Abhishek Chanda

Network Programming with Rust

Build fast and resilient network servers and clients by leveraging Rust's memory-safety and concurrency features

Packt>

Network Programming with Rust

Build fast and resilient network servers and clients by leveraging Rust’s memory-safety and concurrency features

Abhishek Chanda

BIRMINGHAM - MUMBAI

Network Programming with Rust

Commissioning Editor: Merint Mathew
Acquisition Editor: Chaitanya Nair
Content Development Editor: Vikas Tiwari
Technical Editor: Diwakar Shukla
Copy Editor: Safis Editing
Project Coordinator: Ulhas Kambali
Proofreader: Safis Editing
Indexer: Mariammal Chettiyar
Graphics: Tania Dutta
Production Coordinator: Shantanu Zagade

First published: February 2018

Production reference: 1260218

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78862-489-3

www.packtpub.com

To my wife, Anasua, for being an amazing partner and friend, and also for all the diagrams in this book.

To the memory of my mother, Sikha, for her sacrifices and for exemplifying the power of determination.

Contributors

About the author

Abhishek Chanda studied computer science at IIEST Shibpur in India and electrical engineering at Rutgers University. He has lived and worked in multiple countries, working on distributed systems since 2008 at big companies such as Microsoft as well as a number smaller start-ups. Currently, he is working with DataSine in London, where he is responsible for the scalable deployment of infrastructure powering the backend systems of the DataSine platform. He contributes to a number of open source projects, including Rust.

I would like to thank Packt and their wonderful team for making this possible, especially Vikas, Chaitanya, Madhunikita, and Diwakar. I am grateful to Pradeep and all the reviewers for their insightful comments. This book would not have been possible without the support of my wife, Anasua.

About the reviewer

Pradeep R is a software professional at Gigamon. He is a technology enthusiast passionate about network programing and security, with wide experience in working on leading enterprise network switching and routing solutions and in development and deployment of traditional network security elements. Currently, he is working on next-generation network pervasive visibility solutions.

He extensively works with C, C++, Python, JavaScript, Perl, and occasionally with Java, .NET, and Rust. Pradeep has recently reviewed Rust Cookbook.

I would like to thank Vigneshwer Dhinakaran, my brother, who has shown me that boundaries exists only in our minds and they are meant to be breached. I would also like to acknowledge with gratitude my mother, Vasanthi, my grandma, Sulochana, and my sister, Deepika, for their constant encouragement and support.

Table of Contents

Preface

Rust has steadily become one of the most important new programming languages in recent years. Like C or C++, Rust enables the developer to write code that is low-level enough to make Rust code quickly. And since Rust is memory-safe by design, it does not allow code that can crash on a null pointer exception. These properties make it a natural choice for writing low-level networking applications. This book will enable developers to get started with writing networking applications with Rust.

Who this book is for

This book’s target audience is a software engineer who is interested in writing networking software using Rust.

What this book covers

Chapter 1, *Introduction to Client/Server Networking*, starts the book with a gentle introduction to computer networking from the ground up. This includes IP addressing, TCP/UDP, and DNS. This forms the basis of our discussions in later chapters.

Chapter 2, *Introduction to Rust and its Ecosystem*, contains an introduction to Rust. This is an overall introduction that should be good enough to get the reader started. We do assume some familiarity with programming.

Chapter 3, *TCP and UDP Using Rust*, dives into using Rust for networking. We start with basic socket programming using the standard library. We then look at some crates from the ecosystem that can be used for network programming.

Chapter 4, *Data Serialization, Deserialization, and Parsing*, explains that an important aspect of networked computing is handling data. This chapter is an introduction to serializing and deserializing data using Serde. We also look at parsing using nom and other frameworks.

Chapter 5, *Application Layer Protocols*, moves up a layer to look at protocols that operate above TCP/IP. We look at a few crates to work with, such as RPC, SMTP, FTP, and TFTP.

Chapter 6, *Talking HTTP in the Internet*, explains that arguably the most common application of the internet is HTTP. We look at crates such as Hyper and Rocket which are used for writing HTTP servers and clients.

Chapter 7, *Asynchronous Network Programming Using Tokio*, looks at the Tokio stack for asynchronous programming using futures, streams, and event loops.

Chapter 8, *Security*, delves into securing the services we have described so far. This is using certificates and secret keys.

Chapter 9, *Appendix*, discusses a number of crates have appeared that propose alternate ways of doing things already covered in this book. This includes the async/await syntax, parsing using Pest, and so on. We will discuss some of these in the appendix.

To get the most out of this book

1. They are either already familiar with Rust or are planning to start learning the language.
2. They have a commercial background in software engineering using other programming languages and are aware about the tradeoffs in developing software using different programming languages.
3. They have a basic familiarity with networking concepts.
4. They can appreciate why distributed systems are important in modern computing.

Download the example code files

You can download the example code files for this book from your account at www.packtpub.com. If you purchased this book elsewhere, you can visit www.packtpub.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packtpub.com.
2. Select the SUPPORT tab.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at https://github.com/PacktPublishing/Network-Programming-with-Rust. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: “The `target` directory contains compilation artifacts.”

A block of code is set as follows:

```
[package]
name = "hello-rust"
version = "0.1.0"
authors = ["Foo Bar <foo.bar@foobar.com>"]
```

Any command-line input or output is written as follows:

```
# cargo new --bin hello-rust
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: “It will not need to call connect for that same connection:”

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Reviews

Introduction to Client/Server Networking

This book is an introduction to writing networking applications in Rust. This title begs two questions: why should anyone care about networking? And why would anyone want to write networking applications in Rust? We attempt to answer the first question in this chapter. We will introduce Rust and network programming using Rust in subsequent chapters. Firstly, in this chapter, we will start with a bit of history and try to understand how network architecture evolved over the last hundred years. In subsequent sections, we will see how modern networks are layered and addressed. Afterwards, we will describe common service models used in networking. We will end with a summary of networking-related programming interfaces that Linux exposes. Note that this book deliberately ignores network programming in other operating systems and focuses only on Linux for the sake of simplicity. While the Rust compiler is platform-agnostic, there can be cases where some things are different in other platforms compared to Linux. We will point out those differences as we progress.

In this chapter, we will cover the following topics:

- History of networking: why and how networks came into use and how the internet evolved
- Layering in networks: how layering and encapsulation works
- Addressing: how networks and individual hosts are uniquely identified on the internet
- How IP routing works
- How DNS works
- Service models for data delivery
- The network programming interface in Linux

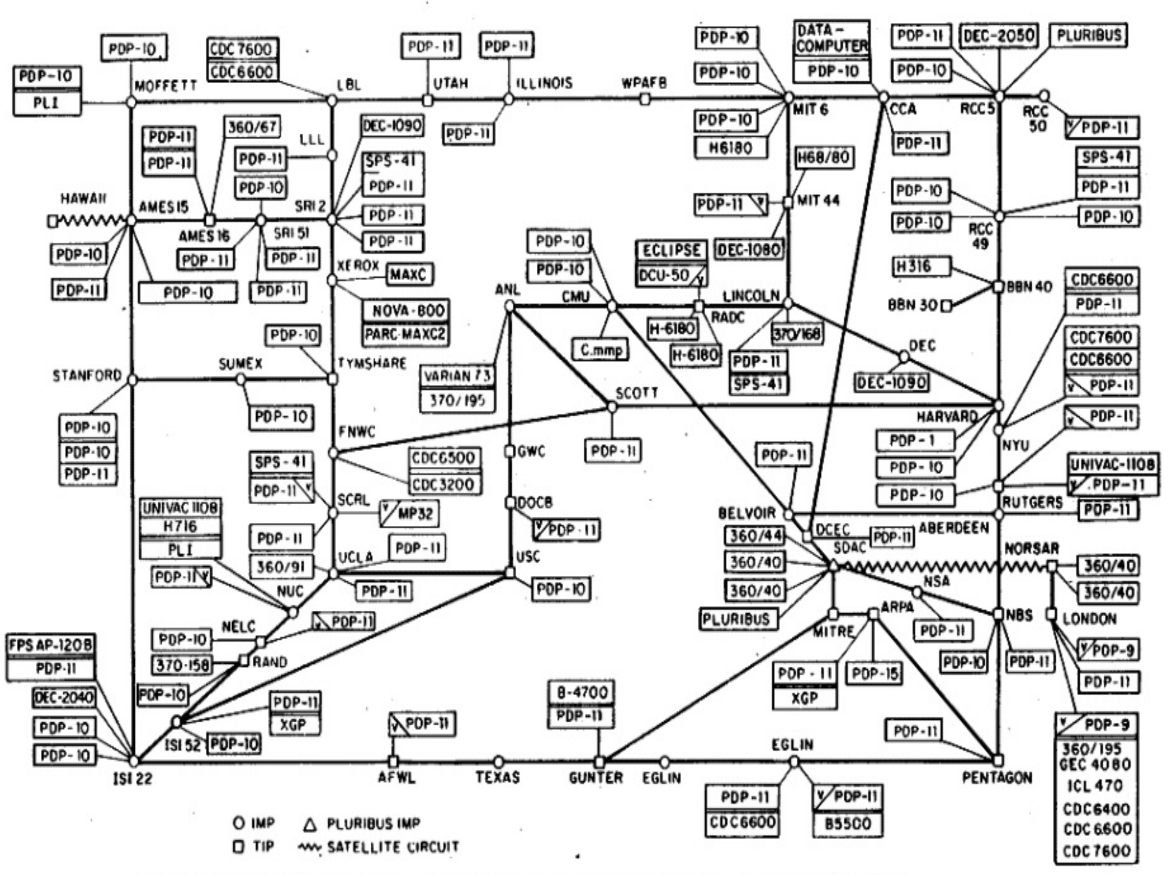
A brief history of networks

The modern internet has revolutionized how we communicate with one another. However, it had humble beginnings in the Victorian era. One of the earliest precursors to the internet was telegraph networks which were operational as early as 1850. Back then, it used to take 10 days to send a message from Europe to North America by sea. Telegraph networks reduced that to 17 hours. By the late 19th century, the telegraph was a fully successful communication technology that was used widely in the two world wars. Around that time, people started building computers to help in cracking enemy codes. Unlike our modern mobile phones and laptops, those computing machines were often huge and needed specialized environments to be able to operate smoothly. Thus, it was necessary to put those in special locations while the operators would sit on a terminal. The terminal needed to be able to communicate with the computer over short distances. A number of local area networking technologies enabled this, the most prominent one being *Ethernet*. Over time, these networks grew and by the 1960s, some of these networks were being connected with one another to form a larger network of networks. The **Advanced Research Projects Agency Network** (**ARPANET**) was established in 1969 and it became the first internetwork that resembles the modern internet. Around 1973, there were a number of such internetworks all around the world, each using their own protocols and methods for communication. Eventually, the protocols were standardized so that the networks could communicate with each other seamlessly. All of these networks were later merged to form what is the internet today.

Since networks evolved in silos all around the world, they were often organized according to geographical proximity. A **Local Area Network** (**LAN**) is a collection of host machines in small proximity like a building or a small neighborhood. A **Wide Area Network** (**WAN**) is one that connects multiple neighborhoods; the global internet is at the top of the hierarchy. The next picture shows a map of the ARPANET in 1977. Each node in this map is a computer (a server, in today's terms). Most of these were located in large universities like Stanford or at national laboratories like Lawrence Berkeley (source: https://commons.wikimedia.org/wiki/File:Arpanet_logical_map,_march_1977.png).

*In networking, a **Request For Comment (RFC)** is a document* *that describes how a proposed system should work. These are the first steps towards standardizing a protocol or a system. The term internet was first used in RFC 675, which proposed a standard for TCP.*

ARPANET LOGICAL MAP, MARCH 1977



(PLEASE NOTE THAT WHILE THIS MAP SHOWS THE HOST POPULATION OF THE NETWORK ACCORDING TO THE BEST INFORMATION OBTAINABLE, NO CLAIM CAN BE MADE FOR ITS ACCURACY)

NAMES SHOWN ARE IMP NAMES, NOT (NECESSARILY) HOST NAMES

Layering in networks

Computer science often focuses on subdividing a problem into smaller, hopefully independent components that can be solved in isolation. Once that is done, all that is needed is a set of rules on how those components should communicate to have a solution to the larger problem. This set of rules, along with a pre-agreed data format, is called a **protocol**. A network is composed of a number of layers, each of which has a fixed purpose. Thus, each of these layers run one or many protocols, forming a stack of protocols. In the early days of networking, different people implemented their networks in different ways. When the internet was conceived, there was a need to make these networks communicate seamlessly. Since they were constructed differently, this turned out to be difficult.

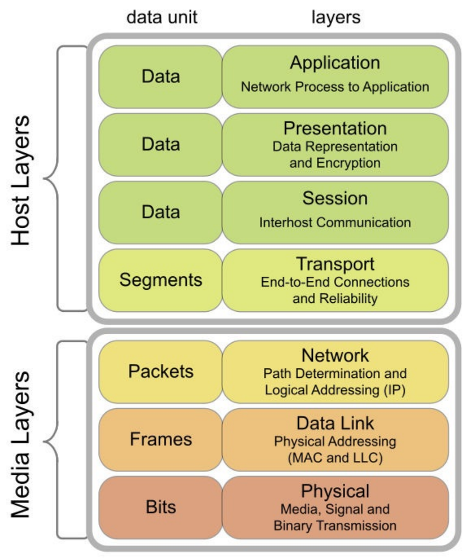
There was a clear need to agree on standard protocols and interfaces to make the internet work. The first attempt at standardizing networking protocols was in 1977, which led to the OSI model. This model has the following layers:

- **Physical layer**: It defines how data is transmitted in the physical medium in terms of its electrical and physical characteristics. This can either be by wire, fiber optic, or a wireless medium.
- **Data link layer**: It defines how data is transmitted between two nodes connected by a physical medium. This layer deals with prioritization between multiple parties trying to access the wire simultaneously. Another important function of this layer is to include some redundancy in the transmitted bits to minimize errors during transmission. This is referred to as coding.
- **Network layer**: It defines how packets (made up of multiple units of data) are transmitted between networks. Thus, this layer needs to define how to identify hosts and networks uniquely.
- **Transport layer**: It defines mechanisms to reliably deliver variable length messages to hosts (in the same or different networks). This layer defines a stream of packets that the receiver can then listen to.
- **Session layer**: It defines how applications running on hosts should communicate. This layer needs to differentiate between applications running on the same host and deliver packets to them.
- **Presentation layer**: It defines common formats for data representation so that different applications can interlink seamlessly. In some cases, this

layer also takes care of security.

- **Application layer**: It defines how user-centric applications should send and receive data. An example is the web browser (a user-centric application) using HTTP (an application layer protocol) to talk to a web server.

The following figure shows a visual representation of this model (source: https://commons.wikimedia.org/wiki/File:Osi-model-jb.svg). This also shows two vertical classifications, the host running the network stack and the physical media (including the wire and the network device). Each layer has its own data unit, the representation of the information it works on, and since each layer encapsulates the one below it, the data units encapsulate too. A number of bits form a frame, a number of frames form a packet, and so on, to the top:


data unit
layers
Host Layers
Data
Application
Network Process to Application
Data
Presentation
Data Representation and Encryption
Data
Session
Interhost Communication
Segments
Transport
End-to-End Connections and Reliability
Media Layers
Packets
Network
Path Determination and Logical Addressing (IP)
Frames
Data Link
Physical Addressing (MAC and LLC)
Bits
Physical
Media, Signal and Binary Transmission

The OSI model and its layers

While OSI was working on standardizing this model, **Defense Advanced Research Projects Agency** (**DARPA**) came up with a full implementation of the much simpler TCP/IP model (also known as the **IP** (**Internet Protocol**) suite). This model has the following layers, from closest to the physical medium to the farthest:

- **Hardware interface layer**: This is a combination of layers one and two of the OSI model. This layer is responsible for managing media access control, handling transmission and reception of bits, retransmission, and coding (some texts on networking differentiate between the hardware interface layer and the link layer. This results in a five layer model instead of four. This hardly matters in practice, though.)
- **IP layer**: This layer corresponds to layer three of the OSI stack. Thus,

this layer is responsible for two major tasks: addressing hosts and networks so that they can be uniquely identified and given a source and a destination address, and computing the path between those given a bunch of constraints (routing).

- **Transport layer**: This layer corresponds to layer four of the OSI stack. This layer converts raw packets to a stream of packets with some guarantees: in-order delivery (for TCP) and randomly ordered delivery (for UDP).
- **Application layer**: This layer combines layers five to seven of the OSI stack and is responsible for identifying the process, data formatting, and interfacing with all user level applications.

Note that the definition of what a particular layer handles changes as we move from one layer to another. The hardware interface layer handles collection of bits and bytes transmitted by hosts, the IP layer handles packets (the collection of a number of bytes sent by a host in a specific format), the transport layer bunches together packets from a given process on a host to another process on another host to form a segment (for TCP) or datagram (for UDP), and the application layer constructs application specific representations from the underlying stream. For each of these layers, the representation of data that they deal with is called a **Protocol Data Unit** (**PDU**) for that layer. As a consequence of this layering, when a process running on a host wants to send data to another host, the data must be broken into individual chunks. As the chunk travels from one layer to another, each layer adds a header (sometimes a trailer) to the chunk, forming the PDU for that layer. This process is called **encapsulation**. Thus, each layer provides a set of services to layers above it, specified in the form of a protocol.

The modern internet exhibits a form of geographical hierarchy. Imagine a number of homes which are served by a number of **Internet Service Providers** (**ISPs**). Each of these homes is in a LAN (either via Ethernet, or more commonly, Wi-Fi). The ISP connects many such LANs in a WAN. Each ISP has one or many WANs that they connect to form their own network. These larger networks, spanning cities, which are controlled by a single business entity, are called **Administrative Systems** (**AS**). Routing between multiple ISPs is often more complex than regular IP routing since they have to take into account things like trading agreements and so on. This is handled by specialized protocols like the **Border Gateway Protocol** (**BGP**).

As mentioned before, one of the earliest and most successful networking technologies is Ethernet. First introduced in 1974, it quickly became the predominant technology for LAN and WAN due to its low cost and relative ease of maintenance. Ethernet is a shared media protocol where all the hosts must use the same physical medium to send and receive frames. Frames are delivered to all hosts, which will check if the destination MAC address (these addresses will be described in the next section) matches its own address. If it does, the frame is accepted, otherwise, it is discarded. Since the physical medium can only carry one signal at any given moment, there is a probability that frames might collide in transit. If that does occur, the sender can sense the collision by sensing transmission from other hosts while it is transmitting its frame. It then aborts the transmission and sends a jam signal to let other hosts know of the collision. Then, it waits for an exponentially backed off amount of time and retries the transmission. After a fixed number of attempts, it gives up if the transmission does not succeed.

This scheme is called **carrier-sense multiple access with collision detection** (**CSMA/CD**). One problem with Ethernet is its relatively short range. Depending on the physical wiring technology used, the maximum length of an Ethernet segment varies between 100 m to 500 m. Thus, multiple segments must be connected to form a larger network. The most common way of doing that is using layer two switches between two adjacent Ethernet segments. Each port of these switches forms different collision domains, reducing the overall probability of collisions. These switches can also monitor traffic to learn which MAC addresses are on which ports so that eventually, they will send out frames for that MAC address only on that port (referred to as a learning switch). In modern homes, Wi-Fi is often the dominant LAN technology compared to Ethernet.

Addressing in networks

We have seen why it is important to identify hosts and networks uniquely to be able to deliver packets reliably. Depending on the scale, there are three major ways of doing this; we will discuss each of those in this section. The end to end process of IP routing will be discussed in the next section. One interesting fact to note is that for each of these addressing modes, one or more addresses are reserved for special use. Often, these are marked by a known set of bits being on or off in a known pattern:

- **Ethernet address**: This is also known as a **Media Access Control** (**MAC**) address. It is a 48-bit long unique identifier assigned to a network device (usually stored on the card) that is used to identify it in a network segment. Usually, these are programmed by the network card manufacturer, but all modern OS's allow one to modify it. The standard way of writing Ethernet addresses are in six groups of two hexadecimal digits (01-23-45-67-89-ab-cd-ef). Another common way is to use a colon to separate the digits (01:23:45:67:89:ab:cd:ef). A few special sequences of bits are reserved for addressing special cases: the sender can request that an Ethernet frame should be received by all hosts in that segment by setting the least significant bit of the first octet to 1; this is called multicasting. If that particular bit is set to 0, the frame should be delivered to only one receiver. Today, these are used widely with Ethernet and Wi-Fi.
- **IP address**: This is an address assigned to each device in an IP network. The original IP address standard (IPv4) defined 32-bit addresses in 1980. However, by 1995, it was obvious that the total number of available addresses on the internet is not enough to cover all devices. This led to the development of IPv6, which expanded the address space to 128 bits. The standard way of dealing with a group of IP addresses is using the CIDR notation, for example, 192.168.100.1/26 (IPv4). The decimal number after the slash counts the number of leading 1s in the network mask. Thus, in this particular case, there are *2^(32-26) = 64* addresses in the network starting from 192.168.100.0 to 192.168.100.63. The **Internet Assigned Numbers Authority** (**IANA**) assigns blocks of publicly routable IP addresses to organizations. A number of IPv4 and v6 addresses are reserved for various purposes like addressing in private networks and so on. In a home network (which will always use special

private range addresses), these are assigned by the **Dynamic Host Configuration Protocol** (**DHCP**) by the Wi-Fi router.

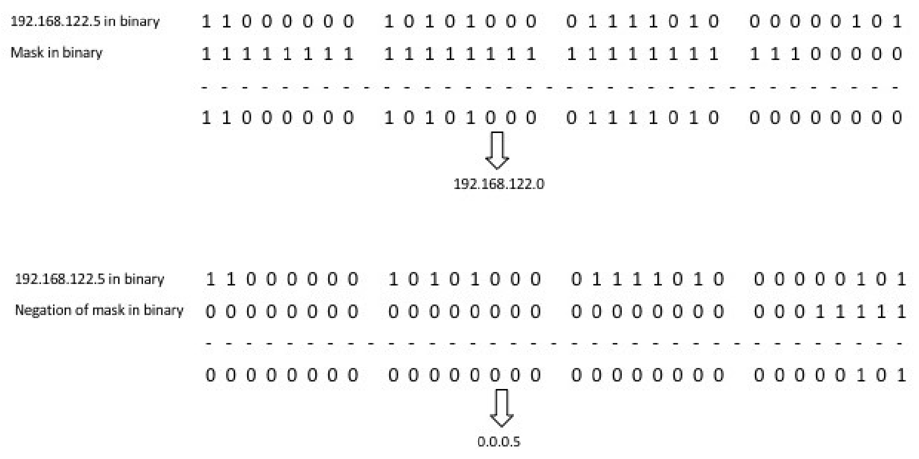
- **Autonomous system number:** This is a 32-bit number used to uniquely identify autonomous systems. Like IP addresses, these are assigned and maintained by the IANA.

Apart from these, communication between hosts often uses a port number to distinguish between processes. When the OS allocates a specific port to a process, it updates its database of the mapping between process identifier and port number. Thus, when it receives incoming packets on that port, it knows what process to deliver those packets to. In case the process has exited by that time, the OS will drop the packets and in the case of TCP, initiate closing of the connection. In the subsequent sections, we will see how TCP works in practice.

A range of port numbers between 0 and 1024 are reserved for common services by the OS. Other applications are free to request any port above 1024.

How IP routing works

To understand how IP routing works, we must first begin with the structure of IPv4 addresses. As described in the last section, these are 32 bits in length. They are written in a dotted decimal notation in groups of 4 bytes (for example, 192.168.122.5). A given number of bits in that network prefix is used to identify the network where the packet should be delivered, and the rest of the bits identify the particular host. Thus, all hosts in the same network must have the same prefix. Conventionally, the prefix is described in the CIDR notation with the starting address and the number of bits in the network portion of the address separated by a slash (192.168.122.0/30). The number can then be used to find out how many addresses are available for hosts in the network (in this case, *2^(32-30) = 4*). Given an IP address and a prefix, the network address can be extracted by bitwise-ANDing the address with a mask of all 1s in the network portion. Calculating the host address is just the reverse; we will need to AND with the network mask's logical negation (the host mask), which has all 0s in the network portion and all 1s in the host portion. Given an address and a prefix like 192.168.122.5/27, we will compute these as shown in the following figure. Thus, for the given CIDR, the network address is 192.168.122.0 and the host address is 0.0.0.5:



```
192.168.122.5 in binary      1 1 0 0 0 0 0 0   1 0 1 0 1 0 0 0   0 1 1 1 1 0 1 0   0 0 0 0 0 1 0 1
Mask in binary               1 1 1 1 1 1 1 1   1 1 1 1 1 1 1 1   1 1 1 1 1 1 1 1   1 1 1 0 0 0 0 0
                             - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                             1 1 0 0 0 0 0 0   1 0 1 0 1 0 0 0   0 1 1 1 1 0 1 0   0 0 0 0 0 0 0 0
                                                       ⇩
                                                 192.168.122.0

192.168.122.5 in binary      1 1 0 0 0 0 0 0   1 0 1 0 1 0 0 0   0 1 1 1 1 0 1 0   0 0 0 0 0 1 0 1
Negation of mask in binary   0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0   0 0 0 1 1 1 1 1
                             - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                             0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0   0 0 0 0 0 1 0 1
                                                       ⇩
                                                    0.0.0.5
```

CIDR to network and host address conversion

As described before, each IP network will have a reserved broadcast address that can be used for a host to send a message to all hosts in that network. This can be computed by ORing with the host mask. In our example, this comes out to be 192.168.122.31. Note that the network address can not be a valid host address.

There are two broad classes of IP address; some blocks of addresses can be routed in the public internet, these are called public IP addresses. Some other blocks can only be used in private networks that do not directly interface with the internet, these are called private addresses. If a router on the internet receives a packet that is destined for a private IP address, it will have to drop that packet. Other than these two, IP addresses are also classified on various parameters: some are reserved for documentation only (192.0.2.0/24), some are reserved for point to point communication between two hosts (169.254.0.0/16), and so on. The Rust standard library has convenience methods to classify IP addresses according to their types.

All routers maintain a routing table which maps prefixes to the outgoing interface of the router (while a router administrator might decide to store individual addresses instead of prefixes, this will quickly lead to a large routing table in a busy router). An entry in the table basically says *If a packet needs to go to this network, it should be sent on this interface*. The next host that receives the packet might be another router or the destination host. How do routers figure out this table? Multiple routers run routing protocols between those which compute those tables. Some common examples are OSPF, RIP, and BGP. Given these primitives, the actual routing mechanism is fairly simple, as shown in the next diagram.

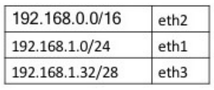
An interesting aspect of IP is the use of the **Time To Live** (**TTL**) field, this is also known as hop limit. The host sends out packets with a fixed value of TTL (usually 64). Each router the packet crossed decreases the TTL. When it reaches 0, the packet is discarded. This mechanism ensures that packets are not stuck in an infinite loop between routers:

```
for each packet do
    if $TTL \leq 1$ then
        drop packet ;
        optionally, send an ICMP error back to sender ;
    else
        compute network prefix for packet ;
        if any outgoing interface is in that network then
            decrement TTL ;
            send packet out on that interface ;
        else if routing table contains a route for that network then
            decrement TTL ;
            send packet out on the interface the route points to ;
        else if default route exists then
            decrement TTL ;
            send packet out on the default outgoing interface ;
        else
            drop packet ;
            optionally, send an ICMP error back to sender ;
        end
    end
end
```

General routing algorithm

***Internet Control Message Protocol** (**ICMP**) is used to exchange operational information between network devices. In the* *preceding example, one or multiple routers might decide to send back an ICMP error if they are configured to do so.*

Note that while trying to match the prefix to routes in the routing table, multiple routes might match. If that happens, the router must select the most specific match and use that for forwarding. Since the most specific routes will have the maximum number of leading 1s, and hence the largest prefix, this is called the longest prefix match. Say our router has the following routing table, as shown in the diagram. **eth1**, **eth2,** and **eth3** are three network interfaces attached to our router, each having a different IP address in different networks:



192.168.0.0/16	eth2
192.168.1.0/24	eth1
192.168.1.32/28	eth3

Longest prefix matching example

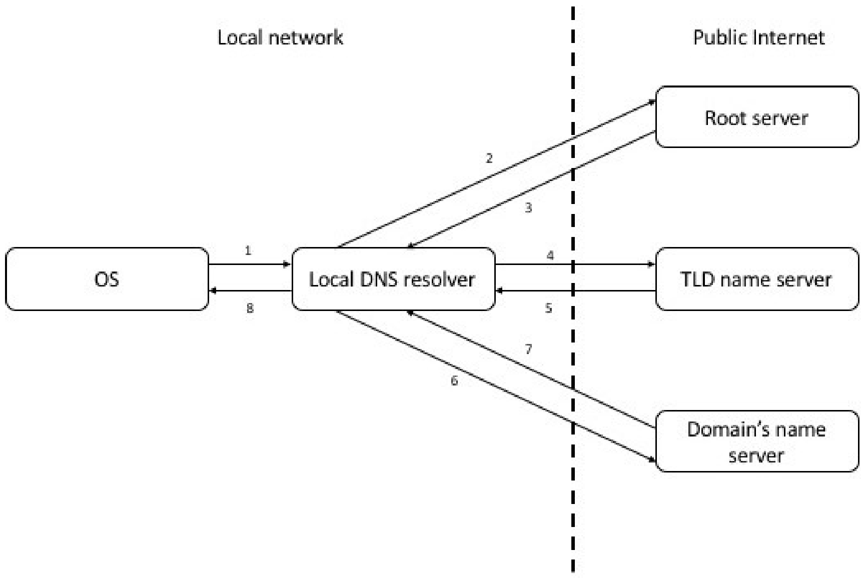
At this point, if our device gets a packet that has a destination address set to 192.168.1.33, all three prefixes have this address but the last one is the largest of the three. So, the packet will go out through **eth3**.

A lot of what we described so far about IPv4 addresses does not change for IPv6, except, of course, it has a larger address space of 128 bits. In this case, the length of the network mask and the host mask depends on the address type.

One might be wondering, how do routers construct the routing table? As always, there are protocols to help with that. Routing protocols are of two major types: interior gateway protocols which are used for routing inside an autonomous system, and exterior gateway protocols which are used in routing between autonomous systems; an example of the latter is BGP. Interior gateway protocols can again be of two types, depending on how they look at the whole network. In link state routing, each router participating in the protocol maintains a view of the whole network topology. In distance vector routing, each router only knows about its one hop neighbors. An example of the former is the **Routing Information Protocol** (**RIP**) and of the latter is **Open Shortest Path First** (**OSPF**). Details about these are beyond the scope of this book. However, we can note that the common theme among all the routing protocols is that they work by exchanging information between routers. Thus, they have their own packet formats for encapsulating that information.

How DNS works

Note that it’s impossible for anyone to remember the IP address of each and every service on the internet. Fortunately, there is a protocol for that! The **Domain Name Server** (**DNS**) solves this problem by maintaining a map of a human readable hierarchical name to the IP address of the service in a distributed database. Thus, when a user enters http://www.google.com in their browser and hits the *Enter* key, the first step is to look up the IP address of the name *www.google.com* using DNS. The next figure shows the steps necessary in such a query. In this discussion, we will use the names **local DNS resolver**, **local DNS server,** and **local DNS nameserver** interchangeably:



How DNS works

An application that needs to resolve a name will use a system call like `getaddrinfo`. This essentially asks the OS to go ahead and resolve the name. This step is not shown in the figure. The next steps are as follows:

1. Typically, each computer in a network will have a local DNS server configured in the file `/etc/resolv.conf`. In most cases, this points to the ISP’s DNS server. This might also point to the home Wi-Fi router’s DNS server. In that case, the DNS will transparently proxy requests to the ISP’s DNS server. The OS will then query that server, asking the IP of the given name *www.google.com.*

2. The local DNS server will, in turn, ask the same question to a pre-populated list of root name servers. These servers are maintained by ICANN and their addresses are well-known. They maintain addresses for the top level domain name servers. This means that they know the addresses of namesevers for the `.com` domain.
3. In this step, the root name server replies with the addresses of TLD name servers for the `.com` domain. These servers maintain a list of addresses for name servers in their own domains.
4. The local DNS server then contacts one of those and asks the same question.
5. The TLD name server replies back with the addresses of servers in the `google.com` domain. An admin of the `google.com` domain maintains a bunch of nameservers for that domain. Those nameservers have full authority over all records in that domain, and each of those records are marked *authoritative* to indicate that.
6. The local DNS server then asks one of those the same question.
7. (Hopefully) that server does know the address of *www.google.com*. If it does, it prepares a response, marks it as authoritative, and sends it back to the local DNS server. The answer can also have a time to live associated with it so that the local DNS server can cache it for future use and evict it after the given time is over. If it does not, name resolution will fail and it will send back a special response called NXDOMAIN.
8. The local DNS server then sends back the same response to the OS, which delivers it to the application. The local server marks the response as non-authoritative, indicating that it got that answer from somewhere else.

Interestingly, DNS is like asking a friend for someone's address, who then says *I do not know, but I know someone who knows someone who knows someone who might know. I can find out for you!* They then go and ask around and return with a reply.

DNS packets are often very small since they have a small question and answer along with some control information, and since DNS does not need very high reliability from the transport layer, this makes it an ideal candidate for using UDP (described in the next section). However, most implementations include an option to fall back to TCP if the transport is too unreliable.

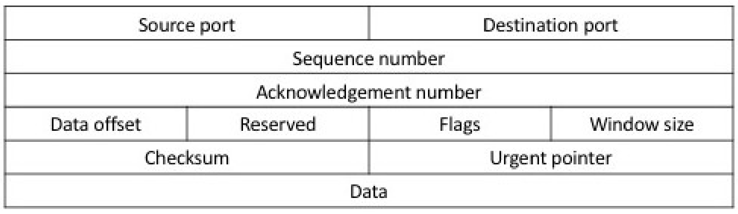
DNS supports multiple record types for various things. The A record maps a name to an IPv4 address, AAAA record maps a *name to a IPv6 address, and so on. Reverse lookups are supported using PTR records.*

Common service models

For two hosts to communicate via a network, they will need to send messages to each other. There are two models of exchanging messages, and each has specific usecases where they work best. In this section, we will explore these. Note that the service models are properties of the protocols and that they set expectations around what a consumer should expect from them.

Connection-oriented service

The service a protocol provides to its consumers is connection oriented when each party involved negotiates a virtual connection before sending the actual data. During the setup process, a number of parameters about the connection must be agreed upon. This is analogous to the older wired telephone systems, where a dedicated connection is set up between the two hosts. In modern networks, an example is TCP. The PDU for TCP is a segment, which consists of a header and a data section. The header has a few fields which are used to transition between states of the protocol state machine. The next figure shows what the TCP header looks like in practice. Each of the rows in this figure are of 32 bits (thus, each row is two octets), and some are divided into multiple segments:



Source port		Destination port	
Sequence number			
Acknowledgement number			
Data offset	Reserved	Flags	Window size
Checksum		Urgent pointer	
Data			

TCP header format

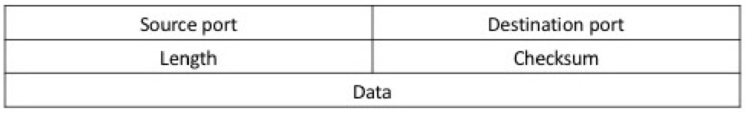
We will look at a few of these which are used for manipulating the connection between hosts:

- Control bits (flags) are a set of 9 bits that are used for various purposes. The flags of interest here are SYN, ACK, FIN, and RST. SYN triggers a synchronization of sequence numbers. The ACK flag indicates that the receiver should care about the corresponding acknowledgment number. The FIN flag starts the process of tearing down a connection. The RST flag resets the connection in case of an error.
- The sequence number is a 32-bit field which is used to reorder messages at the receiver. When the SYN flag is set (which should be the case only for the first packet in a connection), the sequence number is the initial sequence number; otherwise, it is the sequence number accumulated so far.
- The acknowledgement number is a 32-bit field which is used to enable reliable delivery of messages. If the ACK flag is set, this value is the next sequence number that the sender is expecting.

Before two hosts running TCP can start exchanging data, they must do a three-way handshake to establish a connection. This works like this: the client that wants to initiate communication sends a SYN packet to the server. The sequence number is set to a random value and the SYN flag is set to 1. The server responds with a packet that has both SYN and ACK set to 1. This packet has the acknowledgment number set to one more than what it got from the client, and the sequence number is set to a random number. Finally, the client responds with a packet that has the ACK flag set, the sequence number set to the received acknowledgement number in the last step, and the acknowledgement number is set to one more than the sequence number in the previous step. After this is done successfully, both the client and the server have agreed on sequence and acknowledgement numbers. The advantage of this model is that is has a reliable connection where both the sender and the receiver knows what to expect. The sender can tune the rate of sending data, depending on how fast or slow the receiver is or how congested the network is. The disadvantage here is the higher connection setup costs. Assuming it takes 100 ms to send a packet to a host in another continent, we will need to exchange at least 3 packets before we can begin sending data. That amounts to a delay of 300 ms. While this might not look like a lot, remember that at any given point, a laptop being used for accessing Facebook might have thousands of connections open to servers all over the world. The connection oriented service model works fine for a large number of use cases, but there are a few cases where the overhead is either significant or unnecessary. An example is video streaming. In this case, a few missing packets do not cause a huge problem since no one notices a small number of misaligned pixels in the video. These applications prefer a connectionless model, as described below.

Connectionless service

The second case here is a connectionless service. This is used when multiple messages bear no relation to one another, and thus these protocols do not need any connection negotiation step before sending any data. An example of this is UDP, which provides no guarantees of the sequence or reliability of transmitted messages (it does, however, have a checksum field to guarantee the correctness of the datagram). One should note that the protocol running above UDP is always free to implement reliability if that is desired. Interestingly, IP routing is also a connectionless service. The UDP header is shown as follows:



Source port	Destination port
Length	Checksum
Data	

UDP header format

It’s easy to see that the header here is far smaller than a TCP header. It also lacks a number of fields that TCP uses to manage the connection and tune it according to network congestion and so on. Since UDP does not have those fields, it cannot provide those guarantees.

The network programming interface in Linux

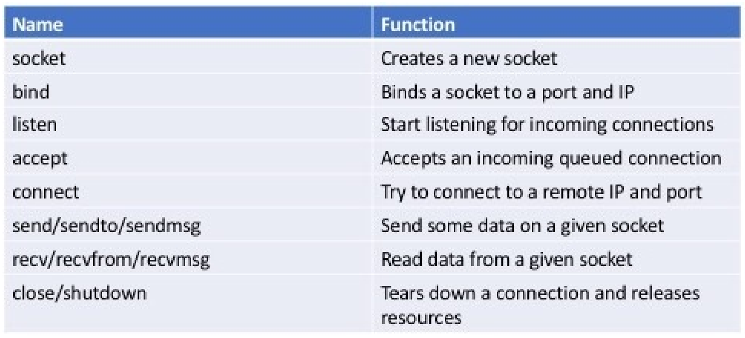
In this section, we will see how Linux (and a lot of other members of the Unix family) implement common network patterns, and how a user will interact with those while writing networking applications. All discussions in this section will be strictly based on a Linux-like OS with the standard C library (glibc). The **Portable OS Interface** (**POSIX**) standard includes all of these, making them portable to any POSIX compliant OS. All functions and data structures here follow C (and C++) coding conventions, but as we will see later, some of these are available in Rust as well through libc bindings.

The most important networking primitive that the OS provides is a *socket.* Now, what is a socket? A socket is a glorified file descriptor, a unique ID that is assigned to each file in a Unix-like OS. This follows from the Unix philosophy that everything should be a file; treating the connection between two hosts over a network as a file enables the OS to expose it as a file descriptor. The programmer is then free to use traditional I/O-related syscalls to write and receive from that file.

Now, obviously, a socket needs to hold some more data than a regular file descriptor. For instance, it needs to track the remote IP and port (and also the local IP and port). Thus, a socket is a logical abstraction for the connection between two hosts, along with all information needed to transfer data between those hosts.

There are two major classes of sockets: UNIX sockets for communicating with processes on the same host, and internet sockets for communication over an IP network.

The standard library also provides a few system calls for interacting with sockets. Some of those are socket specific and some of them are generic I/O syscalls that support writing to file descriptors. Since a socket is basically a file descriptor, those can be used to interact with sockets. Some of these are described in the next image. Note that not all applications will need to use all of these syscalls. A server, for instance, will need to call listen to start listening for incoming connections once it has created a socket. It will not need to call connect for that same connection:



Name	Function
socket	Creates a new socket
bind	Binds a socket to a port and IP
listen	Start listening for incoming connections
accept	Accepts an incoming queued connection
connect	Try to connect to a remote IP and port
send/sendto/sendmsg	Send some data on a given socket
recv/recvfrom/recvmsg	Read data from a given socket
close/shutdown	Tears down a connection and releases resources

Common networking system calls

Any Unix-like OS will have detailed documentation for each of these syscalls in the manpages. The docs for the socket syscall, for example, can be accessed using the command `man socket`. The second argument to the `man` command is the section of the manpages.

Let’s look at the signatures of these syscalls in more detail. Unless otherwise mentioned, all of these return 0 on success or -1 on failure, and set the value of errno accordingly.

