other gateway services like a bastion can benefit from using an EIP. Subnets can automatically assign public IP addresses to EC2 instances, but that address could change; using an EIP would prevent that.

Security controls

There are two fundamental security controls within AWS networking: security groups and network access control lists (NACLs). In our experience, lots of issues arise from misconfigured security groups and NACLs. Developers and network engineers need to understand the differences between the two and the impacts of changes on them.

Security groups. Security groups operate at the instance or network interface level and act as a firewall for those devices associated with them. A security group is a group of network devices that require common network access to each other and other devices on the network. In Figure 6-5 ,we can see that security works across AZs. Security groups have two tables, for inbound and outbound traffic flow. Security groups are stateful, so if traffic is allowed on the inbound flow, the outgoing traffic is allowed. Each security group has a list of rules that define the filter for traffic. Each rule is evaluated before a forwarding decision is made.

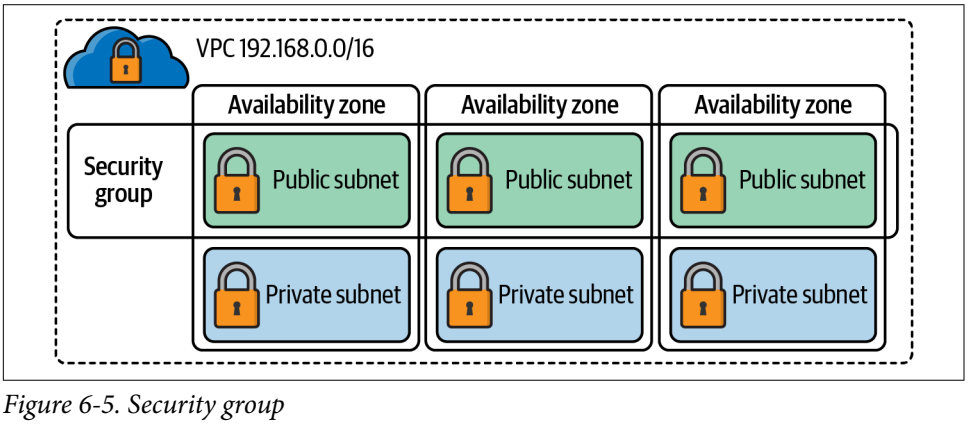

Figure 6-5. Security group

The following is a list of components of security group rules:

Source/destination
: Source (inbound rules) or destination (outbound rules) of the traffic inspected:

  - Individual or range of IPv4 or IPv6 addresses
  - Another security group
  - Other ENIs, gateways, or interfaces

Protocol
: Which layer 4 protocol being filtered, 6 (TCP), 17 (UDP), and 1 (ICMP)

Port range
: Specific ports for the protocol being filtered

Description
: User-defined field to inform others of the intent of the security group

Security groups are similar to the Kubernetes network policies we discussed in earlier chapters. They are a fundamental network technology and should always be used to secure your instances in the AWS VPC. EKS deploys several security groups for communication between the AWS-managed data plane and your worker nodes.

Network access control lists. Network access control lists operate similarly to how they do in other firewalls so that network engineers will be familiar with them. In Figure 6-6, you can see each subnet has a default NACL associated with it and is bounded to an AZ, unlike the security group. Filter rules must be defined explicitly in both directions. The default rules are quite permissive, allowing all traffic in both directions. Users can define their own NACLs to use with a subnet for an added security layer if the security group is too open. By default, custom NACLs deny all traffic, and therefore add rules when deployed; otherwise, instances will lose connectivity.

Here are the components of an NACL:

Rule number
: Rules are evaluated starting with the lowest numbered rule.

Type
: The type of traffic, such as SSH or HTTP.

Protocol
: Any protocol that has a standard protocol number: TCP/UDP or ALL.

Port range
: The listening port or port range for the traffic. For example, 80 for HTTP traffic.

Source
: Inbound rules only; the CIDR range source of the traffic.

Destination
: Outbound rules only; the destination for the traffic.

Allow/Deny
: Whether to allow or deny the specified traffic.

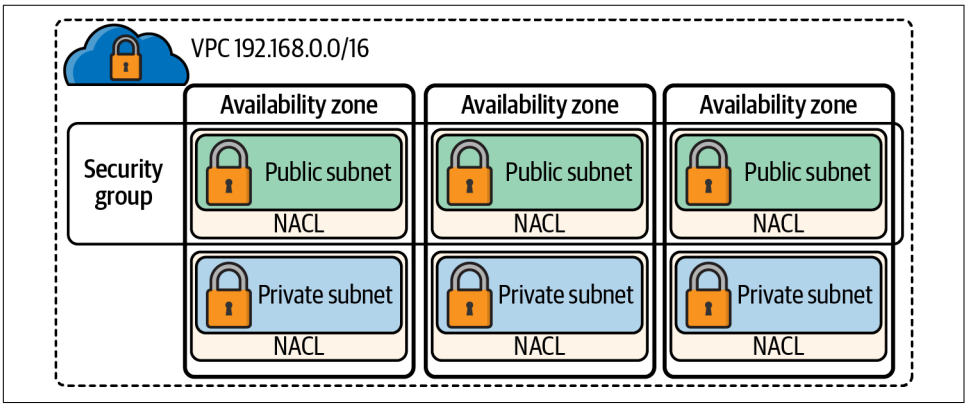

Figure 6-6. NACL

NACLs add an extra layer of security for subnets that may protect from lack or misconfiguration of security groups.

Table 6-1 summarizes the fundamental differences between security groups and network ACLs.

Table 6-1. Security and NACL comparison table

Security group	Network ACL
Operates at the instance level.	Operates at the subnet level.
Supports allow rules only.	Supports allow rules and deny rules.
Stateful: Return traffic is automatically allowed, regardless of any rules.	Stateless: Return traffic must be explicitly allowed by rules.
All rules are evaluated before a forwarding decision is made.	Rules are processed in order, starting with the lowest numbered rule.
Applies to an instance or network interface.	All rules apply to all instances in the subnets that it's associated with.

It is crucial to understand the differences between NACL and security groups. Network connectivity issues often arise due to a security group not allowing traffic on a specific port or someone not adding an outbound rule on an NACL. When troubleshooting issues with AWS networking, developers and network engineers alike should add checking these components to their troubleshooting list.

All the components we have discussed thus far manage traffic flow inside the VPC. The following services manage traffic into the VPC from client requests and ultimately to applications running inside a Kubernetes cluster: network address translation devices, internet gateway, and load balancers. Let's dig into those a little more.

Network address translation devices

Network address translation (NAT) devices are used when instances inside a VPC require internet connectivity, but network connections should not be made directly to instances. Examples of instances that should run behind a NAT device are database instances or other middleware needed to run applications.

In AWS, network engineers have several options for running NAT devices. They can manage their own NAT devices deployed as EC2 instances or use the AWS Managed Service NAT gateway (NAT GW). Both require public subnets deployed in multiple AZs for high availability and EIP. A restriction of a NAT GW is that the IP address of it cannot change after you deploy it. Also, that IP address will be the source IP address used to communicate with the internet gateway.

In the VPC route table in Figure 6-7, we can see how the two route tables exist to establish a connection to the internet. The main route table has two rules, a local route for the inter-VPC and a route for `0.0.0.0/0` with a target of the NAT GW ID. The private subnet's database servers will route traffic to the internet via that NAT GW rule in their route tables.

Pods and instances in EKS will need to egress the VPC, so a NAT device must be deployed. Your choice of NAT device will depend on the operational overhead, cost, or availability requirements for your network design.

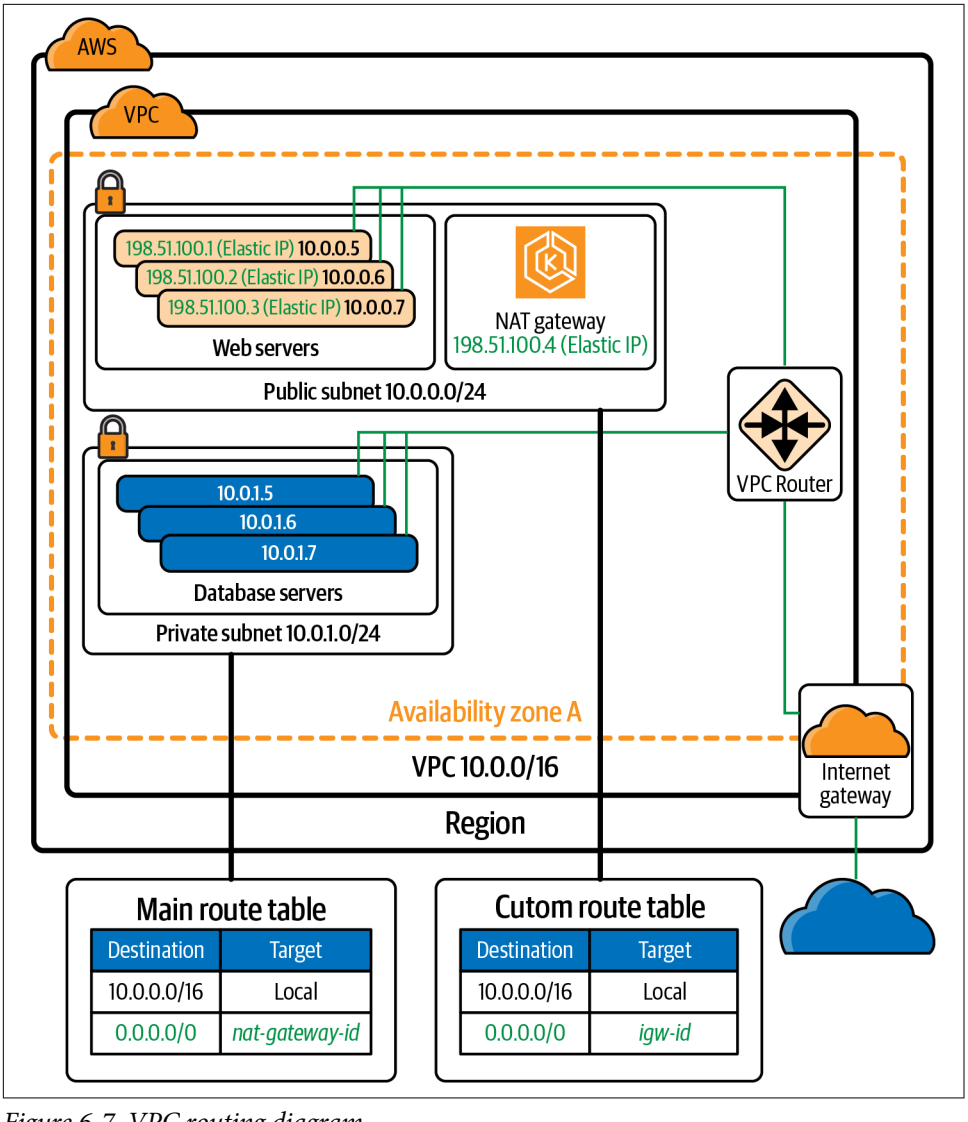

Figure 6-7. VPC routing diagram

Internet gateway

The internet gateway is an AWS-managed service and device in the VPC network that allows connectivity to the internet for all devices in the VPC. Here are the steps to ensure access to or from the internet in a VPC:

1. Deploy and attach an IGW to the VPC.
2. Define a route in the subnet's route table that directs internet-bound traffic to the IGW.
3. Verify NACLs and security group rules allow the traffic to flow to and from instances.

All of this is shown in the VPC routing from Figure 6-7. We see the IGW deploy for the VPC, a custom route table setup that routes all traffic, `0.0.0.0/0`, to the IGW. The web instances have an IPv4 internet routable address, `198.51.100.1-3`.

Elastic load balancers

Now that traffic flows from the internet and clients can request access to applications running inside a VPC, we will need to scale and distribute the load for requests. AWS has several options for developers, depending on the type of application load and network traffic requirements needed.

The elastic load balancer has four options:

Classic
: A classic load balancer provides fundamental load balancing of EC2 instances. It operates at the request and the connection level. Classic load balancers are limited in functionality and are not to be used with containers.

Application
: Application load balancers are layer 7 aware. Traffic routing is made with request-specific information like HTTP headers or HTTP paths. The application load balancer is used with the application load balancer controller. The ALB controller allows devs to automate the deployment and ALB without using the console or API, instead just a few YAML lines.

Network
: The network load balancer operates at layer 4. Traffic can be routed based on incoming TCP/UDP ports to individual hosts running services on that port. The network load balancer also allows admins to deploy then with an EIP, a feature unique to the network load balancer.

Gateway
: The gateway load balancer manages traffic for appliances at the VPC level. Such network devices like deep packet inspection or proxies can be used with a gateway load balancer. The gateway load balancer is added here to complete the AWS service offering but is not used within the EKS ecosystem.

AWS load balancers have several attributes that are important to understand when working with not only containers but other workloads inside the VPC:

Rule
: (ALB only) The rules that you define for your listener determine how the load balancer routes all requests to the targets in the target groups.

Listener
: Checks for requests from clients. They support HTTP and HTTPS on ports 1–65535.

Target
: An EC2 instance, IP address, pods, or lambda running application code.

Target Group
: Used to route requests to a registered target.

Health Check
: Test to ensure targets are still able to accept client requests.

Each of these components of an ALB is outlined in Figure 6-8. When a request comes into the load balancer, a listener is continually checking for requests that match the protocol and port defined for it. Each listener has a set of rules that define where to direct the request. The rule will have an action type to determine how to handle the request:

authenticate-cognito
: (HTTPS listeners) Use Amazon Cognito to authenticate users.

authenticate-oidc
: (HTTPS listeners) Use an identity provider that is compliant with OpenID Connect to authenticate users.

fixed-response
: Returns a custom HTTP response.

forward
: Forward requests to the specified target groups.

redirect
: Redirect requests from one URL to another.

The action with the lowest order value is performed first. Each rule must include exactly one of the following actions: forward, redirect, or fixed-response. In Figure 6-8, we have target groups, which will be the recipient of our forward rules. Each target in the target group will have health checks so the load balancer will know which instances are healthy and ready to receive requests.

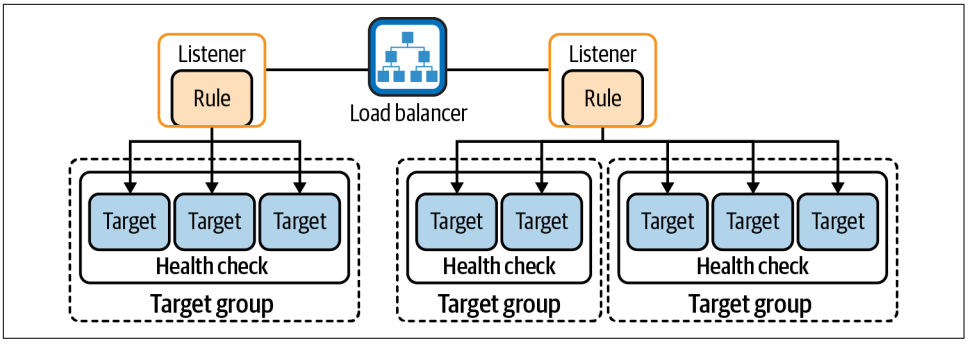

Figure 6-8. Load balancer components

Now that we have a basic understanding of how AWS structures its networking components, we can begin to see how EKS leverages these components to the network and secure the managed Kubernetes cluster and network.

Amazon Elastic Kubernetes Service

Amazon Elastic Kubernetes Service (EKS) is AWS's managed Kubernetes service. It allows developers, cluster administrators, and network administrators to quickly deploy a production-scale Kubernetes cluster. Using the scaling nature of the cloud and AWS network services, with one API request, many services are deployed, including all the components we reviewed in the previous sections.

How does EKS accomplish this? Like with any new service AWS releases, EKS has gotten significantly more feature-rich and easier to use. EKS now supports on-prem deploys with EKS Anywhere, serverless with EKS Fargate, and even Windows nodes. EKS clusters can be deployed traditionally with the AWS CLI or console. `eksctl` is a command-line tool developed by Weaveworks, and it is by far the easiest way to date to deploy all the components needed to run EKS. Our next section will detail the requirements to run an EKS cluster and how `eksctl` accomplishes this for cluster admins and devs.

Let's discuss the components of EKS cluster networking.

EKS nodes

Workers nodes in EKS come in three flavors: EKS-managed node groups, self-managed nodes, and AWS Fargate. The choice for the administrator is how much control and operational overhead they would like to accrue.

Managed node group

Amazon EKS managed node groups create and manage EC2 instances for you. All managed nodes are provisioned as part of an EC2 Auto Scaling group that's managed by Amazon EKS as well. All resources including EC2 instances and

Auto Scaling groups run within your AWS account. A managed-node group's Auto Scaling group spans all the subnets that you specify when you create the group.

Self-managed node group
: Amazon EKS nodes run in your AWS account and connect to your cluster's control plane via the API endpoint. You deploy nodes into a node group. A node group is a collection of EC2 instances that are deployed in an EC2 Auto Scaling group. All instances in a node group must do the following:

  - Be the same instance type
  - Be running the same Amazon Machine Image
  - Use the same Amazon EKS node IAM role

Fargate
: Amazon EKS integrates Kubernetes with AWS Fargate by using controllers that are built by AWS using the upstream, extensible model provided by Kubernetes. Each pod running on Fargate has its own isolation boundary and does not share the underlying kernel, CPU, memory, or elastic network interface with another pod. You also cannot use security groups for pods with pods running on Fargate.

The instance type also affects the cluster network. In EKS the number of pods that can run on the nodes is defined by the number of IP addresses that instance can run. We discuss this further in "AWS VPC CNI" on page 113 and "eksctl" on page 111.