```
int socket(int domain, int type, int protocol);
```

The first parameter for the `socket` syscall tells it what kind of communication `socket` will be used. Common types are AF_INET for IPv4, AF_INET6 for IPv6, AF_UNIX for IPC, and so on. The second parameter tells it what type of socket should be created, common values being SOCK_STREAM for a TCP socket, SOCK_DGRAM for a UDP socket, SOCK_RAW for a raw socket which provides direct access to the network hardware at packet level, and so on. The last parameter denotes the layer 3 protocol to be used; in our case, this is exclusively IP. A complete list of supported protocols is available in the file `/etc/protocols`.

On success, this returns a new file descriptor that the kernel assigns to the socket created.

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

The first parameter for bind is a file descriptor, generally one returned by the socket system call. The second parameter is the address to be assigned to the given socket, passed as a pointer to a structure. The third parameter is the length of the given address.

```
int listen(int sockfd, int backlog);
```

`listen` is a function that takes in the file descriptor for the socket. Note that when an application is listening for incoming connections on a socket, it might not be able to read from it as fast as packets arrive. To handle cases like this, the kernel maintains a queue of packets for each socket. The second parameter here is the maximum length of the queue for the given socket. If more clients are trying to connect after the given number here, the connection will be closed with a connection refused error.

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

This call is used to accept connections on TCP sockets. It takes a connection of the queue for the given socket, creates a new socket, and returns the file descriptor for the new socket back to the caller. The second argument is a pointer to a socket address `struct` that is filled in with the info of the new socket. The third argument is its length.

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

This function connects the socket given by the first argument to the address specified in the second argument (the third argument being the length of the address `struct`).

```
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

This is used to send data over a socket. The first argument tells it which socket to use. The second argument is a pointer to the data to be sent, and the third argument is its length. The last argument is bitwise OR of a number of options which dictates how packets should be delivered in this connection.

This system call returns the number of bytes sent on success.

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

This one is the counterpart of send. As usual, the first argument tells it which socket to read from. The second argument is a pointer to an allocated space where it should write the data it reads, and the third argument is its length. `flags` here has the same meaning as in the case of send.

This function returns the number of bytes received on success:

```
int shutdown(int sockfd, int how);
```

This function shuts down a socket. The first argument tells it which socket to shut down. The second argument dictates if any further transmission or reception should be allowed before the socket is shut down.

```
int close(int fd);
```

This system call is used to destroy file descriptors. Consequently, this can be used to close and clean up a socket as well, given its file descriptor number. While `shutdown` allows the socket to receive pending data and not accept new connections, a `close` will drop all existing connections and cleanup resources.

Other than the ones noted above, a host will also need to resolve the IP of a remote host using DNS. The `getaddrinfo` syscall does that. There are some other syscalls that provide various useful information for writing applications: `gethostname` returns the host name of the current computer, `setsockopt` sets various control options on a socket, and so on.

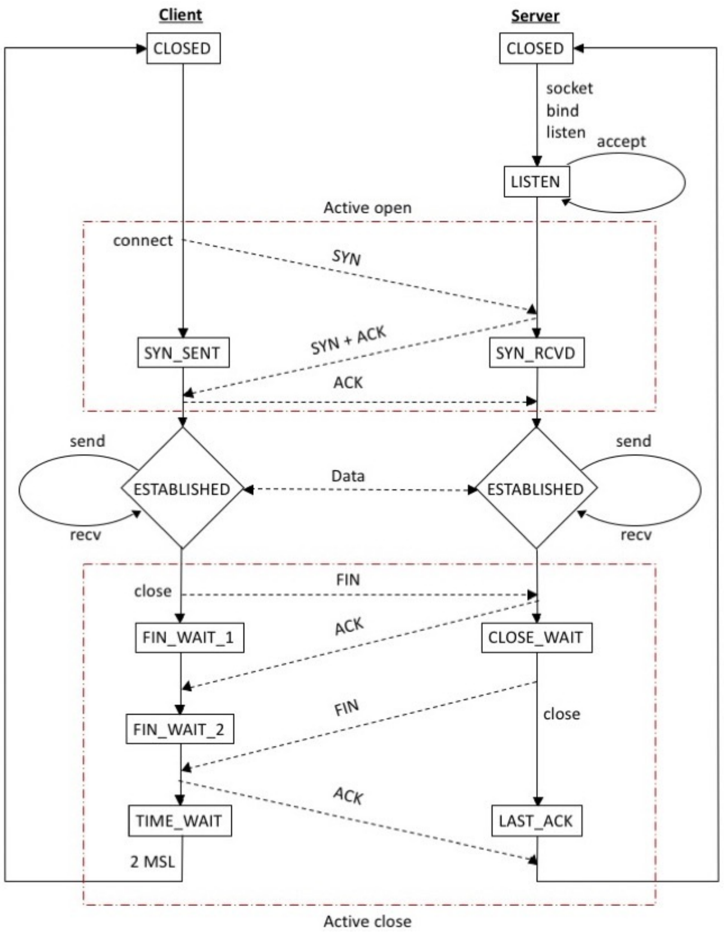
Note that a lot of syscalls described above are blocking, which means they block the thread they are invoked in waiting for the given operation to finish. For example, the *read* syscall will block on the socket if enough data is not available to fill the buffer provided. Often, this is not desirable, especially in modern multithreaded environments where a blocking call will not be able to take full advantage of the computing power available since the thread will loop around doing nothing useful.

Unix provides some more syscalls that enable asynchronous, non-blocking applications using the standard C library. There are two standard ways of doing this:

- Using the *select* system call: This syscall monitors a list of given sockets and lets the caller know if any of those has data to read from. The caller can then retrieve those file descriptors using some special macros and read from those.
- Using the *poll* system call: The high-level semantics here is similar to that of *select*: it takes in a list of socket file descriptors and a timeout. It monitors those asynchronously for the given timeout, and if any of those have some data, it lets the caller know. Unlike *select*, which checks for all conditions (readability, writability, and error) on all file descriptors, poll only cares about the list of file descriptors and conditions it receives. This makes *poll* easier to work with and faster than *select*.

In practice, however, select and poll are both very slow for applications which might need to monitor a lot of sockets for connections. For such applications, either *epoll* or an event-based networking library like libevent or libev might be more suitable. The gain in performance comes at the cost of portability; those libraries are not available in all systems since they are not part of the standard library. The other cost is complexity in writing and maintaining applications based on external libraries.

In the following section, we will walk through the state transitions of a TCP server and client that is communicating over a network. There are some idealistic assumptions here for the sake of simplicity: we assume that there are no intermediate errors or delays of any kind, that the server and the client can process data at the same rate, and that neither the server nor the client crash while communicating. We also assume that the client initiates the connection (**Active open**) and closes it down (**Active close**). We do not show all the possible states of the state machine since that will be way too cumbersome:



TCP state transition for a server and a client

Both the server and the client start from the **CLOSED** state. Assuming the server starts up first, it will first acquire a *socket*, *bind* an address to it, and start *listening* on it. The client starts up and calls **connect** to the server's address and port. When the server sees the connection, it calls **accept** on it. That call returns a new socket from which the server can read data from. But before actual data transmission can occur, the server and the client must do the three-way handshake. The client initiates that by sending a **SYN**, the server reads that, responds with a **SYN + ACK** message, and goes to the **SYN_RCVD** state. The client goes to the **SYN_SENT** state.

When the client gets the **SYN + ACK**, it sends out a final **ACK** and goes to the **ESTABLISHED** state. The server goes to **ESTABLISHED** when it gets the final **ACK**. The actual connection is established only when both parties are in the **ESTABLISHED** state. At this point, both the server and the client can **send** and **receive** data. These operations do not cause a state change. After some time, the client might want to **close** the connection. For that, it sends out a **FIN** packet and goes to the **FIN_WAIT_1** state. The server receives that, sends an **ACK**, and goes to the **CLOSE_WAIT** state. When the client gets that, it goes to the **FIN_WAIT_2** state. This concludes the first round of connection termination. The server then calls **close**, sends out a **FIN**, and goes to the **LAST_ACK** state. When the client gets that, it sends out an **ACK** and goes to the **TIME_WAIT** state. When the server receives the final **ACK**, it goes back to the **CLOSED** state. After this point, all server resources for this connection are released. The client, however, waits for a timeout before moving on to the **CLOSED** state where it releases all client-side resources.

Our assumptions here are pretty basic and idealistic. In the real world, communication will often be more complex. For example, the server might want to push data, and then it will have to initiate the connection. Packets might be corrupted in transit, causing either of the parties to request retransmission, and so on.

Maximum Segment Lifetime *(MSL) is defined to be the maximum time a TCP segment can exist in the network. In most modern systems, it is set to 60 seconds.*

Summary

This chapter started with motivations for writing networking applications in the modern world. We also took a look at how networking evolved. We studied common networking technologies and ideas, and looked at how they work together; starting from simple IP routing and DNS to TCP and UDP. We then studied how Linux (and POSIX) in general supports synchronous and asynchronous network programming.

In the next chapter, we will look at Rust and try to understand its benefits over existing platforms. Having motivated both networking and Rust, we will move on to network programming using Rust.

Introduction to Rust and its Ecosystem

The Rust programming language is sponsored by Mozilla and supported by a community of developers from across the globe. Rust is promoted as a systems programming language that supports automatic memory management without the overhead of a runtime or a garbage collector, concurrency without data races enforced by the compiler, and zero cost abstractions and generics. In subsequent sections, we will discuss these features in more detail. Rust is statically typed and borrows a number of functional programming ideas. A fascinating aspect of Rust is the use of the type system to guarantee memory safety without using a runtime. This makes Rust uniquely suitable for low-resource embedded devices and real-time systems, which require strong guarantees around code correctness. On the flip side, this often means that the compiler has to do a lot more work to ensure syntactical correctness and then translate source code, resulting in higher build times. While the community is working on methods to reduce compile time as much as possible, this is still an important issue encountered by a lot of developers.

The **Low Level Virtual Machine** (**LLVM**) project started as a university research project aimed at developing a set of tools for building compilers that can generate machine code for a range of CPU architectures. This was achieved using the **LLVM intermediate representation** (**LLVM IR**). The toolchain can compile any higher-level language to LLVM IR, which can then be targeted for a given CPU. The Rust compiler depends heavily on the LLVM project for interoperability, using it as a backend. It actually translates Rust code into LLVM's intermediate representation and optimizes it as necessary. LLVM then translates that into machine code for the specific platform, which gets run on the CPU.

In this chapter, we will cover the following topics:

- An introduction to the ecosystem and how Rust works
- Installing Rust and setting up the toolchain
- An introduction to its major features, starting from the borrow checker and how ownership works
- Generics and how the trait system works with the ownership model
- Error handling and the macro system

- Concurrency primitives
- Testing primitives

Note that this chapter is a very high-level overview of the language and some of its most distinctive features, not a deep dive.

The Rust ecosystem

The success or failure of an open source project is often determined by the strength of the community around it. Having a coherent ecosystem helps in building a strong community. Since Rust is primarily driven by Mozilla, they have been able to build a strong ecosystem around it, the primary components being:

- Source code: Rust hosts all the source code in GitHub. Developers are encouraged to report bugs and submit pull requests in there. At the time of writing, the Rust repository on GitHub has 1,868 unique contributors, over 2,700 open bug reports, and 90 open pull requests. The core Rust team is composed of Mozilla employees and contributors from other organizations (like Google, Baidu, and so on). The team uses GitHub for all collaborations; even major changes to any component have to be first proposed by writing a **Request For Comments** (**RFC**). This way, everyone has a chance to take a look at it and collaborate on improving it. Once it is approved, the actual change can be implemented.
- Compiler: The Rust compiler is named *rustc*. Since Rust follows semantic versioning for compiler releases, there cannot be any backward incompatible breaking changes between minor releases. At the time of writing this book, the compiler has already reached version 1.0, thus it can be assumed that there will not be any breaking changes till version 2.0. Note that breaking changes do slip once in a while. But in all those cases, they are treated as bugs and fixed as soon as possible.

To facilitate adding new compiler features without breaking existing dependent libraries, Rust releases new compiler versions in stages. At any point, three different versions of the compiler (and the standard library) are maintained.

- The first one is called **nightly**. As the name implies, it is built each night from the tip of the source tree. Since this is only tested by unit and integration tests, this release often has more bugs in the real world.
- The second stage is **beta**, which is a planned release. By the time a nightly has reached this stage, it has gone through multiple rounds of unit, integration, and regression testing. Additionally, the community has

had the time to use it in real projects and share feedback with the Rust team.
- Once everyone is confident about the release, it is tagged as a **stable** release and pushed out. Since the compiler supports a variety of platforms (from Windows to Redox) and architectures (amd64), each release has pre-built binaries for all combinations of platforms and architectures.

- Installation mechanism: The community supported installation mechanism is via a tool called *rustup.* This tool can install a given version of Rust along with everything needed to use it (including the compiler, standard library, package manager, and so on).
- Package manager: Rust's package manager is called *Cargo*, while individual packages are called *crates*. All external libraries and applications can be packaged in a crate and published using the Cargo CLI tool. A user can then use it to search for and install packages. All crates can be searched using the following website: https://crates.io/. For all packages hosted on *crates.io*, the corresponding documentation is available at: https://docs.rs/.

Getting started with Rust

The Rust toolchain installer is available at: https://www.rustup.rs/. The following commands will install all three versions of the toolchain on a system. For the examples in this book, we will use a Linux machine running Ubuntu 16.04. While most of Rust should not depend on the OS, there can be minor differences.

We will point out any strict dependencies on the OS:

```
# curl https://sh.rustup.rs -sSf | sh
# source $HOME/.cargo/env
# rustup install nightly beta
```

We will need to put Cargo's bin directory to our **PATH** by editing **.bashrc**. Run the following to do that:

```
$ echo "export PATH=$HOME/.cargo/bin:$PATH" >> ~/.bashrc
```

A Rust installation comes with a lot of documentation built in; they can be accessed by running the following command. This should open up the documentation in a browser window:

```
# rustup doc
```

The next step is to set up a Rust project and run it, all using Cargo:

```
# cargo new --bin hello-rust
```

This tells Cargo to set up a new project called `hello-rust` in the current directory. Cargo will create a directory of that name and set up the basic structure. Since the type of this project is set to be a binary, Cargo will generate a file called `main.rs` which will have an empty `main` function, the entry point for the application. The other (default) option here is that of a library, in this case, a file named `lib.rs` will be generated. The file named `Cargo.toml` has a bunch of metadata for the current project and is used by Cargo. All source code is located in the `src` directory:

```
# tree hello-rust/
hello-rust/
├── Cargo.toml
└── src
    └── main.rs

1 directory, 2 files
```

The project can then be built and run using the following command. Note that this command should be run from the `hello-rust` directory that Cargo created previously:

```
# cargo run
```

Interestingly, this command modifies the directory quite a bit. The `target` directory contains compilation artifacts. The structure of this is heavily platform dependent, but always includes everything necessary to run the application in the given build mode. The default build mode is `debug`, which includes debugging information and symbols to be used with a debugger:

```
# tree hello-rust/
hello-rust/
├── Cargo.lock
├── Cargo.toml
├── src
│   └── main.rs
└── target
    └── debug
        ├── build
        ├── deps
        │   └── hello_rust-392ba379262c5523
        ├── examples
        ├── hello-rust
        ├── hello-rust.d
        ├── incremental
        └── native

8 directories, 6 files
```

A packaged Rust application is called a crate. Rust crates are published to *crates.io*. Once published, anyone can use the web interface or the cargo CLI to search for crates, as shown in the following code snippet. This operation needs a working internet connection so that cargo can communicate with *crates.io*:

```
# cargo search term
    Updating registry `https://github.com/rust-lang/crates.io-index`
term = "0.4.6" # A terminal formatting library
ansi_term = "0.9.0" # Library for ANSI terminal colours and styles (bold,
underline)
term-painter = "0.2.4" # Coloring and formatting terminal output
term_size = "0.3.0" # functions for determining terminal sizes and dimensions
rust_erl_ext = "0.2.1" # Erlang external term format codec.
slog-term = "2.2.0" # Unix terminal drain and formatter for slog-rs
colored = "1.5.2" # The most simple way to add colors in your terminal
term_grid = "0.1.6" # Library for formatting strings into a grid layout
rust-tfidf = "1.0.4" # Library to calculate TF-IDF (Term Frequency - Inverse
Document Frequency) for generic documents
aterm = "0.20.0" # Implementation of the Annotated Terms data structure
... and 1147 crates more (use --limit N to see more)
```

Now, say we want to use the `term` crate in our application; we will need to edit the `Cargo.toml` file and include it in the `[dependencies]` section, as shown in the following code snippet. This particular crate helps in formatting terminal colors. Using this crate, we would like to print the words `hello` and `world` in green and red, respectively:

```
[package]
name = "hello-rust"
version = "0.1.0"
authors = ["Foo Bar <foo.bar@foobar.com>"]
```

```
[dependencies]
term = "0.4.6"
```

To actually use the crate in our application, we need to modify the `main.rs` file, as shown in the following code snippet. We will discuss the language in detail in subsequent sections, but here is a small overview. The following sample is a simple program that prints the string `hello world!` on the screen. Further, it prints `hello` in green and `world!` in red:

```
// chapter2/hello-rust/src/main.rs

extern crate term;

fn main() {
    let mut t = term::stdout().unwrap();
    t.fg(term::color::GREEN).unwrap();
    write!(t, "hello, ").unwrap();

    t.fg(term::color::RED).unwrap();
    writeln!(t, "world!").unwrap();

    t.reset().unwrap();
}
```

In Rust, each application must have a single entry point called `main`, which should be defined as a function that does not take in parameters. Functions are defined using the `fn` keyword. The phrase `extern crate term` tells the toolchain that we want to use an external crate as a dependency of our current application.

Now, we can run it using Cargo. It automatically downloads and builds the library we need and all of its dependencies. Finally, it calls the Rust compiler so that our application is linked with the library and runs the executable. Cargo also generates a file called `Cargo.lock` that has a snapshot of everything needed to run the application in a consistent manner. This file should never be edited manually. Since cargo caches all dependencies locally, subsequent invocations do not need internet access:

```
$ cargo run
    Updating registry `https://github.com/rust-lang/crates.io-index`
   Compiling term v0.4.6
   Compiling hello-rust v0.1.0 (file:///Users/Abhishek/Desktop/rust-
book/src/chapter2/hello-rust)
    Finished dev [unoptimized + debuginfo] target(s) in 3.20 secs
     Running `target/debug/hello-rust`
hello, world!
```

Introduction to the borrow checker

The most important aspect of Rust is the ownership and borrowing model. Based on the strict enforcing of borrowing rules, the compiler can guarantee memory safety without an external garbage collector. This is done by the borrow checker, a subsystem of the compiler. By definition, every resource created has a lifetime and an owner associated with it, which operates under the following rules:

- Each resource has exactly one owner at any point in time. By default, the owner is the variable that created that resource, and its lifetime is the lifetime of the enclosing scope. Others can borrow or copy the resource if they need to. Note that a resource can be anything from a variable or a function. A function takes ownership of a resource from its caller; returning from the function transfers back ownership.
- When the owner's scope has finished executing, all resources owned by it will be dropped. This is statically computed by the compiler, which then produces machine code accordingly.

Some examples of these rules are shown in the following code snippet:

```
// chapter2/ownership-heap.rs

fn main() {
    let s = String::from("Test");
    heap_example(s);
}

fn heap_example(input: String) {
    let mystr = input;
    let _otherstr = mystr;
    println!("{}", mystr);
}
```

In Rust, variables are declared using the `let` keyword. All variables are immutable by default and can be made mutable by using the `mut` keyword. The `::` syntax refers to an object in the given namespace, in this case, the `from` function. `println!` is a built-in macro that the compiler provides; it is used to write to the standard output with a trailing newline. Functions are defined using the `fn` keyword. When we try to build it, we get the following error:

```
# rustc ownership-heap.rs
error[E0382]: use of moved value: `mystr`
 --> ownership-heap.rs:9:20
  |
8 | let _otherstr = mystr;
  | --------- value moved here
```

```
9 | println!("{}", mystr);
  | ^^^^^ value used here after move
  |
  = note: move occurs because `mystr` has type `std::string::String`, which does
not implement the `Copy` trait

error: aborting due to previous error
```

In this example, a string resource is created which is owned by the variable `mystr`, in the function `heap_example`. Thus, its lifetime is the same as its scope. Under the hood, since the compiler does not know the length of the string at compile time, it must place it on the heap. The owner variable is created on the stack and points to the resource on the heap. When we assign that resource to a new variable, the resource is now owned by the new variable. Rust will mark `mystr` as invalid at this point to prevent situations where the memory associated with the resource might be freed multiple times. Thus, compilation fails here to guarantee memory safety. We can force the compiler to copy the resource and have the second owner point to the newly created resource. For that, we will need to `.clone()` the resource named `mystr`. Here is how it looks:

```
// chapter2/ownership-heap-fixed.rs

fn main() {
    let s = String::from("Test");
    heap_example(s);
}

fn heap_example(input: String) {
    let mystr = input;
    let _otherstr = mystr.clone();
    println!("{}", mystr);
}
```

As expected, this does not throw any errors on compilation and prints the given string `"Test"` on running. Notice that till now, we have been using Cargo for running our code. Since in this case, we just have a simple file and no external dependency, we will use the Rust compiler directly to compile our code, and we will run it manually:

```
$ rustc ownership-heap-fixed.rs && ./ownership-heap-fixed
Test
```

Consider the code sample below which shows the case when resources are stored on the stack:

```
// chapter2/ownership-stack.rs

fn main() {
    let i = 42;
    stack_example(i);
}

fn stack_example(input: i32) {
    let x = input;
    let _y = x;
    println!("{}", x);
}
```

Interestingly, though it looks exactly the same as the code block before, this does not throw a compilation error. We build and run this using the Rust compiler directly from the command line:

```
# rustc ownership-stack.rs && ./ownership-stack
42
```

The difference is in the types of the variables. Here, the original owner and the resource are both created on the stack. When the resource is reassigned, it is copied over to the new owner. This is possible only because the compiler knows that the size of an integer is always fixed (and hence can be placed on the stack). Rust provides a special way to say that a type can be placed on the stack via the `Copy` trait. Our example works only because built-in integers (and some other types) are marked with this trait. We will explain the trait system in more detail in subsequent sections.

One might have noticed that copying a resource of unknown length to a function might lead to memory bloats. In a lot of languages, the caller will pass a pointer to a memory location, and then to the function. Rust does this by using references. These allow you to refer to a resource without actually owning it. When a function receives a reference to a resource, we say that it borrowed that resource. In the following example, the function `heap_example` borrows the resource owned by the variable `s`. Since borrowing is not absolute ownership, the scope of the borrowing variable does not affect how memory associated with the resource is freed. That also means there is no chance of freeing the borrowed resource multiple times in the function, since nobody in the function's scope actually owns that resource. Thus, the earlier code that failed works in this case:

```
// chapter2/ownership-borrow.rs

fn main() {
    let s = String::from("Test");
    heap_example(&s);
}

fn heap_example(input: &String) {
    let mystr = input;
    let _otherstr = mystr;
    println!("{}", mystr);
}
```

The borrowing rules also imply that the borrow is immutable. However, there can be cases where one needs to mutate the borrow. To handle those cases, Rust allows mutable references (or borrowing). As one would expect, this takes us back to the problem we had in the first example, and compilation fails with the following code:

```
// chapter2/ownership-mut-borrow.rs

fn main() {
    let mut s = String::from("Test");
    heap_example(&mut s);
}

fn heap_example(input: &mut String) {
    let mystr = input;
    let _otherstr = mystr;
    println!("{}", mystr);
}
```

Note that a resource can be mutably borrowed only once in a scope. The compiler will refuse to compile code that tries to do otherwise. While this might look like an annoying error, you need to remember that in a working application, these functions will often be called from competing threads. If there is a synchronization error due to a programming fault, we will end up with a data race where multiple unsynchronized threads race to modify the same resource. This feature helps prevent such situations.

Another language feature that is closely related to references is that of a lifetime. A reference lives as long as it is in scope, thus its lifetime is that of the enclosing scope. All variables declared in Rust can have a lifetime explicit elision that puts a name to its lifetime. This is useful for the borrow checker to reason about the relative lifetimes of variables. In general, one does not need to put an explicit lifetime name to each and every variable since the compiler manages that. In certain scenarios, this is needed, especially when automatic lifetime determination cannot work. Let's look at an example where this happens:

```
// chapter2/lifetime.rs

fn main() {
    let v1 = vec![1, 2, 3, 4, 5];
    let v2 = vec![1, 2];

    println!("{:?}", longer_vector(&v1, &v2));
}

fn longer_vector(x: &[i32], y: &[i32]) -> &[i32] {
    if x.len() > y.len() { x } else { y }
}
```

The `vec!` macro constructs a vector from the given list of objects. Note that unlike previous examples, our function here needs to return a value back to the caller. We need to specify the return type using the arrow syntax. Here, we are given two vectors, and we want to print the longest of the two. Our `longer_vector` function does just that. It takes in references to two vectors, computes their length, and returns a reference to the one with the larger length. This fails to compile with the following error:

```
# rustc lifetime.rs
```

```
error[E0106]: missing lifetime specifier
 --> lifetime.rs:8:43
  |
8 | fn longer_vector(x: &[i32], y: &[i32]) -> &[i32] {
  | ^ expected lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but the signature
does not say whether it is borrowed from `x` or `y`

error: aborting due to previous error
```

This tells us that the compiler could not determine if the returned reference should refer to the first parameter or the second, so it could not determine how long it should live. There is no way this can be determined at compile time since we have no control of the inputs. A key insight here is that we do not need to know the lifetimes of all the references at compile time. We need to make sure the following things hold true:

- The two inputs should have the same lifetimes since we want to compare their lengths in the function
- The return value should have the same lifetime as that of the input, which is the longer of the two

Given these two axioms, it follows that the two inputs and the return should have the same lifetime. We can annotate this, as shown in the following code snippet:

```
fn longer_vector<'a>(x: &'a[i32], y: &'a[i32]) -> &'a[i32] {
    if x.len() > y.len() { x } else { y }
}
```

This works as expected since the compiler can happily guarantee code correctness. Lifetime parameters can also be attached to structs and method definitions. There is a special lifetime called `'static` that refers to the entire duration of the program.

Rust recently accepted a proposal to add a new designated lifetime called `'fn` that will be equal to the scope of the innermost function or closure.

Generics and the trait system

Rust supports writing generic code that is later bound with more concrete types, either during compile time or during runtime. People who are familiar with templates in C++ might notice that generics in Rust are pretty similar to templates, as far as syntax goes. The following example illustrates how to use generic programming. We also introduce some new constructs which we haven't discussed before, which we will explain as we proceed.

Much like C and C++, a Rust `struct` defines a user-defined type that aggregates multiple logically connected resources in one unit. Our struct here defines a tuple of two variables. We define a generic struct and we use a generic *type parameter*, written here as `<T>`. Each member of the struct is defined to be of that type. We later define a generic function that sums the two elements of the tuple. Let's look at a naive implementation of this:

```
// chapter2/generic-function.rs

struct Tuple<T> {
    first: T,
    second: T,
}

fn main() {
    let tuple_u32: Tuple<u32> = Tuple {first: 4u32, second: 2u32 };
    let tuple_u64: Tuple<u64> = Tuple {first: 5u64, second: 6u64 };
    println!("{}", sum(tuple_u32));
    println!("{}", sum(tuple_u64));

    let tuple: Tuple<String> = Tuple {first: "One".to_owned(), second:
"Two".to_owned() };
    println!("{}", sum(tuple));
}

fn sum<T>(tuple: Tuple<T>) -> T
{
    tuple.first + tuple.second
}
```

This fails to compile, and the compiler throws the following error:

```
$ rustc generic-function.rs
error[E0369]: binary operation `+` cannot be applied to type `T`
  --> generic-function-error.rs:18:5
   |
18 | tuple.first + tuple.second
   | ^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
   = note: `T` might need a bound for `std::ops::Add`

error: aborting due to previous error
```

This error is important. The compiler is telling us that it does not know how to add two operands of type `T`. It also (correctly) guessed that the `T` type needs to be bound by the `Add` trait. This means that the list of possible concrete types for `T` should only have types that implement the `Add` trait, which are types whose concrete references can be added. Let's go ahead and put the trait bound in the `sum` function. Our code should look like this now:

```
// chapter2/generic-function-fixed.rs

use std::ops::Add;

struct Tuple<T> {
    first: T,
    second: T,
}

fn main() {
    let tuple_u32: Tuple<u32> = Tuple {first: 4u32, second: 2u32 };
    let tuple_u64: Tuple<u64> = Tuple {first: 5u64, second: 6u64 };
    println!("{}", sum(tuple_u32));
    println!("{}", sum(tuple_u64));

    // These lines fail to compile
    let tuple: Tuple<String> = Tuple {first: "One".to_owned(), second:
"Two".to_owned() };
    println!("{}", sum(tuple));
}

// We constrain the possible types of T to those which implement the Add trait
fn sum<T: Add<Output = T>>(tuple: Tuple<T>) -> T
{
    tuple.first + tuple.second
}
```

For this to work, the elements must be summable; there should be a logical meaning of summing them. So, we have constrained the possible types the `T` parameter can have to those which have the `Add` trait implemented. We also need to let the compiler know that the output for this function should be of the type `T`. Given this information, we can construct our tuples and call the sum function on them, and they will behave as expected. Also, note that a tuple of strings will fail to compile with the error, since the `Add` trait is not implemented for strings.

From the last example, one might notice that traits are essential to properly implement generics. They help the compiler reason about properties of generic types. In essence, a trait defines properties of a type. The library defines a bunch of commonly used traits and their implementations for built-in types. For any user-defined types, it's up to the user to define what properties those types should have by defining and implementing traits.

```
// chapter2/traits.rs

trait Max<T> {
    fn max(&self) -> T;
}

struct ThreeTuple<T> {
    first: T,
    second: T,
    third: T,
}

// PartialOrd enables comparing
impl<T: PartialOrd + Copy> Max<T> for ThreeTuple<T> {
    fn max(&self) -> T {
        if self.first >= self.second && self.first >= self.third {
            self.first
        }
        else if self.second >= self.first && self.second >= self.third {
            self.second
        }
        else {
            self.third
        }
    }
}

struct TwoTuple<T> {
    first: T,
    second: T,
}

impl<T: PartialOrd + Copy> Max<T> for TwoTuple<T> {
    fn max(&self) -> T {
        if self.first >= self.second {
            self.first
        } else { self.second }
    }
}

fn main() {
    let two_tuple: TwoTuple<u32> = TwoTuple {first: 4u32, second: 2u32 };
    let three_tuple: ThreeTuple<u64> = ThreeTuple {first: 6u64,
                                       second: 5u64, third: 10u64 };

    println!("{}", two_tuple.max());
    println!("{}", three_tuple.max());
}
```

We start by defining a generic trait of type `T`. Our trait has a single function that will return the maximum of the given type that implements it. What counts as the maximum is an implementation detail and is not important at this stage. We then define a tuple of three elements, each of the same generic type. Later, we implement our trait for that type we defined. In Rust, a function returns the last expression if there is no explicit return statement. Using this style is considered idiomatic in the community. Our `max` function uses this feature in the `if...else` block. For the implementation to work, the generic types must have an ordering relation defined between them so that we can compare them. In Rust, this is achieved by constraining the possible types to those which implement the `PartialOrd` trait. We also need to put a constraint on the `Copy` trait so that the compiler can make copies of the self parameter before returning from the function. We move on to defining another tuple, which has two elements. We implement the same trait here in a similar fashion. When we actually use these in our `main` function, they work as expected.

Traits can also be used to extend built-in types and add new functionalities. Let's look at the following example:

```
// chapter2/extending-types.rs

// Trait for our behavior
trait Sawtooth {
    fn sawtooth(&self) -> Self;
}

// Extending the builtin f64 type
impl Sawtooth for f64 {
    fn sawtooth(&self) -> f64 {
        self - self.floor()
    }
}

fn main() {
    println!("{}", 2.34f64.sawtooth());
}
```

Here, we want to implement the sawtooth (https://en.wikipedia.org/wiki/Sawtooth_wave) function on the built-in `f64` type. This function is not available in the standard library, so we would need to write some code to get it working by extending the standard library. To have this seamlessly integrated into the type system, we will need to define a trait and implement it for the `f64` type. This enables us to use the new function using the dot notation like any other built-in function on the `f64` type.

The standard library provides a number of built-in traits; the most common of these are `Display` and `Debug`. These two are used to format types while printing. `Display` corresponds to the empty formatter `{}` and `Debug` corresponds to format debugging output. All of the mathematical operations are defined as traits, such as `Add`, `Div`, and so on. The compiler attempts to provide default implementations for user-defined types if they are marked with the `#[derive]` attribute. However, an implementation might choose to override any of these if necessary. An example of this is shown in the following code snippet:

```
// chapter2/derive.rs

use std::fmt;
use std::fmt::Display;

#[derive(Debug, Hash)]
struct Point<T> {
    x: T,
    y: T,
}

impl<T> fmt::Display for Point<T> where T: Display {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "({}, {})", self.x, self.y)
    }
}

fn main() {
    let p: Point<u32> = Point { x: 4u32, y: 2u32 };
    // uses Display
    println!("{}", p);

    // uses Debug
    println!("{:?}", p);
}
```

We define a generic point structure with two fields. We let the compiler generate implementations of some common traits. We must, however, implement the `Display` trait by hand, since the compiler cannot determine the best way to display user-defined types. We have to constrain the generic type to types that implement `Display` using the `where` clause. This example also demonstrates the alternate way to constrain types based on traits. Having set all of this up, we can display our point using the default formatter. This produces the following output:

```
# rustc derive.rs && ./derive
(4, 2)
Point { x: 4, y: 2 }
```

Error handling

One of Rust's major goals is enabling the developer to write robust software. An essential component of this is advanced error handling. In this section, we will take a deeper look at how Rust does error handling. But before that, let's take a detour and look at some type theory. Specifically, we are interested in **algebraic data types** (**ADT**), types formed by combining other types. The two most common ADTs are sum and product types. A `struct` in Rust is an example of a product type. This name derives from the fact that given a struct, the range of its type is essentially the Cartesian product of the ranges of each of its components, since an instance of the type has values for all of its constituent types. In contrast, a sum type is when the ADT can assume the type of only one of its constituents. An example of this is an `enum` in Rust. While similar to enums in C and other languages, Rust enums provide a number of enhancements: they allow variants to carry data.

Now, back to error handling. Rust mandates that operations which can result in an error must return a special `enum` that carries the result. Conveniently, this `enum` looks like this:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

The two possible options are called variants. In this case, they represent the non-error case and the error case, respectively. Note that this is generically defined so that an implementation is free to define the types in both cases. This is useful in applications that want to expand on the standard error type and implement custom errors. Let's look at an example of this in action:

```
// chapter2/custom-errors.rs

use std::fmt;
use std::error::Error;

#[derive(Debug)]
enum OperationsError {
    DivideByZeroError,
}

// Useful for displaying the error nicely
impl fmt::Display for OperationsError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match *self {
        OperationsError::DivideByZeroError => f.write_str("Cannot divide by zero"),
        }
    }
}
```

```
// Registers the custom error as an error
impl Error for OperationsError {
    fn description(&self) -> &str {
        match *self {
            OperationsError::DivideByZeroError => "Cannot divide by zero",
        }
    }
}

// Divides the dividend by the divisor and returns the result. Returns
// an error if the divisor is zero
fn divide(dividend: u32, divisor: u32) -> Result<u32, OperationsError> {
    if divisor == 0u32 {
        Err(OperationsError::DivideByZeroError)
    } else {
        Ok(dividend / divisor)
    }
}

fn main() {
    let result1 = divide(100, 0);
    println!("{:?}", result1);

    let result2 = divide(100, 2);
    println!("{:?}", result2.unwrap());
}
```

For this example, we define a function that simply returns the quotient when the first operand is divided by the second. This function must handle the error case when the divisor is zero. We also want it to signal an error to its caller if that is the case. Also, let's assume that this is a part of a library that will expand to include more such operations. To make the code manageable, we create an error class for our library, with one element that represents the divide by zero error. For the Rust compiler to know that the enum is an error type, our enum has to implement the `Error` trait from the standard library. It also needs to implement the `Display` trait manually. Having set up this boilerplate, we can define our division method. We will take advantage of the generic `Result` trait to annotate that on success, it should return a `u32`, the same type as the operands. On failure, it should return an error of type `OperationsError`. In the function, we raise the error if our divisor is zero. Otherwise, we carry out the division, wrap the result in a `Ok` so that it becomes a variant of the `Result` enum, and return it. In our `main` function, we call this with a zero divisor. The result will be an error, as shown by the first print macro. In the second invocation, we know that the divisor is not zero. Thus, we can safely unwrap the result to convert it from `Ok(50)` to `50`. The standard library has a number of utility functions to handle `Result` types, safely reporting the error to the caller.

Here is a sample run of the last example:

```
$ rustc custom-errors.rs && ./custom-errors
Err(DivideByZeroError)
50
```

A related idiom in the standard library is the `Option` type, as shown in the following code snippet. This is used to indicate the nullability of an operation, indicated by the `None` variant. The `Some` variant handles the case where it holds the actual value of the type `T`:

```
pub enum Option<T> {
    None,
    Some(T),
}
```

Given this type, we could have written our divide function like this:

```
// chapter2/options.rs

fn divide(dividend: u32, divisor: u32) -> Option<u32> {
    if divisor == 0u32 {
        None
    } else {
        Some(dividend / divisor)
    }
}

fn main() {
    let result1 = divide(100, 0);

    match result1 {
        None => println!("Error occurred"),
        Some(result) => println!("The result is {}", result),
    }

    let result2 = divide(100, 2);
    println!("{:?}", result2.unwrap());
}
```

We modify our function to return an `Option` of type `u32`. In our `main` function, we call our function. In this case, we can match on the return type. If it happens to be `None`, we know that the function did not succeed. In that case, we can print an error. In case it returned `Some`, we extract the underlying value and print it. The second invocation works fine since we know it did not get a zero divisor. Using `Option` for error handling can be a bit easier to manage since it involves less boilerplate. However, this can be a bit difficult to manage in a library with custom error types since errors are not handled by the type system.

Note that `Option` can be represented as a `Result` of a given type and the unit type

`type Option<T> = Result<T, ()>;`.

What we have described so far for error handling has been done with recoverable errors. In some cases, though, it might be wise to abort execution if an error occurs. The standard library provides the `panic!` macro to handle such cases. Calling this stops the execution of the current thread, prints a message on the screen, and unwinds the call stack. However, one needs to use this cautiously since in a lot of cases, a better option is to handle the error properly and bubble the error up to the caller.

A number of built-in methods and functions call this macro in case of an error. Let's look at the following example:

```
// chapter2/panic.rs

fn parse_int(s: String) -> u64 {
    return s.parse::<u64>().expect("Could not parse as integer")
}

fn main() {
    // works fine
    let _ = parse_int("1".to_owned());

    // panics
    let _ = parse_int("abcd".to_owned());
}
```

This fails with the following error:

```
# ./panic
thread 'main' panicked at 'Could not parse as integer: ParseIntError { kind:
InvalidDigit }', src/libcore/result.rs:906:4
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

Some methods that call panic are `expect()` and `unwrap()`.

The macro system

Rust supports a macro system that has evolved quite a lot over the years. A distinctive feature of Rust macros are that they are guaranteed to not refer to identifiers outside their scope by accident, and so the macro implementation in Rust is *hygienic*. As one would expect, Rust macros are expanded to source code ahead of the compilation in place, and are compiled with the translation unit. The compiler enforces scoping rules on expanded macros to make them hygienic. Rust macros differ from other constructs in that they always end in an exclamation mark `!`.

Modern Rust has two ways of working with macros; the older, syntactic macro way, and the newer, procedural macro way. Let’s look at each of these:

Syntactic macros

This system of macros has existed as part of Rust since pre- 1.0 releases. These macros are defined using a macro called `macro_rules!`. Let’s look at an example:

```
// chapter2/syntactic-macro.rs

macro_rules! factorial {
    ($x:expr) => {
        {
            let mut result = 1;
            for i in 1..($x+1) {
                result = result * i;
            }
            result
        }
    };
}

fn main() {
    let arg = std::env::args().nth(1).expect("Please provide only one argument");
    println!("{:?}", factorial!(arg.parse::<u64>().expect("Could not parse to an
integer")));
}
```

We start with defining the factorial macro. Since we do not want the compiler to refuse to compile our code as it might overflow the macro stack, we will use a non-recursive implementation. A syntactic macro in Rust is a collection of rules where the left-hand side dictates how the rule should be matched to an input, and the right-hand side dictates what it should expand to. A rule maps to an expression on the right-hand side via the `=>` operator. Variables local to a rule are declared using the `$` sign. Match rules are expressed using a special macro language which has its own set of reserved keywords. Our declaration says that we want to take in any valid Rust expression; in this specific case, it should evaluate to an integer. We will leave it to the caller to make sure that is true. We then loop over from 1 to the last integer in the range while accumulating the result. Once done, we return back the result using the implicit return syntax.

Our caller is the main function, in that we take input from the user using the `std::env` module. We fetch the first in the list of inputs and throw an error if there are no inputs. We then print the result from our macro, and we try to parse the input as a `u64` before passing to it. We also handle the case where parsing might fail. This works as expected:

```
# rustc syntactic-macro.rs && ./syntactic-macro 5
120
```

Rust also provides a few tools to debug macros. One might be interested to see what the expanded macro looks like. The `trace_macros!` macro does exactly that. To make it work, we will need to enable a feature gate, as shown in the following code snippet (since it is not stable in Rust yet, this code will only work in Rust nightly):

```
#![feature(trace_macros)]
trace_macros!(true);
```

Note that the expansion also includes `println!` since it is a macro defined in the standard library.

The same expansion can also be examined using the following command to invoke the compiler:

`rustc -Z unstable-options --pretty expanded syntactic-macro.rs`.

Procedural macros

While regular syntactic macros are useful in a number of scenarios, some applications need advanced code generation features that are better done using the AST that the compiler operates on. Thus, there was a need to extend the macro system to include this. It was later decided that the old macro system and this new system called *procedural macros* would co-exist. Over time, this is intended to replace the syntactic macro system. The compiler supports loading plugins from external crates; these can receive the AST once the compiler has generated it. There are APIs available to modify the AST in-place to add new code as required. A detailed discussion of this system is beyond the scope of this book.

Functional features in Rust

Rust has been inspired by functional languages such as Haskell and OCaml. Unsurprisingly, Rust has rich support for functional programming both in the language and in the standard library. In this section, we will look at some of these.

Higher-order functions

We have seen previously how Rust functions define an isolated scope in which all local variables live. Thus, variables outside the scope can never leak into it unless they are explicitly passed as arguments. There can be cases where this is not the desired behavior; closures provide an anonymous function like mechanism, which has access to all the resources defined in the scope in which it is defined. This enables the compiler to enforce the same borrow checking rules while making it easier to reuse code. In Rust terminology, a typical closure borrows all bindings of its surrounding scope. A closure can be forced to own those by marking it with the move keyword. Let’s look at some examples:

```
// chapter2/closure-borrow.rs

fn main() {
    // closure with two parameters
    let add = |a, b| a + b;
    assert_eq!(add(2, 3), 5);

    // common use cases are on iterators
    println!("{:?}", (1..10).filter(|x| x % 2 == 0).collect::<Vec<u32>>());

    // using a variable from enclosing scope
    let times = 2;
    println!("{:?}", (1..10).map(|x| x * times).collect::<Vec<i32>>());
}
```

The first example is a simple closure that adds two numbers given to it. The second example is more involved; it shows a real example of closures for functional programming. We are interested in filtering a list of integers to collect only the even ones out of it. So, we start with a range from 1 to 10, which returns an instance of the built-in type `Range`. Since that type implements the `IntoIterator` trait, that type behaves as an iterator. Thus, we can filter it by passing a closure that returns true only if the input can be divided by two. Finally, we collect the resultant iterator into a vector of `u32` and print it out. The last example is similar in construction. It borrows the variable times from the closure’s enclosing scope and uses it to map to items of the range.

Let’s look at an example of using the `move` keyword in closures:

```
// chapter2/closure-move.rs

fn main() {
    let mut times = 2;
    {
        // This is in a new scope
        let mut borrow = |x| times += x;
        borrow(5);
    }
    assert_eq!(times, 7);

    let mut own = move |x| times += x;
    own(5);
    assert_eq!(times, 7);

}
```

The difference between the first closure (`borrow`) and the ones we have discussed so far is that this mutates the variable it inherits from the enclosing scope. We must declare the variable and the closure as `mut`. We also need to put the closure in a different scope so that the compiler does not complain about double borrowing when we try to assert its value. As asserted, the closure called `borrow` borrows the variables from its parent scope, and that's why its original value changes to `7`. The second closure called `own` is a move closure, thus it gets a copy of the variable `times`. For this to work, the variable has to implement the `Copy` trait so that the compiler can copy it to the closure, which all built-in types do. Since the variable that the closure gets and the original variable are not the same, the compiler does not complain about borrowing it twice. Also, the original value of the variable does not change. These types of closures are immensely important in implementing threads, as we will see in a later section. The standard library also supports accepting and returning closures in user-defined functions or methods using a number of built-in traits, as shown in the following table:

Trait name	**Function**
`std::ops::Fn`	Implemented by closures that do not receive mutable captured variables.
`std::ops::FnMut`	Implemented by closures that need to mutate the captured variables.
`std::ops::FnOnce`	Implemented by all closures. Indicates that the closure can be called exactly once.

Iterators

Another important functional aspect is that of lazy iteration. Given a collection of types, one should be able to loop over those or a subset in any given order. In Rust, a common iterator is a range which has a start and an end. Let's look at how these work:

```
// chapter2/range.rs

#![feature(inclusive_range_syntax)]

fn main() {
    let numbers = 1..5;
    for number in numbers {
        println!("{}", number);
    }
    println!("------------------");
    let inclusive = 1..=5;
    for number in inclusive {
        println!("{}", number);
    }
}
```

The first range is an exclusive range that spans from the first element to the last but one. The second range is an inclusive one which spans till the last element. Note that inclusive range is an experimental feature that might change in the future.

As one would expect, Rust does provide interfaces with which a user-defined type can be iterated on. The type just needs to implement the trait `std::iterator::Iterator`. Let's look at an example. We are interested in generating the Collatz sequence (https://en.wikipedia.org/wiki/Collatz_conjecture), given an integer. This is given by the recurrence relation below, given an integer:

- If it is even, divide it by two
- If it is odd, multiply it by 3 and add one

According to the conjecture, this sequence will always terminate at 1. We will assume that is true and define our code respecting that:

```
// chapter2/collatz.rs

// This struct holds state while iterating
struct Collatz {
    current: u64,
    end: u64,
}

// Iterator implementation
impl Iterator for Collatz {
    type Item = u64;
    fn next(&mut self) -> Option<u64> {
        if self.current % 2 == 0 {
            self.current = self.current / 2;
        } else {
            self.current = 3 * self.current + 1;
        }

        if self.current == self.end {
            None
        } else {
            Some(self.current)
        }
    }
}

// Utility function to start iteration
fn collatz(start: u64) -> Collatz {
    Collatz { current: start, end: 1u64 }
}

fn main() {
    let input = 10;

    // First 2 items
    for n in collatz(input).take(2) {
        println!("{}", n);
    }

    // Dropping first 2 items
    for n in collatz(input).skip(2) {
        println!("{}", n);
    }
}
```

In our code, the state of the current iteration is represented by the struct called `Collatz`. We implement the `Iterator` protocol on it. For that, we need to implement the `next` function, which takes in the current state and produces the next state. When it reaches the end state, it must return a `None` so that the caller knows that the iterator has been exhausted. This is represented by the nullable return value of the function. Given the recurrence, the implementation is straightforward. In our main function, we instantiate the initial state and we can iterate using regular `for` loops. The `Iterator` trait automatically implements a number of useful functions; the `take` function takes the given number of elements from the iterator, while the `skip` function skips the given number of elements. All these are very important for working with iterable collections.

The following is the output of a run of our example:

```
$ rustc collatz.rs && ./collatz
5
16
8
4
2
```

Concurrency primitives

One of the promises of Rust is to enable *fearless concurrency*. Quite naturally, Rust has support for writing concurrent code through a number of mechanisms. In this chapter, we will discuss a few of these. We have seen how the Rust compiler uses borrow checking to ensure correctness of programs at compile time. It turns out that those primitives are also useful in verifying correctness of concurrent code. Now, there are multiple ways of implementing threading in a language. The simplest possible way is to create a new OS thread for each thread created in the platform. This is often called 1:1 threading. On the other hand, a number of application threads can be mapped to one OS thread. This is called N:1 threading. While this approach is resource-light since we end up with fewer actual threads, there is a higher overhead of context switches. A middle ground is called M:N threading, where multiple application threads are mapped to multiple OS level threads. This approach requires the maximum amount of safeguarding and is implemented using a runtime, something that Rust avoids. Thus, Rust uses the 1:1 model. A thread in Rust corresponds to one OS thread in contrast to languages like Go. Let’s start with a look at how Rust enables writing multithreaded applications:

```
// chapter2/threads.rs

use std::thread;

fn main() {
    for i in 1..10 {
        let handle = thread::spawn(move || {
            println!("Hello from thread number {}", i);
        });
        let _ = handle.join();
    }
}
```

We start by importing the threading library. In our main function, we create an empty vector that we will use to store references to the threads we create so that we can wait for them to exit. The threads are actually created using `thread::spawn`, to which we must pass a closure that will be executed in each of those threads. Since we must borrow a variable from the enclosing scope (the loop index `i`) in our closure, the closure itself must be a move closure. Right before exiting the closure, we call join on the current thread handle so that all threads wait for one another. This produces the following output:

```
# rustc threads.rs && ./threads
```

```
Hello from thread number 1
Hello from thread number 2
Hello from thread number 3
Hello from thread number 4
Hello from thread number 5
Hello from thread number 6
Hello from thread number 7
Hello from thread number 8
Hello from thread number 9
```

The real power of multithreaded applications is when threads can cooperate to do meaningful work. For that, two important things are necessary. Threads need to be able to pass data from one another and there should be ways to coordinate how the threads are scheduled so that they don’t step over one another. For the first problem, Rust provides a message, passing mechanisms via channels. Let’s look at the following example:

```
// chapter2/channels.rs

use std::thread;
use std::sync::mpsc;

fn main() {
    let rhs = vec![10, 20, 30, 40, 50, 60, 70];
    let lhs = vec![1, 2, 3, 4, 5, 6, 7];
    let (tx, rx) = mpsc::channel();

    assert_eq!(rhs.len(), lhs.len());
    for i in 1..rhs.len() {
        let rhs = rhs.clone();
        let lhs = lhs.clone();
        let tx = tx.clone();
        let handle = thread::spawn(move || {
            let s = format!("Thread {} added {} and {}, result {}", i,
            rhs[i], lhs[i], rhs[i] + lhs[i]);
            tx.clone().send(s).unwrap();
        });
        let _ = handle.join().unwrap();
    }

    drop(tx);
    for result in rx {
        println!("{}", result);
    }
}
```

This example is much like the previous one. We import the necessary modules to be able to work with channels. We define two vectors, and we will create a thread for each pair of elements in the two vectors so that we can add those and return the result. We create the channel, which returns handles to the sending and the receiving ends. As a safety check, we make sure that the two vectors are indeed of equal length. Then, we move on to creating our threads. Since we would need to access outside variables here, the threads need to take in a move closure like the last example. Further, the compiler will try to use the `Copy` trait to copy those variables to the threads. In this case, that will fail since the vector type does not implement `Copy`. We need to explicitly `clone` the resources so that they do not need to be copied. We run the computation and send the result on the sending end of the pipe. Later, we join all the threads. Before we loop over the receiving end and print the results, we need to explicitly drop the reference to the original handle to the sending end so that all senders are destroyed before we start receiving (the cloned senders will be automatically destroyed when the threads exit). This prints the following, as expected:

```
# rustc channels.rs && ./channels
Thread 1 added 20 and 2, result 22
Thread 2 added 30 and 3, result 33
Thread 3 added 40 and 4, result 44
Thread 4 added 50 and 5, result 55
Thread 5 added 60 and 6, result 66
Thread 6 added 70 and 7, result 77
```

Also note that mpsc stands for multiple producer single consumer.

While working with multiple threads, another common idiom is that of sharing a common state between all of those. That, however, can be a can of worms in a lot of cases. The caller needs to carefully set up exclusion mechanisms so that the state is shared in a race-free manner. Luckily, the borrow checker can help in ensuring this is easier. Rust has a number of smart pointers for dealing with the shared state. The library also provides a generic mutex type that can be used as a lock while working with multiple threads. But perhaps the most important are the `Send` and the `Sync` traits. Any type that implements the `Send` trait can be shared safely between multiple threads. The `Sync` trait indicates that access from multiple threads is safe for the given data. There are a few rules around these traits:

- All built-in types implement both `Send` and `Sync` with the exception of anything `unsafe`, a few smart pointer types like `Rc<T>` and `UnsafeCell<T>`
- A composite type will automatically implement both, as long as it does not have any type that does not implement `Send` and `Sync`

The `std::sync` package has a lot of types and helpers for working with parallel code.

In the previous paragraph, we mentioned unsafe Rust. Let’s take a detour and look at that in a bit more detail. The Rust compiler provides some strong guarantees around safe programming by using a robust type system. However, there can be cases where these become more of an overhead. To handle such cases, the language provides a way to opt out of those guarantees. A block of code marked with the `unsafe` keyword can do everything Rust can do, and the following:

- Dereference raw pointer types (*mut T or `*const T`)
- Call unsafe functions or methods
- Implement a trait marked as `unsafe`
- Mutate a static variable

Let’s look at an example which uses the `unsafe` block of code to dereference a pointer:

```
// chapter2/unsafe.rs

fn main() {
    let num: u32 = 42;
    let p: *const u32 = &num;

    unsafe {
        assert_eq!(*p, num);
    }
}
```

Here, we create a variable and a pointer to it; if we try to dereference the pointer without using the `unsafe` block, the compiler will refuse to compile. Inside the `unsafe` block, we get back the original value on dereferencing. While unsafe code can be dangerous to work with, it’s very useful in lower level programming like Kernel (RedoxOS) and embedded systems.

Testing

Rust treats testing as a first-class construct; all tools in the ecosystem supports testing. The compiler provides a built-in configuration attribute that designates a module for testing. There is also a test attribute that designates functions as tests. When Cargo generates a project from scratch, it sets up this boilerplate. Let's look at an example project; we will call it factorial. It will export a macro that computes the factorial given an integer. Since we have conveniently written such a macro before, we will just re-use that code here. Note that since this crate will be used as a library, this does not have a main function:

```
# cargo new factorial --lib
# cargo test
   Compiling factorial v0.1.0 (file:///Users/Abhishek/Desktop/rust-
book/src/ch2/factorial)
    Finished dev [unoptimized + debuginfo] target(s) in 1.6 secs
     Running target/debug/deps/factorial-364286f171614349

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

   Doc-tests factorial

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Running `cargo test` runs the stub tests that Cargo generates for us. We will copy the code for the factorial macro to the `lib.rs`, which will look like this:

```
// chapter2/factorial/src/lib.rs

#[allow(unused_macros)]
#[macro_export]
macro_rules! factorial {
    ($x:expr) => {
        {
            let mut result = 1;
            for i in 1..($x+1) {
                result = result * i;
            }
            result
        }
    };
}

#[cfg(test)]
mod tests {
    #[test]
    fn test_factorial() {
        assert_eq!(factorial!(5), 120);
    }
}
```

We also added a test to make sure factorial actually works as one would expect. The `#[macro_export]` attribute tells the compiler that this macro is to be used outside the crate. The compiler built-in `assert_eq!` macro checks that the two arguments are indeed equal. We also need to put the `#[allow(unused_macros)]` attribute since, without it, the compiler will complain that the macro is not being used in non-test code. If we add one more test that looks like this:

```
    #[test]
    fn test_factorial_fail() {
        assert_eq!(factorial!(5), 121);
    }
```

This is obviously wrong, and as expected, fails and gives us a descriptive error. The compiler also supports an attribute called `#[should_panic]` that marks tests that should panic. In this case, the tests pass only if there is a panic. Another way of writing tests is in the documentation which is also run on a Cargo invocation.

This is a very important tool in documenting code with working examples, which are guaranteed to work as the codebase evolves. Let’s go ahead and add some doctest for our factorial macro:

````
// chapter2/factorial/src/lib.rs

/// The factorial crate provides a macro to compute factorial of a given
/// integer
/// # Examples
///
/// ```
/// # #[macro_use] extern crate factorial;
/// # fn main() {
/// assert_eq!(factorial!(0), 1);
/// assert_eq!(factorial!(6), 720);
/// # }
/// ```
///

#[macro_export]
macro_rules! factorial {
    ($x:expr) => {
        {
            let mut result = 1;
            for i in 1..($x+1) {
                result = result * i;
            }
            result
        }
    };
}
````

Doctests for macros differ a bit from doctests for everything else in the following way:

- They must use the `#[macro_use]` attribute to mark that the macro is being used here. Note that an external crate that depends on the crate that exports a macro must use that attribute too.

- They must define the main function and include an `extern crate` directive in the doctests. For everything else, the compiler generates the main function as needed. The extra `#` marks hide those from the generated documentation.

In general, the tests module, doctests, and the `#[test]` attributes should be used only for unit tests. Integration tests should be placed in a top-level tests directory.

The Rust team is working on adding support for running benchmarks in the test system. This is only available on nightly for now.

Summary

This chapter was a very short introduction to the Rust language and the ecosystem. Given this background in Rust, let’s look at a frequently asked question: should a company adopt Rust? Like a lot of things in engineering, the correct answer is that it depends on a lot of factors. One of the primary reasons for adopting Rust would be the ability to write robust code with less of a footprint as possible. Thus, Rust is suitable for projects targeting embedded devices. This area has traditionally used assembly, C, and C++. Rust can provide the same performance guarantees while ensuring code correctness. Rust also works well for offloading performance intensive computation from Python or Ruby. The primary pain point with Rust is that the learning curve can be steep. Thus, a team trying to adopt Rust might spend a lot of time fighting with the compiler, trying to run code. This, however, eases out with time. Luckily, the compiler error messages are generally very helpful. In 2017, the Rust team decided to make ergonomics a priority. This push has made onboarding new developers a lot easier. For large Rust projects, compile time can be larger than C, C++, or Go. This can become a problem for some teams. There are a few ways to work around this problem, one of them being incremental compilation. Thus, it is difficult to arrive at a one size fits all solution. Hopefully, this short introduction will help in deciding whether to choose Rust in a new project.

In the next chapter, we will build on what we studied here by looking at how Rust handles TCP and UDP connections between two hosts in a network.

TCP and UDP Using Rust

Being a system programming language, the Rust Standard Library has support for interacting with the network stack. All the networking-related functionality is located in the `std::net` namespace; reading and writing to sockets also uses `Read` and `Write` traits from `std::io`. Some of the most important structures here are `IpAddr`, which represents a generic IP address that can either be v4 or v6, `SocketAddr`, which represents a generic socket address (a combination of an IP and a port on a host), `TcpListener` and `TcpStream` for communicating over TCP, `UdpSocket` for UDP, and more. Currently, the standard library does not provide any APIs to deal with the network stack at a lower level. While this might change in the future, a number of crates fill that gap. The most important of these is `libpnet`, which provides a set of APIs for lower-level networking.

Some other important crates for networking are `net2` and `socket2`. These were meant to be incubators for APIs that might be moved to the standard library. Some of the functionality here is ported to Rust core repo when it is deemed to be useful and stable enough. Unfortunately, this doesn't work out as planned in all cases. On the whole, the community now suggests using the tokio ecosystem of crates for writing high-performance networking applications that do not require fine-grained control of socket semantics. Note that tokio is not in the scope of this chapter, and that we will cover it in a following chapter.

In this chapter, we will cover the following topics:

- What a simple multithreaded TCP client and server looks like in Rust
- Writing a simple multithreaded UDP client and server
- A number of functionalities in `std::net`
- Learning how to use `net2`, `ipnetwork`, and `libpnet`

For the sake of simplicity, all code in this chapter will deal with IPv4 only. Extending the given examples to IPv6 should be trivial.

A Simple TCP server and client

Most networking examples start with an echo server. So, let’s go ahead and write a basic echo server in Rust to see how all the pieces fit together. We will use the threading model from the standard library for handling multiple clients in parallel. The code is as follows:

```
// chapter3/tcp-echo-server.rs

use std::net::{TcpListener, TcpStream};
use std::thread;

use std::io::{Read, Write, Error};

// Handles a single client
fn handle_client(mut stream: TcpStream) -> Result<(), Error> {
    println!("Incoming connection from: {}", stream.peer_addr()?);
    let mut buf = [0; 512];
    loop {
        let bytes_read = stream.read(&mut buf)?;
        if bytes_read == 0 { return Ok(()); }
        stream.write(&buf[..bytes_read])?;
    }
}

fn main() {
    let listener = TcpListener::bind("0.0.0.0:8888")
                               .expect("Could not bind");
    for stream in listener.incoming() {
        match stream {
            Err(e) => { eprintln!("failed: {}", e) }
            Ok(stream) => {
                thread::spawn(move || {
                    handle_client(stream)
                    .unwrap_or_else(|error| eprintln!("{:?}", error));
                });
            }
        }
    }
}
```

In our `main` function, we create a new `TcpListener`, which in Rust, represents a TCP socket that is listening for incoming connections from clients. For our example, we have hardcoded the local address and the port; the local address being set to `0.0.0.0` tells the kernel to bind this socket to all available interfaces on this host. Setting a well-known port here is important since we will need to know that to connect from the client. In a real application, this should be a configurable parameter taken from the CLI or a configuration file. We call `bind` on the local IP and port pair to create a local listening socket. As discussed earlier, our given choice of IP will bind this socket to all interfaces available on the host, on port 8888. As a result, any client that can reach a network connected to this host will be able to talk to this host. As we have seen in the last chapter, the `expect` function returns the listener if there were no errors. If that is not the case, it panics with the given message. Panicking on failing to bind to the port is actually okay here, since if that fails, there is no way the server will continue working. The `incoming` method on `listener` returns an iterator over streams that have connected to the server. We loop over them and check if any of those have encountered an error. In that case, we can print the error and move on to the next connected client. Note that panicking in this case is not appropriate since the server can function fine if some of the clients run into errors for some reason.