Nodes must be able to communicate with the Kubernetes control plane and other AWS services. The IP address space is crucial to run an EKS cluster. Nodes, pods, and all other services will use the VPC CIDR address ranges for components. The EKS VPC requires a NAT gateway for private subnets and that those subnets be tagged for use with EKS:

```
Key - kubernetes.io/cluster/<cluster-name>
Value - shared
```

The placement of each node will determine the network "mode" that EKS operates; this has design considerations for your subnets and Kubernetes API traffic routing.

EKS mode

Figure 6-9 outlines EKS components. The Amazon EKS control plane creates up to four cross-account elastic network interfaces in your VPC for each cluster. EKS uses two VPCs, one for the Kubernetes control plane, including the Kubernetes API masters, API loadbalancer, and etcd depending on the networking model; the other is the customer VPC where the EKS worker nodes run your pods. As part of the boot process for the EC2 instance, the Kubelet is started. The node's Kubelet reaches out to the Kubernetes cluster endpoint to register the node. It connects either to the public

endpoint outside the VPC or to the private endpoint within the VPC. `kubectl` commands reach out to the API endpoint in the EKS VPC. End users reach applications running in the customer VPC.

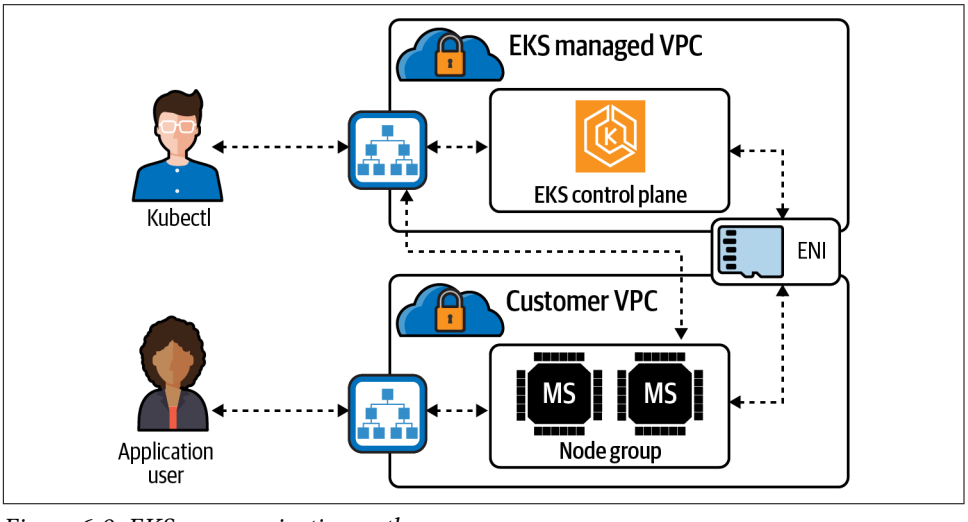

Figure 6-9. EKS communication path

There are three ways to configure cluster control traffic and the Kubernetes API endpoint for EKS, depending on where the control and data planes of the Kubernetes components run.

The networking modes are as follows:

Public-only
: Everything runs in a public subnet, including worker nodes.

Private-only
: Runs solely in a private subnet, and Kubernetes cannot create internet-facing load balancers.

Mixed
: Combo of public and private.

The public endpoint is the default option; it is public because the load balancer for the API endpoint is on a public subnet, as shown in Figure 6-10. Kubernetes API requests that originate from within the cluster's VPC, like when the worker node reaches out to the control plane, leave the customer VPC, but not the Amazon network. One security concern to consider when using a public endpoint is that the API endpoints are on a public subnet and reachable on the internet.

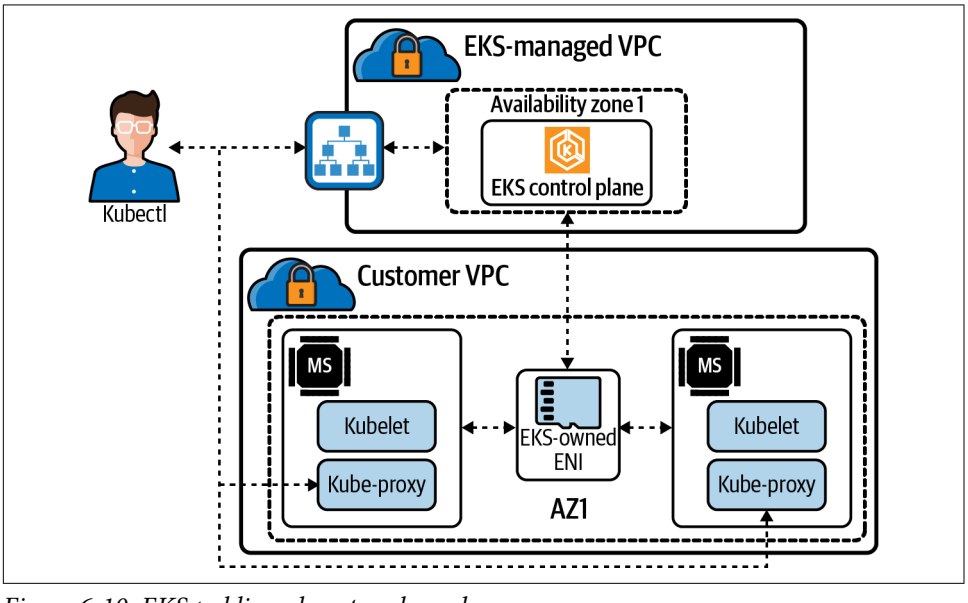

Figure 6-10. EKS public-only network mode

Figure 6-11 shows the private endpoint mode; all traffic to your cluster API must come from within your cluster's VPC. There's no internet access to your API server; any `kubectl` commands must come from within the VPC or a connected network. The cluster's API endpoint is resolved by public DNS to a private IP address in the VPC.

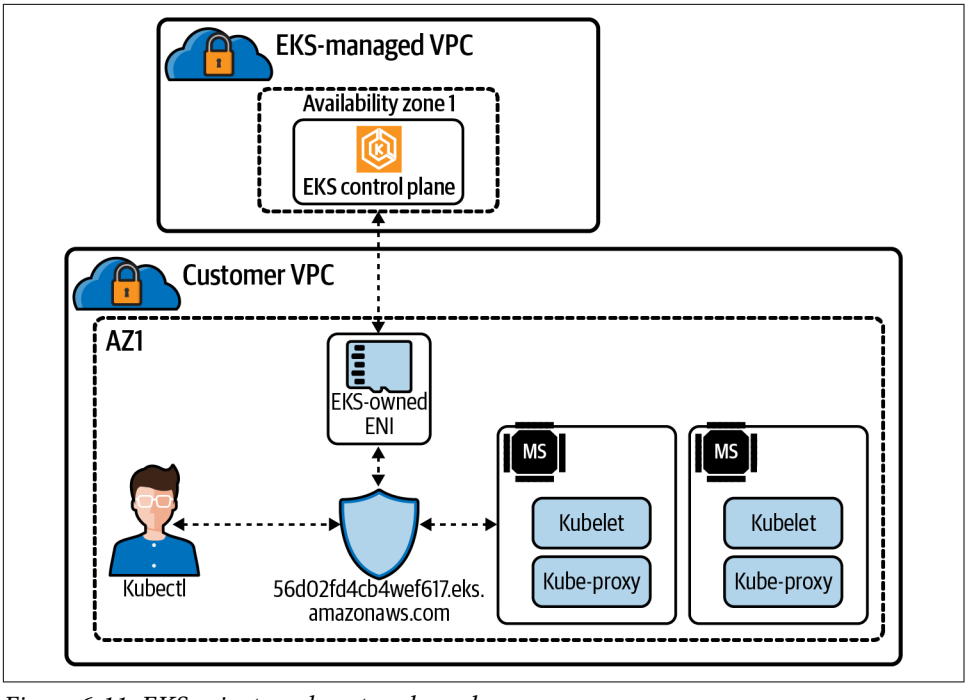

Figure 6-11. EKS private-only network mode

When both public and private endpoints are enabled, any Kubernetes API requests from within the VPC communicate to the control plane by the EKS-managed ENIs within the customer VPC, as demonstrated in Figure 6-12. The cluster API is still accessible from the internet, but it can be limited using security groups and NACLs.

Please see the AWS documentation for more ways to deploy an EKS.

Determining what mode to operate in is a critical decision administrators will make. It will affect the application traffic, the routing for load balancers, and the security of the cluster. There are many other requirements when deploying a cluster in EKS as well. `eksctl` is one tool to help manage all those requirements. But how does `eksctl` accomplish that?

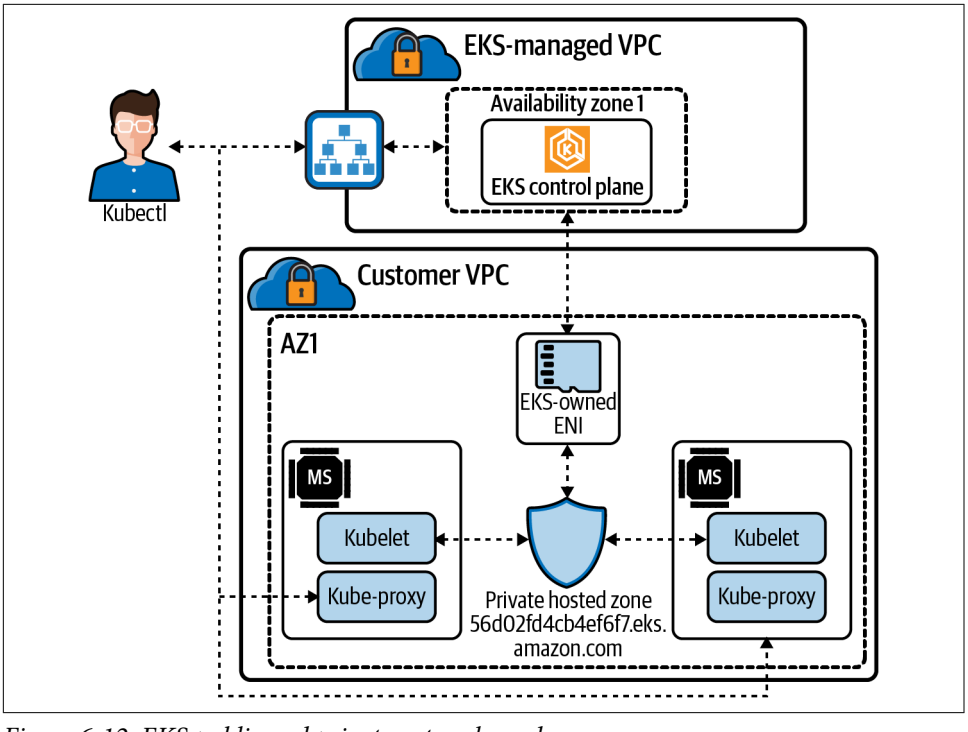

Figure 6-12. EKS public and private network mode

eksctl

`eksctl` is a command-line tool developed by Weaveworks, and it is by far the easiest way to deploy all the components needed to run EKS.

All the information about `eksctl` is available on its website.

`eksctl` defaults to creating a cluster with the following default parameters:

- An autogenerated cluster name
- Two m5.large worker nodes
- Use of the official AWS EKS AMI
- Us-west-2 default AWS region
- A dedicated VPC

A dedicated VPC with 192.168.0.0/16 CIDR range, `eksctl` will create by default 8 /19 subnets: three private, three public, and two reserved subnets. `eksctl` will also deploy a NAT GW that allows for communication of nodes placed in private subnets and an internet gateway to enable access for needed container images and communication to the Amazon S3 and Amazon ECR APIs.

Two security groups are set up for the EKS cluster:

Ingress inter node group SG
: Allows nodes to communicate with each other on all ports

Control plane security group
: Allows communication between the control plane and worker node groups

Node groups in public subnets will have SSH disabled. EC2 instances in the initial node group get a public IP and can be accessed on high-level ports.

One node group containing two m5.large nodes is the default for `eksctl`. But how many pods can that node run? AWS has a formula based on the node type and the number of interfaces and IP addresses it can support. That formula is as follows:

```
(Number of network interfaces for the instance type ×
(the number of IP addresses per network interface - 1)) + 2
```

Using the preceding formula and the default instance size on `eksctl`, an m5.large can support a maximum of 29 pods.

System pods count toward the maximum pods. The CNI plugin and `kube-proxy` pods run on every node in a cluster, so you're only able to deploy 27 additional pods to an m5.large instance. CoreDNS runs on nodes in the cluster, which further decrements the maximum number of pods a node can run.

Teams running clusters must decide on cluster sizing and instance types to ensure no deployment issues with hitting node and IP limitations. Pods will sit in the "waiting" state if there are no nodes available with the pod's IP address. Scaling events for the EKS node groups can also hit EC2 instance type limits and cause cascading issues.

All of these networking options are configurable via the `eksctl` config file.

`eksctl` VPC options are available in the eksctl documentation.

We have discussed how the size node is important for pod IP addressing and the number of them we can run. Once the node is deployed, the AWS VPC CNI manages pod IP addressing for nodes. Let's dive into the inner workings of the CNI.

AWS VPC CNI

AWS has its open source implementation of a CNI. AWS VPC CNI for the Kubernetes plugin offers high throughput and availability, low latency, and minimal network jitter on the AWS network. Network engineers can apply existing AWS VPC networking and security best practices for building Kubernetes clusters on AWS. It includes using native AWS services like VPC flow logs, VPC routing policies, and security groups for network traffic isolation.

The open source for AWS VPC CNI is on GitHub.

There are two components to the AWS VPC CNI:

CNI plugin
: The CNI plugin is responsible for wiring up the host's and pod's network stack when called. It also configures the interfaces and virtual Ethernet pairs.

ipamd
: A long-running node-local IPAM daemon is responsible for maintaining a warm pool of available IP addresses and assigning an IP address to a pod.

Figure 6-13 demonstrates what the VPC CNI will do for nodes. A customer VPC with a subnet 10.200.1.0/24 in AWS gives us 250 usable addresses in this subnet. There are two nodes in our cluster. In EKS, the managed nodes run with the AWS CNI as a daemon set. In our example, each node has only one pod running, with a secondary IP address on the ENI, `10.200.1.6` and `10.200.1.8`, for each pod. When a worker node first joins the cluster, there is only one ENI and all its addresses in the ENI. When pod three gets scheduled to node 1, ipamd assigns the IP address to the ENI for that pod. In this case, `10.200.1.7` is the same thing on node 2 with pod 4.

When a worker node first joins the cluster, there is only one ENI and all of its addresses in the ENI. Without any configuration, `ipamd` always tries to keep one extra ENI. When several pods running on the node exceeds the number of addresses on a single ENI, the CNI backend starts allocating a new ENI. The CNI plugin works by allocating multiple ENIs to EC2 instances and then attaches secondary IP addresses to these ENIs. This plugin allows the CNI to allocate as many IPs per instance as possible.

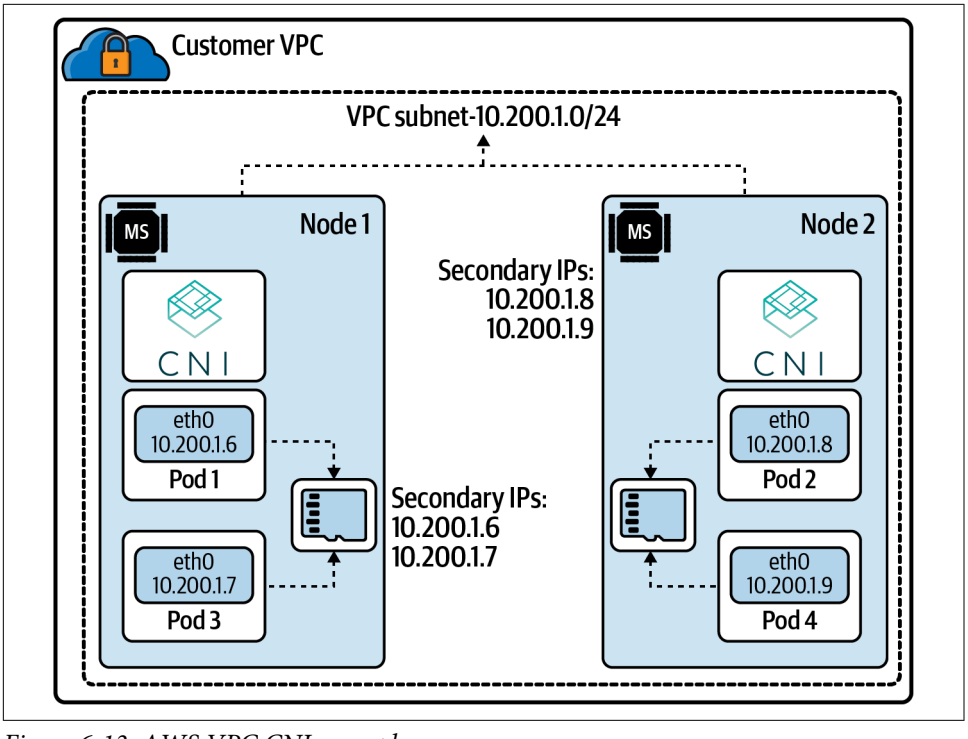

Figure 6-13. AWS VPC CNI example

The AWS VPC CNI is highly configurable. This list includes just a few options:

AWS_VPC_CNI_NODE_PORT_SUPPORT
: Specifies whether NodePort services are enabled on a worker node's primary network interface. This requires additional `iptables` rules and that the kernel's reverse path filter on the primary interface is set to loose.

AWS_VPC_K8S_CNI_CUSTOM_NETWORK_CFG
: Worker nodes can be configured in public subnets, so you need to configure pods to be deployed in private subnets, or if pods' security requirement needs are different from others running on the node, setting this to `true` will enable that.

AWS_VPC_ENI_MTU
: Default: 9001. Used to configure the MTU size for attached ENIs. The valid range is from 576 to 9001.

WARM_ENI_TARGET
: Specifies the number of free elastic network interfaces (and all of their available IP addresses) that the `ipamd` daemon should attempt to keep available for pod assignment on the node. By default, `ipamd` attempts to keep one elastic network

interface and all of its IP addresses available for pod assignment. The number of IP addresses per network interface varies by instance type.

AWS_VPC_K8S_CNI_EXTERNALSNAT
: Specifies whether an external NAT gateway should be used to provide SNAT of secondary ENI IP addresses. If set to `true`, the SNAT `iptables` rule and external VPC IP rule are not applied, and these rules are removed if they have already been applied. Disable SNAT if you need to allow inbound communication to your pods from external VPNs, direct connections, and external VPCs, and your pods do not need to access the internet directly via an internet gateway.

For example, if your pods with a private IP address need to communicate with others' private IP address spaces, you enable `AWS_VPC_K8S_CNI_EXTERNALSNAT` by using this command:

```
kubectl set env daemonset
-n kube-system aws-node AWS_VPC_K8S_CNI_EXTERNALSNAT=true
```

All the information for EKS pod networking can be found in the EKS documentation.

The AWS VPC CNI allows for maximum control over the networking options on EKS in the AWS network.

There is also the AWS ALB ingress controller that makes managing and deploying applications on the AWS cloud network smooth and automated. Let's dig into that next.

AWS ALB ingress controller

Let's walk through the example in Figure 6-14 of how the AWS ALB works with Kubernetes. For a review of what an ingress controller is, please check out Chapter 5.

Let's discuss all the moving parts of ALB Ingress controller:

1. The ALB ingress controller watches for ingress events from the API server. When requirements are met, it will start the creation process of an ALB.
2. An ALB is created in AWS for the new ingress resource. Those resources can be internal or external to the cluster.
3. Target groups are created in AWS for each unique Kubernetes service described in the ingress resource.

4. Listeners are created for every port detailed in your ingress resource annotations. Default ports for HTTP and HTTPS traffic are set up if not specified. NodePort services for each service create the node ports that are used for our health checks.
5. Rules are created for each path specified in your ingress resource. This ensures traffic to a specific path is routed to the correct Kubernetes service.

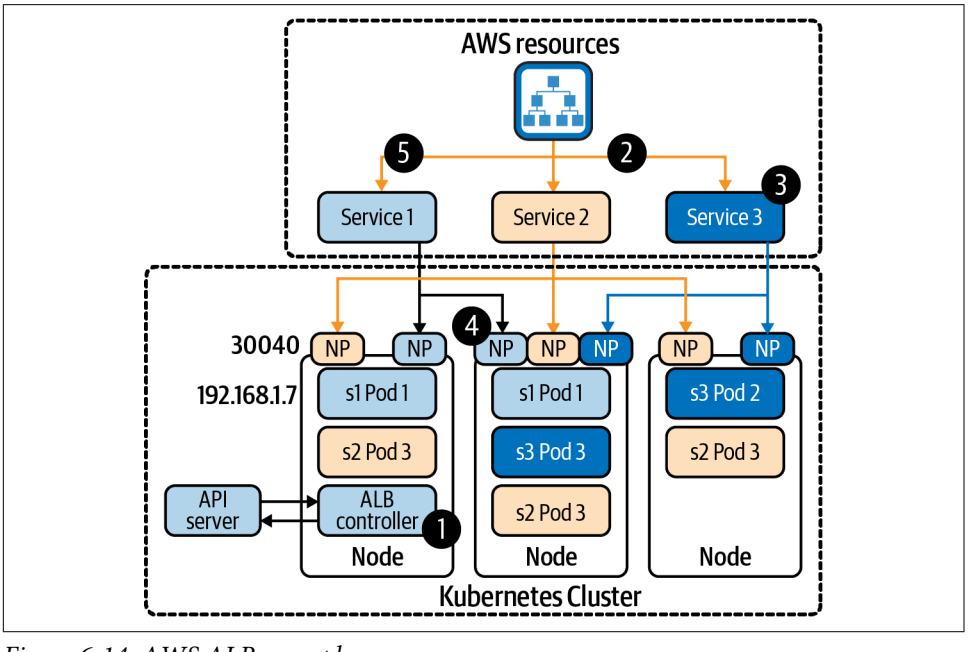

Figure 6-14. AWS ALB example

How traffic reaches nodes and pods is affected by one of two modes the ALB can run:

Instance mode
: Ingress traffic starts at the ALB and reaches the Kubernetes nodes through each service's NodePort. This means that services referenced from ingress resources must be exposed by `type:NodePort` to be reached by the ALB.

IP mode
: Ingress traffic starts at the ALB and reaches directly to the Kubernetes pods. CNIs must support a directly accessible pod IP address via secondary IP addresses on ENI.

The AWS ALB ingress controller allows developers to manage their network needs like their application components. There is no need for other tool sets in the pipeline.

The AWS networking components are tightly integrated with EKS. Understanding the basic options of how they work is fundamental for all those looking to scale their

applications on Kubernetes on AWS using EKS. The size of your subnets, the placements of the nodes in those subnets, and of course the size of nodes will affect how large of a network of pods and services you can run on the AWS network. Using a managed service such as EKS, with open source tools like `eksctl`, will greatly reduce the operational overhead of running an AWS Kubernetes cluster.

Deploying an Application on an AWS EKS Cluster

Let's walk through deploying an EKS cluster to manage our Golang web server:

1. Deploy the EKS cluster.
2. Deploy the web server Application and LoadBalancer.
3. Verify.
4. Deploy ALB Ingress Controller and Verify.
5. Clean up.

Deploy EKS cluster

Let's deploy an EKS cluster, with the current and latest version EKS supports, 1.20:

```
export CLUSTER_NAME=eks-demo
eksctl create cluster -N 3 --name ${CLUSTER_NAME} --version=1.20
eksctl version 0.54.0
using region us-west-2
setting availability zones to [us-west-2b us-west-2a us-west-2c]
subnets for us-west-2b - public:192.168.0.0/19 private:192.168.96.0/19
subnets for us-west-2a - public:192.168.32.0/19 private:192.168.128.0/19
subnets for us-west-2c - public:192.168.64.0/19 private:192.168.160.0/19
nodegroup "ng-90b7a9a5" will use "ami-0a1abe779ecfc6a3e" [AmazonLinux2/1.20]
using Kubernetes version 1.20
creating EKS cluster "eks-demo" in "us-west-2" region with un-managed nodes
will create 2 separate CloudFormation stacks for cluster itself and the initial
nodegroup
if you encounter any issues, check CloudFormation console or try
'eksctl utils describe-stacks --region=us-west-2 --cluster=eks-demo'
CloudWatch logging will not be enabled for cluster "eks-demo" in "us-west-2"
you can enable it with
'eksctl utils update-cluster-logging --enable-types={SPECIFY-YOUR-LOG-TYPES-HERE
(e.g. all)} --region=us-west-2 --cluster=eks-demo'
Kubernetes API endpoint access will use default of
{publicAccess=true, privateAccess=false} for cluster "eks-demo" in "us-west-2"
2 sequential tasks: { create cluster control plane "eks-demo",
3 sequential sub-tasks: { wait for control plane to become ready, 1 task:
{ create addons }, create nodegroup "ng-90b7a9a5" } }
building cluster stack "eksctl-eks-demo-cluster"
deploying stack "eksctl-eks-demo-cluster"
waiting for CloudFormation stack "eksctl-eks-demo-cluster"
```

```
<truncate>
building nodegroup stack "eksctl-eks-demo-nodegroup-ng-90b7a9a5"
--nodes-min=3 was set automatically for nodegroup ng-90b7a9a5
deploying stack "eksctl-eks-demo-nodegroup-ng-90b7a9a5"
waiting for CloudFormation stack "eksctl-eks-demo-nodegroup-ng-90b7a9a5"
<truncated>
waiting for the control plane availability...
saved kubeconfig as "/Users/strongjz/.kube/config"
no tasks
all EKS cluster resources for "eks-demo" have been created
adding identity
"arn:aws:iam::1234567890:role/
eksctl-eks-demo-nodegroup-ng-9-NodeInstanceRole-TLKVDDVTW2TZ" to auth ConfigMap
nodegroup "ng-90b7a9a5" has 0 node(s)
waiting for at least 3 node(s) to become ready in "ng-90b7a9a5"
nodegroup "ng-90b7a9a5" has 3 node(s)
node "ip-192-168-31-17.us-west-2.compute.internal" is ready
node "ip-192-168-58-247.us-west-2.compute.internal" is ready
node "ip-192-168-85-104.us-west-2.compute.internal" is ready
kubectl command should work with "/Users/strongjz/.kube/config",
try 'kubectl get nodes'
EKS cluster "eks-demo" in "us-west-2" region is ready
```

In the output we can see that EKS creating a nodegroup, `eksctl-eks-demo-nodegroup-ng-90b7a9a5`, with three nodes:

```
ip-192-168-31-17.us-west-2.compute.internal
ip-192-168-58-247.us-west-2.compute.internal
ip-192-168-85-104.us-west-2.compute.internal
```

They are all inside a VPC with three public and three private subnets across three AZs:

```
public:192.168.0.0/19 private:192.168.96.0/19
public:192.168.32.0/19 private:192.168.128.0/19
public:192.168.64.0/19 private:192.168.160.0/19
```

We used the default settings of eksctl, and it deployed the k8s API as a public endpoint, `{publicAccess=true, privateAccess=false}`.

Now we can deploy our Golang web application in the cluster and expose it with a LoadBalancer service.

Deploy test application

You can deploy applications individually or all together. *dnsutils.yml* is our `dnsutils` testing pod, *database.yml* is the Postgres database for pod connectivity testing, *web.yml* is the Golang web server and the LoadBalancer service:

```
kubectl apply -f dnsutils.yml,database.yml,web.yml
```

Let's run a `kubectl get pods` to see if all the pods are running fine:

```
kubectl get pods -o wide
NAME                   READY   STATUS    IP               NODE
app-6bf97c555d-5mzfb   1/1     Running   192.168.15.108   ip-192-168-0-94
app-6bf97c555d-76fgm   1/1     Running   192.168.52.42    ip-192-168-63-151
app-6bf97c555d-gw4k9   1/1     Running   192.168.88.61    ip-192-168-91-46
dnsutils               1/1     Running   192.168.57.174   ip-192-168-63-151
postgres-0             1/1     Running   192.168.70.170   ip-192-168-91-46
```

Now check on the LoadBalancer service:

```
kubectl get svc clusterip-service
NAME                TYPE           CLUSTER-IP
EXTERNAL-IP                                                                  PORT(S)        AGE
clusterip-service   LoadBalancer   10.100.159.28
a76d1c69125e543e5b67c899f5e45284-593302470.us-west-2.elb.amazonaws.com   80:32671/TCP   29m
```

The service has endpoints as well:

```
kubectl get endpoints clusterip-service
NAME                ENDPOINTS                                            AGE
clusterip-service   192.168.15.108:8080,192.168.52.42:8080,192.168.88.61:8080   58m
```

We should verify the application is reachable inside the cluster, with the ClusterIP and port, `10.100.159.28:8080`; service name and port, `clusterip-service:80`; and finally pod IP and port, `192.168.15.108:8080`:

```
kubectl exec dnsutils -- wget -qO- 10.100.159.28:80/data
Database Connected

kubectl exec dnsutils -- wget -qO- 10.100.159.28:80/host
NODE: ip-192-168-63-151.us-west-2.compute.internal, POD IP:192.168.52.42

kubectl exec dnsutils -- wget -qO- clusterip-service:80/host
NODE: ip-192-168-91-46.us-west-2.compute.internal, POD IP:192.168.88.61

kubectl exec dnsutils -- wget -qO- clusterip-service:80/data
Database Connected

kubectl exec dnsutils -- wget -qO- 192.168.15.108:8080/data
Database Connected

kubectl exec dnsutils -- wget -qO- 192.168.15.108:8080/host
NODE: ip-192-168-0-94.us-west-2.compute.internal, POD IP:192.168.15.108
```

The database port is reachable from `dnsutils`, with the pod IP and port `192.168.70.170:5432`, and the service name and port - `postgres:5432`:

```
kubectl exec dnsutils -- nc -z -vv -w 5 192.168.70.170 5432
192.168.70.170 (192.168.70.170:5432) open
sent 0, rcvd 0
```

```
kubectl exec dnsutils -- nc -z -vv -w 5 postgres 5432
postgres (10.100.106.134:5432) open
sent 0, rcvd 0
```

The application inside the cluster is up and running. Let's test it from external to the cluster.

Verify LoadBalancer services for Golang web server

`kubectl` will return all the information we will need to test, the ClusterIP, the external IP, and all the ports:

```
kubectl get svc clusterip-service
NAME                TYPE           CLUSTER-IP
EXTERNAL-IP                                                                PORT(S)        AGE
clusterip-service   LoadBalancer   10.100.159.28
a76d1c69125e543e5b67c899f5e45284-593302470.us-west-2.elb.amazonaws.com   80:32671/TCP   29m
```

Using the external IP of the load balancer:

```
wget -qO-
a76d1c69125e543e5b67c899f5e45284-593302470.us-west-2.elb.amazonaws.com/data
Database Connected
```

Let's test the load balancer and make multiple requests to our backends:

```
wget -qO-
a76d1c69125e543e5b67c899f5e45284-593302470.us-west-2.elb.amazonaws.com/host
NODE: ip-192-168-63-151.us-west-2.compute.internal, POD IP:192.168.52.42

wget -qO-
a76d1c69125e543e5b67c899f5e45284-593302470.us-west-2.elb.amazonaws.com/host
NODE: ip-192-168-91-46.us-west-2.compute.internal, POD IP:192.168.88.61

wget -qO-
a76d1c69125e543e5b67c899f5e45284-593302470.us-west-2.elb.amazonaws.com/host
NODE: ip-192-168-0-94.us-west-2.compute.internal, POD IP:192.168.15.108

wget -qO-
a76d1c69125e543e5b67c899f5e45284-593302470.us-west-2.elb.amazonaws.com/host
NODE: ip-192-168-0-94.us-west-2.compute.internal, POD IP:192.168.15.108
```

`kubectl get pods -o wide` again will verify our pod information matches the load-balancer requests:

```
kubectl get pods -o wide
NAME                  READY   STATUS    IP              NODE
app-6bf97c555d-5mzfb  1/1     Running   192.168.15.108  ip-192-168-0-94
app-6bf97c555d-76fgm  1/1     Running   192.168.52.42   ip-192-168-63-151
app-6bf97c555d-gw4k9  1/1     Running   192.168.88.61   ip-192-168-91-46
dnsutils              1/1     Running   192.168.57.174  ip-192-168-63-151
postgres-0            1/1     Running   192.168.70.170  ip-192-168-91-46
```

We can also check the nodeport, since `dnsutils` is running inside our VPC, on an EC2 instance; it can do a DNS lookup on the private host, `ip-192-168-0-94.us-west-2.compute.internal`, and the `kubectl get service` command gave us the node port, 32671:

```
kubectl exec dnsutils -- wget -qO-
ip-192-168-0-94.us-west-2.compute.internal:32671/host
NODE: ip-192-168-0-94.us-west-2.compute.internal, POD IP:192.168.15.108
```

Everything seems to running just fine externally and locally in our cluster.

Deploy ALB ingress and verify

For some sections of the deployment, we will need to know the AWS account ID we are deploying. Let's put that into an environment variable. To get your account ID, you can run the following:

```
aws sts get-caller-identity
{
    "UserId": "AIDA2RZMTHAQTEUI3Z537",
    "Account": "1234567890",
    "Arn": "arn:aws:iam::1234567890:user/eks"
}

export ACCOUNT_ID=1234567890
```

If it is not set up for the cluster already, we will have to set up an OIDC provider with the cluster.

This step is needed to give IAM permissions to a pod running in the cluster using the IAM for SA:

```
eksctl utils associate-iam-oidc-provider \
--region ${AWS_REGION} \
--cluster ${CLUSTER_NAME}  \
--approve
```

For the SA role, we will need to create an IAM policy to determine the permissions for the ALB controller in AWS:

```
aws iam create-policy \
--policy-name AWSLoadBalancerControllerIAMPolicy \
--policy-document iam_policy.json
```

Now we need to create the SA and attach it to the IAM role we created:

```
eksctl create iamserviceaccount \
> --cluster ${CLUSTER_NAME} \
> --namespace kube-system \
> --name aws-load-balancer-controller \
> --attach-policy-arn
arn:aws:iam::${ACCOUNT_ID}:policy/AWSLoadBalancerControllerIAMPolicy \
> --override-existing-serviceaccounts \
```

```
> --approve
eksctl version 0.54.0
using region us-west-2
1 iamserviceaccount (kube-system/aws-load-balancer-controller) was included
(based on the include/exclude rules)
metadata of serviceaccounts that exist in Kubernetes will be updated,
as --override-existing-serviceaccounts was set
1 task: { 2 sequential sub-tasks: { create IAM role for serviceaccount
"kube-system/aws-load-balancer-controller", create serviceaccount
"kube-system/aws-load-balancer-controller" } }
building iamserviceaccount stack
deploying stack
waiting for CloudFormation stack
waiting for CloudFormation stack
waiting for CloudFormation stack
created serviceaccount "kube-system/aws-load-balancer-controller"
```

We can see all the details of the SA with the following:

```
kubectl get sa aws-load-balancer-controller -n kube-system -o yaml
apiVersion: v1
kind: ServiceAccount
metadata:
annotations:
eks.amazonaws.com/role-arn:
arn:aws:iam::1234567890:role/eksctl-eks-demo-addon-iamserviceaccount-Role1
creationTimestamp: "2021-06-27T18:40:06Z"
labels:
app.kubernetes.io/managed-by: eksctl
name: aws-load-balancer-controller
namespace: kube-system
resourceVersion: "16133"
uid: 30281eb5-8edf-4840-bc94-f214c1102e4f
secrets:
- name: aws-load-balancer-controller-token-dtq48
```

The `TargetGroupBinding` CRD allows the controller to bind a Kubernetes service endpoint to an AWS `TargetGroup`:

```
kubectl apply -f crd.yml
customresourcedefinition.apiextensions.k8s.io/ingressclassparams.elbv2.k8s.aws
configured
customresourcedefinition.apiextensions.k8s.io/targetgroupbindings.elbv2.k8s.aws
configured
```

Now we're ready to the deploy the ALB controller with Helm.