Now, we must read data from each of the clients in an infinite loop. But running an infinite loop in the main thread will block it and no other clients will be able to connect. That behavior is definitely not desirable. Thus, we must spawn a worker thread to handle each client connection. The logic of reading from each stream and writing it back is encapsulated in the function called `handle_client`. Each thread receives a closure that calls this function. This closure must be a `move` closure, since this must read a variable (`stream`) from the enclosing scope. In the function, we print the remote endpoint address and port, and then define a buffer to hold data temporarily. We also make sure that the buffer is zeroed out. We then run an infinite loop in which we read all data in the stream. The read method in the stream returns the length of the data it has read. It can return zero in two cases, if it has reached the end of the stream or if the given buffer was zero in length. We know for sure that the second case is not true. Thus, we break out of the loop (and the function) when the read method returns a zero. In that case, we return a `Ok()`. We then write the same data back to the stream using the slice syntax. Note that we have used `eprintln!` to output errors. This macro writes the given string to a standard error, and has been stabilized recently.

One might notice the apparent lack of error handling in reading from and writing to the stream. But that is not actually the case. We have used the `?` operator to handle errors in these invocations. This operator unwraps the result to an `Ok` if everything was fine; otherwise, it does an early return of the error to the calling function. Given this setup, the return type of the function must be either the empty type, to handle success cases, or the `io::Error` type, to handle error cases. Note that it might be a good idea to implement custom errors in such cases and return those instead of built-in errors. Also note that the `?` operator cannot be used in the `main` function currently since the `main` function does not return a `Result`.

Rust recently accepted an RFC which proposes the ability to use the `?` operator in the `main` function.

Interacting with the server from the terminal is easy. When we run the server on a Linux machine and `nc` on another terminal, any text entered to `nc` should be echoed back. Note that if the client and the server are running on the same node, we can use 127.0.0.1 as the server address:

```
$ nc <server ip> 8888
test
test
foobar
foobar
foobarbaz
foobarbaz
^C
```

While using `nc` to interact with the server is fine, it is much more fun to write a client from scratch. In this section, we will see what a simple TCP client might look like. This client will read input from `stdin` as a string and send it over to the server. When it gets back a reply, it will print that in `stdout`. In our example here, the client and the server are running on the same physical host, so we can use 127.0.0.1 as the server address:

```
// chapter3/tcp-client.rs

use std::net::TcpStream;
use std::str;
use std::io::{self, BufRead, BufReader, Write};

fn main() {
    let mut stream = TcpStream::connect("127.0.0.1:8888")
                               .expect("Could not connect to server");
    loop {
        let mut input = String::new();
        let mut buffer: Vec<u8> = Vec::new();
        io::stdin().read_line(&mut input)
                   .expect("Failed to read from stdin");
        stream.write(input.as_bytes())
              .expect("Failed to write to server");

        let mut reader = BufReader::new(&stream);

        reader.read_until(b'\n', &mut buffer)
              .expect("Could not read into buffer");
        print!("{}", str::from_utf8(&buffer)
               .expect("Could not write buffer as string"));
    }
}
```

In this case, we start with importing all required libraries. We then set up a connection to the server using `TcpStrem::connect`, which takes in the remote endpoint address as a string. Like all TCP connections, the client needs to know the remote IP and port to connect. In case setting up the connection fails, we will abort our program with an error message. We then start an infinite loop, in which we will initialize an empty string to read user input locally and a vector of `u8` to read responses from the server. Since a vector in Rust grows as necessary, we will not need to manually chunk the data at each iteration. The `read_line` function reads a line from standard input and stores it in the variable called `input`. Then, it is written to the connection as a stream of bytes. At this point, if everything worked as expected, the server should have sent back a response. We will read that using a `BufReader` that takes care of chunking the data internally. This also makes reading more efficient since there will not be more system calls than necessary. The `read_until` method reads the data in our buffer, which grows as needed. Finally, we can print out the buffer as a string, which has been converted using the `from_utf8` method.

Running the client is easy, and as expected, behaves exactly like `nc`:

```
$ rustc tcp-client.rs && ./tcp-client
test
test
foobar
foobar
foobarbaz
foobarbaz
^C
```

Real-world applications are often more complex than this. A server might need some time to process the input before serving back the response. Let's simulate that by sleeping for a random amount of time in the `handle_client` function; the `main` function will remain exactly the same as the previous example. The first step is to create our project using `cargo`:

```
$ cargo new --bin tcp-echo-random
```

Note that we will need to add the `rand` crate in our `Cargo.toml`, as shown in the following code snippet:

```
[package]
name = "tcp-echo-random"
version = "0.1.0"
authors = ["Foo <foo@bar.com>"]

[dependencies]
rand = "0.3.17"
```

Having set up our dependencies, let's modify the `handle_client` function to sleep for a random delay before sending the response back:

```
// chapter3/tcp-echo-random/src/main.rs

extern crate rand;

use std::net::{TcpListener, TcpStream};
use std::thread;
use rand::{thread_rng, Rng};
use std::time::Duration;
use std::io::{Read, Write, Error};

fn handle_client(mut stream: TcpStream) -> Result<(), Error> {
    let mut buf = [0; 512];
    loop {
        let bytes_read = stream.read(&mut buf)?;
        if bytes_read == 0 { return Ok(()) }
        let sleep = Duration::from_secs(*thread_rng()
                              .choose(&[0, 1, 2, 3, 4, 5])
                              .unwrap());
        println!("Sleeping for {:?} before replying", sleep);
        std::thread::sleep(sleep);
        stream.write(&buf[..bytes_read])?;
    }
}

fn main() {
    let listener = TcpListener::bind("127.0.0.1:8888").expect("Could
    not bind");
    for stream in listener.incoming() {
        match stream {
            Err(e) => eprintln!("failed: {}", e),
            Ok(stream) => {
                thread::spawn(move || {
                    handle_client(stream).unwrap_or_else(|error|
                    eprintln!("{:?}", error));
                });
            }
        }
    }
}
```

In our main file, we must declare a dependency on the `rand` crate and declare it as an `extern crate`. We use the `thread_rng` function to select an integer between zero and five randomly and then sleep for that time duration using `std::thread::sleep`. On the client side, we will set read and connect timeouts, since replies won't be instantaneous from the server:

```
// chapter3/tcp-client-timeout.rs

use std::net::TcpStream;
use std::str;
use std::io::{self, BufRead, BufReader, Write};
use std::time::Duration;
use std::net::SocketAddr;

fn main() {
    let remote: SocketAddr = "127.0.0.1:8888".parse().unwrap();
    let mut stream = TcpStream::connect_timeout(&remote,
    Duration::from_secs(1))
                           .expect("Could not connect to server");
    stream.set_read_timeout(Some(Duration::from_secs(3)))
          .expect("Could not set a read timeout");
    loop {
        let mut input = String::new();
        let mut buffer: Vec<u8> = Vec::new();
        io::stdin().read_line(&mut input).expect("Failed to read from
        stdin");
        stream.write(input.as_bytes()).expect("Failed to write to
        server");

        let mut reader = BufReader::new(&stream);

        reader.read_until(b'\n', &mut buffer)
              .expect("Could not read into buffer");
        print!("{}", str::from_utf8(&buffer)
                    .expect("Could not write buffer as string"));
    }
}
```

Here, we use `set_read_timeout` to set the timeout to three seconds. Thus, if the server sleeps for more than three seconds, the client will abort the connection. This function is curious since it takes in `Option<Duration>` to be able to specify a `Duration` that is `None`. Hence, we will need to wrap our `Duration` in a `Some` before passing to this function. Now, if we open two sessions, running the server using cargo in one and the client in another, here is what we'll see; the server prints how long it slept, for each client it accepts:

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/tcp-echo-random`
Sleeping for Duration { secs: 2, nanos: 0 } before replying
Sleeping for Duration { secs: 1, nanos: 0 } before replying
Sleeping for Duration { secs: 1, nanos: 0 } before replying
Sleeping for Duration { secs: 5, nanos: 0 } before replying
```

On the client side, we have a single file (not a cargo project) that we will build using `rustc` and run the executable directly after compiling:

```
$ rustc tcp-client-timeout.rs && ./tcp-client-timeout
test
test
foo
foo
bar
bar
baz
thread 'main' panicked at 'Could not read into buffer: Error { repr: Os { code: 35,
message: "Resource temporarily unavailable" } }', src/libcore/result.rs:906:4
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

For the first three inputs, the server chose delays which were less than three seconds. The client got a response back within three seconds and did not abort the connection. For the last message, the delay was five seconds, which caused the client to abort reading.

A Simple UDP server and client

There are a few semantic differences between the UDP server and the TCP server we wrote earlier. Unlike TCP, UDP does not have a stream structure. This derives from the semantic differences between the two protocols. Let's take a look at what a UDP server might look like:

```
// chapter3/udp-echo-server.rs

use std::thread;
use std::net::UdpSocket;

fn main() {
    let socket = UdpSocket::bind("0.0.0.0:8888")
                           .expect("Could not bind socket");

    loop {
        let mut buf = [0u8; 1500];
        let sock = socket.try_clone().expect("Failed to clone socket");
        match socket.recv_from(&mut buf) {
            Ok((_, src)) => {
                thread::spawn(move || {
                    println!("Handling connection from {}", src);
                    sock.send_to(&buf, &src)
                        .expect("Failed to send a response");
                });
            },
            Err(e) => {
                eprintln!("couldn't recieve a datagram: {}", e);
            }
        }
    }
}
```

As with TCP, we start with binding to the local address on a given port and we handle the possibility that binding can fail. Since UDP is a connectionless protocol, we will not need to do a sliding window to read all the data. Thus, we can just allocate a static buffer of a given size. It will be a better idea to dynamically detect the MTU of the underlying network card and set the buffer size to be that, since that is the maximum size each UDP packet can have. However, since MTU for a common LAN is around 1,500, we can get away with allocating a buffer of that size here. The `try_clone` method clones the given socket and returns a new one, which is moved into the closure.

We then read from the socket, which returns the length of the data read and the source in the `Ok()` case. We then spawn a new thread, in which we write back the same buffer to the given socket. For anything that can fail, we will need to handle the error like we did for the TCP server.

Interacting with this server is exactly the same as last time—using `nc`. The only difference is that in this case, we will need to pass the `-u` flag to force `nc` to make it use only UDP. Take a look at the following example:

```
$ nc -u 127.0.0.1 8888
test
test
test
test
^C
```

Now, let’s write a simple UDP client to achieve the same results. As we will see, there are some subtle differences between the TCP server and this:

```
// chapter3/udp-client.rs

use std::net::UdpSocket;
use std::{str,io};

fn main() {
    let socket = UdpSocket::bind("127.0.0.1:8000")
                           .expect("Could not bind client socket");
    socket.connect("127.0.0.1:8888")
          .expect("Could not connect to server");
    loop {
        let mut input = String::new();
        let mut buffer = [0u8; 1500];
        io::stdin().read_line(&mut input)
                   .expect("Failed to read from stdin");
        socket.send(input.as_bytes())
              .expect("Failed to write to server");

        socket.recv_from(&mut buffer)
              .expect("Could not read into buffer");
        print!("{}", str::from_utf8(&buffer)
                         .expect("Could not write buffer as string"));
    }
}
```

There is a major difference between this basic client and the TCP client we saw in the last section. In this case, it is absolutely essential to `bind` to a client-side socket first before connecting to the server. Once that is done, the rest of the example is essentially the same. Running it on the client side and the server side produces similar results like the TCP case. Here is a session on the server side:

```
$ rustc udp-echo-server.rs && ./udp-echo-server
Handling connection from 127.0.0.1:8000
Handling connection from 127.0.0.1:8000
Handling connection from 127.0.0.1:8000
^C
```

Here is what we see on the client side:

```
$ rustc udp-client.rs && ./udp-client
test
test
foo
foo
bar
bar
^C
```

UDP multicasting

The `UdpSocket` type has a number of methods that the corresponding TCP types do not. Of these, the most interesting ones are for multicasting and broadcasting. Let's look at how multicasting works with an example server and client. For this example, we will combine the client and the server in one file. In the `main` function, we will check whether a CLI argument has been passed. If there has, we will run the client; otherwise, we will run the server. Note that the value of the argument will not be used; it will be treated as a Boolean:

```
// chapter3/udp-multicast.rs

use std::{env, str};
use std::net::{UdpSocket, Ipv4Addr};

fn main() {
    let mcast_group: Ipv4Addr = "239.0.0.1".parse().unwrap();
    let port: u16 = 6000;
    let any = "0.0.0.0".parse().unwrap();
    let mut buffer = [0u8; 1600];
    if env::args().count() > 1 {
        // client case
        let socket = UdpSocket::bind((any, port))
                               .expect("Could not bind client socket");
        socket.join_multicast_v4(&mcast_group, &any)
              .expect("Could not join multicast group");
        socket.recv_from(&mut buffer)
              .expect("Failed to write to server");
        print!("{}", str::from_utf8(&buffer)
                         .expect("Could not write buffer as string"));
    } else {
        // server case
        let socket = UdpSocket::bind((any, 0))
                               .expect("Could not bind socket");
        socket.send_to("Hello world!".as_bytes(), &(mcast_group, port))
              .expect("Failed to write data");
    }
}
```

Both the client and the server parts here are mostly similar to what we discussed before. One difference is that the `join_multicast_v4` call makes the current socket join a multicast group with the address passed. For both the server and the client, we do not specify a single address while binding. Instead, we use the special address `0.0.0.0` that denotes any available address. This is equivalent to passing `INADDR_ANY` to the underlying `setsockopt` call. In the server case, we send it to the multicast group instead. Running this is a bit more tricky. Since there is no way to set `SO_REUSEADDR` and `SO_REUSEPORT` in the standard library, we will need to run the client on multiple different machines and the server on another. For this to work, all of those need to be in the same network and the address of the multicast group needs to be a valid multicast address (the first four bits should be 1110). The `UdpSocket` type also supports leaving multicast groups, broadcasting, and so on. Note that broadcasting does not make sense for TCP since it is a connection between two hosts by definition.

Running the previous example is simple; on one host, we will run the server, and on the other, the client. Given this setup, the output should look like this on the server side:

```
$ rustc udp-multicast.rs && ./udp-multicast server
Hello world!
```

Miscellaneous utilities in std::net

Another important type in the standard library is `IpAddr`, which represents an IP address. Not surprisingly, it is an enum with two variants, one for v4 addresses and the other for v6 addresses. All of these types have methods to classify addresses according to their types (global, loopback, multicast, and so on). Note that a number of these methods are not stabilized yet and hence are only available in the nightly compiler. They are behind a feature flag named `ip` which must be included in the crate root so that you can use those methods. A closely related type is `SocketAddr`, which is a combination of an IP address and a port number. Thus, this also has two variants, one for v4 and one for v6. Let's look at some examples:

```
// chapter3/ip-socket-addr.rs

#![feature(ip)]

use std::net::{IpAddr, SocketAddr};

fn main() {
    // construct an IpAddr from a string and check it
    // represents the loopback address
    let local: IpAddr = "127.0.0.1".parse().unwrap();
    assert!(local.is_loopback());

    // construct a globally routable IPv6 address from individual
    octets
    // and assert it is classified correctly
    let global: IpAddr = IpAddr::from([0, 0, 0x1c9, 0, 0, 0xafc8, 0,
    0x1]);
    assert!(global.is_global());

    // construct a SocketAddr from a string an assert that the
    underlying
    // IP is a IPv4 address
    let local_sa: SocketAddr = "127.0.0.1:80".parse().unwrap();
    assert!(local_sa.is_ipv4());

    // construct a SocketAddr from a IPv6 address and a port, assert
    that
    // the underlying address is indeed IPv6
    let global_sa = SocketAddr::new(global, 80u16);
    assert!(global_sa.is_ipv6());
}
```

The `feature(ip)` declaration is necessary since the `is_global` function is not stabilized yet. This example does not produce any output since all asserts should evaluate to true.

A common functionality is that of a DNS lookup, given a hostname. Rust does this using the `lookup_host` function that returns the `LookupHost` type, which is actually an iterator over DNS responses. Let's look at how this can be used.

This function is gated by the `looup_host` flag and must be included to use this function with the nightly compiler:

```
// chapter3/lookup-host.rs

#![feature(lookup_host)]

use std::env;
use std::net::lookup_host;

fn main() {
    let args: Vec<_> = env::args().collect();
    if args.len() != 2 {
        eprintln!("Please provide only one host name");
        std::process::exit(1);
    } else {
        let addresses = lookup_host(&args[1]).unwrap();
        for address in addresses {
            println!("{}", address.ip());
        }
    }
}
```

Here, we read a CLI argument and exit if we were not given exactly one name to resolve. Otherwise, we call `lookup_host` with the given hostname. We iterate over the returned results and print the IP of each. Note that each of the returned results is of type `SocketAddr`; since we are only interested in the IP, we extract that using the `ip()` method. This function corresponds to the `getaddrinfo` call in libc, and thus it returns only `A` and `AAAA` record types. Running this is as expected:

```
$ rustc lookup-host.rs && ./lookup-host google.com
2a00:1450:4009:810::200e
216.58.206.110
```

Currently, reverse DNS lookup is not possible in the standard library. In the next section, we will discuss some crates in the ecosystem that can be used for the advanced networking functionality. For instance, the `trust-dns` crate supports interacting with DNS servers in more detail, and it also supports querying all record types and also reverse DNS.

Some related crates

A careful reader might have noticed that a lot of common networking-related functionalities are missing from the standard library. For instance, there is no way to deal with IP networks (CIDRs). Let’s look at how the `ipnetwork` crate helps with that. Since we are going to use an external crate, the example has to be in a cargo project. We will need to add it as a dependency to `Cargo.toml`. Let’s start by setting up a project:

```
$ cargo new --bin ipnetwork-example
```

This generates a `Cargo.toml` file that we need to modify to declare our dependency. Once we do that, it should look like this:

```
[package]
name = "ipnetwork-example"
version = "0.1.0"
authors = ["Foo <foo@bar.com>"]

[dependencies]
ipnetwork = "0.12.7"
```

Having set up the project, let’s look at our `main` function:

```
// chapter3/ipnetwork-example/src/main.rs

extern crate ipnetwork;

use std::net::Ipv4Addr;
use ipnetwork::{IpNetwork, Ipv4Network, Ipv6Network};

fn main() {
    let net = IpNetwork::new("192.168.122.0".parse().unwrap(), 22)
                        .expect("Could not construct a network");
    let str_net: IpNetwork = "192.168.122.0/22".parse().unwrap();

    assert!(net == str_net);
    assert!(net.is_ipv4());

    let net4: Ipv4Network = "192.168.121.0/22".parse().unwrap();
    assert!(net4.size() == 2u64.pow(32 - 22));
    assert!(net4.contains(Ipv4Addr::new(192, 168, 121, 3)));

    let _net6: Ipv6Network = "2001:db8::0/96".parse().unwrap();
    for addr in net4.iter().take(10) {
        println!("{}", addr);
    }
}
```

The first two lines show two different ways of constructing `IpNetwork` instances, either using the constructor or by parsing a string. The next `assert` makes sure they are indeed identical. The `assert` after that ensures that the network we created is a v4 network. Next, we specifically create `Ipv4Network` objects and as expected, the size of the network matches *2^(32 - prefix)*. The next `assert` makes sure the `contains` method works correctly for an IP in that network. We then create a `Ipv6Network`, and since all of these types implement the iterator protocol, we can iterate over the network and print individual addresses in a `for` loop. Here is the output that we should see by running the last example:

```
$ cargo run
   Compiling ipnetwork v0.12.7
   Compiling ipnetwork-example v0.1.0 (file:///Users/Abhishek/Desktop/rust-
book/src/chapter3/ipnetwork-example)
    Finished dev [unoptimized + debuginfo] target(s) in 1.18 secs
     Running `target/debug/ipnetwork-example`
192.168.120.0
192.168.120.1
192.168.120.2
192.168.120.3
192.168.120.4
192.168.120.5
192.168.120.6
192.168.120.7
192.168.120.8
192.168.120.9
```

The standard library also lacks fine-grained control over sockets and connections, one example being the ability to set `SO_REUSEADDR`, as we described before. The primary reason for this is that the community has not been able to reach a strong consensus on how to best expose these features while maintaining a clean API. A useful library in this context is `mio`, which provides an alternative to thread-based concurrency. `mio` essentially runs an event loop where all parties register. When there is an event, every listener is alerted and they have the option to handle that event. Let's look at the following example. Like last time, we will need to set up the project using cargo:

```
$ cargo new --bin mio-example
```

The next step is to add `mio` as a dependency; the `Cargo.toml` file should look like this:

```
[package]
name = "mio-example"
version = "0.1.0"
authors = ["Foo <foo@bar.com>"]

[dependencies]
mio = "0.6.11"
```

Like all other cargo projects, we will need to declare `mio` as a dependency in `Cargo.toml` and pin it to a specific version so that cargo can download it and link it against our app:

```
// chapter3/mio-example/src/main.rs

extern crate mio;

use mio::*;
use mio::tcp::TcpListener;

use std::net::SocketAddr;
use std::env;

// This will be later used to identify the server on the event loop
const SERVER: Token = Token(0);

// Represents a simple TCP server using mio
struct TCPServer {
    address: SocketAddr,
}

// Implementation for the TCP server
impl TCPServer {
    fn new(port: u32) -> Self {
        let address = format!("0.0.0.0:{}", port)
            .parse::<SocketAddr>().unwrap();

        TCPServer {
            address,
        }
    }

    // Actually binds the server to a given address and runs it
    // This function also sets up the event loop that dispatches
    // events. Later, we use a match on the token on the event
    // to determine if the event is for the server.
    fn run(&mut self) {
        let server = TcpListener::bind(&self.address)
            .expect("Could not bind to port");
        let poll = Poll::new().unwrap();
        poll.register(&server,
                       SERVER,
                       Ready::readable(),
                       PollOpt::edge()).unwrap();

        let mut events = Events::with_capacity(1024);
        loop {
            poll.poll(&mut events, None).unwrap();

            for event in events.iter() {
                match event.token() {
                    SERVER => {
                        let (_stream, remote) =
                        server.accept().unwrap();
                        println!("Connection from {}", remote);
                    }
                    _ => {
                        unreachable!();
                    }
                }
            }
        }
    }
}

fn main() {
    let args: Vec<String> = env::args().collect();
    if args.len() != 2 {
        eprintln!("Please provide only one port number as argument");
        std::process::exit(1);
    }
    let mut server = TCPServer::new(args[1].parse::<u32>()
                               .expect("Could not parse as u32"));
    server.run();
}
```

Unlike our previous examples, this is a TCP server that just prints the client's source IP and port. In `mio`, every listener on the event loop is assigned a token, which can then be used to differentiate between the listeners when an event is delivered. We define a struct for our server (`TCPServer`) in its constructor, and we `bind` to all local addresses and return an instance of the struct. The `run` method of that struct binds the socket to the given socket address; it then uses the `Poll` struct to instantiate the event loop.

It then registers the server socket, with a token on the instance. We also indicate that we should be alerted when the event is ready for reading or writing. Lastly, we indicate that we want edge-triggered events only, which means that the event should be wholly consumed when it is received, otherwise subsequent calls on the same token might block it. We then set up an empty container for our events. Having done all the boilerplate, we enter an infinite loop and start polling with the events container we just created. We loop over the list of events, and if any of the event's tokens match the server's token, we know it is meant for the server. We can then accept the connection and print the remote end's information. We then go back to the next event, and so on. In our `main` function, we first deal with CLI arguments, making sure that we passed a port number as an integer. Then, we instantiate the server and call the run method on it.

Here is a sample session on running the server when two clients are connected to it. Note that either `nc` or the TCP clients from earlier can be used to connect to this server:

```
$ cargo run 4321
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/mio-example 4321`
Connection from 127.0.0.1:60955
Connection from 127.0.0.1:60956
^C
```

Some other major things that are missing from the standard library and the crates discussed here is the ability to work with the physical network device, a nicer API to craft and parse packets, and so on. One crate that helps in dealing with lower level network-related things in `libpnet`. Let's write a small packet dumper using it:

```
$ cat Cargo.toml
[package]
name = "pnet-example"
version = "0.1.0"
authors = ["Foo Bar <foo@bar.com>"]

[dependencies]
pnet = "0.20.0"
```

We initialize our Cargo project like this:

```
$ cargo new --bin pnet-example
```

We then add `pnet` as a dependency, pinning it to a specific version (the latest one currently available). We can then move on to our source, which should look like this:

```
// chapter3/pnet-example/src/main.rs

extern crate pnet;

use pnet::datalink::{self, NetworkInterface};
use pnet::datalink::Channel::Ethernet;
use pnet::packet::ethernet::{EtherTypes, EthernetPacket};
use pnet::packet::ipv4::Ipv4Packet;
use pnet::packet::tcp::TcpPacket;
use pnet::packet::ip::IpNextHeaderProtocols;
use pnet::packet::Packet;

use std::env;

// Handles a single ethernet packet
fn handle_packet(ethernet: &EthernetPacket) {
    match ethernet.get_ethertype() {
        EtherTypes::Ipv4 => {
            let header = Ipv4Packet::new(ethernet.payload());
            if let Some(header) = header {
                match header.get_next_level_protocol() {
                    IpNextHeaderProtocols::Tcp => {
                        let tcp = TcpPacket::new(header.payload());
                        if let Some(tcp) = tcp {
                            println!(
                                "Got a TCP packet {}:{} to {}:{}",
                                header.get_source(),
                                tcp.get_source(),
                                header.get_destination(),
                                tcp.get_destination()
                            );
                        }
                    }
                    _ => println!("Ignoring non TCP packet"),
                }
            }
        }
        _ => println!("Ignoring non IPv4 packet"),
    }
}

fn main() {
    let interface_name = env::args().nth(1).unwrap();

    // Get all interfaces
    let interfaces = datalink::interfaces();
    // Filter the list to find the given interface name
    let interface = interfaces
        .into_iter()
        .filter(|iface: &NetworkInterface| iface.name == interface_name)
        .next()
        .expect("Error getting interface");

    let (_tx, mut rx) = match datalink::channel(&interface, Default::default()) {
        Ok(Ethernet(tx, rx)) => (tx, rx),
        Ok(_) => panic!("Unhandled channel type"),
        Err(e) => {
            panic!(
                "An error occurred when creating the datalink channel:
                {}",e
            )
        }
    };
    // Loop over packets arriving on the given interface
    loop {
        match rx.next() {
            Ok(packet) => {
                let packet = EthernetPacket::new(packet).unwrap();
                handle_packet(&packet);
            }
            Err(e) => {
            panic!("An error occurred while reading: {}", e);
            }
        }
    }
}
```

Like always, we start with declaring `pnet` as an external crate. We then import a bunch of things that we will use later. We take in the name of the interface we should sniff as a CLI argument. The `datalink::interfaces()` gets us a list of all available interfaces in the current host, and we filter that list by the name of the interface we were given. In case we do not find a match, we throw an error and exit. The `datalink::channel()` call gives us a channel to send and receive packets. In this case, we do not care about the sending end since we are just interested in sniffing packets. We match on the returned channel type to make sure we work with Ethernet only. The receiving end of the channel, `rx`, gives us an iterator that yields packets on each `next()` call.

The packets are then passed to the `handle_packet` function, which extracts relevant information and prints those. For this toy example, we will only deal with IPv4-based TCP packets. A real network will obviously get IPv6 and ICMP packets with UDP and TCP. All those combinations will be ignored here.

In the `handle_packet` function, we match on the ethertype of the packet to make sure we only process IPv4 packets. Since the whole payload of the Ethernet packet is the IP packet (refer to Chapter 1, *Introduction to Client/Server Networking*), we construct an IP packet from the payload.
The `get_next_level_protocol()` call returns the transport protocol, and if that matches TCP, we construct a TCP packet from the payload of the preceding layer. At this point, we can print the source and destination port from the TCP packet. The source and destination IP will be in the enclosing IP packet. An example run is shown in the following code block. We need to pass the name of the interface to listen on to our program as command line arguments. Here is how we can get the interface name in Linux:

```
$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
```

```
       valid_lft forever preferred_lft forever
2: enp1s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
group default qlen 1000
    link/ether f4:4d:30:ac:88:ee brd ff:ff:ff:ff:ff:ff
    inet 192.168.5.15/22 brd 192.168.7.255 scope global enp1s0
       valid_lft forever preferred_lft forever
    inet6 fe80::58c6:9ccc:e78c:caa6/64 scope link
       valid_lft forever preferred_lft forever
```

For this example, we will ignore the loopback interface `lo` since it does not receive a lot of traffic, and use the other interface `enp1s0`. We will also run this example with root privileges (using sudo) since it need to access the network device directly.

The first step is to build the project using cargo and the run the executable. Note that the exact output of this example might be a bit different, depending on what packets arrive:

```
$ cargo build
$ sudo ./target/debug/pnet-example enp1s0
Got a TCP packet 192.168.0.2:53041 to 104.82.249.116:443
Got a TCP packet 104.82.249.116:443 to 192.168.0.2:53041
Got a TCP packet 192.168.0.2:53064 to 17.142.169.200:443
Got a TCP packet 192.168.0.2:53064 to 17.142.169.200:443
Got a TCP packet 17.142.169.200:443 to 192.168.0.2:53064
Got a TCP packet 17.142.169.200:443 to 192.168.0.2:53064
Got a TCP packet 192.168.0.2:53064 to 17.142.169.200:443
Got a TCP packet 192.168.0.2:52086 to 52.174.153.60:443
Ignoring non IPv4 packet
Got a TCP packet 52.174.153.60:443 to 192.168.0.2:52086
Got a TCP packet 192.168.0.2:52086 to 52.174.153.60:443
Ignoring non IPv4 packet
Ignoring non IPv4 packet
Ignoring non IPv4 packet
Ignoring non IPv4 packet
```

In the previous section, we saw how the DNS-related functionality in the standard library is rather limited. One crate that is widely popular for DNS-related things in `trust-dns`. Let's look at an example of using this for querying a given name. Let's start with the empty project:

```
$ cargo new --bin trust-dns-example
```

We will then add the versions of required crates in `Cargo.toml` first:

```
[package]
name = "trust-dns-example"
version = "0.1.0"
authors = ["Foo <foo@bar.com>"]

[dependencies]
trust-dns-resolver = "0.6.0"
trust-dns = "0.12.0"
```

Our app depends on `trust-dns` for DNS-related things. As usual, we will add it to our `Cargo.toml` before using it in our app:

```
// chapter3/trust-dns-example/src/main.rs

extern crate trust_dns_resolver;
extern crate trust_dns;

use std::env;

use trust_dns_resolver::Resolver;
use trust_dns_resolver::config::*;

use trust_dns::rr::record_type::RecordType;

fn main() {
    let args: Vec<String> = env::args().collect();
    if args.len() != 2 {
        eprintln!("Please provide a name to query");
        std::process::exit(1);
    }
    let resolver = Resolver::new(ResolverConfig::default(),
                                 ResolverOpts::default()).unwrap();

    // Add a dot to the given name
    let query = format!("{}.", args[1]);

    // Run the DNS query
    let response = resolver.lookup_ip(query.as_str());
    println!("Using the synchronous resolver");
    for ans in response.iter() {
        println!("{:?}", ans);
    }

    println!("Using the system resolver");
    let system_resolver = Resolver::from_system_conf().unwrap();
    let system_response = system_resolver.lookup_ip(query.as_str());
    for ans in system_response.iter() {
        println!("{:?}", ans);
    }

    let ns = resolver.lookup(query.as_str(), RecordType::NS);
    println!("NS records using the synchronous resolver");
    for ans in ns.iter() {
        println!("{:?}", ans);
    }
}
```

We set up all required imports and `extern` crate declarations. Here, we expect to get the name to resolve as a CLI argument, and if everything goes well, it should be in `args[1]`. This crate supports two types of synchronous DNS resolver. `Resolver::new` creates a synchronous resolver, and with default options, it will use Google's public DNS as upstream servers. The `Resolver::from_system_conf` creates a synchronous resolver with configurations from the system's `resolv.conf`. Thus, this second option is only available on Unix systems. Before we pass the query to the `resolver`, we format it to FQDN by appending a `.` to the name, using the `format!` macro. We pass the query using the `lookup_ip` function, which then returns an iterator over the answers of the DNS question. Once we get that, we can iterate over it and print out each answer. As the name suggests, the `lookup_ip` function only looks up `A` and `AAAA` records. There is a more general `lookup` function that can take in a record type to query. In the last step, we want to get all `NS` records for the given name. Once we get back an answer, we loop over it and print the results.

`trust-dns` also supports a tokio-based asynchronous DNS resolver.

An example session will look like this:

```
$ cargo run google.com
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/trust-dns-example google.com`
Using the synchronous resolver
LookupIp(Lookup { rdatas: [AAAA(2a00:1450:4009:811::200e), A(216.58.206.142)] })
Using the system resolver
LookupIp(Lookup { rdatas: [A(216.58.206.110), AAAA(2a00:1450:4009:810::200e)] })
NS records using the synchronous resolver
Lookup { rdatas: [NS(Name { is_fqdn: true, labels: ["ns3", "google", "com"] }),
NS(Name { is_fqdn: true, labels: ["ns1", "google", "com"] }), NS(Name { is_fqdn:
true, labels: ["ns4", "google", "com"] }), NS(Name { is_fqdn: true, labels: ["ns2",
"google", "com"] })] }
```

In this example, all prints are using the debug representation of the structures. A real application will want to format these as required.

Summary

This chapter was a short introduction to the basic networking functionality in Rust. We started with given functionality in `std::net`, and we wrote a few TCP and UDP servers using those. We then looked at some other utilities in the same namespace. At the end, we went over examples of a number of crates which are aimed at extending the standard library's functionality around networking. Bear in mind that it is always possible to just use the libc crate to write networking code, which is based on POSIX-compatible networking code with access to fine-grained control over sockets and network devices. The problem with this approach is that the code might be unsafe, breaking Rust's guarantee of safety. Another crate called nix aims to provide libc's functionality native Rust so that it preserves all the memory and type safety guarantees that the compiler provides: this might be a useful alternative for someone who needs very fine control over networking.

In the next chapter, we will look at how to handle data once we receive it in a server or a client using a number of serialization/de-serialization methods in the Rust ecosystem.