Set the version environment to deploy:

```
export ALB_LB_VERSION="v2.2.0"
```

Now deploy it, add the `eks` Helm repo, get the VPC ID the cluster is running in, and finally deploy via Helm.

```
helm repo add eks https://aws.github.io/eks-charts

export VPC_ID=$(aws eks describe-cluster \
--name ${CLUSTER_NAME} \
--query "cluster.resourcesVpcConfig.vpcId" \
--output text)

helm upgrade -i aws-load-balancer-controller \
eks/aws-load-balancer-controller \
-n kube-system \
--set clusterName=${CLUSTER_NAME} \
--set serviceAccount.create=false \
--set serviceAccount.name=aws-load-balancer-controller \
--set image.tag="${ALB_LB_VERSION}" \
--set region=${AWS_REGION} \
--set vpcId=${VPC_ID}

Release "aws-load-balancer-controller" has been upgraded. Happy Helming!
NAME: aws-load-balancer-controller
LAST DEPLOYED: Sun Jun 27 14:43:06 2021
NAMESPACE: kube-system
STATUS: deployed
REVISION: 2
TEST SUITE: None
NOTES:
AWS Load Balancer controller installed!
```

We can watch the deploy logs here:

```
kubectl logs -n kube-system -f deploy/aws-load-balancer-controller
```

Now to deploy our ingress with ALB:

```
kubectl apply -f alb-rules.yml
ingress.networking.k8s.io/app configured
```

With the `kubectl describe ing app` output, we can see the ALB has been deployed.

We can also see the ALB public DNS address, the rules for the instances, and the endpoints backing the service.

```
kubectl describe ing app
Name:             app
Namespace:        default
Address:
k8s-default-app-d5e5a26be4-2128411681.us-west-2.elb.amazonaws.com
Default backend:  default-http-backend:80
(<error: endpoints "default-http-backend" not found>)
Rules:
Host        Path  Backends
  ----        ----  --------
*
          /data   clusterip-service:80 (192.168.3.221:8080,
192.168.44.165:8080,
```

```
192.168.89.224:8080)
          /host   clusterip-service:80 (192.168.3.221:8080,
192.168.44.165:8080,
192.168.89.224:8080)
Annotations:  alb.ingress.kubernetes.io/scheme: internet-facing
kubernetes.io/ingress.class: alb
Events:
Type     Reason                  Age                    From
Message
----     ------                  ----                   ----
-------
Normal   SuccessfullyReconciled  4m33s (x2 over 5m58s)  ingress
Successfully reconciled
```

It's time to test our ALB!

```
wget -qO- k8s-default-app-d5e5a26be4-2128411681.us-west-2.elb.amazonaws.com/data
Database Connected

wget -qO- k8s-default-app-d5e5a26be4-2128411681.us-west-2.elb.amazonaws.com/host
NODE: ip-192-168-63-151.us-west-2.compute.internal, POD IP:192.168.44.165
```

Cleanup

Once you are done working with EKS and testing, make sure to delete the applications pods and the service to ensure that everything is deleted:

```
kubectl delete -f dnsutils.yml,database.yml,web.yml
```

Clean up the ALB:

```
kubectl delete -f alb-rules.yml
```

Remove the IAM policy for ALB controller:

```
aws iam  delete-policy
--policy-arn arn:aws:iam::${ACCOUNT_ID}:policy/AWSLoadBalancerControllerIAMPolicy
```

Verify there are no leftover EBS volumes from the PVCs for test application. Delete any EBS volumes found for the PVC's for the Postgres test database:

```
aws ec2 describe-volumes --filters
Name=tag:kubernetes.io/created-for/pv/name,Values=*
--query "Volumes[].{ID:VolumeId}"
```

Verify there are no load balancers running, ALB or otherwise:

```
aws elbv2 describe-load-balancers --query "LoadBalancers[].LoadBalancerArn"

aws elb describe-load-balancers --query "LoadBalancerDescriptions[].DNSName"
```

Let's make sure we delete the cluster, so you don't get charged for a cluster doing nothing:

```
eksctl delete cluster --name ${CLUSTER_NAME}
```

We deployed a service load balancer that will for each service deploy a classical ELB into AWS. The ALB controller allows developers to use ingress with ALB or NLBs to expose the application externally. If we were to scale our application to multiple backend services, the ingress allows us to use one load balancer and route based on layer 7 information.

In the next section, we will explore GCP in the same manner we just did for AWS.

Google Compute Cloud (GCP)

In 2008, Google announced App Engine, a platform as a service to deploy Java, Python, Ruby, and Go applications. Like its competitors, GCP has extended its service offerings. Cloud providers work to distinguish their offerings, so no two products are ever the same. Nonetheless, many products do have a lot in common. For instance, GCP Compute Engine is an infrastructure as a service to run virtual machines. The GCP network consists of 25 cloud regions, 76 zones, and 144 network edge locations. Utilizing both the scale of the GCP network and Compute Engine, GCP has released Google Kubernetes Engine, its container as a service platform.

GCP Network Services

Managed and unmanaged Kubernetes clusters on GCP share the same networking principles. Nodes in either managed or unmanaged clusters run as Google Compute Engine instances. Networks in GCP are VPC networks. GCP VPC networks, like in AWS, contain functionality for IP management, routing, firewalling, and peering.

The GCP network is divided into tiers for customers to choose from; there are premium and standard tiers. They differ in performance, routing, and functionality, so network engineers must decide which is suitable for their workloads. The premium tier is the highest performance for your workloads. All the traffic between the internet and instances in the VPC network is routed within Google's network as far as possible. If your services need global availability, you should use premium. Make sure to remember that the premium tier is the default unless you make configuration changes.

The standard tier is a cost-optimized tier where traffic between the internet and VMs in the VPC network is routed over the internet in general. Network engineers should pick this tier for services that are going to be hosted entirely within a region. The standard tier cannot guarantee performance as it is subject to the same performance that all workloads share on the internet.

The GCP network differs from the other providers by having what is called *global* resources. Global because users can access them in any zone within the same project. These resources include such things as VPC, firewalls, and their routes.

See the GCP documentation for a more comprehensive overview of the network tiers.

Regions and zones

Regions are independent geographic areas that contain multiple zones. Regional resources offer redundancy by being deployed across multiple zones for that region. Zones are deployment areas for resources within a region. One zone is typically a data center within a region, and administrators should consider them a single fault domain. In fault-tolerant application deployments, the best practice is to deploy applications across multiple zones within a region, and for high availability, you should deploy applications across various regions. If a zone becomes unavailable, all the zone resources will be unavailable until owners restore services.

Virtual private cloud

A VPC is a virtual network that provides connectivity for resources within a GCP project. Like accounts and subscriptions, projects can contain multiple VPC networks, and by default, new projects start with a default auto-mode VPC network that also includes one subnet in each region. Custom-mode VPC networks can contain no subnets. As stated earlier, VPC networks are global resources and are not associated with any particular region or zone.

A VPC network contains one or more regional subnets. Subnets have a region, CIDR, and globally unique name. You can use any CIDR for a subnet, including one that overlaps with another private address space. The specific choice of subnet CIDR impacts which IP addresses you can reach and which networks you can peer.

Google creates a "default" VPC network, with randomly generated subnets for each region. Some subnets may overlap with another VPC's subnet (such as the default VPC network in another Google Cloud project), which will prevent peering.

VPC networks support peering and shared VPC configuration. Peering a VPC network allows the VPC in one project to route to the VPC in another, placing them on the same L3 network. You cannot peer with any overlapping VPC network, as some IP addresses exist in both networks. A shared VPC allows another project to use specific subnets, such as creating machines that are part of that subnet. The VPC documentation has more information.

Peering VPC networks is standard, as organizations often assign different teams, applications, or components to their project in Google Cloud. Peering has upsides for access control, quota, and reporting. Some admins may also create multiple VPC networks within a project for similar reasons.

Subnet

Subnets are portions within a VPC network with one primary IP range with the ability to have zero or more secondary ranges. Subnets are regional resources, and each subnet defines a range of IP addresses. A region can have more than one subnet. There are two modes of subnet formulation when you create them: auto or custom. When you create an auto-mode VPC network, one subnet from each region is automatically created within it using predefined IP ranges. When you define a custom-mode VPC network, GCP does not provision any subnets, giving administrators control over the ranges. Custom-mode VPC networks are suited for enterprises and production environments for network engineers to use.

Google Cloud allows you to "reserve" static IP addresses for internal and external IP addresses. Users can utilize reserved IP addresses for GCE instances, load balancers, and other products beyond our scope. Reserved internal IP addresses have a name and can be generated automatically or assigned manually. Reserving an internal static IP address prevents it from being randomly automatically assigned while not in use.

Reserving external IP addresses is similar; although you can request an automatically assigned IP address, you cannot choose what IP address to reserve. Because you are reserving a globally routable IP address, charges apply in some circumstances. You cannot secure an external IP address that you were assigned automatically as an ephemeral IP address.

Routes and firewall rules

When deploying a VPC, you can use firewall rules to allow or deny connections to and from your application instances based on the rules you deploy. Each firewall rule can apply to ingress or egress connections, but not both. The instance level is where GCP enforces rules, but the configuration pairs with the VPC network, and you cannot share firewall rules among VPC networks, peered networks included. VPC firewall rules are stateful, so when a TCP session starts, firewall rules allow bidirectional traffic similar to an AWS security group.

Cloud load balancing

Google Cloud Load Balancer (GCLB) offers a fully distributed, high-performance, scalable load balancing service across GCP, with various load balancer options. With GCLB, you get a single Anycast IP that fronts all your backend instances across the globe, including multiregion failover. In addition, software-defined load balancing

services enable you to apply load balancing to your HTTP(S), TCP/SSL, and UDP traffic. You can also terminate your SSL traffic with an SSL proxy and HTTPS load balancing. Internal load balancing enables you to build highly available internal services for your internal instances without requiring any load balancers to be exposed to the internet.

The vast majority of GCP users make use of GCP's load balancers with Kubernetes ingress. GCP has internal-facing and external-facing load balancers, with L4 and L7 support. GKE clusters default to creating a GCP load balancer for ingresses and `type: LoadBalancer` services.

To expose applications outside a GKE cluster, GKE provides a built-in GKE ingress controller and GKE service controller, which deploys a Google Cloud load balancer on behalf of GKE users. GKE provides three different load balancers to control access and spread incoming traffic across your cluster as evenly as possible. You can configure one service to use multiple types of load balancers simultaneously:

External load balancers
: Manage traffic from outside the cluster and outside the VPC network. External load balancers use forwarding rules associated with the Google Cloud network to route traffic to a Kubernetes node.

Internal load balancers
: Manage traffic coming from within the same VPC network. Like external load balancers, internal ones use forwarding rules associated with the Google Cloud network to route traffic to a Kubernetes node.

HTTP load balancers
: Specialized external load balancers used for HTTP traffic. They use an ingress resource rather than a forwarding rule to route traffic to a Kubernetes node.

When you create an ingress object, the GKE ingress controller configures a Google Cloud HTTP(S) load balancer according to the ingress manifest and the associated Kubernetes service rules manifest. The client sends a request to the load balancer. The load balancer is a proxy; it chooses a node and forwards the request to that node's NodeIP:NodePort combination. The node uses its `iptables` NAT table to select a pod. As we learned in earlier chapters, `kube-proxy` manages the `iptables` rules on that node.

When an ingress creates a load balancer, the load balancer is "pod aware" instead of routing to all nodes (and relying on the service to route requests to a pod), and the load balancer routes to individual pods. It does this by tracking the underlying `Endpoints/EndpointSlice` object (as covered in Chapter 5) and using individual pod IP addresses as target addresses.

Cluster administrators can use an in-cluster ingress provider, such as ingress-Nginx or Contour. A load balancer points to applicable nodes running the ingress proxy in such a setup, which routes requests to the applicable pods from there. This setup is cheaper for clusters that have many ingresses but incurs performance overhead.

GCE instances

GCE instances have one or more network interfaces. A network interface has a network and subnetwork, a private IP address, and a public IP address. The private IP address must be part of the subnetwork. Private IP addresses can be automatic and ephemeral, custom and ephemeral, or static. External IP addresses can be automatic and ephemeral, or static. You can add more network interfaces to a GCE instance. Additional network interfaces don't need to be in the same VPC network. For example, you may have an instance that bridges two VPCs with varying levels of security. Let's discuss how GKE uses these instances and manages the network services that empower GKE.

GKE

Google Kubernetes Engine (GKE) is Google's managed Kubernetes service. GKE runs a hidden control plane, which cannot be directly viewed or accessed. You can only access specific control plane configurations and the Kubernetes API.

GKE exposes broad cluster config around things like machine types and cluster scaling. It reveals only some network-related settings. At the time of writing, NetworkPolicy support (via Calico), max pods per node (`maxPods` in the kubelet, `--node-CIDR-mask-size` in `kube-controller-manager`), and the pod address range (`--cluster-CIDR` in `kube-controller-manager`) are the customizable options. It is not possible to directly set `apiserver/kube-controller-manager` flags.

GKE supports public and private clusters. Private clusters don't issue public IP addresses to nodes, which means nodes are accessible only within your private network. Private clusters also allow you to restrict access to the Kubernetes API to specific IP addresses. GKE runs worker nodes using automatically managed GCE instances by creating creates *node pools*.

GCP GKE nodes

Networking for GKE nodes is comparable to networking for self-managed Kubernetes clusters on GKE. GKE clusters define *node pools,* which are a set of nodes with an identical configuration. This configuration contains GCE-specific settings as well as general Kubernetes settings. Node pools define (virtual) machine type, autoscaling, and the GCE service account. You can also set custom taints and labels per node pool.

A cluster exists on exactly one VPC network. Individual nodes can have their network tags for crafting specific firewall rules. Any GKE cluster running 1.16 or later will have a `kube-proxy` DaemonSet so that all new nodes in the cluster will automatically have the `kube-proxy` start. The size of the subnet allows will affect the size of the cluster. So, pay attention to the size of that when you deploy clusters that scale. There is a formula you can use to calculate the maximum number of nodes, `N`, that a given netmask can support. Use `S` for the netmask size, whose valid range is between 8 and 29:

```
N = 2(32 -S) - 4
```

Calculate the size of the netmask, `S`, required to support a maximum of `N` nodes:

```
S = 32 - ⌈log2(N + 4)⌉
```

Table 6-2 also outlines cluster node and how it scales with subnet size.

Table 6-2. Cluster node scale with subnet size

Subnet primary IP range	Maximum nodes
/29	Minimum size for a subnet's primary IP range: 4 nodes
/28	12 nodes
/27	28 nodes
/26	60 nodes
/25	124 nodes
/24	252 nodes
/23	508 nodes
/22	1,020 nodes
/21	2,044 nodes
/20	The default size of a subnet's primary IP range in auto mode networks: 4,092 nodes
/19	8,188 nodes
/8	Maximum size for a subnet's primary IP range: 16,777,212 nodes

If you use GKE's CNI, one end of the veth pair is attached to the pod in its namespace and connects the other side to the Linux bridge device cbr0.1, exactly how we outlined it in Chapters 2 and 3].

Clusters span either the zone or region boundary; zonal clusters have only a single control plane. Regional clusters have multiple replicas of the control plane. Also, when you deploy clusters, there are two cluster modes with GKE: VPC-native and routes based. A cluster that uses alias IP address ranges is considered a VPC-native cluster. A cluster that uses custom static routes in a VPC network is called a *routes-based cluster*. Table 6-3 outlines how the creation method maps with the cluster mode.

Table 6-3. Cluster mode with cluster creation method

Cluster creation method	Cluster network mode
Google Cloud Console	VPC-native
REST API	Routes-based
gcloud v256.0.0 and higher or v250.0.0 and lower	Routes-based
gcloud v251.0.0–255.0.0	VPC-native

When using VPC-native, administrators can also take advantage of network endpoint groups (NEG), which represent a group of backends served by a load balancer. NEGs are lists of IP addresses managed by an NEG controller and are used by Google Cloud load balancers. IP addresses in an NEG can be primary or secondary IP addresses of a VM, which means they can be pod IPs. This enables container-native load balancing that sends traffic directly to pods from a Google Cloud load balancer.

VPC-native clusters have several benefits:

- Pod IP addresses are natively routable inside the cluster's VPC network.
- Pod IP addresses are reserved in network before pod creation.
- Pod IP address ranges are dependent on custom static routes.
- Firewall rules apply to just pod IP address ranges instead of any IP address on the cluster's nodes.
- GCP cloud network connectivity to on-premise extends to pod IP address ranges.

Figure 6-15 shows the mapping of GKE to GCE components.

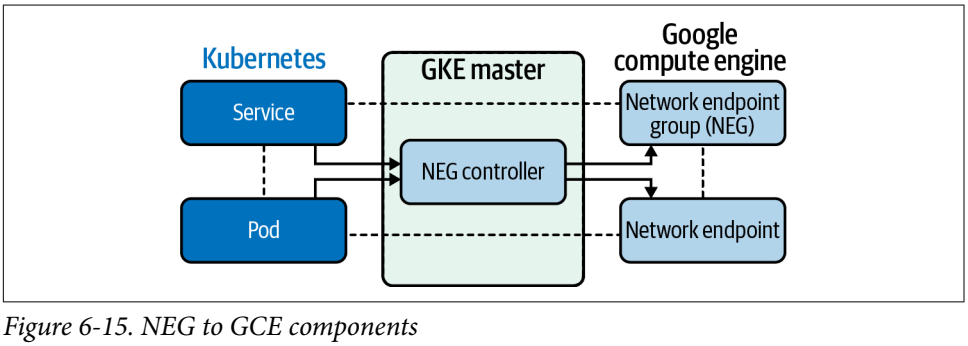

Figure 6-15. NEG to GCE components

Here is a list of improvements that NEGs bring to the GKE network:

Improved network performance
: The container-native load balancer talks directly with the pods, and connections have fewer network hops; both latency and throughput are improved.

Increased visibility
: With container-native load balancing, you have visibility into the latency from the HTTP load balancer to the pods. The latency from the HTTP load balancer to each pod is visible, which was aggregated with node IP-based container-native load balancing. This increased visibility makes troubleshooting your services at the NEG level easier.

Support for advanced load balancing
: Container-native load balancing offers native support in GKE for several HTTP load-balancing features, such as integration with Google Cloud services like Google Cloud Armor, Cloud CDN, and Identity-Aware Proxy. It also features load-balancing algorithms for accurate traffic distribution.

Like most managed Kubernetes offerings from major providers, GKE is tightly integrated with Google Cloud offerings. Although much of the software driving GKE is opaque, it uses standard resources such as GCE instances that can be inspected and debugged like any other GCP resources. If you really need to manage your own clusters, you will lose out on some functionality, such as container-aware load balancing.

It's worth noting that GCP does not yet support IPv6, unlike AWS and Azure.

Finally, we'll look at Kubernetes networking on Azure.

Azure

Microsoft Azure, like other cloud providers, offers an assortment of enterprise-ready network solutions and services. Before we can discuss how Azure AKS networking works, we should discuss Azure deployment models. Azure has gone through some significant iterations and improvements over the years, resulting in two different deployment models that can encounter Azure. These models differ in how resources are deployed and managed and may impact how users leverage the resources.

The first deployment model was the classic deployment model. This model was the initial deployment and management method for Azure. All resources existed independently of each other, and you could not logically group them. This was cumbersome; users had to create, update, and delete each component of a solution, leading to errors, missed resources, and additional time, effort, and cost. Finally, these resources could not even be tagged for easy searching, adding to the difficulty of the solution.

In 2014, Microsoft introduced the Azure Resource Manager as the second model. This new model is the recommended model from Microsoft, with the recommendation going so far as to say that you should redeploy your resources using the Azure Resource Manager (ARM). The primary change with this model was the introduction of the resource group. Resource groups are a logical grouping of resources that allows

for tracking, tagging, and configuring the resources as a group rather than individually.

Now that we understand the basics of how resources are deployed and managed in Azure, we can discuss the Azure network service offerings and how they interact with the Azure Kubernetes Service (AKS) and non-Azure Kubernetes offerings.

Azure Networking Services

The core of Azure networking services is the virtual network, also known as an Azure Vnet. The Vnet establishes an isolated virtual network infrastructure to connect your deployed Azure resources such as virtual machines and AKS clusters. Through additional resources, Vnets connect your deployed resources to the public internet as well as your on-premise infrastructure. Unless the configuration is changed, all Azure Vnets can communicate with the internet through a default route.

In Figure 6-16, an Azure Vnet has a single CIDR of `192.168.0.0/16`. Vnets, like other Azure resources, require a subscription to place the Vnet into a resource group for the Vnet. The security of the Vnet can be configured while some options, such as IAM permissions, are inherited from the resource group and the subscription. The Vnet is confined to a specified region. Multiple Vnets can exist within a single region, but a Vnet can exist within only one region.

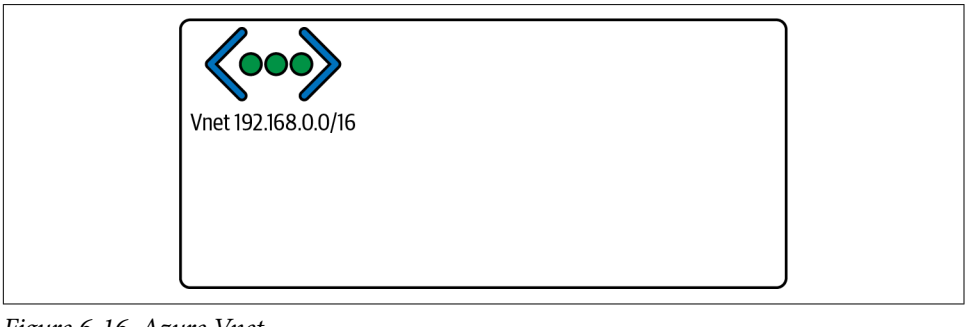

Figure 6-16. Azure Vnet

Azure backbone infrastructure

Microsoft Azure leverages a globally dispersed network of data centers and zones. The foundation of this dispersal is the Azure region, which comprises a set of data centers within a latency-defined area, connected by a low-latency, dedicated network infrastructure. A region can contain any number of data centers that meet these criteria, but two to three are often present per region. Any area of the world containing at least one Azure region is known as Azure geography.

Availability zones further divide a region. Availability zones are physical locations that can consist of one or more data centers maintained by independent power,

cooling, and networking infrastructure. The relationship of a region to its availability zones is architected so that a single availability zone failure cannot bring down an entire region of services. Each availability zone in a region is connected to the other availability zones in the region but not dependent on the different zones. Availability zones allow Azure to offer 99.99% uptime for supported services. A region can consist of multiple availability zones, as shown in Figure 6-17, which can, in turn, consist of numerous data centers.

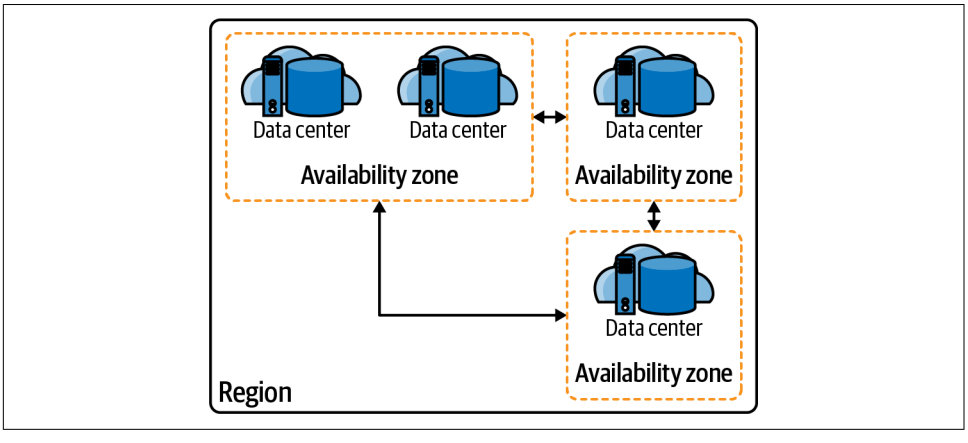

Figure 6-17. Region

Since a Vnet is within a region and regions are divided into availability zones, Vnets are also available across the availability zones of the region they are deployed. As shown in Figure 6-18, it is a best practice when deploying infrastructure for high availability to leverage multiple availability zones for redundancy. Availability zones allow Azure to offer 99.99% uptime for supported services. Azure allows for the use of load balancers for networking across redundant systems such as these.

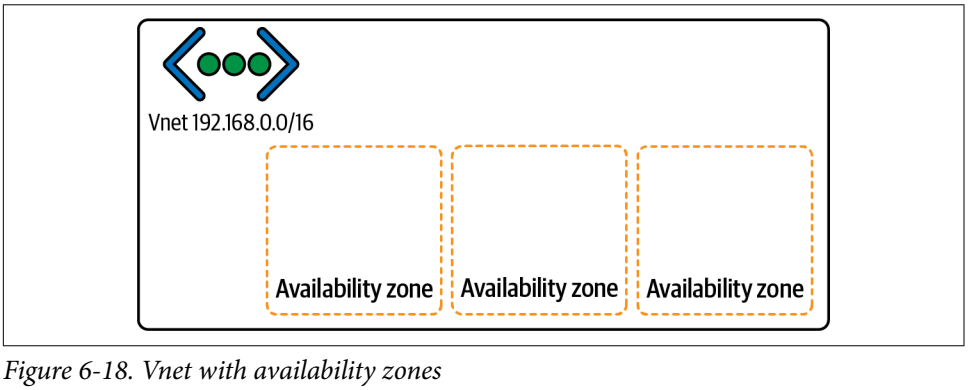

Figure 6-18. Vnet with availability zones

The Azure documentation has an up-to-date list of Azure geographies, regions, and availability zones.

Subnets

Resource IPs are not assigned directly from the Vnet. Instead, subnets divide and define a Vnet. The subnets receive their address space from the Vnet. Then, private IPs are allocated to provisioned resources within each subnet. This is where the IP addressing AKS clusters and pods will come. Like Vnets, Azure subnets span availability zones, as depicted in Figure 6-19.

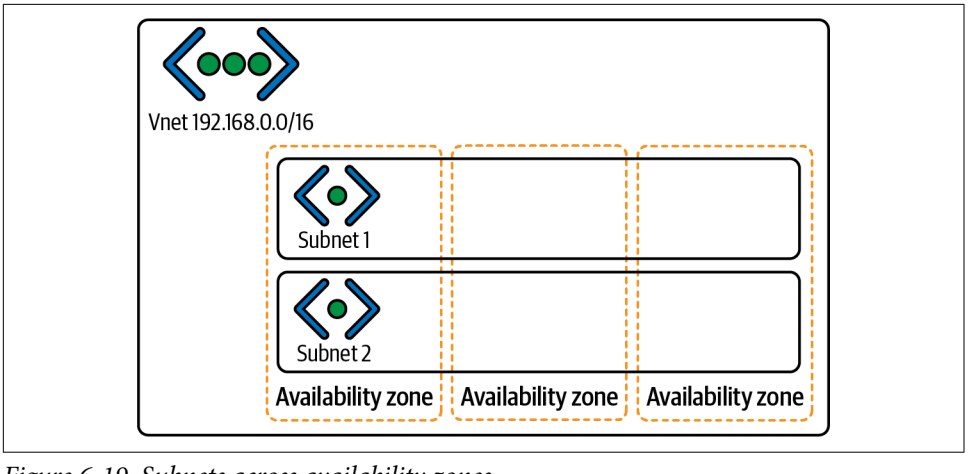

Figure 6-19. Subnets across availability zones

Route tables

As mentioned in previous sections, a route table governs subnet communication or an array of directions on where to send network traffic. Each newly provisioned subnet comes equipped with a default route table populated with some default system routes. This route cannot be deleted or changed. The system routes include a route to the Vnet the subnet is defined within, routes for `10.0.0.0/8` and `192.168.0.0/16` that are by default set to go nowhere, and most importantly a default route to the internet. The default route to the internet allows any newly provisioned resource with an Azure IP to communicate out to the internet by default. This default route is an essential difference between Azure and some other cloud service providers and requires adequate security measures to protect each Azure Vnet.

Figure 6-20 shows a standard route table for a newly provisioned AKS setup. There are routes for the agent pools with their CIDRs as well as their next-hop IP. The next-hop IP is the route the table has defined for the path, and the next-hop type is set for

a virtual appliance, which would be the load balancer in this case. What is not present are those default system routes. The default routes are still in the configuration, just not viewable in the route table. Understanding Azure's default networking behavior is critical from a security perspective and from troubleshooting and planning perspectives.

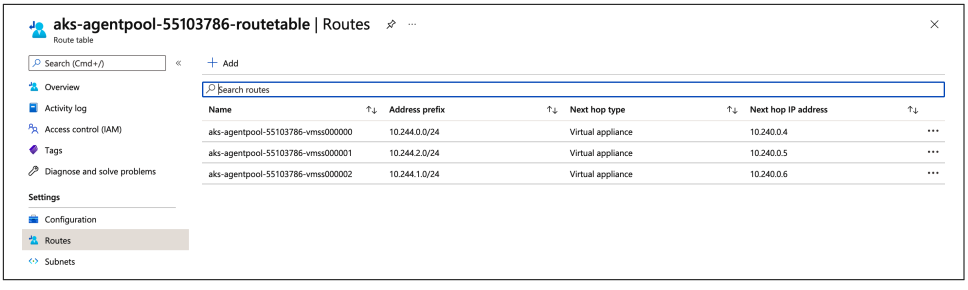

Figure 6-20. Route table

Some system routes, known as optional default routes, affect only if the capabilities, such as Vnet peering, are enabled. Vnet peering allows Vnets anywhere globally to establish a private connection across the Azure global infrastructure backbone to communicate.

Custom routes can also populate route tables, which the Border Gateway Protocol either creates if leveraged or uses user-defined routes. User-defined routes are essential because they allow the network administrators to define routes beyond what Azure establishes by default, such as proxies or firewall routes. Custom routes also impact the system default routes. While you cannot alter the default routes, a customer route with a higher priority can overrule it. An example of this is to use a user-defined route to send traffic bound for the internet to a next-hop of a virtual firewall appliance rather than the internet directly. Figure 6-21 defines a custom route called Google with a next-hop type of internet. As long as the priorities are set up correctly, this custom route will send that traffic out the default system route for the internet, even if another rule redirects the remaining internet traffic.

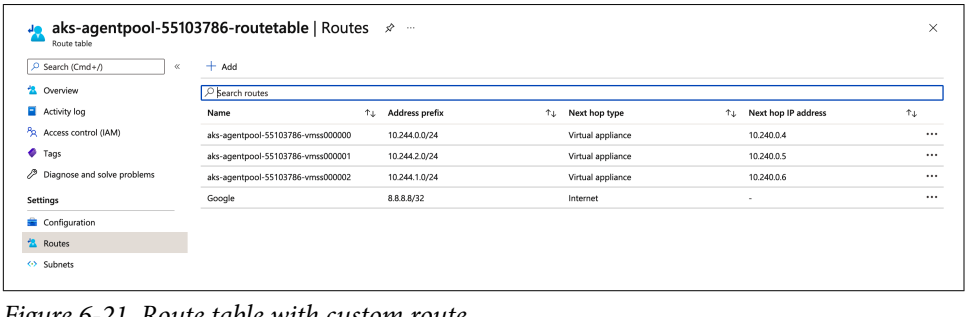

Figure 6-21. Route table with custom route

Route tables can also be created on their own and then used to configure a subnet. This is useful for maintaining a single route table for multiple subnets, especially when there are many user-defined routes involved. A subnet can have only one route table associated with it, but a route table can be associated with multiple subnets. The rules of configuring a user-created route table and a route table created as part of the subnet's default creation are the same. They have the same default system routes and will update with the same optional default routes as they come into effect.

While most routes within a route table will use an IP range as the source address, Azure has begun to introduce the concept of using service tags for sources. A service tag is a phrase that represents a collection of service IPs within the Azure backend, such as SQL.EastUs, which is a service tag that describes the IP address range for the Microsoft SQL Platform service offering in the eastern US. With this feature, it could be possible to define a route from one Azure service, such as AzureDevOps, as the source, and another service, such as Azure AppService, as the destination without knowing the IP ranges for either.

The Azure documentation has a list of available service tags.

Public and private IPs

Azure allocates IP addresses as independent resources themselves, which means that a user can create a public IP or private IP without attaching it to anything. These IP addresses can be named and built in a resource group that allows for future allocation. This is a crucial step when preparing for AKS cluster scaling as you want to make sure that enough private IP addresses have been reserved for the possible pods if you decide to leverage Azure CNI for networking. Azure CNI will be discussed in a later section.

IP address resources, both public and private, are also defined as either dynamic or static. A static IP address is reserved to not change, while a dynamic IP address can change if it is not allocated to a resource, such as a virtual machine or AKS pod.

Network security groups

NSGs are used to configure Vnets, subnets, and network interface cards (NICs) with inbound and outbound security rules. The rules filter traffic and determine whether the traffic will be allowed to proceed or be dropped. NSG rules are flexible to filter traffic based on source and destination IP addresses, network ports, and network protocols. An NSG rule can use one or multiple of these filter items and can apply many NSGs.

An NSG rule can have any of the following components to define its filtering:

Priority
: This is a number between 100 and 4096. The lowest numbers are evaluated first, and the first match is the rule that is used. Once a match is found, no further rules are evaluated.

Source/destination
: Source (inbound rules) or destination (outbound rules) of the traffic inspected. The source/destination can be any of the following:

  - Individual IP address
  - CIDR block (i.e., 10.2.0.0/24)
  - Microsoft Azure service tag
  - Application security groups

Protocol
: TCP, UDP, ICMP, ESP, AH, or Any.

Direction
: The rule for inbound or outbound traffic.

Port range
: Single ports or ranges can be specified here.

Action
: Allow or deny the traffic.