Data Serialization, Deserialization, and Parsing

In the previous chapter, we covered writing simple socket servers in Rust. Transport protocols such as TCP and UDP only provide mechanisms to transport messages, so it is up to a higher-level protocol to actually craft and send those messages. Also, TCP and UDP protocols always deal with bytes; we saw this when we called `as_bytes` on our strings before sending those out on the socket. This process of converting a piece of data into a format that can be stored or transmitted (a stream of bytes in the case of networking) is called serialization. The reverse process is deserialization, which turns a raw data format into a data structure. Any networking software must deal with serializing and deserializing data that has been received, or is about to be sent out. This simple conversion is not always possible for more complex types such as user-defined types, or even simple collection types. The Rust ecosystem has special crates that can handle these in a wide range of cases.

In this chapter, we will cover the following topics:

- Serialization and deserialization using Serde. We will start with basic usage and then move on to writing custom serializers using Serde.
- Parsing textual data using nom.
- The last topic will be on parsing binary data, a very frequently used technique in networking.

Serialization and deserialization using Serde

Serde is the de-facto standard way of serializing and deserializing data in Rust. Serde supports a number of data structures that it can serialize out of the box to a number of given data formats (including JSON, and TOML, CSV). The easiest way to understand Serde is to think of it as an invertible function that transforms a given data structure into a stream of bytes. Other than standard data types, Serde also provides a few macros that can be implemented on user defined data types, making them (de)serializable.

In Chapter 2, *Introduction to Rust and its Ecosystem*, we discussed how procedural macros can be used to implement custom derives for given data types. Serde uses that mechanism to provide two custom derives, named `Serialize` and `Deserialize`, that can be implemented for user-defined data types that are composed of data types that Serde supports. Let us look at a small example of how this works. We start with creating the empty project using Cargo:

```
$ cargo new --bin serde-basic
```

Here is what the Cargo manifest should look like:

```
[package]
name = "serde-basic"
version = "0.1.0"
authors = ["Foo<foo@bar.com>"]

[dependencies]
serde = "1.0"
serde_derive = "1.0"
serde_json = "1.0"
serde_yaml = "0.7.1"
```

The `serde` crate is the core of the Serde ecosystem. The `serde_derive` crate provides necessary instrumentation that uses procedural macros for deriving `Serialize` and `Deserialize`. The next two crates provide Serde-specific functionality to and from JSON and YAML, respectively:

```
// chapter4/serde-basic/src/main.rs

#[macro_use]
extern crate serde_derive;

extern crate serde;
extern crate serde_json;
extern crate serde_yaml;

// We will serialize and deserialize instances of
// this struct
#[derive(Serialize, Deserialize, Debug)]
struct ServerConfig {
    workers: u64,
    ignore: bool,
    auth_server: Option<String>
}

fn main() {
    let config = ServerConfig {
                workers: 100,
                ignore: false,
                auth_server: Some("auth.server.io".to_string())
            };
    {
        println!("To and from YAML");
        let serialized = serde_yaml::to_string(&config).unwrap();
        println!("{}", serialized);
        let deserialized: ServerConfig =
        serde_yaml::from_str(&serialized).unwrap();
        println!("{:?}", deserialized);
    }
    println!("\n\n");
    {
        println!("To and from JSON");
        let serialized = serde_json::to_string(&config).unwrap();
        println!("{}", serialized);
        let deserialized: ServerConfig =
        serde_json::from_str(&serialized).unwrap();
        println!("{:?}", deserialized);
    }
}
```

Since the `serde_derive` crate exports macros, we will need to mark it with a `macro_use` declaration; we then declare all our dependencies as `extern` crates. Having set this up, we can define our custom data type. In this case, we are interested in a config for a server that has a bunch of parameters of different types. The `auth_server` parameter is optional and that is why it is wrapped in an `Option`. Our struct derives the two traits from Serde, and also the compiler-provided `Debug` trait that we will use later to display after deserialization. In our main function, we instantiate our class and call `serde_yaml::to_string` on it to serialize it to a string; the reverse of this is `serde_yaml::from_str`.

A sample run should look like this:

```
$ cargo run
   Compiling serde-basic v0.1.0 (file:///Users/Abhishek/Desktop/rust-
book/src/chapter4/serde-basic)
    Finished dev [unoptimized + debuginfo] target(s) in 1.88 secs
     Running `target/debug/serde-basic`
To and from YAML
---
workers: 100
ignore: false
auth_server: auth.server.io
ServerConfig { workers: 100, ignore: false, auth_server: Some("auth.server.io") }



To and from JSON
{"workers":100,"ignore":false,"auth_server":"auth.server.io"}
ServerConfig { workers: 100, ignore: false, auth_server: Some("auth.server.io") }
```

Let us move on to a more advanced example of using Serde over a network. In this example, we will set up a TCP server and a client. This part will be exactly the same as how we did it in the last chapter. But this time, our TCP server will function as a calculator that takes in a point in a 3D space with three components along the three axes, and returns its distance from the origin in the same reference frame. Let us set up our Cargo project like this:

```
$ cargo new --bin serde-server
```

The manifest should look like this:

```
$ cat Cargo.toml
[package]
name = "serde-server"
version = "0.1.0"
authors = ["Foo <foo@bar.com>"]

[dependencies]
serde = "1.0"
serde_derive = "1.0"
serde_json = "1.0"
```

With this, we can then move on to defining our code. In this example, the server and the client will be in the same binary. The application will take in a flag that dictates whether it should run as the server or the client. As we did in the last chapter, in the server case, we will bind to all local interfaces on a known port and listen for incoming connections. The client case will connect to the server on that known port and wait for user input on the console. The client expects input as a comma-separated list of three integers, one for each axis. On getting the input, the client constructs a struct of a given definition, serializes it using Serde, and sends the stream of bytes to the server. The server deserializes the stream into a struct of the same type. It then computes the distance and sends back the result, which the client then displays. The code is as follows:

```
// chapter4/serde-server/src/main.rs

#[macro_use]
extern crate serde_derive;

extern crate serde;
extern crate serde_json;

use std::net::{TcpListener, TcpStream};
use std::io::{stdin, BufRead, BufReader, Error, Write};
use std::{env, str, thread};

#[derive(Serialize, Deserialize, Debug)]
struct Point3D {
    x: u32,
    y: u32,
    z: u32,
}

// Like previous examples of vanilla TCP servers, this function handles
// a single client.
```

```
fn handle_client(stream: TcpStream) -> Result<(), Error> {
    println!("Incoming connection from: {}", stream.peer_addr()?);
    let mut data = Vec::new();
    let mut stream = BufReader::new(stream);

    loop {
        data.clear();

        let bytes_read = stream.read_until(b'\n', &mut data)?;
        if bytes_read == 0 {
            return Ok(());
        }
        let input: Point3D = serde_json::from_slice(&data)?;
        let value = input.x.pow(2) + input.y.pow(2) + input.z.pow(2);

        write!(stream.get_mut(), "{}", f64::from(value).sqrt())?;
        write!(stream.get_mut(), "{}", "\n")?;
    }
}

fn main() {
    let args: Vec<_> = env::args().collect();
    if args.len() != 2 {
        eprintln!("Please provide --client or --server as argument");
        std::process::exit(1);
    }
    // The server case
    if args[1] == "--server" {
        let listener = TcpListener::bind("0.0.0.0:8888").expect("Could
        not bind");
        for stream in listener.incoming() {
            match stream {
                Err(e) => eprintln!("failed: {}", e),
                Ok(stream) => {
                    thread::spawn(move || {
                        handle_client(stream).unwrap_or_else(|error|
                        eprintln!("{:?}", error));
                    });
                }
            }
        }
    }
    // Client case begins here
    else if args[1] == "--client" {
        let mut stream = TcpStream::connect("127.0.0.1:8888").expect("Could not
connect to server");
        println!("Please provide a 3D point as three comma separated
        integers");
        loop {
            let mut input = String::new();
            let mut buffer: Vec<u8> = Vec::new();
            stdin()
                .read_line(&mut input)
                .expect("Failed to read from stdin");
            let parts: Vec<&str> = input
                                   .trim_matches('\n')
                                   .split(',')
                                   .collect();
            let point = Point3D {
                x: parts[0].parse().unwrap(),
                y: parts[1].parse().unwrap(),
                z: parts[2].parse().unwrap(),
            };
            stream
                .write_all(serde_json::to_string(&point).unwrap().as_bytes())
                .expect("Failed to write to server");
            stream.write_all(b"\n").expect("Failed to write to
            server");
            let mut reader = BufReader::new(&stream);
            reader
                .read_until(b'\n', &mut buffer)
                .expect("Could not read into buffer");
            let input = str::from_utf8(&buffer).expect("Could not write
            buffer as string");
            if input == "" {
                eprintln!("Empty response from server");
            }
            print!("Response from server {}", input);
        }
    }
}
```

We start with setting up Serde as we did in the last example. We then define our 3D point as a struct of three elements. In our main function, we handle CLI arguments and branch out to the client or the server, depending on what was passed. In both cases, we signal the end of transmission by sending a newline character. The client reads a line from `stdin`, cleans it, and creates an instance of the struct in a loop. In both cases, we wrap our streams in a `BufReader` for easier handling. We run our code using Cargo. An example session on the server is as follows:

```
server$ cargo run -- --server
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/serde-server --server`
Incoming connection from: 127.0.0.1:49630
```

And, on the client side, we see the following interaction with the server. As expected, the client reads input, serializes that, and sends it to the server. It then waits for a response and, when it gets one, prints the result to standard output:

```
client$ cargo run -- --client
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/serde-server --client`
Please provide a 3D point as three comma separated integers
1,2,3
Response from server 3.7416573867739413
3,4,5
Response from server 7.0710678118654755
4,5,6
Response from server 8.774964387392123
```

Custom serialization and deserialization

As we saw before, Serde provides built-in serialization and deserialization for all primitive data types, and a number of complex data types, via macros. In some cases, however, Serde might fail to auto-implement. This might happen for more complex data types. In those cases, you will need to implement these manually. These cases demonstrate advanced usage of Serde, which also allows renaming fields in the output. For everyday usage, using these advanced feature is almost never necessary. These might be more common for networking, to handle a new protocol, and more.

Let’s say we have a struct of three fields. We will just assume that Serde fails to implement `Serialize` and `Deserialize` on this, and so we will need to implement those manually. We initialize our project using Cargo:

```
$ cargo new --bin serde-custom
```

We then declare our dependencies; the resulting Cargo config file should look like this:

```
[package]
name = "serde-custom"
version = "0.1.0"
authors = ["Foo <foo@bar.com>"]

[dependencies]
serde = "1.0"
serde_derive = "1.0"
serde_json = "1.0"
serde_test = "1.0"
```

Our struct looks like this:

```
// chapter4/serde-custom/src/main.rs

// We will implement custom serialization and deserialization
// for this struct
#[derive(Debug, PartialEq)]
struct KubeConfig {
    port: u8,
    healthz_port: u8,
    max_pods: u8,
}
```

We need to derive `Debug` and `PartialEq` for Serde to use internally. In the real world, it might be necessary to manually implement those as well. Now, we will need to implement the `Serialize` trait for kubeconfig. This trait looks like this:

```
pub trait Serialize {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
        where S: Serializer;
}
```

The basic workflow for serializing our struct will simply be to serialize the struct name, then each of the elements, and then signal the end of serialization, in that order. Serde has built-in methods to serialize that can work with all basic types, therefore an implementation does not need to worry about handling built-in types. Let’s look at how we can serialize our struct:

```
// chapter4/serde-custom/src/main.rs

// Implementing Serialize for our custom struct defines
// how instances of that struct should be serialized.
// In essence, serialization of an object is equal to
// sum of the serializations of it's components
impl Serialize for KubeConfig {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
        where S: Serializer
    {
        let mut state = serializer.serialize_struct("KubeConfig", 3)?;
        state.serialize_field("port", &self.port)?;
        state.serialize_field("healthz_port", &self.healthz_port)?;
        state.serialize_field("max_pods", &self.max_pods)?;
        state.end()
    }
}
```

Serialization of a struct will always begin with a call of `serialize_struct` with the struct name and number of fields as parameter (there are similarly named methods for other types). We then serialize each field in the order they appear while passing a key name that will be used in the resultant json. Once done, we call the special `end` method as a signal.

Implementing deserialization is a bit more involved, with a bit of boilerplate code. The related trait looks like this:

```
pub trait Deserialize<'de>: Sized {
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
        where D: Deserializer<'de>;
}
```

Implementing this for a type requires implementing the visitor pattern. Serde defines a special `Visitor` trait, as shown in the following sample. Note that this has `visit_*` methods for all built-in types, those are not shown here. Also, in the following sample, we use the symbol `...` to indicate that there are more methods here that are not important for our discussion.

```
pub trait Visitor<'de>: Sized {
    type Value;
    fn expecting(&self, formatter: &mut Formatter) -> Result;
    fn visit_bool<E>(self, v: bool) -> Result<Self::Value,
     E>
    where
        E: Error,
    { }
    ...
}
```

An implementation of this trait is used internally by the deserializer to construct the resultant type. In our case, it will look like this:

```
// chapter4/serde-custom/src/main.rs

// Implementing Deserialize for our struct defines how
// an instance of the struct should be created from an
// input stream of bytes
impl<'de> Deserialize<'de> for KubeConfig {
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
        where D: Deserializer<'de>
    {
        enum Field { Port, HealthzPort, MaxPods };

        impl<'de> Deserialize<'de> for Field {
            fn deserialize<D>(deserializer: D) ->
            Result<Field,
            D::Error>
                where D: Deserializer<'de>
            {
                struct FieldVisitor;

                impl<'de> Visitor<'de> for FieldVisitor {
                    type Value = Field;

                    fn expecting(&self, formatter: &mut fmt::Formatter)
                    -> fmt::Result {
                        formatter.write_str("`port` or `healthz_port`
                        or `max_pods`")
                    }

                    fn visit_str<E>(self, value: &str) ->
                    Result<Field,
                    E>
                        where E: de::Error
                    {
                        match value {
                            "port" => Ok(Field::Port),
                            "healthz_port" =>
                            Ok(Field::HealthzPort),
                            "max_pods" => Ok(Field::MaxPods),
                            _ => Err(de::Error::unknown_field(value,
                            FIELDS)),
                        }
                    }
                }

                deserializer.deserialize_identifier(FieldVisitor)
            }
        }
}
```

Now, the input to the deserializer is json, which can be treated as a map. Thus, we will only need to implement `visit_map` from the `Visitor` trait. If any non-json data is passed to our deserializer, it will error out on a call to some other function from that trait. Most of the previous implementation is boilerplate. It boils down to a few parts: implementing `Visitor` for the fields, and implementing `visit_str` (since all of our fields are strings). At this point, we should be able to deserialize individual fields. The second part is to implement `Visitor` for the overall struct, and to implement `visit_map`. Errors must be handled appropriately in all cases. In the end, we can call `deserializer.deserialize_struct` and pass the name of the struct, the list of fields, and the visitor implementation for the whole struct.

This implementation will look like this:

```
// chapter4/serde-custom/src/main.rs

impl<'de> Deserialize<'de> for KubeConfig {
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
        where D: Deserializer<'de>
    {
        struct KubeConfigVisitor;

        impl<'de> Visitor<'de> for KubeConfigVisitor {
            type Value = KubeConfig;

            fn expecting(&self, formatter: &mut fmt::Formatter) ->
            fmt::Result {
                formatter.write_str("struct KubeConfig")
            }

            fn visit_map<V>(self, mut map: V) ->
            Result<KubeConfig,
            V::Error>
                where V: MapAccess<'de>
            {
                let mut port = None;
                let mut hport = None;
                let mut max = None;
                while let Some(key) = map.next_key()? {
                    match key {
                        Field::Port => {
                            if port.is_some() {
                                return
                            Err(de::Error::duplicate_field("port"));
                            }
                            port = Some(map.next_value()?);
                        }
                        Field::HealthzPort => {
                            if hport.is_some() {
                                return
                            Err(de::Error::duplicate_field
                            ("healthz_port"));
                            }
                            hport = Some(map.next_value()?);
                        }
                        Field::MaxPods => {
                            if max.is_some() {
                                return
                                Err(de::Error::duplicate_field
                                ("max_pods"));
                            }
                            max = Some(map.next_value()?);
                        }
                    }
                }
                let port = port.ok_or_else(||
                de::Error::missing_field("port"))?;
                let hport = hport.ok_or_else(||
                de::Error::missing_field("healthz_port"))?;
                let max = max.ok_or_else(||
                de::Error::missing_field("max_pods"))?;
                Ok(KubeConfig {port: port, healthz_port: hport,
                max_pods: max})
            }
        }

        const FIELDS: &'static [&'static str] = &["port",
        "healthz_port", "max_pods"];
        deserializer.deserialize_struct("KubeConfig", FIELDS,
        KubeConfigVisitor)
    }
}
```

Serde also provides a crate that can be used to unit test custom serializers and deserializers using a token-stream-like interface. To use it, we will need to add `serde_test` to our `Cargo.toml` and declare it as an extern crate in our main file. Here is a test for our deserializer:

```
// chapter4/serde-custom/src/main.rs

#[test]
fn test_ser_de() {
    let c = KubeConfig { port: 10, healthz_port: 11, max_pods: 12};

    assert_de_tokens(&c, &[
        Token::Struct { name: "KubeConfig", len: 3 },
        Token::Str("port"),
        Token::U8(10),
        Token::Str("healthz_port"),
        Token::U8(11),
        Token::Str("max_pods"),
        Token::U8(12),
        Token::StructEnd,
    ]);
}
```

The `assert_de_tokens` call checks if the given stream of tokens deserializes to our struct or not, thereby testing our deserializer. We can also add a main function to drive the serializer, like this:

```
// chapter4/serde-custom/src/main.rs

fn main() {
    let c = KubeConfig { port: 10, healthz_port: 11, max_pods: 12};
    let serialized = serde_json::to_string(&c).unwrap();
    println!("{:?}", serialized);
}
```

All this can now be run using Cargo. Using `cargo test` should run the test we just wrote, which should pass. `cargo run` should run the main function and print the serialized json:

```
$ cargo test
   Compiling serde-custom v0.1.0 (file:///serde-custom)
    Finished dev [unoptimized + debuginfo] target(s) in 0.61 secs
     Running target/debug/deps/serde_custom-81ee5105cf257563

running 1 test
test test_ser_de ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

$ cargo run
   Compiling serde-custom v0.1.0 (file:///serde-custom)
    Finished dev [unoptimized + debuginfo] target(s) in 0.54 secs
     Running `target/debug/serde-custom`
    "{\"port\":10,\"healthz_port\":11,\"max_pods\":12}"
```

Parsing textual data

Data parsing is a problem closely related to that of deserialization. The most common way of thinking about parsing is to start with a formal grammar and construct parsers based on that. This results in a bottom-up parser where smaller rules parse smaller components of the whole input. A final combinatorial rule combines all smaller rules in a given order to form the final parser. This way of formally defining a finite set of rules is called a **Parsing Expression Grammar** (**PEG**). This ensures that parsing is unambiguous; that there is only one valid parse tree if parsing succeeds. In the Rust ecosystem, there are a few distinct ways of implementing PEGs, and each of those have their own strengths and weaknesses. The first way is using macros to define a domain-specific language for parsing.

This method integrates well with the compiler through the new macro system, and can produce fast code. However, this is often harder to debug and maintain. Since this method does not allow overloading operators, the implementation must define a DSL, which might be more of a cognitive load for a learner. The second method is using the trait system. This method helps in defining custom operators and is often easier to debug and maintain. An example of a parser that uses the first approach is nom; examples of parsers using the second approach are pom and pest.

Our use case for parsing is mostly in the context of networking applications. In these cases, sometimes it is more useful to deal with raw strings (or byte streams) and parse required information instead of deserializing to a complex data structure. A common case for this is any text-based protocol, such as HTTP. A server might receive a raw request as a stream of bytes over a socket and parse it to extract information. In this section, we will study some common parsing techniques in the Rust ecosystem.

Now, nom is a parser combinator framework, which means it can combine smaller parsers to build more powerful parsers. This is a bottom-up approach that usually starts with writing very specific parsers that parse a well-defined thing from the input. The framework then provides methods to chain these small parsers into a complete one. This approach is in contrast to a top-down approach in the case of lex and yacc, where one would start with defining the grammar. It can handle both byte streams (binary data) or strings, and provides all of Rust’s usual guarantees. Let us start with parsing a simple string, which in this case is an HTTP GET or POST request. Like all cargo projects, we will first set up the structure:

```
$ cargo new --bin nom-http
```

Then we will add our dependencies (nom in this case). The resultant manifest should look like this:

```
$ cat Cargo.toml
[package]
name = "nom-http"
version = "0.1.0"
authors = ["Foo<foo@bar.com>"]

[dependencies.nom]
version = "3.2.1"
features = ["verbose-errors", "nightly"]
```

The crate provides a few extra features that are often useful for debugging; these are disabled by default and can be turned on by passing the list to the `features` flag, as shown in the preceding sample. Now, let's move on to our main file:

```
// chapter4/nom-http/src/main.rs

#[macro_use]
extern crate nom;

use std::str;
use nom::{ErrorKind, IResult};

#[derive(Debug)]
enum Method {
    GET,
    POST,
}

#[derive(Debug)]
struct Request {
    method: Method,
    url: String,
    version: String,
}

// A parser that parses method out of a HTT request
named!(parse_method<&[u8], Method>,
       return_error!(ErrorKind::Custom(12), alt!(map!(tag!("GET"), |_| Method::GET)
| map!(tag!("POST"), |_| Method::POST))));

// A parser that parses the request part
named!(parse_request<&[u8], Request>, ws!(do_parse!(
    method: parse_method >>
    url: map_res!(take_until!(" "), str::from_utf8) >>
    tag!("HTTP/") >>
    version: map_res!(take_until!("\r"), str::from_utf8) >>
    (Request { method: method, url: url.into(), version: version.into() })
)));

// Driver function for running the overall parser
fn run_parser(input: &str) {
    match parse_request(input.as_bytes()) {
      IResult::Done(rest, value) => println!("Rest: {:?} Value: {:?}",
      rest, value),
      IResult::Error(err) => println!("{:?}", err),
        IResult::Incomplete(needed) => println!("{:?}", needed)
    }
}

fn main() {
    let get = "GET /home/ HTTP/1.1\r\n";
    run_parser(get);
    let post = "POST /update/ HTTP/1.1\r\n";
    run_parser(post);
    let wrong = "WRONG /wrong/ HTTP/1.1\r\n";
    run_parser(wrong);
}
```

As might be obvious, nom makes heavy use of macros for code generation, the most important one being `named!`, which takes in a function signature and defines a parser based on that. A nom parser returns an instance of the `IResult` type; this is defined as an enum and has three variants:

- The `Done(rest, value)` variant represents the case where the current parser was successful. In this case, the value will have the current parsed value and the rest will have the remaining input to be parsed.
- The `Error(Err<E>)` variant represents an error during parsing. The underlying error will have the error code, position in error, and more. In a large parse tree, this can also hold pointers to more errors.
- The last variant, `Incomplete(needed)`, represents the case where parsing was incomplete for some reason. Needed is an enum that again has two variants; the first one represents the case where it is not known how much data is needed. The second one represents the exact size of data needed.

We start with representations for HTTP methods and the full request as structs. In our toy example, we will only deal with GET and POST, and ignore everything else. We then define a parser for the HTTP method; our parser will take in a slice of bytes and return the `Method` enum. This is simply done by reading the input and looking for the strings GET or POST. In each case, the base parser is constructed using the `tag!` macro, which parses input to extract the given string. And, if the parsing was successful, we convert the result to `Method` using the `map!` macro, which maps the result of a parser to a function. Now, for parsing the method, we will either have a POST or a GET, but never both. We use the `alt!` macro to express the logical OR of both the parsers we constructed before. The `alt!` macro will construct a parser that parses the input if any one of it's constituent macros can parse the given input. Finally, all this is wrapped in the `return_error!` macro, which returns early if parsing fails in the current parser, instead of passing onto the next parser in the tree.

We then move on to parsing the whole request by defining `parse_request`. We start with trimming extra whitespace from the input using the `ws!` macro. We then invoke the `do_parse!` macro that chains multiple sub-parsers. This one is different from other combinators because this allows storing results from intermediate parsers. This is useful in constructing instances of our structs while returning results. In `do_parse!`, we first call `parse_method` and store its result in a variable. Having removed the method from a request, we should encounter empty whitespace before we find the location of the object. This is handled by the `take_until!(" ")` call, which consumes input till it finds an empty space. The result is converted to a `str` using `map_res!`. The next parser in the list is one that removes the sequence `HTTP/` using the `tag!` macro. Next, we parse the HTTP version by reading input till we see a `\r`, and map it back to a `str`. Once we are done with all the parsing, we construct a `Request` object and return it. Note the use of the `>>` symbol as a separator between parsers in the sequence.

We also define a helper function called `run_parser` to run our parsers in a given input and print the result. This function calls the parser and matches on the result to display either the resultant structure or error. We then define our main function with three HTTP requests, the first two being valid, and the last one being invalid since the method is wrong. On running this, the output is as follows:

```
$ cargo run
   Compiling nom-http v0.1.0 (file:///Users/Abhishek/Desktop/rust-book/src/ch4/nom-http)
    Finished dev [unoptimized + debuginfo] target(s) in 0.60 secs
     Running `target/debug/nom-http`
Rest: [] Value: Request { method: GET, url: "/home/", version: "1.1" }
Rest: [] Value: Request { method: POST, url: "/update/", version: "1.1" }
NodePosition(Custom(128), [87, 82, 79, 78, 71, 32, 47, 119, 114, 111, 110, 103, 47,
32, 72, 84, 84, 80, 47, 49, 46, 49, 13, 10], [Position(Alt, [87, 82, 79, 78, 71,
32, 47, 119, 114, 111, 110, 103, 47, 32, 72, 84, 84, 80, 47, 49, 46, 49, 13, 10])])
```

In the first two cases, everything was parsed as expected and we got the result back. As expected, parsing failed in the last case with the custom error being returned.

As we discussed before, a common problem with nom is debugging, since it is much harder to debug macros. Macros also encourage the use of specific DSLs (like using the `>>` separator), which some people might find difficult to work with. At the time of writing, some error messages from nom are not helpful enough in finding what is wrong with a given parser. These will definitely improve in the future, but in the meantime, nom provides a few helper macros for debugging.

For instance, `dbg!` prints the result and the input if the underlying parser did not return a `Done`. The `dbg_dump!` macro is similar but also prints out a hex dump of the input buffer. In our experience, a few techniques can be used for debugging:

- Expanding the macro by passing compiler options to `rustc`. Cargo enables this using the following invocations: `cargo rustc -- -Z unstable-options --pretty=expanded` expands and pretty prints all macros in the given project. One might find it useful to expand the macros to trace execution and debug. A related command in Cargo, `rustc -- -Z trace-macros`, only expands the macros.
- Running smaller parsers in isolation. Given a series of parsers and another one combining those, it might be easier to run each of the sub-parsers till one of those errors out. Then, one can go on to debug only the small parser that is failing. This is very useful in isolating faults.
- Using the provided debugging macros `dbg!` and `dbg_dump!`. These can be used like debugging print statements to trace execution.

TIP

`pretty=expanded` is an unstable compiler option right now. Sometime in the future, it will be stabilized (or removed). In that case, one will not need to pass the `-Z unstable-options` flag to use it.

Let us look at an example of another parser combinator called `pom`. As we discussed before, this one relies heavily on traits and operator-overloading to implement parser combinations. As the time of writing, the current version is 1.1.0, and we will use that for our example project. Like always, the first step is to set up our project and add `pom` to our dependencies:

```
$ cargo new --bin pom-string
```

The `Cargo.toml` file will look like this:

```
[package]
name = "pom-string"
version = "0.1.0"
authors = ["Foo<foo@bar.com>"]

[dependencies]
pom = "1.1.0"
```

In this example, we will parse an example HTTP request, like last time. This is how it will look:

```
// chapter4/pom-string/src/main.rs

extern crate pom;

use pom::DataInput;
use pom::parser::{sym, one_of, seq};
use pom::parser::*;
use std::str;

// Represents one or more occurrence of an empty whitespace
fn space() -> Parser<'static, u8, ()> {
    one_of(b" \t\r\n").repeat(0..).discard()
}

// Represents a string in all lower case
fn string() -> Parser<'static, u8, String> {
    one_of(b"abcdefghijklmnopqrstuvwxyz").repeat(0..).convert(String::from_utf8)
}

fn main() {
    let get = b"GET /home/ HTTP/1.1\r\n";
    let mut input = DataInput::new(get);
    let parser = (seq(b"GET") | seq(b"POST")) * space() * sym(b'/') *
    string() * sym(b'/') * space() * seq(b"HTTP/1.1");
    let output = parser.parse(&mut input);
    println!("{:?}", str::from_utf8(&output.unwrap()));
}
```

We start with declaring our dependency on `pom`. In our main function, we define the final parser as a sequence of multiple sub-parsers. The `*` operator has been overloaded to make it apply multiple parsers in sequence.
The `seq` operator is a built-in parser that matches the given string from input. The `|` operator does a logical OR of the two operands. We define a function called `space()` that represents empty white spaces in input. This function takes one of each empty whitespace characters, repeats it 0 or more times, and then discards it. Consequently, the function returns a `Parser` with no return type, indicated by `()`. The string function is similarly defined to be one of the characters in the English alphabet, repeated 0 or more times, and then converted to an `std::String`.

The return type of this function is a `Parser` that has a `String`, as expected. Having set those up, our main parser will have a space, followed by the symbol `/`, followed by a string, a symbol `/`, a space again, and ending with the sequence `HTTP/1.1`. And, as expected, when we parse an example string with the parser we wrote, it produces an `Ok`:

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/pom-string`
Ok("HTTP/1.1")
```

PEG-based parser combinators can be easier to debug and to work with. They also tend to produce better error messages, but, unfortunately, those are not mature enough right now. The community around them is not as large as the community around nom. Consequently, it is often easier to get help with nom issues. At the end of the day, it is up to the programmer to choose something that works for them.

Parsing binary data

A related problem is that of parsing binary data. Common cases where this is applicable include parsing binary files and binary protocols. Let us look at how `nom` can be used to parse binary data. In our toy example, we will write a parser for the IPv6 header. Our `Cargo.toml` will look exactly the same as last time. Set up the project using the CLI:

```
$ cargo new --bin nom-ipv6
```

Our main file will look like this:

```
// chapter4/nom-ipv6/src/main.rs

#[macro_use]
extern crate nom;

use std::net::Ipv6Addr;

use nom::IResult;

// Struct representing an IPv6 header
#[derive(Debug, PartialEq, Eq)]
pub struct IPv6Header {
    version: u8,
    traffic_class: u8,
    flow_label: u32,
    payload_length: u16,
    next_header: u8,
    hop_limit: u8,
    source_addr: Ipv6Addr,
    dest_addr: Ipv6Addr,
}

// Converts a given slice of [u8] to an array of 16 u8 given by
// [u8; 16]
fn slice_to_array(input: &[u8]) -> [u8; 16] {
    let mut array = [0u8; 16];
    for (&x, p) in input.iter().zip(array.iter_mut()) {
        *p = x;
    }
    array
}

// Converts a reference to a slice [u8] to an instance of
// std::net::Ipv6Addr
fn to_ipv6_address(i: &[u8]) -> Ipv6Addr {
    let arr = slice_to_array(i);
    Ipv6Addr::from(arr)
}

// Parsers for each individual section of the header
named!(parse_version<&[u8], u8>, bits!(take_bits!(u8, 4)));
named!(parse_traffic_class<&[u8], u8>, bits!(take_bits!(u8, 8)));
named!(parse_flow_label<&[u8], u32>, bits!(take_bits!(u32, 20)));
named!(parse_payload_length<&[u8], u16>, bits!(take_bits!(u16, 16)));
named!(parse_next_header<&[u8], u8>, bits!(take_bits!(u8, 8)));
named!(parse_hop_limit<&[u8], u8>, bits!(take_bits!(u8, 8)));
named!(parse_address<&[u8], Ipv6Addr>, map!(take!(16), to_ipv6_address));
```

```
// The primary parser
named!(ipparse<&[u8], IPv6Header>,
       do_parse!(
            ver: parse_version >>
            cls: parse_traffic_class >>
            lbl: parse_flow_label >>
            len: parse_payload_length >>
            hdr: parse_next_header >>
            lim: parse_hop_limit >>
            src: parse_address >>
            dst: parse_address >>
              (IPv6Header {
                  version: ver,
                  traffic_class: cls,
                  flow_label: lbl,
                  payload_length: len,
                  next_header: hdr,
                  hop_limit: lim,
                  source_addr: src,
                  dest_addr : dst
              })
));