Figure 6-22 shows an example of an NSG.

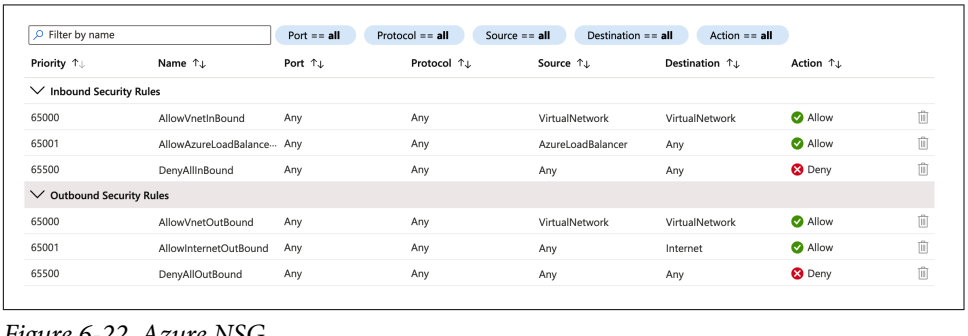

Figure 6-22. Azure NSG

There are some considerations to keep in mind when configuring Azure network security groups. First, two or more rules cannot exist with the same priority and direction. The priority or direction can match as long as the other does not. Second, port ranges can be used only in the Resource Manager deployment model, not the

classic deployment model. This limitation also applies to IP address ranges and service tags for the source/destination. Third, when specifying the IP address for an Azure resource as the source/destination, if the resource has both a public and private IP address, use the private IP address. Azure performs the translation from public to private IP addressing outside this process, and the private IP address will be the right choice at the time of processing.

Communication outside the virtual network

The concepts described so far have mainly pertained to Azure networking within a single Vnet. This type of communication is vital in Azure networking but far from the only type. Most Azure implementations will require communication outside the virtual network to other networks, including, but not limited to, on-premise networks, other Azure virtual networks, and the internet. These communication paths require many of the same considerations as the internal networking processes and use many of the same resources, with a few differences. This section will expand on some of those differences.

Vnet peering can connect Vnets in different regions using global virtual network peering, but there are constraints with certain services such as load balancers.

For a list of these constraints, see the Azure documentation.

Communication outside of Azure to the internet uses a different set of resources. Public IPs, as discussed earlier, can be created and assigned to a resource in Azure. The resource uses its private IP address for all networking internal to Azure. When the traffic from the resource needs to exit the internal networks to the internet, Azure translates the private IP address into the resource's assigned public IP. At this point, the traffic can leave to the internet. Incoming traffic bound for the public IP address of an Azure resource translates to the resource's assigned private IP address at the Vnet boundary, and the private IP is used from then on for the rest of the traffic's trip to its destination. This traffic path is why all subnet rules for things like NSGs are defined using private IP addresses.

NAT can also be configured on a subnet. If configured, resources on a subnet with NAT enabled do not need a public IP address to communicate with the internet. NAT is enabled on a subnet to allow outbound-only internet traffic with a public IP from a pool of provisioned public IP addresses. NAT will enable resources to route traffic to the internet for requests such as updates or installs and return with the requested traffic but prevents the resources from being accessible on the internet. It is important to note that, when configured, NAT takes priority over all other outbound rules and

replaces the default internet destination for the subnet. NAT also uses port address translation (PAT) by default.

Azure load balancer

Now that you have a method of communicating outside the network and communication to flow back into the Vnet, a way to keep those lines of communication available is needed. Azure load balancers are often used to accomplish this by distributing traffic across backend pools of resources rather than a single resource to handle the request. There are two primary load balancer types in Azure: the standard load balancer and the application gateway.

Azure standard load balancers are layer 4 systems that distribute incoming traffic based on layer 4 protocols such as TCP and UDP, meaning traffic is routed based on IP address and port. These load balancers filter incoming traffic from the internet, but they can also load balance traffic from one Azure resource to a set of other Azure resources. The standard load balancer uses a zero-trust network model. This model requires an NSG to "open" traffic to be inspected by the load balancer. If the attached NSG does not permit the traffic, the load balancer will not attempt to route it.

Azure application gateways are similar to standard load balancers in that they distribute incoming traffic but differently in that they do so at layer 7. This allows for the inspection of incoming HTTP requests to filter based on URI or host headers. Application gateways can also be used as web application firewalls to further secure and filter traffic. Additionally, the application gateway can also be used as the ingress controller for AKS clusters.

Load balancers, whether standard or application gateways, have some basic concepts that sound be considered:

Frontend IP address
: Either public or private depending on the use, this is the IP address used to target the load balancer and, by extension, the backend resources it is balancing.

SKU
: Like other Azure resources, this defines the "type" of the load balancer and, therefore, the different configuration options available.

Backend pool
: This is the collection of resources that the load balancer is distributing traffic to, such as a collection of virtual machines or the pods within an AKS cluster.

Health probes
: These are methods used by the load balancer to ensure the backend resource is available for traffic, such as a health endpoint that returns an OK status:

Listener
: A configuration that tells the load balancer what type of traffic to expect, such as HTTP requests.

Rules
: Determines how to route the incoming traffic for that listener.

Figure 6-23 illustrates some of these primary components within the Azure load balancer architecture. Traffic comes into the load balancer and is compared to the listeners to determine if the load balancer balances the traffic. Then the traffic is evaluated against the rules and finally sent on to the backend pool. Backend pool resources with appropriately responding health probes will process the traffic.

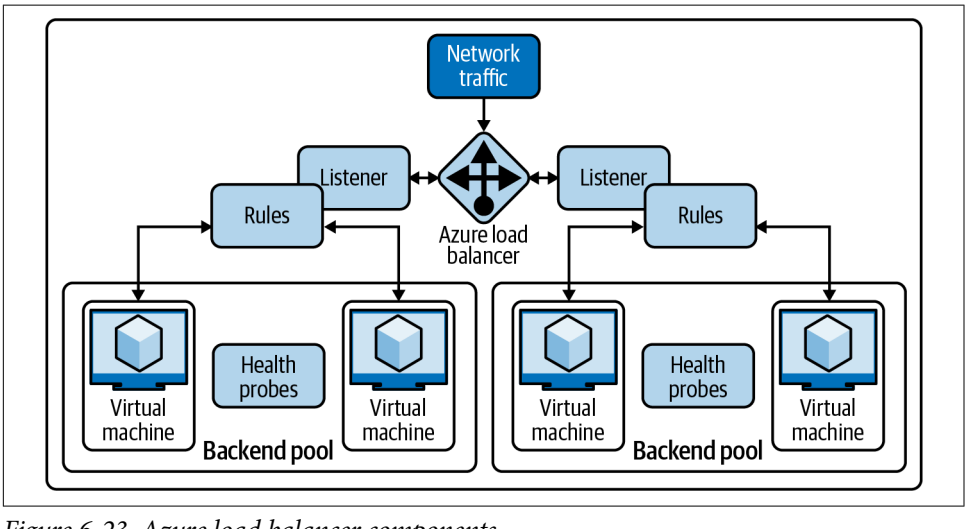

Figure 6-23. Azure load balancer components

Figure 6-24 shows how AKS would use the load balancer.

Now that we have a basic knowledge of the Azure network, we can discuss how Azure uses these constructs in its managed Kubernetes offering, Azure Kubernetes Service.

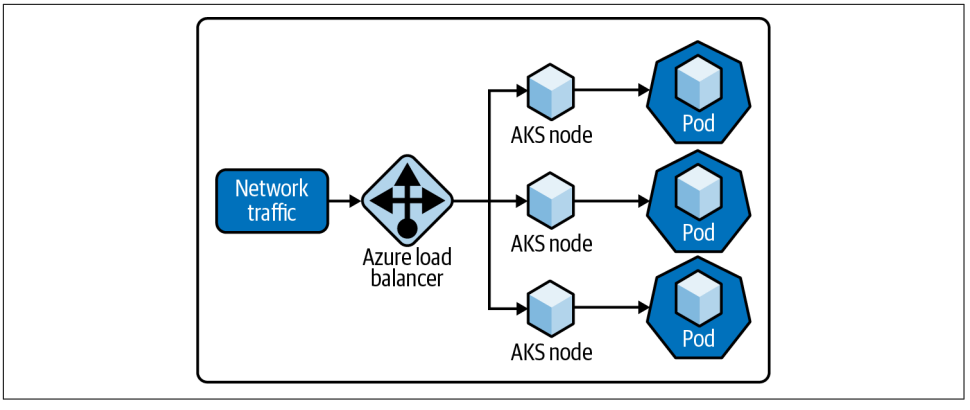

Figure 6-24. AKS load balancing

Azure Kubernetes Service

Like other cloud providers, Microsoft understood the need to leverage the power of Kubernetes and therefore introduced the Azure Kubernetes Service as the Azure Kubernetes offering. AKS is a hosted service offering from Azure and therefore handles a large portion of the overhead of managing Kubernetes. Azure handles components such as health monitoring and maintenance, leaving more time for development and operations engineers to leverage the scalability and power of Kubernetes for their solutions.

AKS can have clusters created and managed using the Azure CLI, Azure PowerShell, the Azure Portal, and other template-based deployment options such as ARM templates and HashiCorp's Terraform. With AKS, Azure manages the Kubernetes masters so that the user only needs to handle the node agents. This allows Azure to offer the core of AKS as a free service where the only payment required is for the agent nodes and peripheral services such as storage and networking.

The Azure Portal allows for easy management and configuration of the AKS environment. Figure 6-25 shows the overview page of a newly provisioned AKS environment. On this page, you can see information and links to many of the crucial integrations and properties. The cluster's resource group, DNS address, Kubernetes version, networking type, and a link to the node pools are visible in the Essentials section.

Figure 6-26 zooms in on the Properties section of the overview page, where users can find additional information and links to corresponding components. Most of the data is the same as the information in the Essentials section. However, the various subnet CIDRs for the AKS environment components can be viewed here for things such as the Docker bridge and the pod subnet.

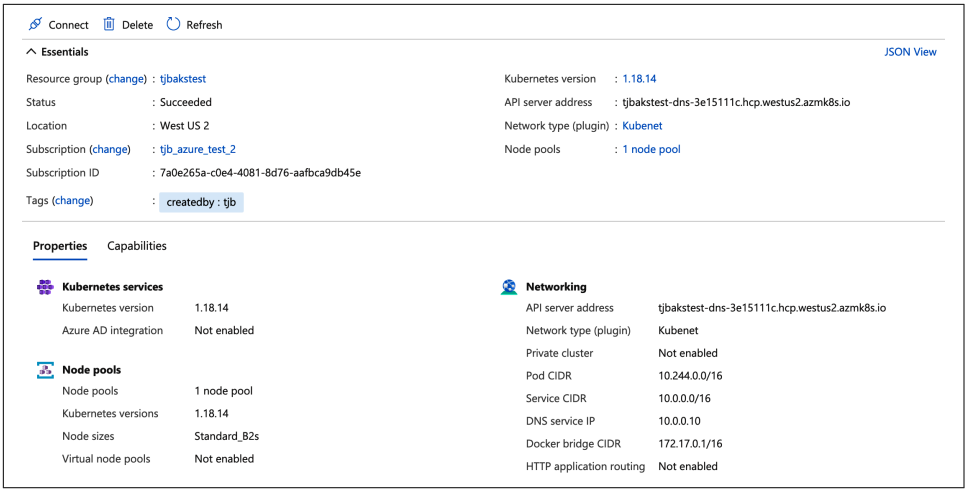

Figure 6-25. Azure Portal AKS overview

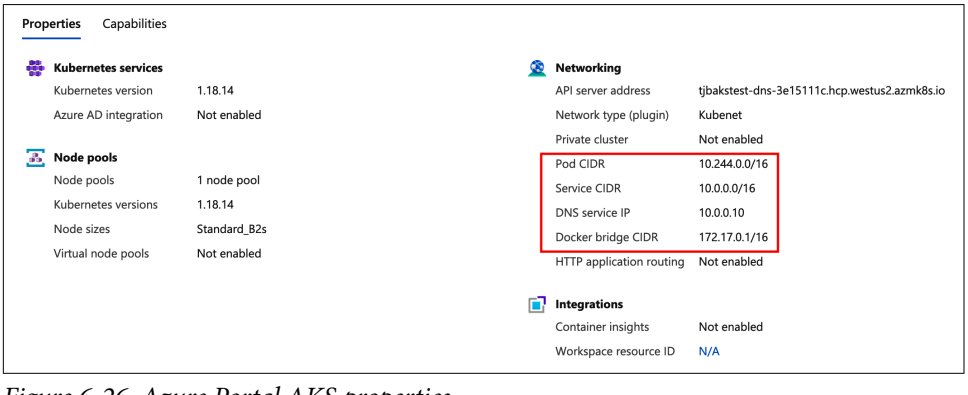

Figure 6-26. Azure Portal AKS properties

Kubernetes pods created within AKS are attached to virtual networks and can access network resources through abstraction. The kube-proxy on each AKS node creates this abstraction, and this component allows for inbound and outbound traffic. Additionally, AKS seeks to make Kubernetes management even more streamlined by simplifying how to roll changes to virtual network changes. Network services in AKS are autoconfigured when specific changes occur. For example, opening a network port to a pod will also trigger relevant changes to the attached NSGs to open those ports.

By default, AKS will create an Azure DNS record that has a public IP. However, the default network rules prevent public access. The private mode can create the cluster to use no public IPs and block public access for only internal use of the cluster. This mode will cause the cluster access to be available only from within the Vnet. By default, the standard SKU will create an AKS load balancer. This configuration can be

changed during deployment if deploying via the CLI. Resources not included in the cluster are made in a separate, auto-generated resource group.

When leveraging the kubenet networking model for AKS, the following rules are true:

- Nodes receive an IP address from the Azure virtual network subnet.
- Pods receive an IP address from a logically different address space than the nodes.
- The source IP address of the traffic switches to the node's primary address.
- NAT is configured for the pods to reach Azure resources on the Vnet.

It is important to note that only the nodes receive a routable IP; the pods do not.

While kubenet is an easy way to administer Kubernetes networking within the Azure Kubernetes Service, it is not the only way. Like other cloud providers, Azure also allows for the use the CNI when managing Kubernetes infrastructure. Let's discuss CNI in the next section.

Azure CNI

Microsoft has provided its own CNI plugin for Azure and AKS, Azure CNI. The first significant difference between this and kubenet is that the pods receive routable IP information and can be accessed directly. This difference places additional importance on the need for IP address space planning. Each node has a maximum number of pods it can use, and many IP addresses are reserved for that use.

More information can be found on the Azure Container Networking GitHub.

With Azure CNI, traffic inside the Vnet is no longer NAT'd to the node's IP address but to the pod's IP itself, as illustrated in Figure 6-27. Outside traffic, such as to the internet, is still NAT'd to the node's IP address. Azure CNI still performs the backend IP address management and routing for these items, though, as all resources on the same Azure Vnet can communicate with each other by default.

The Azure CNI can also be used for Kubernetes deployments outside AKS. While there is additional work to be done on the cluster that Azure would typically handle, this allows you to leverage Azure networking and other resources while maintaining more control over the customarily managed aspects of Kubernetes under AKS.

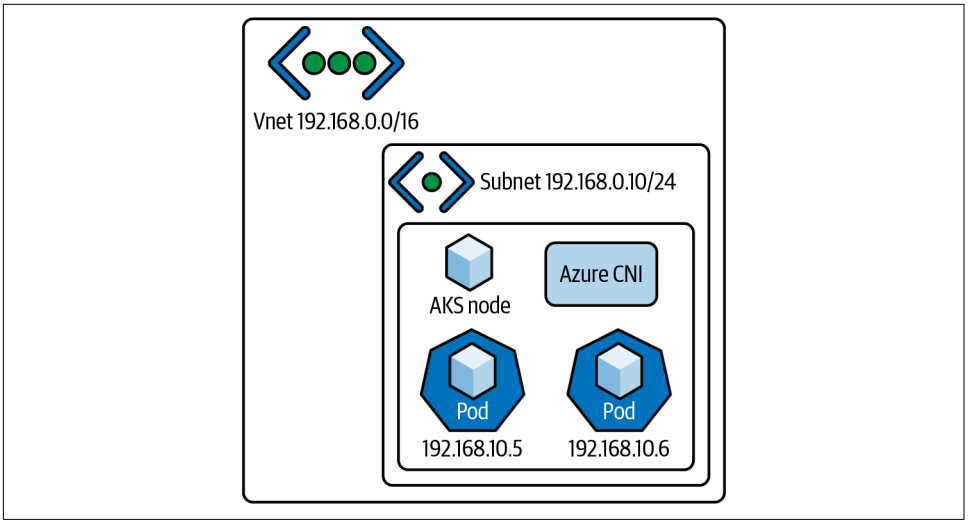

Figure 6-27. Azure CNI

Azure CNI also provides the added benefit of allowing for the separation of duties while maintaining the AKS infrastructure. The Azure CNI creates the networking resources in a separate resource group. Being in a different resource group allows for more control over permissions at the resource group level within the Azure Resource Management deployment model. Different teams can access some components of AKS, such as the networking, without needing access to others, such as the application deployments.

Azure CNI is not the only way to leverage additional Azure services to enhance your Kubernetes network infrastructure. The next section will discuss the use of an Azure application gateway as a means of controlling ingress into your Kubernetes cluster.

Application gateway ingress controller

Azure allows for the deployment of an application gateway inside the AKS cluster deployment to serve as the application gateway ingress controller (AGIC). This deployment model eliminates the need for maintaining a secondary load balancer outside the AKS infrastructure, thereby reducing maintenance overhead and error points. AGIC deploys its pods in the cluster. It then monitors other aspects of the cluster for configuration changes. When a change is detected, AGIC updates the Azure Resource Manager template that configures the load balancer and then applies the updated configuration. Figure 6-28 illustrates this.

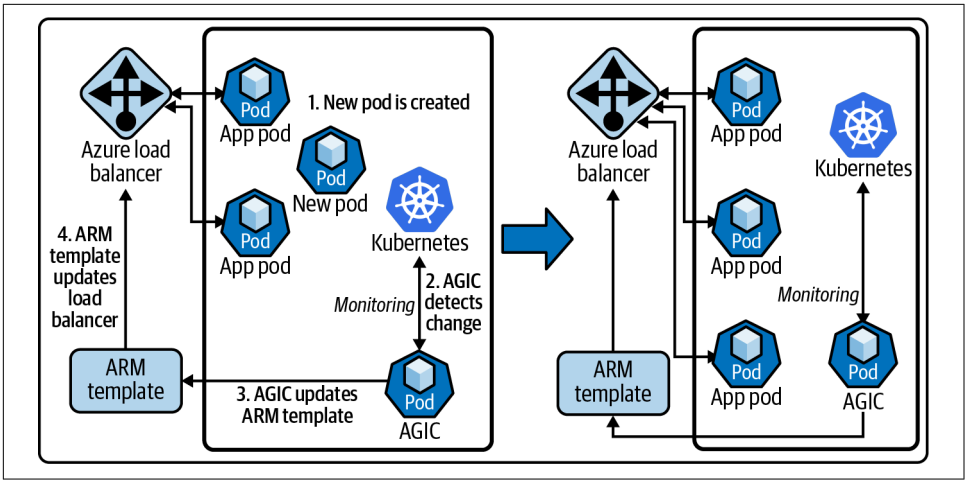

Figure 6-28. Azure AGIC

There are AKS SKU limitations for the use of the AGIC, only supporting Standard_v2 and WAF_v2, but those SKUs also have autoscaling capabilities. Use cases for using such a form of ingress, such as the need for high scalability, have the potential for the AKS environment to scale. Microsoft supports the use of both Helm and the AKS add-on as deployment options for the AGIC. These are the critical differences between the two options:

- Helm deployment values cannot be edited when using the AKS add-on.
- Helm supports Prohibited Target configuration. An AGIC can configure the application gateway to target only the AKS instances without impacting other backend components.
- The AKS add-on, as a managed service, will be automatically updated to its current and more secure versions. Helm deployments will need manual updating.

Even though AGIC is configured as the Kubernetes ingress resource, it still carries the full benefit of the cluster's standard layer 7 application gateway. Application gateway services such as TLS termination, URL routing, and the web application firewall capability are all configurable for the cluster as part of the AGIC.

While many Kubernetes and networking fundamentals are universal across cloud providers, Azure offers its own spin on Kubernetes networking through its enterprise-focused resource design and management. Whether you have a need for a single cluster using basic settings and kubenet or a large-scale deployment with advanced networking through the use of deployed load balancers and application gateways, Microsoft's Azure Kubernetes Service can be leveraged to deliver a reliable, managed Kubernetes infrastructure.

Deploying an Application to Azure Kubernetes Service

Standing up an Azure Kubernetes Service cluster is one of the basic skills needed to begin exploring AKS networking. This section will go through the steps of standing up a sample cluster and deploying the Golang web server example from Chapter 1 to that cluster. We will be using a combination of the Azure Portal, the Azure CLI, and `kubectl` to perform these actions.

Before we begin with the cluster deployment and configuration, we should discuss the Azure Container Registry (ACR). The ACR is where you store container images in Azure. For this example, we will use the ACR as the location for the container image we will be deploying. To import an image to the ACR, you will need to have the image locally available on your computer. Once you have the image available, we have to prep it for the ACR.

First, identify the ACR repository you want to store the image in and log in from the Docker CLI with `docker login <acr_repository>.azurecr.io`. For this example, we will use the ACR repository `tjbakstestcr`, so the command would be `docker login tjbakstestcr.azurecr.io`. Next, tag the local image you wish to import to the ACR with `<acr_repository>.azurecr.io\<imagetag>`. For this example, we will use an image currently tagged `aksdemo`. Therefore, the tag would be `tjbakstestcr.azure.io/aksdemo`. To tag the image, use the command `docker tag <local_image_tag> <acr_image_tag>`. This example would use the command `docker tag aksdemo tjbakstestcr.azure.io/aksdem`. Finally, we push the image to the ACR with `docker push tjbakstestcr.azure.io/aksdem`.

You can find additional information on Docker and the Azure Container Registry in the official documentation.

Once the image is in the ACR, the final prerequisite is to set up a service principal. This is easier to set up before you begin, but you can do this during the AKS cluster creation. An Azure service principal is a representation of an Azure Active Directory Application object. Service principals are generally used to interact with Azure through application automation. We will be using a service principal to allow the AKS cluster to pull the `aksdemo` image from the ACR. The service principal needs to have access to the ACR repository that you store the image in. You will need to record the client ID and secret of the service principal you want to use.

You can find additional information on Azure Active Directory service principals in the documentation.

Now that we have our image in the ACR and our service principal client ID and secret, we can begin deploying the AKS cluster.

Deploying an Azure Kubernetes Service cluster

The time has come to deploy our cluster. We are going to start in the Azure Portal. Go to *portal.azure.com* to log in. Once logged in, you should see a dashboard with a search bar at the top that will be used to locate services. From the search bar, we will be typing **kubernetes** and selecting the Kubernetes Service option from the drop-down menu, which is outlined in Figure 6-29.

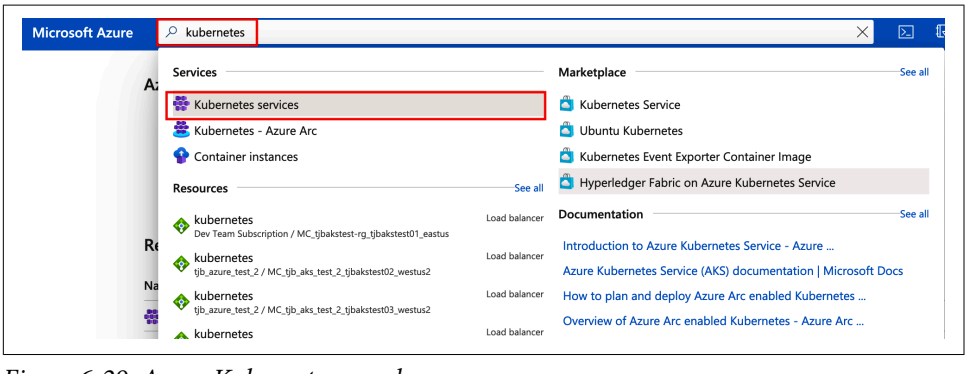

Figure 6-29. Azure Kubernetes search

Now we are on the Azure Kubernetes Services blade. Deployed AKS clusters are viewed from this screen using filters and queries. This is also the screen for creating new AKS clusters. Near the top of the screen, we are going to select Create as shown in Figure 6-30. This will cause a drop-down menu to appear, where we will select "Create a Kubernetes cluster."

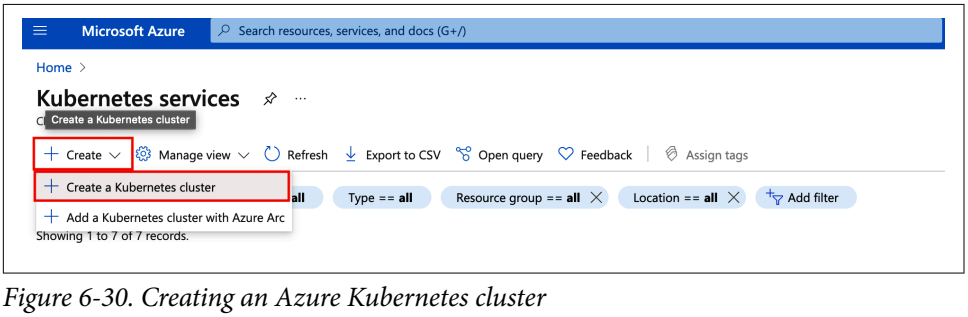

Figure 6-30. Creating an Azure Kubernetes cluster

Next we will define the properties of the AKS cluster from the "Create Kubernetes cluster" screen. First, we will populate the Project Details section by selecting the subscription that the cluster will be deployed to. There is a drop-down menu that allows for easier searching and selection. For this example, we are using the `tjb_azure_test_2` subscription, but any subscription can work as long as you have access to it. Next, we have to define the resource group we will use to group the AKS cluster. This can be an existing resource group or a new one can be created. For this example, we will create a new resource group named `go-web`.

After the Project Details section is complete, we move on to the Cluster Details section. Here, we will define the name of the cluster, which will be "go-web" for this example. The region, availability zones, and Kubernetes version fields are also defined in this section and will have predefined defaults that can be changed. For this example, however, we will use the default "(US) West 2" region with no availability zones and the default Kubernetes version of 1.19.11.

Not all Azure regions have availability zones that can be selected. If availability zones are part of the AKS architecture that is being deployed, the appropriate regions should be considered. You can find more information on AKS regions in the availability zones documentation.

Finally, we will complete the Primary Node Pool section of the "Create Kubernetes cluster" screen by selecting the node size and node count. For this example, we are going to keep the default node size of DS2 v2 and the default node count of 3. While most virtual machines, sizes are available for use within AKS, there are some restrictions. Figure 6-31 shows the options we have selected filled in.

You can find more information on AKS restrictions, including restricted node sizes, in the documentation.

Click the "Next: Node pools" button to move to the Node Pools tab. This page allows for the configuration of additional node pools for the AKS cluster. For this example, we are going to leave the defaults on this page and move on to the Authentication page by clicking the "Next: Authentication" button at the bottom of the screen.

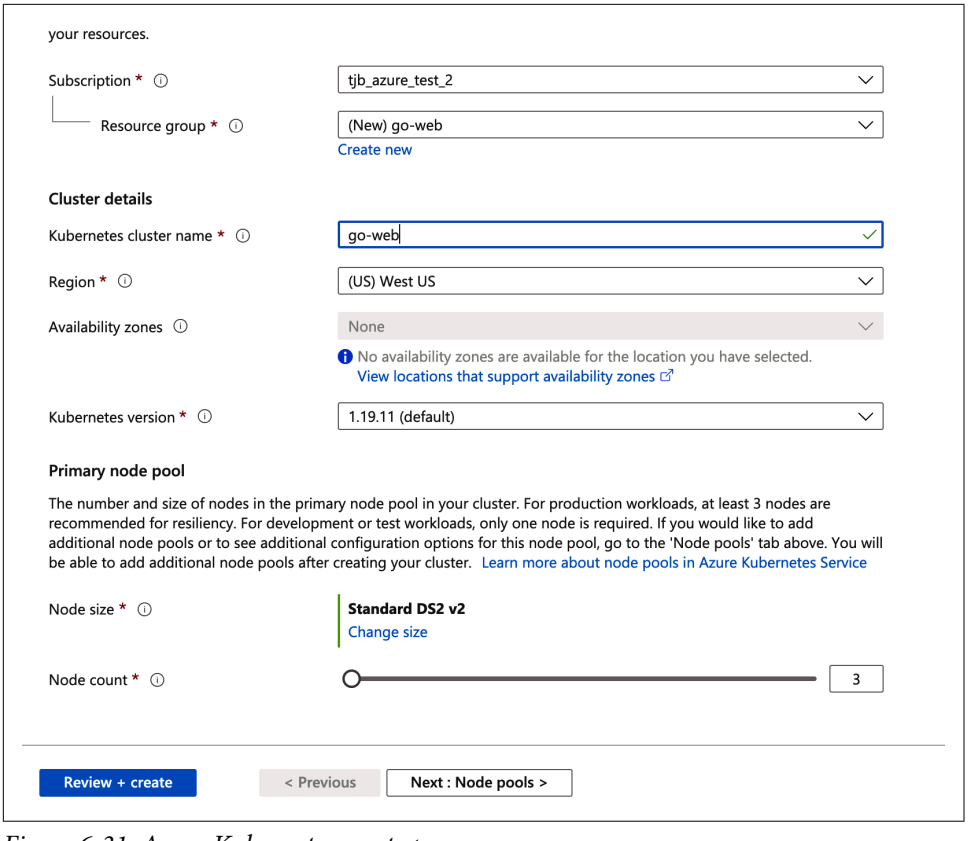

Figure 6-31. Azure Kubernetes create page

Figure 6-32 shows the Authentication page, where we will define the authentication method that the AKS cluster will use to connect to attached Azure services such as the ACR we discussed previously in this chapter. "System-Assigned Managed Identity" is the default authentication method, but we are going to select the "Service principal" radio button.

If you did not create a service principal at the beginning of this section, you can create a new one here. If you create a service principal at this stage, you will have to go back and grant that service principal permissions to access the ACR. However, since we will use a previously created service principal, we are going to click the "Configure service principal" link and enter the client ID and secret.

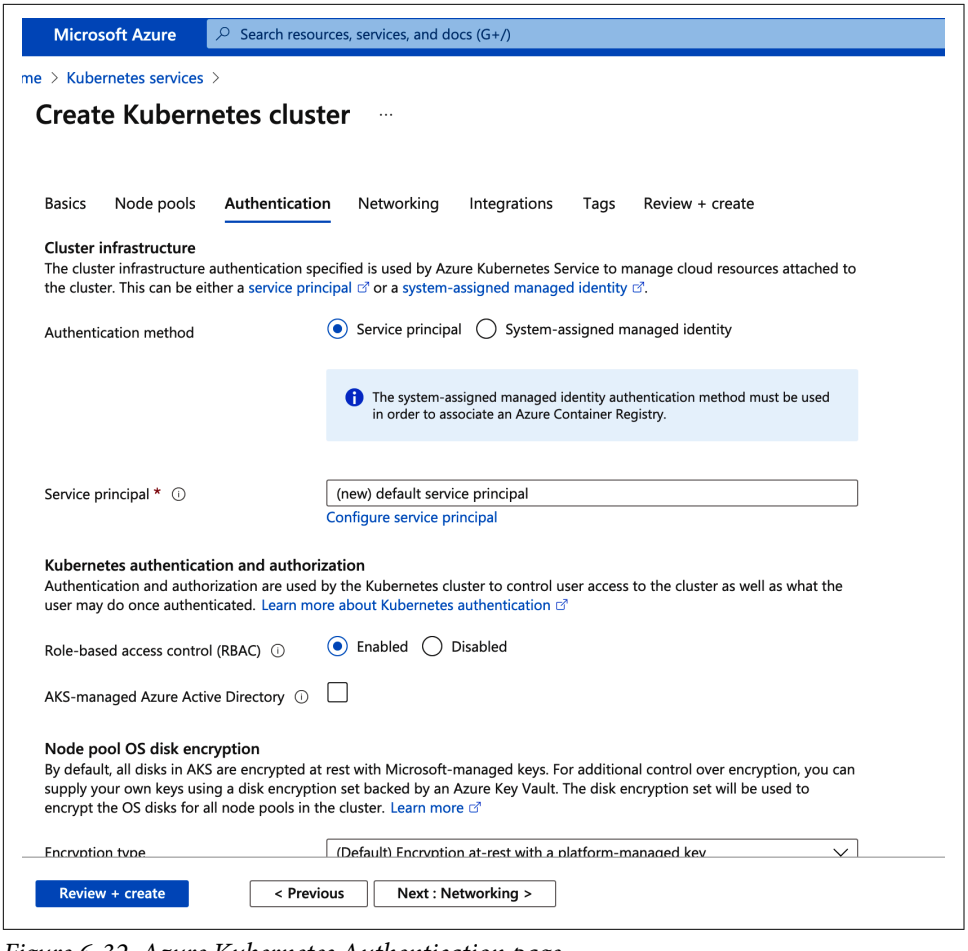

Figure 6-32. Azure Kubernetes Authentication page

The remaining configurations will remain at the defaults at this time. To complete the AKS cluster creation, we are going to click the "Review + create" button. This will take us to the validation page. As shown in Figure 6-33, if everything is defined appropriately, the validation will return a "Validation Passed" message at the top of the screen. If something is misconfigured, a "Validation Failed" message will be there instead. As long as validation passes, we will review the settings and click Create.