// Wrapper for the parser
pub fn parse_ipv6_header(i: &[u8]) -> IResult<&[u8], IPv6Header> {
    ipparse(i)
}

fn main() {
    const EMPTY_SLICE: &'static [u8] = &[];
    let bytes = [0x60,
                 0x00,
                 0x08, 0x19,
                 0x80, 0x00, 0x14, 0x06,
                 0x40,
                 0x2a, 0x02, 0x0c, 0x7d, 0x2e, 0x5d, 0x5d, 0x00,
                 0x24, 0xec, 0x4d, 0xd1, 0xc8, 0xdf, 0xbe, 0x75,
                 0x2a, 0x00, 0x14, 0x50, 0x40, 0x0c, 0x0c, 0x0b,
                 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xbd
                 ];

    let expected = IPv6Header {
        version: 6,
        traffic_class: 0,
        flow_label: 33176,
        payload_length: 20,
        next_header: 6,
        hop_limit: 64,
        source_addr:
        "2a02:c7d:2e5d:5d00:24ec:4dd1:c8df:be75".parse().unwrap(),
        dest_addr: "2a00:1450:400c:c0b::bd".parse().unwrap(),
    };
    assert_eq!(ipparse(&bytes), IResult::Done(EMPTY_SLICE, expected));
}
```

Here, we start with declaring a struct for the IPv6 fixed header, as defined in RFC 2460 (https://tools.ietf.org/html/rfc2460). We first define a helper function called `to_ipv6_address` that takes in a slice of `u8` and converts to an IPv6 address. To do that, we need another helper function that converts a slice to a fixed-size array (`16` in this case). Having set those up, we define a number of parsers for parsing each of the members of the struct using the `named!` macro.

The `parse_version` function takes in a slice of bytes and returns the version as a `u8`. This is done by reading 4 bits from the input as a `u8`, using the `take_bits!`

macro. That is then wrapped in the `bits!` macro which transforms the input to a bit stream for the underlying parser. In the same way, we go on to define parsers for all the other fields in the header structure. For each one, we take the number of bits they occupy according to the RFC and convert to a type large enough to hold it. The last case of parsing the address is different. Here, we read 16 bytes using the `take!` macro and map it to the `to_ipv6_address` function to convert the byte stream, using the `map!` macro.

At this point, all small pieces to parse the whole struct are ready, and we can define a function using the `do_parse!` macro. In there, we accumulate results in temporary variables and construct an instance of the `IPv6Header` struct, which is then returned. In our main function, we have an array of bytes that was taken from a IPv6 packet dump and should represent a valid IPv6 header. We parse that using the parser we defined and assert that the output matches what is expected. Thus, a successful run of our parser previously will not throw an exception.

Let us recap all the macros from `nom` that we used so far:

Macro	**Purpose**
`named!`	Creates a parsing function by combining smaller functions. This (or a variant) is always the top-level call in a chain.
`ws!`	Enables a parser to consume all whitespaces (`\t`, `\r` and `\n`) between tokens.
`do_parse!`	Applies subparsers in a given sequence, can store intermediate results.
`tag!`	Declares a static sequence of bytes that the enclosing parser should recognize.
`take_until!`	Consumes input till the given tag.
`take_bits!`	Consumes the given number of bits from the input and casts them to the given type.
`take!`	Consumes the specified number of bytes from input.
`map_res!`	Maps a function (returning a result) on the output of a parser.
`map!`	Maps a function to the output of a parser.
`bits!`	Transforms the given slice to a bit stream.

Summary

In this section, we studied handling data in more detail. Specifically, (de)serializing and parsing. At the time of writing, Serde and related crates are the community-supported way of (de)serializing data in Rust, while `nom` is the most frequently used parser combinator. These tools tend to produce better error messages on the nightly compiler, and with a few feature flags turned on, since they often depend on a few cutting edge night-only features. With time, these features will be available in the stable compiler, and these tools will work seamlessly.

In the next chapter, we will talk about the next steps after having made sense of incoming data on a socket. More often than not, this involves dealing with application-level protocols.

Application Layer Protocols

As we saw in the previous few chapters, two hosts in a network exchange bytes, either in a stream or in discrete packets. It is often up to a higher-level application to process those bytes to something that makes sense to the application. These applications define a new layer of protocol over the transport layer, often called application layer protocols. In this chapter, we will look into some of those protocols.

There are a number of important considerations for designing application layer protocols. An implementation needs to know at least the following details:

- Is the communication broadcast or point-to-point? In the first case, the underlying transport protocol must be UDP. In the second case, it can be either TCP or UDP.
- Does the protocol need a reliable transport? If yes, TCP is the only option. Otherwise, UDP might be suitable, too.
- Does the application need a byte stream (TCP), or can it work on packet-by-packet basis (UDP)?
- What signals the end of input between the parties?
- What is the data format and encoding used?

Some very commonly used application layer protocols are DNS (which we studied in the previous chapters) and HTTP (which we will study in a subsequent chapter). Other than these, a very important application layer toolset commonly used for microservice-based architectures is gRPC. Another application layer protocol everyone has used at least a few times is SMTP, the protocol that powers email.

In this chapter, we will study the following topics:

- How RPC works. Specifically, we will look at gRPC and write a small server and client using the toolkit.
- We will take a look at a crate caller `lettre` that can be used to send emails programmatically.
- The last topic will be on writing a simple FTP client and a TFTP server in Rust.

Introduction to RPC

In regular programming, it is often useful to encapsulate frequently used logic in a function so that it can be reused in multiple places. With the rise of networked and distributed systems, it became necessary to let a common set of operations be accessible over a network, so that validated clients can call them. This is often called a **Remote Procedure Call** (**RPC**). In Chapter 4, *Data Serialization, De-Serialization, and Parsing*, we saw a simple example of this when a server returned the distance of a given point from the origin. Real world RPC has a number of application layer protocols defined, which are far more complex. One of the most popular RPC implementations is gRPC, which was initially introduced by Google and later moved to an open source model. gRPC offers high performance RPC over internet scale networks and is widely used in a number of projects, including Kubernetes.

Before digging deeper into gRPC, let's look at protocol buffers, a related tool. It is a set of mechanisms to build language and platform neutral exchanging structured data between applications. It defines its own **Interface Definition Language** (**IDL**) to describe the data format, and a compiler that can take that format and generate code to convert to and from it. The IDL also allows for defining abstract services: input and output message formats that the compiler can use to generate stubs in a given language. We will see an example of a definition of a data format in a subsequent example. The compiler has plugins to generate output code in a large number of languages, including Rust. In our example, we will use such a plugin in a build script to autogenerate Rust modules. Now, gRPC uses protocol buffers to define the underlying data and messages. Messages are exchanged over HTTP/2 on top of TCP/IP. This mode of communication is often faster in practice, since it can make better use of existing connections, and also since HTTP/2 supports bidirectional asynchronous connections. gRPC, being an opinionated system, makes a lot of assumptions about the considerations we discussed in the previous section, on our behalf. Most of these defaults (like HTTP/2 over TCP) were chosen because they support the advanced features gRPC offers (like bidirectional streaming). Some other defaults, like using `protobuf`, can be swapped for another message format implementation.

For our gRPC example, we will build a service that is a lot like Uber. It has a central server where clients (cabs) can record their names and locations. And then, when a user requests a cab with their location, the server sends back a list of cabs near that user. Ideally, this server should have two classes of clients, one for cabs and one for users. But for simplicity’s sake, we will assume that we have only one type of client.

Let us start with setting up the project. Like always, we will use Cargo CLI to initialize the project:

```
$ cargo new --bin grpc_example
```

All of our dependencies are listed in the `dependencies` section of `Cargo.toml`, as shown in the following code snippet. In this example, we will use the build script feature to generate Rust source code from our protocol definition, using the `protoc-rust-grpc` crate. Hence, we will need to add that as a build dependency:

```
$ cat Cargo.toml
[package]
name = "grpc_example"
version = "0.1.0"
authors = ["Foo<foo@bar.com>"]

[dependencies]
protobuf = "1.4.1"
grpc = "0.2.1"
tls-api = "0.1.8"

[build-dependencies]
protoc-rust-grpc = "0.2.1"
```

The following script is our build script. It is simply a Rust executable (which has a main function) that Cargo builds and runs right before calling the compiler on the given project. Note that the default name for this script is `build.rs`, and it must be located in the project root. However, these parameters can be configured in the Cargo config file:

```
// ch5/grpc/build.rs

extern crate protoc_rust_grpc;

fn main() {
    protoc_rust_grpc::run(protoc_rust_grpc::Args {
        out_dir: "src",
        includes: &[],
        input: &["foobar.proto"],

        rust_protobuf: true,
    }).expect("Failed to generate Rust src");
}
```

TIP *One of the most common use cases for build scripts is code generation (like our current project). They can also be used to find and configure native libraries on the host, and so on.*

In the script, we use the `protoc_rust_grpc` crate to generate Rust modules from our `proto` file (called `foobar.proto`). We also set the `rust_protobuf` flag to make it generate protobuf messages. Note that the `protoc` binary must be available in

`$PATH` for this to work. This is a part of the protobuf package. Follow these steps to install it from the source:

1. Download the pre-built binaries from GitHub:

```
$ curl -Lo
https://github.com/google/protobuf/releases/download/v3.5.1/protoc-3.5.1-
linux-x86_64.zip
```

2. Unzip the archive:

```
$ unzip protoc-3.5.1-linux-x86_64.zip -d protoc3
```

3. Copy the binary to somewhere in `$PATH`:

```
$ sudo mv protoc3/bin/* /usr/local/bin/
```

This current example has been tested on Ubuntu 16.04 with protoc version 3.5.1.

Next, we will need the protocol definition, as shown in the following code snippet:

```
// ch5/grpc/foobar.proto

syntax = "proto3";

package foobar;

// Top level gRPC service with two RPC calls
service FooBarService {
    rpc record_cab_location(CabLocationRequest) returns
    (CabLocationResponse);
    rpc get_cabs(GetCabRequest) returns (GetCabResponse);
}

// A request to record location of a cab
// Name: unique name for a cab
// Location: current location of the given cab
message CabLocationRequest {
    string name = 1;
    Location location = 2;
}

// A response for a CabLocationRequest
// Accepted: a boolean indicating if this
// request was accepted for processing
message CabLocationResponse {
    bool accepted = 1;
}

// A request to return cabs at a given location
// Location: a given location
message GetCabRequest {
    Location location = 1;
}

// A response for GetCabLocation
// Cabs: list of cabs around the given location
```

```
message GetCabResponse {
    repeated Cab cabs = 1;
}

// Message that the CabLocationRequest passes
// to the server
message Cab {
    string name = 1;
    Location location = 2;
}

// Message with the location of a cab
message Location {
    float latitude = 1;
    float longitude = 2;
}
```

The proto file starts with a declaration of the version of protobuf IDP spec; we will be using version 3. The package declaration indicates that all the generated code will be placed in a Rust module called `foobar`, and all other generated code will be placed in a module called `foobar_grpc`. We define a service called `FooBarService` that has two RPC functions; `record_cab_location` records the location of a cab, given its name and location, and `get_cabs` returns a set of cabs, given a location. We will also need to define all associated `protobuf` messages for each of the requests and responses. The spec also defines a number of built-in data types that closely correspond to those in a programming language (string, float, and so on).

Having set up everything related to the `protobuf` message formats and functions, we can use Cargo to generate actual Rust code. The generated code will be located in the `src` directory and will be called `foobar.rs` and `foobar_grpc.rs`. These names are automatically assigned by the compiler. The `lib.rs` file should re-export those using the pub mod syntax. Note that Cargo build will not modify the `lib.rs` file for us; that needs to be done by hand. Let us move on to our server and client. Here is what the server will look:

```
// ch5/grpc/src/bin/server.rs

extern crate grpc_example;
extern crate grpc;
extern crate protobuf;

use std::thread;

use grpc_example::foobar_grpc::*;
use grpc_example::foobar::*;

struct FooBarServer;

// Implementation of RPC functions
    impl FooBarService for FooBarServer {
    fn record_cab_location(&self,
                           _m: grpc::RequestOptions,
                           req: CabLocationRequest)
                           ->
        grpc::SingleResponse<CabLocationResponse> {
        let mut r = CabLocationResponse::new();
        println!("Recorded cab {} at {}, {}", req.get_name(),
        req.get_location().latitude, req.get_location().longitude);

        r.set_accepted(true);
        grpc::SingleResponse::completed(r)
    }

    fn get_cabs(&self,
                    _m: grpc::RequestOptions,
                    _req: GetCabRequest)
                    -> grpc::SingleResponse<GetCabResponse> {
        let mut r = GetCabResponse::new();

        let mut location = Location::new();
        location.latitude = 40.7128;
        location.longitude = -74.0060;

        let mut one = Cab::new();
        one.set_name("Limo".to_owned());
        one.set_location(location.clone());

        let mut two = Cab::new();
        two.set_name("Merc".to_owned());
        two.set_location(location.clone());

        let vec = vec![one, two];
        let cabs = ::protobuf::RepeatedField::from_vec(vec);

        r.set_cabs(cabs);

        grpc::SingleResponse::completed(r)
    }
}

fn main() {
    let mut server = grpc::ServerBuilder::new_plain();
    server.http.set_port(9001);
    server.add_service(FooBarServiceServer::new_service_def(FooBarServer));
    server.http.set_cpu_pool_threads(4);
    let _server = server.build().expect("Could not start server");
    loop {
        thread::park();
    }
}
```

Note that this server is very different from the servers we wrote in previous chapters. This is because `grpc::ServerBuilder` encapsulates a lot of the complexity in writing servers. `FooBarService` is the service `protobuf` compiler generated for us, defined as a trait in the file `foobar_grpc.rs`. As expected, this trait has two methods: `record_cab_location` and `get_cabs`. Thus, for our server, we will need to implement this trait on a struct and pass that struct to `ServerBuilder` to run on a given port.

In our toy example, we will not actually record cab locations. A real world app would want to put these in a database to be looked up later. Instead, we will just print a message saying that we received a new location. We also need to work with some boilerplate code here, to make sure all gRPC semantics are fulfilled. In the `get_cabs` function, we always return a static list of cabs for all requests. Note that since all `protobuf` messages are generated for us, we get a bunch of utility functions, like `get_name` and `get_location`, for free. Finally, in the `main` function, we pass our server struct to gRPC to create a new server on a given port and run it on an infinite loop.

Our client is actually defined as a struct in the source generated by the `protobuf` compiler. We just need to make sure the client has the same port number we are running our server on. We use the `new_plain` method on the client struct and pass an address and port to it, along with some default options. We can then call the `record_cab_location` and `get_cabs` methods over RPC and process the responses:

```
// ch5/grpc/src/bin/client.rs

extern crate grpc_example;
extern crate grpc;

use grpc_example::foobar::*;
use grpc_example::foobar_grpc::*;


fn main() {
    // Create a client to talk to a given server
    let client = FooBarServiceClient::new_plain("127.0.0.1", 9001,
    Default::default()).unwrap();

    let mut req = CabLocationRequest::new();
    req.set_name("foo".to_string());

    let mut location = Location::new();
    location.latitude = 40.730610;
    location.longitude = -73.935242;
    req.set_location(location);

    // First RPC call
    let resp = client.record_cab_location(grpc::RequestOptions::new(),
    req);
    match resp.wait() {
        Err(e) => panic!("{:?}", e),
        Ok((_, r, _)) => println!("{:?}", r),
    }

    let mut nearby_req = GetCabRequest::new();
    let mut location = Location::new();
    location.latitude = 40.730610;
    location.longitude = -73.935242;
    nearby_req.set_location(location);

    // Another RPC call
    let nearby_resp = client.get_cabs(grpc::RequestOptions::new(),
    nearby_req);
    match nearby_resp.wait() {
        Err(e) => panic!("{:?}", e),
        Ok((_, cabs, _)) => println!("{:?}", cabs),
    }
}
```

Here is how a run of the client will look like. As noted before, this is not as dynamic as it should be, since it returns only hardcoded values:

```
$ cargo run --bin client
   Blocking waiting for file lock on build directory
   Compiling grpc_example v0.1.0 (file:///rust-book/src/ch5/grpc)
   Finished dev [unoptimized + debuginfo] target(s) in 3.94 secs
     Running `/rust-book/src/ch5/grpc/target/debug/client`
accepted: true
cabs {name: "Limo" location {latitude: 40.7128 longitude: -74.006}} cabs {name:
"Merc" location {latitude: 40.7128 longitude: -74.006}}
```

Notice how it exits right after talking to the server. The server, on the other hand, runs in an infinite loop, and does not exit till it gets a signal:

```
$ cargo run --bin server
   Compiling grpc_example v0.1.0 (file:///rust-book/src/ch5/grpc)
    Finished dev [unoptimized + debuginfo] target(s) in 5.93 secs
     Running `/rust-book/src/ch5/grpc/target/debug/server`
Recorded cab foo at 40.73061, -73.93524
```

Introduction to SMTP

Internet email uses a protocol called **Simple Mail Transfer Protocol** (**SMTP**), which is an IETF standard. Much like HTTP, it is a simple text protocol over TCP, using port `25` by default. In this section, we will look at a small example of using `lettre` for sending emails. For this to work, let us set up our project first:

```
$ cargo new --bin lettre-example
```

Now, our `Cargo.toml` file should look like this:

```
$ cat Cargo.toml
[package]
name = "lettre-example"
version = "0.1.0"
authors = ["Foo<foo@bar.com>"]

[dependencies]
lettre = "0.7"
uuid = "0.5.1"
native-tls = "0.1.4"
```

Let’s say we want to send crash reports for a server automatically. For this to work, we need to have an SMTP server running somewhere accessible. We also need to have a user who can authenticate using a password set up on that server. Having set those up, our code will look like this:

```
// ch5/lettre-example/src/main.rs

extern crate uuid;
extern crate lettre;
extern crate native_tls;

use std::env;
use lettre::{SendableEmail, EmailAddress, EmailTransport};
use lettre::smtp::{SmtpTransportBuilder, SUBMISSION_PORT};
use lettre::smtp::authentication::Credentials;
use lettre::smtp::client::net::ClientTlsParameters;

use native_tls::TlsConnector;

// This struct represents our email with all the data
// we want to send.
struct CrashReport {
    to: Vec<EmailAddress>,
    from: EmailAddress,
    message_id: String,
    message: Vec<u8>,
}

// A simple constructor for our email.
impl CrashReport {
    pub fn new(from_address: EmailAddress,
        to_addresses: Vec<EmailAddress>,
        message_id: String,
        message: String) -> CrashReport {
            CrashReport { from: from_address,
            to: to_addresses,
            message_id: message_id,
            message: message.into_bytes()
            }
        }
}

impl<'a> SendableEmail<'a, &'a [u8]> for CrashReport {
    fn to(&self) -> Vec<EmailAddress> {
        self.to.clone()
    }

    fn from(&self) -> EmailAddress {
        self.from.clone()
    }

    fn message_id(&self) -> String {
        self.message_id.clone()
    }

    fn message(&'a self) -> Box<&[u8]> {
        Box::new(self.message.as_slice())
    }
}

fn main() {
    let server = "smtp.foo.bar";
    let connector = TlsConnector::builder().unwrap().build().unwrap();
    let mut transport = SmtpTransportBuilder::new((server, SUBMISSION_PORT),
lettre::ClientSecurity::Opportunistic(<ClientTlsParameters>::new(server.to_string(),
 connector)))
.expect("Failed to create transport")
    .credentials(Credentials::new(env::var("USERNAME").unwrap_or_else(|_|
"user".to_string()),
env::var("PASSWORD").unwrap_or_else
(|_| "password".to_string())))
    .build();
    let report = CrashReport::new(EmailAddress::
    new("foo@bar.com".to_string()), vec!
    [EmailAddress::new("foo@bar.com".to_string())]
    , "foo".to_string(), "OOPS!".to_string());
    transport.send(&report).expect("Failed to send the report");
}
```

Our email is represented by the `CrashReport` struct; as expected, it has a `from` email address. The `to` field is a vector of email addresses, enabling us to send an email to multiple addresses. We implement a constructor for the struct. The crate `lettre` defines a trait called `SendableEmail` that has a bunch of properties an SMTP server needs to send an email. For a user-defined email to be sendable, it needs to implement that trait. In our case, `CrashReport` needs to implement it. We go on to implement all required methods in the trait. At this point, a new instance of `CrashReport` should be sendable as an email.

In our main function, we will need to `auth` against the SMTP server to send our emails. We create a transport object which has all of the required info to talk to the SMTP server. The username and password can be passed as environment variables (or defaults). We then create an instance of our `CrashReport` and use the `send` method of the transport to send it. Running this does not output any information (if it ran successfully).

One might have noticed that the API exposed by `lettre` is not very easy to use. This is primarily because the library is largely immature, being at version 0.7 at the time of writing. Thus, one should expect breaking changes in the API till it reaches a 1.0 release.

Introduction to FTP and TFTP

Another common application layer protocol is the **File Transfer Protocol** (**FTP**). This is a text-based protocol, where the server and clients exchange text commands to upload and download files. The Rust ecosystem has a crate called rust-ftp to interact with FTP servers programmatically. Let us look at an example of its use. We set up our project using Cargo:

```
$ cargo new --bin ftp-example
```

Our `Cargo.toml` should look like this:

```
[package]
name = "ftp-example"
version = "0.1.0"
authors = ["Foo<foo@bar.com>"]

[dependencies.ftp]
version = "2.2.1"
```

For this example to work, we will need a running FTP server somewhere. Once we have set that up and made sure a regular FTP client can connect to it, we can move on to our main code:

```
// ch5/ftp-example/src/main.rs

extern crate ftp;

use std::str;
use std::io::Cursor;
use ftp::{FtpStream, FtpError};

fn run_ftp(addr: &str, user: &str, pass: &str) -> Result<(), FtpError> {
    let mut ftp_stream = FtpStream::connect((addr, 21))?;
    ftp_stream.login(user, pass)?;
    println!("current dir: {}", ftp_stream.pwd()?);

    let data = "A random string to write to a file";
    let mut reader = Cursor::new(data);
    ftp_stream.put("my_file.txt", &mut reader)?;

    ftp_stream.quit()
}

fn main() {
    run_ftp("ftp.dlptest.com", "dlpuser@dlptest.com", "eiTqR7EMZD5zy7M").unwrap();
}
```

For our example, we will connect to a free public FTP server, located at `ftp.dlptest.com`. The credentials for using the server are in this website: https://dlptest.com/ftp-test/. Our helper function, called `run_ftp`, takes in the address of an FTP server, with its username and password as strings. It then connects to the server on port `21` (the default port for FTP). It goes on to log in using the given credentials, and then prints the current directory (which should be `/`). We then write a file there using the `put` function, and, at the end, close our connection to the server. In our `main` function, we simply call the helper with the required parameters.

A thing to note here is the usage of `cursor`; it represents an inmemory buffer and provides implementations of `Read`, `Write`, and `Seek` over that buffer. The `put` function expects an input that implements `Read`; wrapping our data in a `Cursor` automatically does that for us.

Here is what we see on running this example:

```
$ cargo run
   Compiling ftp-example v0.1.0 (file:///Users/Abhishek/Desktop/rust-
book/src/ch5/ftp-example)
    Finished dev [unoptimized + debuginfo] target(s) in 1.5 secs
     Running `target/debug/ftp-example`
current dir: /
```

We have used an open FTP server in this example. Since this server is not under our control, it might be taken offline without notice. If that happens, the example will need to be modified to use another server.

A protocol closely related to FTP is called **Trivial File Transfer Protocol** (**TFTP**). TFTP is text-based, like FTP, but unlike FTP, it is way simpler to implement and maintain. It uses UDP for transport and does not provide any authentication primitives. Since it is faster and lighter, it is frequently implemented in embedded systems and boot protocols, like PXE and BOOTP. Let us look at a simple TFTP server using the crate called `tftp_server`. For this example, we will start with Cargo, like this:

```
$ cargo new --bin tftp-example
```

Our manifest is very simple and looks like this:

```
[package]
name = "tftp-example"
version = "0.1.0"
authors = ["Foo <foo@bar.com>"]

[dependencies]
tftp_server = "0.0.2"
```

Our main file will look like this:

```
// ch5/tftp-example/src/main.rs

extern crate tftp_server;

use tftp_server::server::TftpServer;

use std::net::SocketAddr;
use std::str::FromStr;

fn main() {
    let addr = format!("0.0.0.0:{}", 69);
    let socket_addr = SocketAddr::from_str(addr.as_str()).expect("Error
    parsing address");
    let mut server =
    TftpServer::new_from_addr(&socket_addr).expect("Error creating
    server");
    match server.run() {
        Ok(_) => println!("Server completed successfully!"),

        Err(e) => println!("Error: {:?}", e),
    }
}
```

This can be easily tested on any Unix machine that has a TFTP client installed. If we run this on one terminal and run the client on another, we will need to connect the client to a localhost on port `69`. We should then be able to download a file from the server.

Running this might require root privileges. If that is the case, use `sudo`*.*

$ sudo ./target/debug/tftp-example

An example session is as follows:

```
$ tftp
tftp> connect 127.0.0.1 69
tftp> status
Connected to 127.0.0.1.
Mode: netascii Verbose: off Tracing: off
Rexmt-interval: 5 seconds, Max-timeout: 25 seconds
tftp> get Cargo.toml
Received 144 bytes in 0.0 seconds
tftp> ^D
```

Summary

In this chapter, we built upon what we studied previously. In essence, we moved the network stack up to the application layer. We studied some major considerations for building application layer protocols. We then looked at RPC, and in particular, gRPC, studying how it enables developers to build large-scale networked services. We then looked at a Rust crate that can be used to send emails via an SMTP server. The last few examples were on writing an FTP client and a TFTP server. Along with other application layer protocols covered elsewhere in this book, we should have a good standing in understanding these protocols.

HTTP is one text-based application layer protocol that deserves a chapter of its own. In the next chapter, we will take a closer look at it and write some code to make it work.

Talking HTTP in the Internet

The single most important application-layer protocol that has changed our lives heavily has to be the HTTP. It forms the backbone of the World Wide Web (WWW). In this chapter, we will look at how Rust makes writing fast HTTP servers easier. We will also look at writing clients to communicate with these servers over a network.

We will cover the following topics in this chapter:

- A short introduction to Hyper, one of the most widely used crates for writing HTTP servers
- We will study Rocket, a new crate that has become widely popular owing to its simpler interface
- We will move on to reqwest, an HTTP client library

Introducing Hyper

Hyper is arguably the most stable and well-known of Rust-based HTTP frameworks. It has two distinct components, one for writing HTTP servers and one for writing clients. Recently, the server component was moved to a new async programming model based on tokio and futures. As a result, it is well-suited for high-traffic workloads. However, like a lot of other libraries in the ecosystem, Hyper has not hit Version 1.0 yet, so one should expect breaking API changes.

We will start with writing a small HTTP server in Hyper. Like always, we will need to set up our project using Cargo.

```
$ cargo new --bin hyper-server
```

Let us now add dependencies that will include `hyper` and `futures`. The `Cargo.toml` file will look as follows:

```
[package]
name = "hyper-server"
version = "0.1.0"
authors = ["Foo<foo@bar.com>"]

[dependencies]
hyper = "0.11.7"
futures = "0.1.17"
```

Our main file is pretty simple. In the real world, HTTP servers often talk to a backend, and all that can take a while to complete. Thus, it is common for replies to be a bit delayed. We will simulate that using a function that sleeps for 200 ms each time it is called, and then returns a fixed string.

```
// ch6/hyper-server/src/main.rs

extern crate hyper;
extern crate futures;

use std::{ thread, time };
use futures::future::FutureResult;
use hyper::{Get, StatusCode};
use hyper::header::ContentLength;
use hyper::server::{Http, Service, Request, Response};

// Simulate CPU intensive work by sleeping for 200 ms
fn heavy_work() -> String {
    let duration = time::Duration::from_millis(200);
    thread::sleep(duration);
    "done".to_string()
}

#[derive(Clone, Copy)]
struct Echo;

impl Service for Echo {
    type Request = Request;
    type Response = Response;
    type Error = hyper::Error;
    type Future = FutureResult<Response, hyper::Error>;

    // This method handles actually processing requests
    // We only handle GET requests on /data and ignore everything else
    // returning a HTTP 404
    fn call(&self, req: Request) -> Self::Future {
        futures::future::ok(match (req.method(), req.path()) {
        (&Get, "/data") => {
        let b = heavy_work().into_bytes();
        Response::new()
        .with_header(ContentLength(b.len() as u64))
        .with_body(b)
}
        _ => Response::new().with_status(StatusCode::NotFound),})
    }
}

fn main() {
    let addr = "0.0.0.0:3000".parse().unwrap();
    let server = Http::new().bind(&addr, || Ok(Echo)).unwrap();
    server.run().unwrap();
}
```

As Hyper heavily relies on `tokio` to do asynchronous handling of requests, an HTTP server in Hyper needs to implement a built-in trait called `Service` from `tokio`. This is essentially a function that maps a `Request` to a `Response` via an implementation of the `call` method. This method returns the result as a `Future`, indicating eventual completion of the given task. In that implementation, we match the method and path of the incoming request. If the method is `GET` and the path is `/data`, we call `heavy_work` and get the result. We then compose a response by setting the `Content-Length` header to the size of the string we are returning and the body of the response. In our `main` function, we construct our server by binding it to a known port. At the end, we call `run` on it to start the server.

Interacting with the server is easy with `curl`; a session should look like this:

```
$ curl http://127.0.0.1:3000/data
done$
```

Let us benchmark our server. For this, we will install ApacheBench (https://httpd.apache.org/docs/trunk/programs/ab.html). We will run 1,000 total requests from 100 clients in parallel by passing some command-line parameters to ApacheBench. This will take a while to complete, and we are waiting 200 ms before returning each response. So, for 1,000 requests, we will wait for at least 200 seconds. On one run, the output looks like this:

```
$ ab -n 1000 -c 100 http://127.0.0.1:3000/data
Benchmarking 127.0.0.1 (be patient)
Completed 100 requests
Completed 200 requests
Completed 300 requests
Completed 400 requests
Completed 500 requests
Completed 600 requests
Completed 700 requests
Completed 800 requests
Completed 900 requests
Completed 1000 requests
Finished 1000 requests


Server Software:
Server Hostname: 127.0.0.1
Server Port: 3000

Document Path: /data
Document Length: 4 bytes

Concurrency Level: 100
Time taken for tests: 203.442 seconds
Complete requests: 1000
Failed requests: 0
Total transferred: 79000 bytes
HTML transferred: 4000 bytes
Requests per second: 4.92 [#/sec] (mean)
Time per request: 20344.234 [ms] (mean)
Time per request: 103.442 [ms] (mean, across all concurrent requests)
Transfer rate: 0.38 [Kbytes/sec] received

Connection Times (ms)
min mean[+/-sd] median max
Connect: 0 2 0.7 2 3
Processing: 5309 20123 8061.9 20396 33029
Waiting: 203 12923 5518.0 14220 20417
Total: 5311 20124 8061.9 20397 33029

Percentage of the requests served within a certain time (ms)
  50% 20397
  66% 25808
  75% 26490
  80% 27263
  90% 28373
  95% 28568
  98% 33029
  99% 33029
 100% 33029 (longest request)
```

Notice that, across all requests, the server takes around 103.4 ms to reply back. This matches our expectation of 100 ms with the extra time being spent on other things. Also, our server is processing 4.92 requests per second, which is way too low for a reasonable server. This is all because our server is single-threaded, and only one thread serves all clients. This server also ignores the fact that multiple CPU cores are available on the host.

Let us go ahead and write a server that largely does the same thing, the difference being that this one uses multi-threading heavily and uses all CPU cores. Cargo setup should be as follows:

```
$ cargo new --bin hyper-server-faster
```

We will need to add some more crates as dependencies, and our `Cargo.toml` should be as follows:

```
[package]
name = "hyper-server-faster"
```

```
version = "0.1.0"
authors = ["Foo<foo@bar.com>"]

[dependencies]
hyper = "0.11.7"
futures = "0.1.17"
net2 = "0.2.31"
tokio-core = "0.1.10"
num_cpus = "1.0"
```

We have a number of extra things here. `tokio-core` will be used to run an event loop (as we did in mio, in Chapter 3, *TCP and UDP Using Rust*), `net2` will be used for some advanced socket configuration, and `num_cpus` will be used to figure out the number of CPU cores on the machine. Having set those up, our main file is pretty simple:

```
// ch6/hyper-server-faster/src/main.rs

extern crate futures;
extern crate hyper;
extern crate net2;
extern crate tokio_core;
extern crate num_cpus;

use futures::Stream;
use net2::unix::UnixTcpBuilderExt;
use tokio_core::reactor::Core;
use tokio_core::net::TcpListener;
use std::{thread, time};
use std::net::SocketAddr;
use std::sync::Arc;
use futures::future::FutureResult;
use hyper::{Get, StatusCode};
use hyper::header::ContentLength;
use hyper::server::{Http, Service, Request, Response};

// Same method like last example
fn heavy_work() -> String {
    let duration = time::Duration::from_millis(200);
    thread::sleep(duration);
    "done".to_string()
}

#[derive(Clone, Copy)]
struct Echo;

impl Service for Echo {
    type Request = Request;
    type Response = Response;
    type Error = hyper::Error;
    type Future = FutureResult<Response, hyper::Error>;

    fn call(&self, req: Request) -> Self::Future {
        futures::future::ok(match (req.method(), req.path()) {
            (&Get, "/data") => {
                let b = heavy_work().into_bytes();
                Response::new()
                    .with_header(ContentLength(b.len() as u64))
                    .with_body(b)
            }
            _ => Response::new().with_status(StatusCode::NotFound),
        })
    }
}

// One server instance
```

```
fn serve(addr: &SocketAddr, protocol: &Http) {
    let mut core = Core::new().unwrap();
    let handle = core.handle();
    let listener = net2::TcpBuilder::new_v4()
        .unwrap()
        .reuse_port(true)
        .unwrap()
        .bind(addr)
        .unwrap()
        .listen(128)
        .unwrap();
    let listener = TcpListener::from_listener(listener, addr,
    &handle).unwrap();
    core.run(listener.incoming().for_each(|(socket, addr)| {
        protocol.bind_connection(&handle, socket, addr, Echo);
        Ok(())
    })).unwrap();
}

// Starts num number of serving threads
fn start_server(num: usize, addr: &str) {
    let addr = addr.parse().unwrap();

    let protocol = Arc::new(Http::new());
    {
        for _ in 0..num - 1 {
            let protocol = Arc::clone(&protocol);
            thread::spawn(move || serve(&addr, &protocol));
        }
    }
    serve(&addr, &protocol);
}


fn main() {
    start_server(num_cpus::get(), "0.0.0.0:3000");
}
```

Functionally, this server is exactly the same as the last one. Architecturally, they are very different. Our implementation of `Service` is the same. What changed majorly is that we split starting the server into two functions; the serve function creates a new event loop (and a handle to it). We create our listener using `net2` so that we can set a bunch of options on it using the `TcpBuilder` pattern. Specifically, we set `SO_REUSEPORT` on the socket so that under high loads the OS can distribute connections to all threads fairly. We also set a backlog of 128 for the listening socket. We then loop over incoming connections on the listener and, for each, we run our service implementation. Our `start_server` method takes in an integer that corresponds to the number of cores on the host and an address as a string. We then start a loop and run the serve method in new threads. In this case, our `Http` instance will be passed to multiple threads. Thus, we need to wrap it in an **Automatically Reference Counting** (**ARC**) pointer because that guarantees thread safety of the underlying type. Finally, we call `start_server` in our `main` function, using `num_cpus::get` to get the number of cores on the machine.

Benchmarking this in the same way as last time shows these results:

```
$ ab -n 1000 -c 100 http://127.0.0.1:3000/data
Benchmarking 127.0.0.1 (be patient)
Completed 100 requests
Completed 200 requests
Completed 300 requests
Completed 400 requests
Completed 500 requests
Completed 600 requests
Completed 700 requests
Completed 800 requests
Completed 900 requests
Completed 1000 requests
Finished 1000 requests


Server Software:
Server Hostname: 127.0.0.1
Server Port: 3000

Document Path: /data
Document Length: 4 bytes

Concurrency Level: 100
Time taken for tests: 102.724 seconds
Complete requests: 1000
Failed requests: 0
Total transferred: 79000 bytes
HTML transferred: 4000 bytes
Requests per second: 9.73 [#/sec] (mean)
Time per request: 10272.445 [ms] (mean)
Time per request: 102.724 [ms] (mean, across all concurrent requests)
Transfer rate: 0.75 [Kbytes/sec] received

Connection Times (ms)
              min mean[+/-sd] median max
Connect: 0 2 1.0 2 6
Processing: 304 10036 1852.8 10508 10826
Waiting: 106 5482 2989.3 5458 10316
Total: 305 10038 1852.7 10510 10828

Percentage of the requests served within a certain time (ms)
  50% 10510
  66% 10569
  75% 10685
  80% 10686
  90% 10756
  95% 10828
  98% 10828
  99% 10828
 100% 10828 (longest request)
```

This server’s throughput is around double the last one, primarily because it uses threading better. Requests still take just over 100 ms to process, as expected. Note that the actual time taken by this will depend on the hardware and conditions of the machine running this.

Introducing Rocket

Perhaps the most widely known web framework for Rust is Rocket. It started as a one-man project and gradually evolved into a simple, elegant, and fast framework over the last year or so. Rocket focuses a lot on simplicity, something that a lot of Flask users will appreciate. Like Flask uses python decorators to declare routes, Rocket uses custom attributes to the same effect. Unfortunately, this means that Rocket has to make heavy use of nightly-only features. Thus, as of now, Rocket applications can only be built using nightly Rust. However, as more and more things are stabilized (moved to stable Rust), this restriction will eventually go away.

Let us start with a basic example of Rocket, beginning with setting up the project:

```
$ cargo new --bin rocket-simple
```

Our Cargo setup needs to add Rocket components as dependencies, and should look like this:

```
[package]
name = "rocket-simple"
version = "0.1.0"
authors = ["Foo <foo@bar.com>"]

[dependencies]
rocket = "0.3.6"
rocket_codegen = "0.3.6"
```

Let us look at the main file. As we will see, Rocket needs a bit of boilerplate setup:

```
// ch6/rocket-simple/src/main.rs

#![feature(plugin)]
#![plugin(rocket_codegen)]

extern crate rocket;

#[get("/")]
fn blast_off() -> &'static str {
    "Hello, Rocket!"
}

fn main() {
    rocket::ignite().mount("/", routes![blast_off]).launch();
}
```

The function called `blast_off` defines a route, a mapping between an incoming request and an output. In this case, a **GET** request to the `/` route should return a static string. In our main function, we initialize Rocket, add our routes, and call `launch`. Run it using Cargo:

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/rocket-simple`
Configured for development.
    => address: localhost
    => port: 8000
    => log: normal
    => workers: 16
    => secret key: generated
    => limits: forms = 32KiB
    => tls: disabled
Mounting '/':
    => GET /
Rocket has launched from http://localhost:8000
```

In another terminal, if we use curl to hit that endpoint, this is what we should see:

```
$ curl http://localhost:8000
Hello, Rocket!$
```

Experienced Python users will find Rocket similar to the framework called Flask.