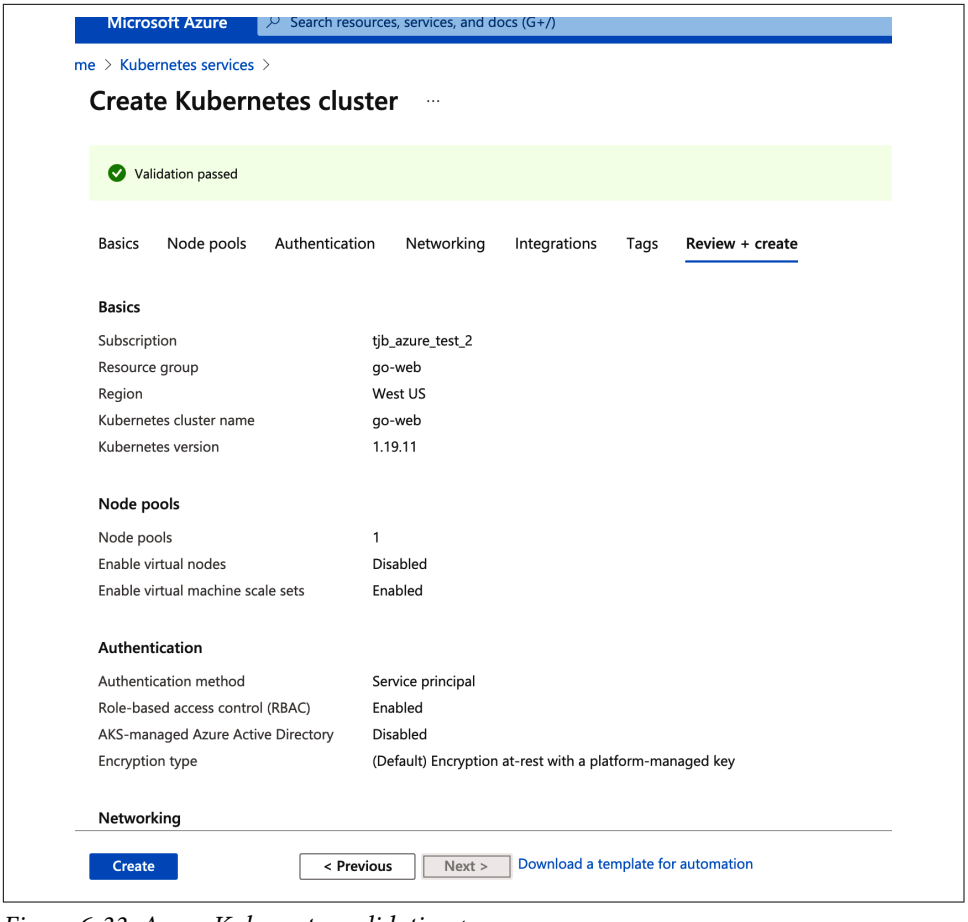

Figure 6-33. Azure Kubernetes validation page

You can view the deployment status from the notification bell on the top of the Azure screen. Figure 6-34 shows our example deployment in progress. This page has information that can be used to troubleshoot with Microsoft should an issue arise such as the deployment name, start time, and correlation ID.

Our example deployed completely without issue, as shown in Figure 6-35. Now that the AKS cluster is deployed, we need to connect to it and configure it for use with our example web server.

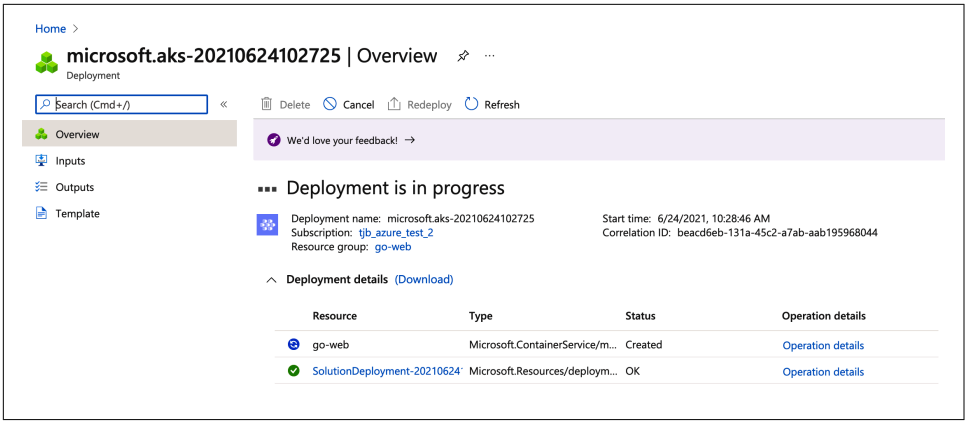

Figure 6-34. Azure Kubernetes deployment progress

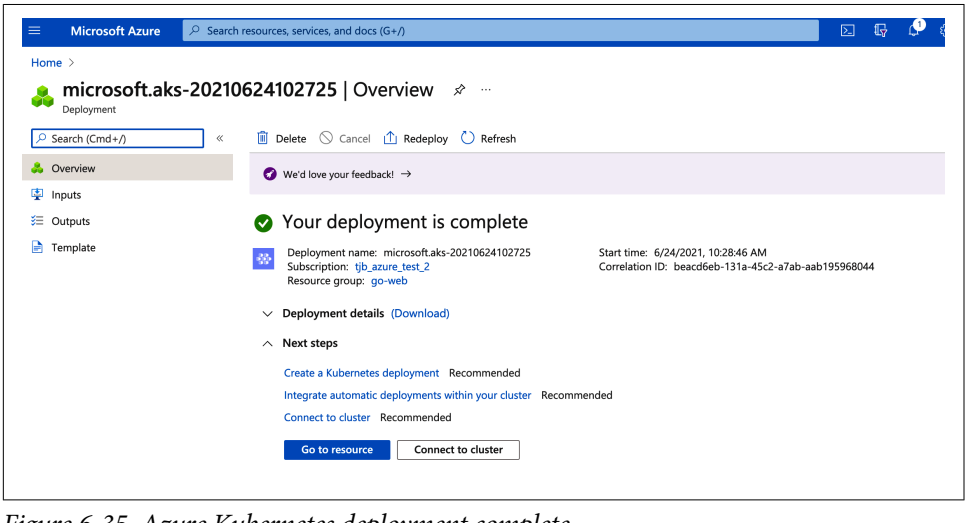

Figure 6-35. Azure Kubernetes deployment complete

Connecting to and configuring AKS

We will now shift to working with the example `go-web` AKS cluster from the command line. To manage AKS clusters from the command line, we will primarily use the `kubectl` command. Azure CLI has a simple command, `az aks install-cli`, to install the `kubectl` program for use. Before we can use `kubectl`, though, we need to gain access to the cluster. The command `az aks get-credentials --resource-group <resource_group_name> --name <aks_cluster_name>` is used to gain access to the AKS cluster. For our example, we will use `az aks get-credentials --resource-group go-web --name go-web` to access our `go-web` cluster in the `go-web` resource group.

Next we will attach the Azure container registry that has our `aksdemo` image. The command `az aks update -n <aks_cluster_name> -g <cluster_resource_group_name> --attach-acr <acr_repo_name>` will attach a named ACR repo to an existing AKS cluster. For our example, we will use the command `az aks update -n tjbakstest -g tjbakstest --attach-acr tjbakstestcr`. Our example runs for a few moments and then produces the output shown in Example 6-1.

Example 6-1. AttachACR output

```
{- Finished ..
  "aadProfile": null,
  "addonProfiles": {
    "azurepolicy": {
      "config": null,
      "enabled": false,
      "identity": null
    },
    "httpApplicationRouting": {
      "config": null,
      "enabled": false,
      "identity": null
    },
    "omsAgent": {
      "config": {
        "logAnalyticsWorkspaceResourceID":
        "/subscriptions/7a0e265a-c0e4-4081-8d76-aafbca9db45e/
        resourcegroups/defaultresourcegroup-wus2/providers/
        microsoft.operationalinsights/
        workspaces/defaultworkspace-7a0e265a-c0e4-4081-8d76-aafbca9db45e-wus2"
      },
      "enabled": true,
      "identity": null
    }
  },
  "agentPoolProfiles": [
    {
      "availabilityZones": null,
      "count": 3,
      "enableAutoScaling": false,
      "enableNodePublicIp": null,
      "maxCount": null,
      "maxPods": 110,
      "minCount": null,
      "mode": "System",
      "name": "agentpool",
      "nodeImageVersion": "AKSUbuntu-1804gen2containerd-2021.06.02",
      "nodeLabels": {},
      "nodeTaints": null,
      "orchestratorVersion": "1.19.11",
      "osDiskSizeGb": 128,
```

```
    "osDiskType": "Managed",
    "osType": "Linux",
    "powerState": {
      "code": "Running"
    },
    "provisioningState": "Succeeded",
    "proximityPlacementGroupId": null,
    "scaleSetEvictionPolicy": null,
    "scaleSetPriority": null,
    "spotMaxPrice": null,
    "tags": null,
    "type": "VirtualMachineScaleSets",
    "upgradeSettings": null,
    "vmSize": "Standard_DS2_v2",
    "vnetSubnetId": null
  }
],
"apiServerAccessProfile": {
  "authorizedIpRanges": null,
  "enablePrivateCluster": false
},
"autoScalerProfile": null,
"diskEncryptionSetId": null,
"dnsPrefix": "go-web-dns",
"enablePodSecurityPolicy": null,
"enableRbac": true,
"fqdn": "go-web-dns-a59354e4.hcp.westus.azmk8s.io",
"id":
"/subscriptions/7a0e265a-c0e4-4081-8d76-aafbca9db45e/
resourcegroups/go-web/providers/Microsoft.ContainerService/managedClusters/go-web",
"identity": null,
"identityProfile": null,
"kubernetesVersion": "1.19.11",
"linuxProfile": null,
"location": "westus",
"maxAgentPools": 100,
"name": "go-web",
"networkProfile": {
  "dnsServiceIp": "10.0.0.10",
  "dockerBridgeCidr": "172.17.0.1/16",
  "loadBalancerProfile": {
    "allocatedOutboundPorts": null,
    "effectiveOutboundIps": [
      {
        "id":
        "/subscriptions/7a0e265a-c0e4-4081-8d76-aafbca9db45e/
        resourceGroups/MC_go-web_go-web_westus/providers/Microsoft.Network/
        publicIPAddresses/eb67f61d-7370-4a38-a237-a95e9393b294",
        "resourceGroup": "MC_go-web_go-web_westus"
      }
    ],
    "idleTimeoutInMinutes": null,
```

```
      "managedOutboundIps": {
        "count": 1
      },
      "outboundIpPrefixes": null,
      "outboundIps": null
    },
    "loadBalancerSku": "Standard",
    "networkMode": null,
    "networkPlugin": "kubenet",
    "networkPolicy": null,
    "outboundType": "loadBalancer",
    "podCidr": "10.244.0.0/16",
    "serviceCidr": "10.0.0.0/16"
  },
  "nodeResourceGroup": "MC_go-web_go-web_westus",
  "powerState": {
    "code": "Running"
  },
  "privateFqdn": null,
  "provisioningState": "Succeeded",
  "resourceGroup": "go-web",
  "servicePrincipalProfile": {
    "clientId": "bbd3ac10-5c0c-4084-a1b8-39dd1097ec1c",
    "secret": null
  },
  "sku": {
    "name": "Basic",
    "tier": "Free"
  },
  "tags": {
    "createdby": "tjb"
  },
  "type": "Microsoft.ContainerService/ManagedClusters",
  "windowsProfile": null
}
```


This output is the CLI representation of the AKS cluster information. This means that the attachment was successful. Now that we have access to the AKS cluster and the ACR is attached, we can deploy the example Go web server to the AKS cluster.

Deploying the Go web server

We are going to deploy the Golang code shown in Example 6-2. As mentioned earlier in this chapter, this code has been built into a Docker image and now is stored in the ACR in the `tjbakstestcr` repository. We will be using the following deployment YAML file to deploy the application.

Example 6-2. Kubernetes Podspec for Golang minimal webserver

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: go-web
spec:
  containers:
  - name: go-web
    image: go-web:v0.0.1
    ports:
    - containerPort: 8080
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 5
    readinessProbe:
      httpGet:
        path: /
        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 5
```

Breaking down this YAML file, we see that we are creating two AKS resources: a deployment and a service. The deployment is configured for the creation of a container named `go-web` and a container port 8080. The deployment also references the `aksdemo` ACR image with the line `image: tjbakstestcr.azurecr.io/aksdemo` as the image that will be deployed to the container. The service is also configured with the name go-web. The YAML specifies the service is a load balancer listening on port 8080 and targeting the `go-web` app.

Now we need to publish the application to the AKS cluster. The command `kubectl apply -f <yaml_file_name>.yaml` will publish the application to the cluster. We will see from the output that two things are created: `deployment.apps/go-web` and `service/go-web`. When we run the command `kubectl get pods`, we can see an output like that shown here:

```
○ → kubectl get pods
NAME                      READY   STATUS    RESTARTS   AGE
go-web-574dd4c94d-2z5lp   1/1     Running   0          5h29m
```

Now that the application is deployed, we will connect to it to verify it is up and running. When a default AKS cluster is stood up, a load balancer is deployed with it with a public IP address. We could go through the portal and locate that load balancer and

public IP address, but `kubectl` offers an easier path. The command `kubectl get` [.keep-together]#`service go-web` produces this output:

```
○ → kubectl get service go-web
NAME     TYPE           CLUSTER-IP   EXTERNAL-IP    PORT(S)          AGE
go-web   LoadBalancer   10.0.3.75    13.88.96.117   8080:31728/TCP   21h
```

In this output, we see the external IP address of 13.88.96.117. Therefore, if everything deployed correctly, we should be able to cURL 13.88.96.117 at port 8080 with the command `curl 13.88.96.117:8080`. As we can see from this output, we have a successful deployment:

```
○ → curl 13.88.96.117:8080 -vvv
*   Trying 13.88.96.117...
* TCP_NODELAY set
* Connected to 13.88.96.117 (13.88.96.117) port 8080 (#0)
> GET / HTTP/1.1
> Host: 13.88.96.117:8080
> User-Agent: curl/7.64.1
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Fri, 25 Jun 2021 20:12:48 GMT
< Content-Length: 5
< Content-Type: text/plain; charset=utf-8
<
* Connection #0 to host 13.88.96.117 left intact
Hello* Closing connection 0
```

Going to a web browser and navigating to http://13.88.96.117:8080 will also be available, as shown in Figure 6-36.

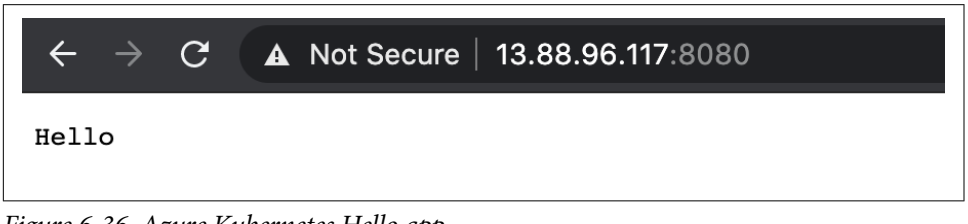

Figure 6-36. Azure Kubernetes Hello app

AKS conclusion

In this section, we deployed an example Golang web server to an Azure Kubernetes Service cluster. We used the Azure Portal, the `az cli`, and `kubectl` to deploy and configure the cluster and then deploy the application. We leveraged the Azure container registry to host our web server image. We also used a YAML file to deploy the application and tested it with cURL and web browsing.

Conclusion

Each cloud provider has its nuanced differences when it comes to network services provided for Kubernetes clusters. Table 6-4 highlights some of those differences. There are lots of factors to choose from when picking a cloud service provider, and even more when selecting the managed Kubernetes platform to run. Our aim in this chapter was to educate administrators and developers on the choices you will have to make when managing workloads on Kubernetes.

Table 6-4. Cloud network and Kubernetes summary

	AWS	Azure	GCP
Virtual network	VPC	Vnet	VPC
Network scope	Region	Region	Global
Subnet boundary	Zone	Region	Region
Routing scope	Subnet	Subnet	VPC
Security controls	NACL/SecGroups	Network security groups/Application SecGroup	Firewall
IPv6	Yes	Yes	No
Kubernetes managed	eks	aks	gke
ingress	AWS ALB controller	Nginx-Ingress	GKE ingress controller
Cloud custom CNI	AWS VPC CNI	Azure CNI	GKE CNI
Load Balancer support	ALB L7, L4 w/NLB, and Nginx	L4 Azure Load Balancer, L7 w/Nginx	L7, HTTP(S)
Network policies	Yes (Calico/Cilium)	Yes (Calico/Cilium)	Yes (Calico/Cilium)

We have covered many layers, from the OSI foundation to running networks in the cloud for our clusters. Cluster administrators, network engineers, and developers alike have many decisions to make, such as the subnet size, the CNI to choose, and the load balancer type, to name a few. Understanding all of those and how they will affect the cluster network was the basis for this book. This is just the beginning of your journey for managing your clusters at scale. We have managed to cover only the networking options available for managing Kubernetes clusters. Storage, compute, and even how to deploy workloads onto those clusters are decisions you will have to make now. The O'Reilly library has an extensive number of books to help, such as *Production Kubernetes* (Rosso et al.), where you learn what the path to production looks like when using Kubernetes, and *Hacking Kubernetes* (Martin and Hausenblas), on how to harden Kubernetes and how to review Kubernetes clusters for security weaknesses.

We hope this guide has helped make those networking choices easy for you. We were inspired to see what the Kubernetes community has done and are excited to see what you build on top of the abstractions Kubernetes provides for you.

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 degree in computer science from the University of Louisville; six AWS certifications, including the Certified Advanced Networking Specialty; and the CNCF's CKA.

Vallery Lancey is a software engineer who specializes in reliability, infrastructure, and distributed systems. Vallery began using Kubernetes in 2017, living through many of the early-adopter challenges and rapidly evolving features. She has built Kubernetes platforms and operated Kubernetes at massive scale at companies such as Lyft. Vallery is a part-time Kubernetes contributor to SIG-Network. There, she has contributed to kube-proxy and early IPv4/IPv6 dual-stack support.

Colophon

The animal on the cover of *Networking and Kubernetes* is a black-throated loon (*Gavia arctica*), an aquatic bird found in the northern hemisphere.

The black-throated loon is about 28 inches long and can weigh between 3 and 7.5 pounds. Adults are mostly black with white patches, white stripes, and white underparts. The loon usually lays a clutch of two eggs, which they incubate for 27 to 29 days. They make their nests on the ground and due to predation and flooding, it is common for only one chick to survive.

Although the overall population of the loon is declining, its conservation staus is considered of least concern due to its large population over an extremely large range. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Susan Thompson, based on a black-and white-engraving from *British Birds*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.