Let us now look at a more complex example: writing an API server using Rocket. Our application is a blog server that has the following endpoints:

Endpoint	Method	Purpose
`/posts`	`GET`	Get all posts
`/posts/<id>`	`GET`	Get the post with the given ID
`/posts`	`POST`	Add a new post
`/posts/<id>`	`PATCH`	Edit a post
`/posts/<id>`	`DELETE`	Delete the post with the given ID

For this example, we will use SQLite Version 3 as our database. A real application should use a more scalable database, such as PostgresSQL or MySQL. We will use the diesel crate as our **Object-Relational Mapping** (**ORM**) tool, and r2d2 for connection-pooling to the database. The first step for this is to install the diesel CLI to work with database schema migrations. This can be installed using Cargo.

```
$ cargo install diesel_cli --no-default-features --features sqlite
```

SQLite Version 3 must be installed on the host’s system for this

to work. For more information, visit the following link:
https://www.sqlite.org/download.html

Now we can set up our database using diesel CLI. It will read our migration scripts and create the whole schema. As SQLite is a file-based database, it will also create an empty `db` file, if one does not exist already.

```
$ DATABASE_URL=db.sql diesel migration run
```

Remember that the last command must be run from the directory that has the migrations directory, otherwise it will fail to find the migrations. We will use Cargo to set up the project:

```
$ cargo new --bin rocket-blog
```

We will then add a number of dependencies here; `Cargo.toml` should look like the following code snippet:

```
[package]
authors = ["Foo <foo@bar.com>"]
name = "rocket-blog"
version = "0.1.0"

[dependencies]
rocket = "0.3.5"
rocket_codegen = "0.3.5"
rocket_contrib = "0.3.5"
diesel = { version = "0.16.0", features = ["sqlite"] }
diesel_codegen = { version = "0.16.0", features = ["sqlite"] }
dotenv = "0.10.1"
serde = "1.0.21"
serde_json = "1.0.6"
serde_derive = "1.0.21"
lazy_static = "0.2.11"
r2d2 = "0.7.4"
r2d2-diesel = "0.16.0"
```

This app is a bit more complex than our previous example. This is composed of multiple modules, each doing a specific thing. Here is how the `src` directory looks:

```
$ tree src/
src/
├── db.rs
├── error.rs
├── main.rs
├── models.rs
├── post.rs
└── schema.rs
```

The first step in running this is to set up connection to the database. We will use r2d2 for database pooling; all the `db` setup-related operations will be in `db.rs`. This will look like the following code snippet:

```
// ch6/rocket-blog/src/db.rs

use dotenv::dotenv;
use std::env;
use diesel::sqlite::SqliteConnection;
use r2d2;
use r2d2_diesel::ConnectionManager;
use rocket::request::{Outcome, FromRequest};
use rocket::Outcome::{Success, Failure};
use rocket::Request;
use rocket::http::Status;

// Statically initialize our DB pool
lazy_static! {
    pub static ref DB_POOL:
    r2d2::Pool<ConnectionManager<SqliteConnection>> = {
        dotenv().ok();

        let database_url = env::var("DATABASE_URL").
        expect("DATABASE_URL must be set");
        let config = r2d2::Config::builder()
            .pool_size(32)
            .build();
        let manager = ConnectionManager::
        <SqliteConnection>::new(database_url);
        r2d2::Pool::new(config, manager).expect("Failed to create
        pool.")
    };
}

pub struct DB(r2d2::PooledConnection<ConnectionManager<SqliteConnection>>);

// Make sure DB pointers deref nicely
impl Deref for DB {
    type Target = SqliteConnection;

    fn deref(&self) -> &Self::Target {
        &self.0
    }
}

impl<'a, 'r> FromRequest<'a, 'r> for DB {
    type Error = r2d2::GetTimeout;
    fn from_request(_: &'a Request<'r>) -> Outcome<Self, Self::Error> {
        match DB_POOL.get() {
            Ok(conn) => Success(DB(conn)),
            Err(e) => Failure((Status::InternalServerError, e)),
        }
    }
}
```

Our `DB` struct has an instance of the database pool and returns that on calling the function called `conn`. `FromRequest`. This is a request guard trait from Rocket that makes sure that particular requests can be fulfilled by the handler that matched. In our case, we use it to make sure a new connection is available in the pool, and we return an HTTP 500 error if that is not the case. Now this trait will be used for all incoming requests throughout the life of the program. Thus, for this to work correctly, the reference to the database pool must live throughout the program and not the local scope. We use the `lazy_static!` crate to make sure the constant `DB_POOL` is initialized only once and lives throughout the life of the program. In the macro, we set up `dotenv`, which will be used later to parse the database location and also the connection pool with a size of 32 connections. We also implement the `Deref` trait for our database wrapper so that `&*DB` is transparently translated to a `&SqliteConnection`.

The next step is to set up the database schema in code. Luckily, `diesel` makes that very easy, as it can read the database schema and generate Rust code to represent that accordingly. The generated Rust code is put in a module corresponding to the name of the file it is in (in this case, the module will be called `schema`). We use the `dotenv` crate to pass this information to a `diesel` macro. This is done in the file `schema.rs`:

```
// ch6/rocket-blog/src/schema.rs

infer_schema!("dotenv:DATABASE_URL");
```

Note that once the new macro system is functional, this call will use the `dotenv!` macro. We can then use the generated schema to build our models. This is done in `models.rs`. This file will look like the following code snippet:

```
// ch6/rocket-blog/src/models.rs

use super::schema::posts;
use rocket::{Request, Data};
use rocket::data::{self, FromData};
use rocket::http::Status;
use rocket::Outcome::*;
use serde_json;

// Represents a blog post in the database
#[derive(Queryable)]
#[derive(Serialize,Deserialize)]
pub struct Post {
    pub id: i32,
    pub title: String,
    pub body: String,
    pub pinned: bool,
}

// Represents a blog post as incoming request data
#[derive(Insertable, Deserialize, AsChangeset)]
#[table_name="posts"]
pub struct PostData {
    pub title: String,
    pub body: String,
    pub pinned: bool,
}

// This enables using PostData from incoming request data
impl FromData for PostData {
    type Error = String;

    #[allow(unused_variables)]
    fn from_data(req: &Request, data: Data) -> data::Outcome<Self,
    String> {
        let reader = data.open();
        match serde_json::from_reader(reader).map(|val| val) {
            Ok(value) => Success(value),
            Err(e) => Failure((Status::BadRequest, e.to_string())),
        }
    }
}
```

We have two major structures here: the `Post` structure represents a blog post in the database, and the `PostData` structure represents a blog post as seen in an incoming create request. As a `PostData` has not been saved in the database yet, it does not have an ID. Diesel necessitates that all types that can be queried should implement the `Queryable` trait, which is done automatically using `#[derive(Queryable)]`. We also enable serialization and deserialization using serde, as this will be passed as JSON to the API. In contrast, the `PostData` struct does not derive `Queryable`; it derives some other traits. The `Insertable` trait indicates that this struct can be used to insert a row of data in a table (with the specified table name). Because we will only need to deserialize this struct from an incoming request, we only implement `Deserialize` on it. Lastly, the `AsChangeSet` trait enables this struct to be used to update records in the database.

The `FromData` trait is from Rocket, which is used to validate incoming data, making sure it parses into JSON correctly. This is in relation to a feature called data guards. When Rocket finds a suitable handler for an incoming request, it calls the data guard of the data type specified in the request handler on the incoming data. The route is actually invoked only if the data guard succeeds. These guards are implemented using the `FromData` trait. In our case, the implementation tries to parse the input as a JSON (using SerDe). In the success case, we return the JSON for further processing, or we return a `Status::BadRequest`, which sends back an HTTP 400 error.

The only database-related thing required now is the model. This will define a number of convenience methods that can be used to manipulate records using diesel. The file `post.rs` hosts these, and it looks like the following code snippet:

```
// ch6/rocket-blog/src/post.rs

use diesel::result::Error;
use diesel;
use diesel::sqlite::SqliteConnection;
use models::*;
use diesel::prelude::*;
use schema::posts;

// Returns post with given id
pub fn get_post(conn: &SqliteConnection, id: i32) -> Result<Post, Error> {
    posts::table
        .find(id)
        .first::<Post>(conn)
}

// Returns all posts
pub fn get_posts(conn: &SqliteConnection) -> Result<Vec<Post>, Error> {
    posts::table
        .load::<Post>(conn)
}

// Creates a post with the given PostData, assigns a ID
pub fn create_post(conn: &SqliteConnection, post: PostData) -> bool {
    diesel::insert(&post)
        .into(posts::table).execute(conn).is_ok()
}

// Deletes a post with the given ID
```

```
pub fn delete_post(conn: &SqliteConnection, id: i32) -> Result<usize, Error> {
    diesel::delete(posts::table.find(id))
        .execute(conn)
}

// Updates a post with the given ID and PostData
pub fn update_post(conn: &SqliteConnection, id: i32, updated_post: PostData) ->
bool {
    diesel::update(posts::table
        .find(id))
        .set(&updated_post).execute(conn).is_ok()
}
```

The function names are very self-explanatory. We make abundant use of a number of diesel APIs that help us to interact with the database without writing SQL directly. All of these functions take in a reference to the database connection. The `get_post` function takes in an additional post ID that it uses on the posts table to look up posts using the `find` method, and then returns the first result as an instance of `Post`. The `get_posts` function is similar, except it returns all records in the posts table as a vector of `Post` instances. `create_post` takes in a reference to `PostData` and inserts that record into the database. This function returns a `bool` indicating success or failure. `delete_post` takes in a post ID and tries to delete it in the database. `update_post` again takes in a reference to `PostData` and a post ID. It then tries to replace the post with the given ID with the new `PostData`.

Let us move on to defining errors for our API. This will be located in the file called `error.rs`. As will see here, the errors will need to implement a number of traits to be used seamlessly in Rocket and Diesel.

```
// ch6/rocket-blog/src/error.rs

use std::error::Error;
use std::convert::From;
use std::fmt;
use diesel::result::Error as DieselError;
use rocket::http::Status;
use rocket::response::{Response, Responder};
use rocket::Request;

#[derive(Debug)]
pub enum ApiError {
    NotFound,
    InternalServerError,
}

impl fmt::Display for ApiError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match *self {
            ApiError::NotFound => f.write_str("NotFound"),
            ApiError::InternalServerError => f.write_str("InternalServerError"),
        }
    }
}

// Translates a database error to an API error
impl From<DieselError> for ApiError {
    fn from(e: DieselError) -> Self {
        match e {
            DieselError::NotFound => ApiError::NotFound,
            _ => ApiError::InternalServerError,
        }
    }
}

impl Error for ApiError {
    fn description(&self) -> &str {
        match *self {
            ApiError::NotFound => "Record not found",
            ApiError::InternalServerError => "Internal server error",
        }
    }
}

// This enables sending back an API error from a route
impl<'r> Responder<'r> for ApiError {
    fn respond_to(self, _request: &Request) -> Result<Response<'r>, Status> {
        match self {
            ApiError::NotFound => Err(Status::NotFound),
            _ => Err(Status::InternalServerError),
        }
    }
}
```

Our error is an `enum` called `ApiError`; for simplicity's sake, we will only return an object-not-found error and a catch-all internal server error. As we have seen in previous chapters, to declare an error in Rust, we will need to implement `fmt::Display` and `std::error::Error` on that type. We also implement `From<DieselError>` for our type so that a failure in the database lookup can be reported appropriately. The final trait we need to implement is `Responder` from Rocket, which enables us to use this as the return type of a request handler.

Having done all the groundwork, the last part of the system is our main file that will run when invoked with Cargo. It should look like the following code snippet:

```
// ch6/rocket-blog/src/main.rs

#![feature(plugin)]
#![plugin(rocket_codegen)]
extern crate rocket;
#[macro_use]
extern crate diesel;
#[macro_use]
extern crate diesel_codegen;
extern crate dotenv;
extern crate serde_json;
#[macro_use]
extern crate lazy_static;
extern crate rocket_contrib;
#[macro_use]
extern crate serde_derive;
extern crate r2d2;
extern crate r2d2_diesel;

mod schema;
mod db;
mod post;
mod models;
mod error;
```

```
use db::DB;
use post::{get_posts, get_post, create_post, delete_post, update_post};
use models::*;
use rocket_contrib::Json;
use rocket::response::status::{Created, NoContent};
use rocket::Rocket;
use error::ApiError;

#[get("/posts", format = "application/json")]
fn posts_get(db: DB) -> Result<Json<Vec<Post>>, ApiError> {
    let posts = get_posts(&db)?;
    Ok(Json(posts))
}

#[get("/posts/<id>", format = "application/json")]
fn post_get(db: DB, id: i32) -> Result<Json<Post>, ApiError> {
    let post = get_post(&db, id)?;
    Ok(Json(post))
}

#[post("/posts", format = "application/json", data = "<post>")]
fn post_create(db: DB, post: PostData) -> Result<Created<String>, ApiError> {
    let post = create_post(&db, post);
    let url = format!("/post/{}", post);
    Ok(Created(url, Some("Done".to_string())))
}

#[patch("/posts/<id>", format = "application/json", data = "<post>")]
fn post_edit(db: DB, id: i32, post: PostData) -> Result<Json<bool>, ApiError> {
    let post = update_post(&db, id, post);
    Ok(Json(post))
}

#[delete("/posts/<id>")]
fn post_delete(db: DB, id: i32) -> Result<NoContent, ApiError> {
    delete_post(&db, id)?;
    Ok(NoContent)
}

// Helper method to setup a rocket instance
fn rocket() -> Rocket {
    rocket::ignite().mount("/", routes![post_create, posts_get, post_delete,
post_edit, post_get])
}

fn main() {
        rocket().launch();
}
```

The most important things here are the route handlers. These are just regular functions with special attributes that determine the path, the format, and the arguments. Also, notice the use of instances of `DB` as request guards in the handlers. We have a helper function called `rocket` that sets up everything, and the `main` function just calls the `ignite` method to start the server. When `rocket` sees an incoming request, this is how a response is generated:

1. It goes over the list of all handlers and finds one that matches the HTTP method, type, and format. If one is found, it ensures that the handler's parameters can be derived from the data in the request using `FormData`. This process continues till a handler works or all handlers have been exhausted. In the latter case, a 404 error is returned.

2. The handler function then receives a copy of the data parsed into the given data type. After it has done processing, it must use the `Responder` implementation to convert the output to a valid return type.
3. Finally, `Rocket` sends back the response to the client.

Having set up everything, running the server is very easy:

```
$ DATABASE_URL=db.sql cargo run
```

You can use `curl` to interact with the server, as shown in the following code snippet. We use `jq` for formatting the JSON nicely by piping curl's output to `jq`:

```
$ curl -X POST -H "Content-Type: application/json" -d '{"title": "Hello Rust!",
"body": "Rust is awesome!!", "pinned": true}' http://localhost:8000/posts
Done
$ curl http://localhost:8000/posts | jq
  % Total % Received % Xferd Average Speed Time Time Time Current
                                 Dload Upload Total Spent Left Speed
100 130 100 130 0 0 130 0 0:00:01 --:--:-- 0:00:01 8125
[
  {
    "id": 1,
    "title": "test",
    "body": "test body",
    "pinned": true
  },
  {
    "id": 2,
    "title": "Hello Rust!",
    "body": "Rust is awesome!!",
    "pinned": true
  }
]
```

For comparison, here is how a load testing session looks:

```
$ ab -n 10000 -c 100 http://localhost:8000/posts
Benchmarking localhost (be patient)
Completed 1000 requests
Completed 2000 requests
Completed 3000 requests
Completed 4000 requests
Completed 5000 requests
Completed 6000 requests
Completed 7000 requests
Completed 8000 requests
Completed 9000 requests
Completed 10000 requests
Finished 10000 requests


Server Software: Rocket
Server Hostname: localhost
Server Port: 8000

Document Path: /posts
Document Length: 130 bytes

Concurrency Level: 100
```

```
Time taken for tests: 2.110 seconds
Complete requests: 10000
Failed requests: 0
Total transferred: 2740000 bytes
HTML transferred: 1300000 bytes
Requests per second: 4740.00 [#/sec] (mean)
Time per request: 21.097 [ms] (mean)
Time per request: 0.211 [ms] (mean, across all concurrent requests)
Transfer rate: 1268.32 [Kbytes/sec] received

Connection Times (ms)
              min mean[+/-sd] median max
Connect: 0 0 0.4 0 4
Processing: 3 21 20.3 19 229
Waiting: 2 20 19.8 18 228
Total: 7 21 20.3 19 229

Percentage of the requests served within a certain time (ms)
  50% 19
  66% 19
  75% 20
  80% 20
  90% 21
  95% 22
  98% 26
  99% 214
 100% 229 (longest request)
```

Now, to be fair to our earlier servers, we had a delay of 100 ms there. In this case, each request takes around 21 ms on average. So, hypothetically, if each request took 100 ms, we would have one fifth of the throughput. That comes up to be around 950 requests per second—way faster than our earlier servers!

Now, obviously, an HTTP server cannot be all about REST endpoints. It must be able to serve static and dynamic content as well. For that, Rocket has a bunch of features to be able to generate HTML. Let us look at an example, which is a simple web page that takes in a name as a URL parameter and outputs that. The page also counts the total number of visits and displays that as well. The Cargo setup for this project is simple: we just run the following command:

```
$ cargo new --bin rocket-templates
```

This time, we will just need Rocket in our `Cargo.toml` file:

```
[package]
authors = ["Foo<foo@bar.com>"]
name = "rocket-templates"
version = "0.1.0"

[dependencies]
rocket = "0.3.5"
rocket_codegen = "0.3.5"

[dependencies.rocket_contrib]
version = "*"
default-features = false
features = ["tera_templates"]
```

Our web page will be generated from a template. We will use a templating engine called Tera, which is inspired by Jinja2 and is written in Rust. Rocket supports templates in a different crate called `rocket_contrib`, which we will pull in with required features. Our template is pretty simple and should look like this:

```
// ch6/rocket-templates/templates/webpage.html.tera

<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Rocket template demo</title>
  </head>
  <body>
    <h1>Hi {{name}}, you are visitor number {{ visitor_number }}</h1>
  </body>
</html>
```

Note that the template has to be in a directory called `templates` in the project root, otherwise `Rocket` will not be able to find it. In this case, the template is pretty simple. It needs to be a complete HTML page, as we intend to display it in a browser. We use two temporary variables, `name` and `visitor_number`, that will be replaced during execution. Our main file will look like the following code snippet:

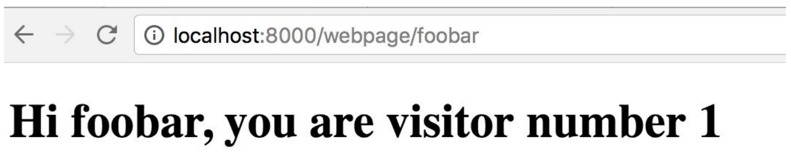
```
// ch6/rocket-templates/src/main.rs

#![feature(plugin)]
#![plugin(rocket_codegen)]

extern crate rocket_contrib;
extern crate rocket;

use rocket_contrib::Template;
use rocket::{Rocket, State};
use std::collections::HashMap;
use std::sync::atomic::{AtomicUsize, Ordering};

struct VisitorCounter {
    visitor_number: AtomicUsize,
}

#[get("/webpage/<name>")]
fn webpage(name: String, visitor: State<VisitorCounter>) -> Template {
    let mut context = HashMap::new();
    context.insert("name", name);
    let current = visitor.visitor_number.fetch_add(1, Ordering::SeqCst);
    context.insert("visitor_number", current.to_string());
    Template::render("webpage", &context)
}

fn rocket() -> Rocket {
    rocket::ignite()
        .manage(VisitorCounter { visitor_number: AtomicUsize::new(1) })
        .mount("/", routes![webpage])
        .attach(Template::fairing())
}

fn main() {
    rocket().launch();
}
```

Our setup is pretty much the same as last time; the only difference being, we have used the template fairing, which is like a middleware in Rocket. To use this, we needed to call `attach(Template::fairing())` on the rocket instance. Another difference is the use of managed state, which we use to manage our counter automatically. This is achieved by calling `manage` on the instance and passing an initial state for the managed object. Our counter is a struct, having only one element that holds the current count. Now our counter will be shared between multiple threads, all running rocket instances. To make the counter thread safe, we have used the primitive `AtomicUsize`, which guarantees thread safety. In our route, we match on the `GET` verb and we take in a name as a URL parameter. To render our template, we will need to build a context and populate it. Whenever an incoming request matches this route, we can insert the name in our context. We then call `fetch_add` on the underlying counter. This method increments the counter and returns the previous value, which we store in the context against a key called `visitor_number`. Once done, we can render our template, which is returned to the client. Note the use of `Ordering::SeqCst` in `fetch_add`, which guarantees a sequentially consistent view of the counter across all competing threads. Also note that the names of the keys in the context have to match the temporary variables used in the template, otherwise rendering will fail.

Running this is easy; we just need to use `cargo run`. This is what we see on the CLI:

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/rocket-templates`
Configured for development.
    => address: localhost
    => port: 8000
    => log: normal
    => workers: 16
    => secret key: generated
    => limits: forms = 32KiB
    => tls: disabled
Mounting '/':
    => GET /webpage/<name>
Rocket has launched from http://localhost:8000
```

We can then use a web browser to access the page to see something like the following screenshot:


localhost:8000/webpage/foobar

Hi foobar, you are visitor number 1

Notice that the counter resets when the `Rocket` instance is restarted. A real-world application may decide to persist such a metric in a database so that it is not lost between restarts. This also works with `curl`, which just dumps the raw HTML in the console:

```
$ curl http://localhost:8000/webpage/foo
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Rocket template demo</title>
  </head>
  <body>
    <h1>Hi foo, you are visitor number 2</h1>
  </body>
</html>
```

The last experiment with this piece of code is performance analysis. As always, we will fire up apache bench and point it to the endpoint. This is what one run shows:

```
$ ab -n 10000 -c 100 http://localhost:8000/webpage/foobar
Benchmarking localhost (be patient)
Completed 1000 requests
Completed 2000 requests
Completed 3000 requests
Completed 4000 requests
Completed 5000 requests
Completed 6000 requests
Completed 7000 requests
Completed 8000 requests
Completed 9000 requests
Completed 10000 requests
Finished 10000 requests


Server Software: Rocket
Server Hostname: localhost
Server Port: 8000

Document Path: /webpage/foobar
Document Length: 191 bytes

Concurrency Level: 100
Time taken for tests: 2.305 seconds
Complete requests: 10000
Failed requests: 0
Total transferred: 3430000 bytes
HTML transferred: 1910000 bytes
Requests per second: 4337.53 [#/sec] (mean)
Time per request: 23.055 [ms] (mean)
Time per request: 0.231 [ms] (mean, across all concurrent requests)
Transfer rate: 1452.90 [Kbytes/sec] received

Connection Times (ms)
              min mean[+/-sd] median max
Connect: 0 0 2.8 0 200
Processing: 3 23 18.6 21 215
Waiting: 3 22 18.0 20 214
Total: 7 23 18.8 21 215

Percentage of the requests served within a certain time (ms)
  50% 21
  66% 21
  75% 22
  80% 22
  90% 24
  95% 25
  98% 28
  99% 202
 100% 215 (longest request)
```

Performance in this case is comparable to last time, as measured in requests per second. This one is slightly slower, as it has to increment the counter and render the template each time. This is also reflected in mean time per request, which increased by 2 ms.

Rocket has a ton of other features, from cookies to streaming data. It also supports SSL out of the box by reading a special config file that can be placed in the root directory of the application. However, those advanced features are outside the scope of this book.

Introducing reqwest

So far, we have only talked about writing servers and used `curl` to access those. Sometimes, programmatically accessing a server becomes a necessity. In this section, we will discuss the `reqwest` crate and look at how to use it; this borrows heavily from the requests library in Python. Thus, it is very easy to set up and use, starting first with the project setup:

```
$ cargo new --bin reqwest-example
```

The next step for our demo is to include our dependencies. Our Cargo config should look like this:

```
[package]
name = "reqwest-example"
version = "0.1.0"
authors = ["Foo<foo@bar.com>"]

[dependencies]
reqwest = "0.8.1"
serde_json = "1.0.6"
serde = "1.0.21"
serde_derive = "1.0.21"
```

Here, we will use Serde to serialize and deserialize our data to JSON. Very conveniently, we will use the `Rocket` server we wrote in the last section. Our main file will look like this:

```
// ch6/reqwest-example/src/main.rs

extern crate serde_json;
#[macro_use]
extern crate serde_derive;
extern crate reqwest;

#[derive(Debug,Serialize, Deserialize)]
struct Post {
    title: String,
    body: String,
    pinned: bool,
}

fn main() {
    let url = "http://localhost:8000/posts";
    let post: Post = Post {title: "Testing this".to_string(), body: "Try to write
something".to_string(), pinned: true};
    let client = reqwest::Client::new();

    // Creates a new blog post using the synchronous client
    let res = client.post(url)
            .json(&post)
            .send()
            .unwrap();
    println!("Got back: {}", res.status());

    // Retrieves all blog posts using the synchronous client
    let mut posts = client.get(url).send().unwrap();
    let json: Vec<Post> = posts.json().unwrap();
    for post in json {
        println!("{:?}", post);
    }
}
```

We start with a struct to represent our blog post, which is exactly the same as the one in the last section. In our `main` function, we create an instance of our client and, using the builder pattern, pass on our post as a JSON to it. Finally, we call `send` on it and print out the return status. Make sure to change the `url` to point to the location where `Rocket` is running. We then issue a `GET` request on the same endpoint. We deserialize the response to a list of `Post` objects and print those out in a loop. Internally, reqwest uses SerDe to serialize and deserialize to/from JSON, making the API very user-friendly.

Here is an example session running the preceding code. In our server, we already had two existing entries and, in our code, we added one more. Then, we got back all three, which are printed here. Take a look at the following code snippet:

```
$ cargo run
   Compiling reqwest-example v0.1.0 (file:///src/ch6/reqwest-example)
    Finished dev [unoptimized + debuginfo] target(s) in 1.94 secs
     Running `target/debug/reqwest-example`
Got back: 201 Created
Post { title: "test", body: "test body", pinned: true }
Post { title: "Hello Rust!", body: "Rust is awesome!!", pinned: true }
Post { title: "Testing this", body: "Try to write something", pinned: true }
```

Recently, `reqwest` added asynchronous programming support using tokio. All those APIs are located in `reqwest::unstable` and, as the name suggests, those are not stable yet. Let’s look at using the asynchronous client for the same purpose. In this case, we will use the futures and tokio crates, thus we will need to include those in our cargo manifest, which will look like this:

```
[package]
name = "reqwest-async"
version = "0.1.0"
authors = ["Foo <foo@bar.com>"]

[dependencies]
serde_json = "1.0.6"
serde = "1.0.21"
serde_derive = "1.0.21"
futures = "0.1.17"
tokio-core = "0.1.10"

[dependencies.reqwest]
version = "0.8.1"
features = ["unstable"]
```

We will need to activate the feature called unstable in reqwest. Our main file will look like the following code snippet:

```
// ch6/reqwest-async/src/main.rs

extern crate serde_json;
```

```
#[macro_use]
extern crate serde_derive;
extern crate reqwest;
extern crate futures;
extern crate tokio_core;

use futures::Future;
use tokio_core::reactor::Core;
use reqwest::unstable::async::{Client, Decoder};
use std::mem;
use std::io::{self, Cursor};
use futures::Stream;

#[derive(Debug, Serialize, Deserialize)]
struct Post {
    title: String,
    body: String,
    pinned: bool,
}

fn main() {
    let mut core = Core::new().expect("Could not create core");
    let url = "http://localhost:8000/posts";
    let post: Post = Post {
        title: "Testing this".to_string(),
        body: "Try to write something".to_string(),
        pinned: true,
    };
    let client = Client::new(&core.handle());

    // Creates a new post using the async client
    let res = client.post(url).json(&post).send().and_then(|res| {
        println!("{}", res.status());
        Ok(())
    });
    core.run(res).unwrap();

    // Gets all current blog posts using the async client
    let posts = client
        .get(url)
        .send()
        .and_then(|mut res| {
            println!("{}", res.status());
            let body = mem::replace(res.body_mut(), Decoder::empty());
            body.concat2().map_err(Into::into)
        })
        .and_then(|body| {
            let mut body = Cursor::new(body);
            let mut writer: Vec<u8> = vec![];
            io::copy(&mut body, &mut writer).unwrap();
            let posts: Vec<Post> =
serde_json::from_str(std::str::from_utf8(&writer).unwrap())
                .unwrap();
            for post in posts {
                println!("{:?}", post);
            }
            Ok(())
        });
    core.run(posts).unwrap();
}
```

Admittedly, this is way more convoluted than the previous version! Some scaffolding for the `Post` struct is the same, and we pull in all the extra libraries we need. In our `main` function, we create a tokio core and then an asynchronous client based on that core. We chain the `json` and `send` methods like last time. Things diverge from there; for the asynchronous client, the `send` call returns a future. Once that future has resolved, the `and_then` call executes another future based on the first one. In here, we print out the status that we got back and resolve the future by returning an `Ok(())`. Finally, we run our future on the core.

Getting data back from the endpoint is a bit more involved, because we have to deal with returned data as well. Here, we chain calls to `get` and `send`. We then chain another future that collects the response body. The second future is then chained to another one that consumes that body and copies it over to a `Vec<u8>` named `writer`. We then convert the vector to a `str` using `std::str::from_utf8`. The `str` is then passed to `serde_json::from_str`, which tries to deserialize it into a vector of `Post` objects that we can then print out by iterating over those. At the end, we resolve the chain of futures by returning an `Ok(())`. On running, this behaves exactly like the last one.

Summary

In this chapter, we covered a number of crates that help with dealing with HTTP-based REST endpoints in Rust, using Hyper and Rocket. We also looked at programmatically accessing these endpoints, using request, which is largely based on Hyper. These crates are at various stages of development. As we saw, Rocket can only run on nightly, because it uses a bunch of features that are not stable yet. We also glossed over tokio, which powers both Hyper and Rocket.

Now, tokio, being the defacto asynchronous programming library in Rust, deserves all the attention it can get. So, we will discuss the tokio stack in detail in the next chapter.

Asynchronous Network Programming Using Tokio

In a sequential programming model, code is always executed in the order dictated by the semantics of the programming language. Thus, if one operation blocks for some reason (waiting for a resource, and so forth), the whole execution blocks and can only move forward once that operation has completed. This often leads to poor utilization of resources, because the main thread will be busy waiting on one operation. In GUI apps, this also leads to poor user interactivity, because the main thread, which is responsible for managing the GUI, is busy waiting for something else. This is a major problem in our specific case of network programming, as we often need to wait for data to be available on a socket. In the past, we worked around these issues using multiple threads. In that model, we delegated a costly operation to a background thread, making the main thread free for user interaction, or some other task. In contrast, an asynchronous model of programming dictates that no operation should ever block. Instead, there should be a mechanism to check whether they have completed from the main thread. But how do we achieve this? A simple way would be to run each operation in its own thread, and then to join on all of those threads. In practice, this is troublesome owing to the large number of potential threads and coordination between them.

Rust provides a few crates that support asynchronous programming using a futures-based, event loop-driven model. We will study that in detail in this chapter. Here are the topics we will cover here:

- Futures abstraction in Rust
- Asynchronous programming using the tokio stack

Looking into the Future

The backbone of Rust's asynchronous programming story is the futures crate. This crate provides a construct called a *future*. This is essentially a placeholder for the result of an operation. As you would expect, the result of an operation can be in one of two states—either the operation is still in progress and the result is not available yet, or the operation has finished and the result is available. Note that in the second case, there might have been an error, making the result immaterial.

The library provides a trait called `Future` (among other things),which any type can implement to be able to act like a future. This is how the trait looks:

```
trait Future {
    type Item;
    type Error;
    fn poll(&mut self) -> Poll<Self::Item, Self::Error>;
    ...
}
```

Here, `Item` refers to the type of the returned result on successful completion of the operation, and `Error` is the type that is returned if the operation fails. An implementation must specify those and also implement the poll method that gets the current state of the computation. If it has already finished, the result will be returned. If not, the future will register that the current task is interested in the outcome of the given operation. This function returns a `Poll`, which looks like this:

```
type Poll<T, E> = Result<Async<T>, E>;
```

A `Poll` is typed to a result of another type called `Async` (and the given error type), which is defined next.

```
pub enum Async<T> {
    Ready(T),
    NotReady,
}
```

`Async`, in turn, is an enum that can either be in `Ready(T)` or `NotReady`. These last two states correspond to the state of the operation. Thus, the poll function can return three possible states:

- `Ok(Async::Ready(result))` when the operation has completed successfully and the result is in the inner variable called `result`.
- `Ok(Async::NotReady)` when the operation has not completed yet and a result is not available. Note that this does not indicate an error condition.

- `Err(e)` when the operation ran into an error. No result is available in this case.

It is easy to note that a `Future` is essentially a `Result` that might still be running something to actually produce that `Result`. If one removes the case that the `Result` might not be ready at any point in time, the only two options we are left with are the `Ok` and the `Err` cases, which exactly correspond to a `Result`.

Thus, a `Future` can represent anything that takes a non-trivial amount of time to complete. This can be a networking event, a disk read, and so on. Now, the most common question at this point is: how do we return a future from a given function? There are a few ways of doing that. Let us look at an example here. The project setup is the same as it always is.

```
$ cargo new --bin futures-example
```

We will need to add some libraries in our Cargo config, which will look like this:

```
[package]
name = "futures-example"
version = "0.1.0"
authors = ["Foo<foo@bar.com>"]

[dependencies]
futures = "0.1.17"
futures-cpupool = "0.1.7"
```

In our main file, we set up everything as usual. We are interested in finding out whether a given integer is a prime or not, and this will represent the part of our operation that takes some time to complete. We have two functions, doing exactly that. These two use two different styles of returning futures, as we will see later. In practice, the naive way of primality testing did not turn out to be slow enough to be a good example. Thus, we had to sleep for a random time to simulate slowness.

```
// ch7/futures-example/src/main.rs

#![feature(conservative_impl_trait)]
extern crate futures;
extern crate futures_cpupool;

use std::io;
use futures::Future;
use futures_cpupool::CpuPool;

// This implementation returns a boxed future
fn check_prime_boxed(n: u64) -> Box<Future<Item = bool, Error = io::Error>> {
    for i in 2..n {
        if n % i == 0 { return Box::new(futures::future::ok(false)); }
    }
    Box::new(futures::future::ok(true))
}

// This returns a future using impl trait
```

```
fn check_prime_impl_trait(n: u64) -> impl Future<Item = bool, Error = io::Error> {
        for i in 2..n {
        if n % i == 0 { return futures::future::ok(false); }
    }
    futures::future::ok(true)
}

// This does not return a future
fn check_prime(n: u64) -> bool {
    for i in 2..n {
        if n % i == 0 { return false }
    }
    true
}

fn main() {
    let input: u64 = 58466453;
    println!("Right before first call");
    let res_one = check_prime_boxed(input);
    println!("Called check_prime_boxed");
    let res_two = check_prime_impl_trait(input);
    println!("Called check_prime_impl_trait");
    println!("Results are {} and {}", res_one.wait().unwrap(),
    res_two.wait().unwrap());

    let thread_pool = CpuPool::new(4);
    let res_three = thread_pool.spawn_fn(move || {
        let temp = check_prime(input);
        let result: Result<bool, ()> = Ok(temp);
        result
    });
    println!("Called check_prime in another thread");
    println!("Result from the last call: {}", res_three.wait().unwrap());
}
```

There are a few major ways of returning futures. The first one is using trait objects, as done in `check_prime_boxed`. Now, `Box` is a pointer type pointing to an object on the heap. It is a managed pointer in the sense that the object will be automatically cleaned up when it goes out of scope. The return type of the function is a trait object, which can represent any future that has its `Item` set to bool and `Error` set to `io:Error`. Thus, this represents dynamic dispatch. The second way of returning a future is using the `impl` trait feature. In the case of `check_prime_impl_trait`, that is what we do. We say that the function returns a type that implements `Future<Item=bool, Error=io::Error>`, and as any type that implements the `Future` trait is a future, our function is returning a future. Note that in this case, we do not need to box before returning the result. Thus, an advantage of this approach is that no allocation is necessary for returning the future. Both of our functions use the `future::ok` function to signal that our computation has finished successfully with the given result. Another option is to not actually return a future and to use the futures-based thread pool crate to do the heavy lifting toward creating a future and managing it. This is the case with `check_prime` that just returns a `bool`. In our main function, we set up a thread pool using the futures-`cpupool` crate, and we run the last function in that pool. We get back a future on which we can call `wait` to get the result. A totally different option for achieving the same goal is to return a custom type that implements the `Future` trait. This one is the least ergonomic, as it involves writing some extra code, but it is the most flexible approach.

The `impl` trait is not a stable feature yet. Thus, `check_prime_impl_trait` will only work on nightly Rust.

Having constructed a future, the next goal is to execute it. There are three ways of doing this:

- In the current thread: This will end up blocking the current thread till the future has finished executing. In our previous example, `res_one` and `res_two` are executed on the main thread, blocking user interaction.
- In a thread pool: This is the case with `res_three`, which is executed in a thread pool named `thread_pool`. Thus, in this case, the calling thread is free to move on with its own processing.
- In an event loop: In some cases, neither of the above is possible. The only option then is to execute futures in an event loop. Conveniently, the tokio-core crate provides futures-friendly APIs to use event loops. We will look deeper into this model in the next section.

In our main function, we call the first two functions in the main thread. Thus, they will block execution of the main thread. The last one, however, is run on a different thread. In that case, the main thread is immediately free to print out that `check_prime` has been called. It blocks again on calling `wait` on the future. Note that the futures are lazily evaluated in all cases. When we run this, we should see the following:

```
$ cargo run
   Compiling futures-example v0.1.0 (file:///src/ch7/futures-example)
    Finished dev [unoptimized + debuginfo] target(s) in 0.77 secs
     Running `target/debug/futures-example`
Right before first call
Called check_prime_boxed
Called check_prime_impl_trait
Results are true and true
Called check_prime in another thread
Result from the last call: true
```

What sets futures apart from regular threads is that they can be chained ergonomically. This is like saying, *download the web page and then parse the html and then extract a given word*. Each of these steps in series is a future, and the next one cannot start unless the first one has finished. The whole operation is a `Future` as well, being made up of a number of constituent futures. When this larger future is being executed, it is called a task. The crate provides a number of APIs for interacting with tasks in the `futures::task` namespace. The library provides a number of functions to work with futures in this manner. When a given type implements the `Future` trait (implements the `poll` method), the compiler can provide implementations for all of these combinators. Let us look at an example of implementing a timeout functionality using chaining. We will use the tokio-timer crate for the timeout future and, in our code, we have two competing functions that sleep for a random amount of time and then return a fixed string to the caller. We will dispatch all these simultaneously and, if we get back the string corresponding to the first function, we declare that it has won. Similarly, this applies for the second one. In case we do not get back either, we know that the timeout future has triggered. Let's start with the project setup:

```
$ cargo new --bin futures-chaining
```

We then add our dependencies in our `Cargo.toml`

```
[package]
name = "futures-chaining"
version = "0.1.0"
authors = ["Foo <foo@bar.com>"]

[dependencies]
tokio-timer = "0.1.2"
futures = "0.1.17"
futures-cpupool = "0.1.7"
rand = "0.3.18"
```

Like last time, we use a thread pool to execute our futures using the futures-`cpupool` crate. Lets us look at the code:

```
// ch7/futures-chaining/src/main.rs

extern crate futures;
extern crate futures_cpupool;
extern crate tokio_timer;
extern crate rand;

use futures::future::select_ok;
use std::time::Duration;

use futures::Future;
use futures_cpupool::CpuPool;
use tokio_timer::Timer;
use std::thread;
use rand::{thread_rng, Rng};

// First player, identified by the string "player_one"
fn player_one() -> &'static str {
    let d = thread_rng().gen_range::<u64>(1, 5);
    thread::sleep(Duration::from_secs(d));
    "player_one"
}

// Second player, identified by the string "player_two"
fn player_two() -> &'static str {
    let d = thread_rng().gen_range::<u64>(1, 5);
    thread::sleep(Duration::from_secs(d));
    "player_two"
}

fn main() {
    let pool = CpuPool::new_num_cpus();
    let timer = Timer::default();

    // Defining the timeout future
    let timeout = timer.sleep(Duration::from_secs(3))
        .then(|_| Err(()));

    // Running the first player in the pool
    let one = pool.spawn_fn(|| {
        Ok(player_one())
    });

    // Running second player in the pool
    let two = pool.spawn_fn(|| {
        Ok(player_two())
    });

    let tasks = vec![one, two];
    // Combining the players with the timeout future
    // and filtering out result
    let winner = select_ok(tasks).select(timeout).map(|(result, _)|
    result);
    let result = winner.wait().ok();
    match result {
        Some(("player_one", _)) => println!("Player one won"),
        Some(("player_two", _)) => println!("Player two won"),
        Some((_, _)) | None => println!("Timed out"),
    }
}
```

Our two players are very similar; both of them generate a random number between `1` and `5` and sleep for that amount of seconds. After that, they return a fixed string corresponding to their names. We will later use these strings to identify them uniquely. In our main function, we initialize the thread pool and the timer. We use the combinator on the timer to return a future that errors out after `3` seconds. We then spawn the two players in the thread pool and return `Result`s from those as futures. Note that those functions are not really running at this point, because futures are lazily evaluated. We then put those futures in a list and use the `select_ok` combinator to run those in parallel. This function takes in a iterable of futures and selects the first successful future; the only restriction here is that all the futures passed to this function should be of the same type. Thus, we cannot pass the timeout future here. We chain the result of `select_ok` to the timeout future using the `select` combinator that takes two futures and waits for either to finish executing. The resultant future will have the one that has finished and the one that hasn't. We then use the `map` combinator to discard the second part. Finally, we block on our futures and signal the end of the chain using `ok()`. We can then compare the result with the known strings to determine which future has won, and print out messages accordingly.

This is how a few runs will look. As our timeout is smaller than the maximum sleep period of either of the two functions, we should see a few timeouts. Whenever a function chooses a time less than the timeout, it gets a shot at winning.

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/futures-chaining`
Player two won
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/futures-chaining`
Player one won
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/futures-chaining`
Timed out
```

Working with streams and sinks

The futures crate provides another useful abstraction for a lazily evaluated series of events, called `Stream`. If `Future` corresponds to `Result`, a `Stream` corresponds to `Iterator`. Semantically, they are very similar to futures, and they look like this:

```
trait Stream {
    type Item;
    type Error;
    fn poll(& mut self) -> Poll<Option<Self::Item>, Self::Error>;
    ...
}
```

The only difference here is that the return type is wrapped in an `Option`, exactly like the `Iterator` trait. Thus, a `None` here would indicate that the stream has terminated. Also, all streams are futures and can be converted using `into_future`. Let us look at an example of using this construct. We will partially reuse our collatz example from a previous chapter. The first step is to set up the project:

```
$ cargo new --bin streams
```

With all the dependencies added, our Cargo config looks like this:

```
[package]
name = "streams"
version = "0.1.0"
authors = ["Foo<foo@bar.com>"]

[dependencies]
futures = "0.1.17"
rand = "0.3.18"
```

Having set everything up, our main file will look as follows. In this case, we have a struct called `CollatzStream` that has two fields for the current state and the end state (which should always be `1`). We will implement the `Stream` trait on this to make this behave as a stream:

```
// ch7/streams/src/main.rs

extern crate futures;
extern crate rand;

use std::{io, thread};
use std::time::Duration;
use futures::stream::Stream;
use futures::{Poll, Async};
use rand::{thread_rng, Rng};
use futures::Future;

// This struct holds the current state and the end condition
// for the stream
#[derive(Debug)]
```

```
struct CollatzStream {
    current: u64,
    end: u64,
}

// A constructor to initialize the struct with defaults
impl CollatzStream {
    fn new(start: u64) -> CollatzStream {
        CollatzStream {
            current: start,
            end: 1
        }
    }
}

// Implementation of the Stream trait for our struct
impl Stream for CollatzStream {
    type Item = u64;
    type Error = io::Error;
    fn poll(&mut self) -> Poll<Option<Self::Item>, io::Error> {
        let d = thread_rng().gen_range::<u64>(1, 5);
        thread::sleep(Duration::from_secs(d));
        if self.current % 2 == 0 {
            self.current = self.current / 2;
        } else {
            self.current = 3 * self.current + 1;
        }
        if self.current == self.end {
            Ok(Async::Ready(None))
        } else {
            Ok(Async::Ready(Some(self.current)))
        }
    }
}

fn main() {
    let stream = CollatzStream::new(10);
    let f = stream.for_each(|num| {
        println!("{}", num);
        Ok(())
    });
    f.wait().ok();
}
```

We simulate a delay in returning the result by sleeping for a random amount of time between `1` and `5` seconds. Our implementation for the poll returns `Ok(Async::Ready(None))` to signal that the stream has finished when it reaches `1`. Otherwise, it returns the current state as `Ok(Async::Ready(Some(self.current)))`. It’s easy to note that, except for the stream semantics, this implementation is the same as that for iterators. In our main function, we initialize the struct and use the `for_each` combinator to print out each item in the stream. This combinator returns a future on which we call `wait` and `ok` to block and get all results. Here is what we see on running the last example:

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/streams`
5
16
8
4
2
```

As it is with the `Future` trait, the `Stream` trait also supports a number of other combinators useful for different purposes. The dual of a `Stream` is a `Sink`, which is a receiver of asynchronous events. This is extremely useful in modeling the sending end of Rust channels, network sockets, file descriptors, and so on.

A common pattern in any asynchronous system is synchronization. This becomes important, as more often than not, components need to communicate with one another to pass data or coordinate tasks. We solved this exact problem in the past using channels. But those constructions are not applicable here, as the channel implementation in the standard library is not asynchronous. Thus, futures has its own channel implementation, which provides all the guarantees you would expect from an asynchronous system. Let us look at an example; our project setup should look like this:

```
$ cargo new --bin futures-ping-pong
```

Cargo config should look like this:

```
[package]
name = "futures-ping-pong"
version = "0.1.0"
authors = ["Foo<foo@bar.com>"]

[dependencies]
futures = "0.1"
tokio-core = "0.1"
rand = "0.3.18"
```

Now we have two functions. One waits for a random amount of time and then randomly returns either `"ping"` or `"pong"`. This function will be our sender. Here is what it looks like:

```
// ch7/futures-ping-pong/src/main

extern crate futures;
extern crate rand;
extern crate tokio_core;

use std::thread;
use std::fmt::Debug;
use std::time::Duration;
use futures::Future;
use rand::{thread_rng, Rng};

use futures::sync::mpsc;
use futures::{Sink, Stream};
use futures::sync::mpsc::Receiver;

// Randomly selects a sleep duration between 1 and 5 seconds. Then
// randomly returns either "ping" or "pong"
fn sender() -> &'static str {
    let mut d = thread_rng();
    thread::sleep(Duration::from_secs(d.gen_range::<u64>(1, 5)));
    d.choose(&["ping", "pong"]).unwrap()
}

// Receives input on the given channel and prints each item
fn receiver<T: Debug>(recv: Receiver<T>) {
    let f = recv.for_each(|item| {
        println!("{:?}", item);
        Ok(())
    });
    f.wait().ok();
}

fn main() {
    let (tx, rx) = mpsc::channel(100);
    let h1 = thread::spawn(|| {
        tx.send(sender()).wait().ok();
    });
    let h2 = thread::spawn(|| {
        receiver::<&str>(rx);
    });
    h1.join().unwrap();
    h2.join().unwrap();
}
```

The futures crate provides two types of channel: a *oneshot* channel that can be used only once to send and receive any messages, and a regular *mpsc* channel that can be used multiple times. In our main function, we get hold of both ends of the channel and spawn our sender in another thread as a future. The receiver is spawned in another thread. In both cases, we record the handles to be able to wait for them to finish (using `join`) later. Note that our receiver takes in the receiving end of the channel as parameter. Because `Receiver` implements `Stream`, we can use the `and_then` combinator on it to print out the value. Finally, we call `wait()` and `ok()` on the future before exiting the receiver function. In the main function, we join on the two thread handles to drive them to completion.

Running the last example will randomly print either `"ping"` or `"pong"`, depending on what was sent via the channel. Note that the actual printing happens on the receiving end.

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/futures-ping-pong`
"ping"
```

The futures crate also provides a locking mechanism in `futures::sync::BiLock` that closely mirrors `std::sync::Mutex`. This is a future-aware mutex that arbitrates sharing a resource between two owners. Note that a `BiLock` is only for two futures, which is an annoying limitation. Here is how it works: we are interested in modifying our last example to show a counter when the sender function is called. Now our counter needs to be thread-safe so that it can be shared across consumers. Set up the project using Cargo:

```
$ cargo new --bin future-bilock
```

Our `Cargo.toml` file should be exactly the same, and here is how the main file looks:

```
// ch7/future-bilock/src/main.rs
```

```
extern crate futures;
extern crate rand;

use std::thread;
use std::fmt::Debug;
use std::time::Duration;
use futures::{Future, Async};
use rand::{thread_rng, Rng};

use futures::sync::{mpsc, BiLock};
use futures::{Sink, Stream};
use futures::sync::mpsc::Receiver;

// Increments the shared counter if it can acquire a lock, then
// sleeps for a random duration between 1 and 5 seconds, then
// randomly returns either "ping" or "pong"
fn sender(send: &BiLock<u64>) -> &'static str {
    match send.poll_lock() {
        Async::Ready(mut lock) => *lock += 1,
        Async::NotReady => ()
    }
    let mut d = thread_rng();
    thread::sleep(Duration::from_secs(d.gen_range::<u64>(1, 5)));
    d.choose(&["ping", "pong"]).unwrap()
}

// Tries to acquire a lock on the shared variable and prints it's
// value if it got the lock. Then prints each item in the given
// stream
fn receiver<T: Debug>(recv: Receiver<T>, recv_lock: BiLock<u64>) {
    match recv_lock.poll_lock() {
        Async::Ready(lock) => println!("Value of lock {}", *lock),
        Async::NotReady => ()
    }
    let f = recv.for_each(|item| {
        println!("{:?}", item);
        Ok(())
    });
    f.wait().ok();
}

fn main() {
    let counter = 0;
    let (send, recv) = BiLock::new(counter);
    let (tx, rx) = mpsc::channel(100);
    let h1 = thread::spawn(move || {
        tx.send(sender(&send)).wait().ok();
    });
    let h2 = thread::spawn(|| {
        receiver::<&str>(rx, recv);
    });
    h1.join().unwrap();
    h2.join().unwrap();
}
```

While this is basically the same as the last example, there are some differences. In our main function, we set the counter to zero. We then create a `BiLock` on the counter. The constructor returns two handles like a channel, which we can then pass around. We then create our channel and spawn the sender. Now, if we look at the sender, it has been modified to take in a reference to a `BiLock`. In the function, we attempt to acquire a lock using `poll_lock`, and, if that works, we increment the counter. Otherwise, we do nothing. We then move on to our usual business of returning `"ping"` or `"pong"`. The receiver has been modified to take a `BiLock` as well. In that, we try to acquire a lock and, if successful, we print out the value of the data being locked. In our main function, we spawn these futures using threads and join on them to wait for those to finish.

Here is what happens on an unsuccessful run, when both parties fail to acquire the lock. In a real example, we would want to handle the error gracefully and retry. We left out that part for the sake of brevity:

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/futures-bilock`
thread '<unnamed>' panicked at 'no Task is currently running',
libcore/option.rs:917:5
note: Run with `RUST_BACKTRACE=1` for a backtrace.
thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value: Any',
libcore/result.rs:945:5
```

Here is what a good run looks like:

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/futures-bilock`
Value of lock 1
"pong"
```

Heading to tokio

The tokio ecosystem is an implementation of a network stack in Rust. It has all the major functionality of the standard library, the major difference being that it is non-blocking (most common calls do not block the current thread). This is achieved by using mio to do all the low-level heavy lifting, and using futures to abstract away long-running operations. The ecosystem has two basic crates, everything else being built around those:

- `tokio-proto` provides primitives for building asynchronous servers and clients. This depends heavily on mio for low-level networking and on futures for abstraction.
- `tokio-core` provides an event loop to run futures in and a number of related APIs. This is useful when an application needs fine-grained control over IO.

As we mentioned in the last section, one way to run futures is on an event loop. An event loop (called a `reactor` in tokio) is an infinite loop that listens for defined events and takes appropriate action once it receives one. Here is how this works: we will borrow our previous example of a function that determines whether the given input is a prime or not. This returns a future with the result, which we then print out. The project setup is the same as it always is:

```
$ cargo new --bin futures-loop
```

Here is what `Cargo.toml` should look like:

```
[package]
name = "futures-loop"
version = "0.1.0"
authors = ["Foo<foo@bar.com>"]

[dependencies]
futures = "0.1"
tokio-core = "0.1"
```

For this example, we will take input in an infinite loop. For each input, we trim out newlines and spaces and try to parse it as an `u64`. Here is how it looks:

```
// ch7/futures-loop/src/main.rs

extern crate futures;
extern crate tokio_core;

use std::io;
use std::io::BufRead;
use futures::Future;
use tokio_core::reactor::Core;

fn check_prime_boxed(n: u64) -> Box<Future<Item = bool, Error = io::Error>> {
    for i in 2..n {
        if n % i == 0 {
            return Box::new(futures::future::ok(false));
        }
    }
    Box::new(futures::future::ok(true))
}

fn main() {
    let mut core = Core::new().expect("Could not create event loop");
    let stdin = io::stdin();

    loop {
        let mut line = String::new();
        stdin
            .lock()
            .read_line(&mut line)
            .expect("Could not read from stdin");
        let input = line.trim()
            .parse::<u64>()
            .expect("Could not parse input as u64");
        let result = core.run(check_prime_boxed(input))
            .expect("Could not run future");
        println!("{}", result);
    }
}
```

In our main function, we create the core and start our infinite loop. We use the `run` method of core to start a task to execute the future asynchronously. The result is collected and printed on the standard output. Here is what a session should look like:

```
$ cargo run
12
false
13
true
991
true
```

The `tokio-proto` crate is an asynchronous server (and client) building toolkit. Any server that uses this crate has the following three distinct layers:

- A codec that dictates how data should be read from and written to the underlying socket forming the transport layer for our protocol. Subsequently, this layer is the bottom-most (closest to the physical medium). In practice, writing a codec amounts to implementing a few given traits from the library that processes a stream of bytes.
- A protocol sits above a codec and below the actual event loop running the protocol. This acts as a glue to bind those together. tokio supports multiple protocol types, depending on the application: a simple request-response type protocol, a multiplexed protocol, and a streaming protocol. We will delve into each of these shortly.
- A service that actually runs all this as a future. As this is just a future, an

easy way to think of this is as an asynchronous function that transforms an input to an eventual response (which could be an error). In practice, most of the computation is done in this layer.

Because the layers are swappable, an implementation is perfectly free to swap the protocol type for another, or the service for another one, or the codec. Let us look at an example of a simple service using `tokio-proto`. This one is a traditional request-response service that provides a text-based interface. It takes in a number and returns its collatz sequence as an array. If the input is not a valid integer, it send back a message indicating the same. Our project setup is pretty simple:

```
$ cargo new --bin collatz-proto
```

The Cargo config looks like the following sample:

```
[package]
name = "collatz-proto"
version = "0.1.0"
authors = ["Foo<foo@bar.com>"]

[dependencies]
bytes = "0.4"
futures = "0.1"
tokio-io = "0.1"
tokio-core = "0.1"
tokio-proto = "0.1"
tokio-service = "0.1"
```

As described earlier, we will need to implement the different layers. In our current case, each of our layers do not need to hold much state. Thus, they can be represented using unit structs. If that was not the case, we would need to put some data in those.

```
// ch7/collatz-proto/src/main.rs

extern crate bytes;
extern crate futures;
extern crate tokio_io;
extern crate tokio_proto;
extern crate tokio_service;

use std::io;
use std::str;
use bytes::BytesMut;
use tokio_io::codec::{Encoder, Decoder};
use tokio_io::{AsyncRead, AsyncWrite};
use tokio_io::codec::Framed;
use tokio_proto::pipeline::ServerProto;
use tokio_service::Service;
use futures::{future, Future};
use tokio_proto::TcpServer;

// Codec implementation, our codec is a simple unit struct
pub struct CollatzCodec;

// Decoding a byte stream from the underlying socket
impl Decoder for CollatzCodec {
    type Item = String;
    type Error = io::Error;

    fn decode(&mut self, buf: &mut BytesMut) -> io::Result<Option<String>> {
        // Since a newline denotes end of input, read till a newline
        if let Some(i) = buf.iter().position(|&b| b == b'\n') {
            let line = buf.split_to(i);
            // and remove the newline
            buf.split_to(1);
            // try to decode into an UTF8 string before passing
            // to the protocol
            match str::from_utf8(&line) {
                Ok(s) => Ok(Some(s.to_string())),
                Err(_) => Err(io::Error::new(io::ErrorKind::Other,
                "invalid UTF-8")),
            }
        } else {
            Ok(None)
        }
    }
}

// Encoding a string to a newline terminated byte stream
impl Encoder for CollatzCodec {
    type Item = String;
    type Error = io::Error;

    fn encode(&mut self, msg: String, buf: &mut BytesMut) ->
    io::Result<()> {
        buf.extend(msg.as_bytes());
        buf.extend(b"\n");
        Ok(())
    }
}

// Protocol implementation as an unit struct
pub struct CollatzProto;

impl<T: AsyncRead + AsyncWrite + 'static> ServerProto<T> for CollatzProto {
    type Request = String;
    type Response = String;
    type Transport = Framed<T, CollatzCodec>;
    type BindTransport = Result<Self::Transport, io::Error>;
    fn bind_transport(&self, io: T) -> Self::BindTransport {
        Ok(io.framed(CollatzCodec))
    }
}

// Service implementation
pub struct CollatzService;

fn get_sequence(n: u64) -> Vec<u64> {
    let mut n = n.clone();
    let mut result = vec![];
    result.push(n);
    while n > 1 {
        if n % 2 == 0 {
            n /= 2;
        } else {
            n = 3 * n + 1;
        }
        result.push(n);
    }
    result
}

impl Service for CollatzService {
    type Request = String;
    type Response = String;
    type Error = io::Error;
    type Future = Box<Future<Item = Self::Response, Error = Self::Error>>;

    fn call(&self, req: Self::Request) -> Self::Future {
        match req.trim().parse::<u64>() {
            Ok(num) => {
                let res = get_sequence(num);
                Box::new(future::ok(format!("{:?}", res)))
            }
            Err(_) => Box::new(future::ok("Could not parse input as an
            u64".to_owned())),
        }
    }
}

fn main() {
    let addr = "0.0.0.0:9999".parse().unwrap();
    let server = TcpServer::new(CollatzProto, addr);
    server.serve(|| Ok(CollatzService));
}
```

As we saw earlier, the first step is to tell the codec how to read data to and from the socket. This is done by implementing `Encoder` and `Decoder` from `tokio_io::codec`. Note that we don't have to deal with raw sockets here; we get a stream of bytes as input, which we are free to process. According to our protocol defined before, a newline indicates an end of input. So, in our decoder, we read till a newline and return the data after removing the said newline as a UTF-8 encoded string. In case of an error, we return a `None`.

The `Encoder` implementation is exactly the reverse: it transforms a string into a stream of bytes. The next step is the protocol definition, this one is really simple, as it does not do multiplexing or streaming. We implement `bind_transport` to bind the codec to our raw socket, which we will get to later. The only catch here is that the `Request` and `Response` types here should match that of the codec. Having set these up, the next step is to implement the service, by declaring an unit struct and implementing the `Service` trait on it. Our helper function `get_sequence` returns the collatz sequence given a `u64` as input. The `call` method in `Service` implements the logic of computing the response. We parse the input as a `u64` (remember that our codec returns input as a String). If that did not error out, we call our helper function and return the result as a static string, otherwise we return an error. Our main function looks similar to such a function as would use standard networking types, but, we use the `TcpServer` type from tokio, which takes in our socket (to bind it to the codec) and our protocol definition. Finally, we call the `serve` method while passing our service as a closure. This method takes care of managing the event loop and cleaning up things on exit.

Let us use `telnet` to interact with it. Here is how a session will look:

```
$ telnet localhost 9999
Connected to localhost.
Escape character is '^]'.
```

```
12
[12, 6, 3, 10, 5, 16, 8, 4, 2, 1]
30
[30, 15, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1]
foobar
Could not parse input as an u64
```

As always, it would be much more useful to write a client for our server. We will borrow a lot from the example of running a future in an event loop. We start with setting up our project:

```
$ cargo new --bin collatz-client
```

Our Cargo setup will look like this:

```
[package]
name = "collatz-client"
version = "0.1.0"
authors = ["Abhishek Chanda <abhishek.becs@gmail.com>"]

[dependencies]
futures = "0.1"
tokio-core = "0.1"
tokio-io = "0.1"
```

Here is our main file:

```
// ch7/collatz-client/src/main.rs

extern crate futures;
extern crate tokio_core;
extern crate tokio_io;

use std::net::SocketAddr;
use std::io::BufReader;
use futures::Future;
use tokio_core::reactor::Core;
use tokio_core::net::TcpStream;

fn main() {
    let mut core = Core::new().expect("Could not create event loop");
    let handle = core.handle();
    let addr: SocketAddr = "127.0.0.1:9999".parse().expect("Could not parse as
SocketAddr");
    let socket = TcpStream::connect(&addr, &handle);
    let request = socket.and_then(|socket| {
        tokio_io::io::write_all(socket, b"110\n")
    });
    let response = request.and_then(|(socket, _request)| {
        let sock = BufReader::new(socket);
        tokio_io::io::read_until(sock, b'\n', Vec::new())
    });
    let (_socket, data) = core.run(response).unwrap();
    println!("{}", String::from_utf8_lossy(&data));
}
```

Of course, this one sends a single integer to the server (110 in decimal), but it is trivial to put this in a loop to read input and send those. We leave that as an exercise for the reader. Here, we create a event loop and get its handle. We then use the asynchronous `TcpStream` implementation to connect to the server on a given address. This returns a future, which we combine with a closure using `and_then` to write to the given socket. The whole construct returns a new future called `request`, which is chained with a reader future. The final future is called `response` and is run on the event loop. Finally, we read the response and print it out. At every step, we have to respect our protocol that a newline denotes end-of-input for both the server and the client. Here is what a session looks like:

```
$ cargo run
   Compiling futures-loop v0.1.0 (file:///src/ch7/collatz-client)
    Finished dev [unoptimized + debuginfo] target(s) in 0.94 secs
     Running `target/debug/futures-loop`
[110, 55, 166, 83, 250, 125, 376, 188, 94, 47, 142, 71, 214, 107, 322, 161, 484,
242, 121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466, 233, 700, 350, 175,
526, 263, 790, 395, 1186, 593, 1780, 890, 445, 1336, 668, 334, 167, 502, 251, 754,
377, 1132, 566, 283, 850, 425, 1276, 638, 319, 958, 479, 1438, 719, 2158, 1079,
3238, 1619, 4858, 2429, 7288, 3644, 1822, 911, 2734, 1367, 4102, 2051, 6154, 3077,
9232, 4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976, 488, 244, 122,
61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1]
```

Socket multiplexing in tokio

One model of asynchronous-request processing in a server is through multiplexing incoming connections. In this case, each connection is assigned a unique ID of some kind and replies are issued whenever one is ready, irrespective of the order in which it was received. Thus, this allows a higher throughput, as the shortest job gets the highest priority implicitly. This model also makes the server highly responsive with a larger number of incoming requests of varying complexity. Traditional Unix-like systems support this using the select and poll system calls for socket multiplexing.

In the tokio ecosystem, this is mirrored by a number of traits that enable implementing multiplexed protocols. The basic anatomy of the server is the same as a simple server: we have the codec, the protocol using the codec, and a service that actually runs the protocol. The only difference here is that we will assign a request ID to each incoming request. This will be used later to disambiguate while sending back responses. We will also need to implement some traits from the `tokio_proto::multiplex` namespace. As an example, we will modify our collatz server and add multiplexing to it. Our project setup is a bit different in this case, as we are planning to run the binaries using Cargo, and our project will be a library. We set it up like this:

```
$ cargo new collatz-multiplexed
```

The Cargo config is similar:

```
[package]
name = "collatz-multiplexed"
version = "0.1.0"
authors = ["Foo <foo@bar.com>"]

[dependencies]
bytes = "0.4"
futures = "0.1"
tokio-io = "0.1"
tokio-core = "0.1"
tokio-proto = "0.1"
tokio-service = "0.1"
```

Here is what the `lib.rs` file looks like:

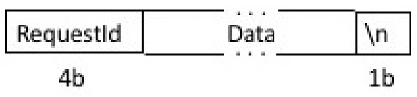
```
// ch7/collatz-multiplexed/src/lib.rs

extern crate bytes;
extern crate futures;
extern crate tokio_core;
extern crate tokio_io;
extern crate tokio_proto;
extern crate tokio_service;
```

```
use futures::{future, Future};

use tokio_io::{AsyncRead, AsyncWrite};
use tokio_io::codec::{Decoder, Encoder, Framed};
use tokio_core::net::TcpStream;
use tokio_core::reactor::Handle;
use tokio_proto::TcpClient;
use tokio_proto::multiplex::{ClientProto, ClientService, RequestId, ServerProto};
use tokio_service::Service;

use bytes::{BigEndian, Buf, BufMut, BytesMut};

use std::{io, str};
use std::net::SocketAddr;

// Everything client side
// Represents a client connecting to our server
pub struct Client {
    inner: ClientService<TcpStream, CollatzProto>,
}

impl Client {
    pub fn connect(
        addr: &SocketAddr,
        handle: &Handle,
    ) -> Box<Future<Item = Client, Error = io::Error>> {
        let ret = TcpClient::new(CollatzProto)
            .connect(addr, handle)
            .map(|service| Client {
                inner: service,
            });

        Box::new(ret)
    }
}

impl Service for Client {
    type Request = String;
    type Response = String;
    type Error = io::Error;
    type Future = Box<Future<Item = String, Error = io::Error>>;

    fn call(&self, req: String) -> Self::Future {
        Box::new(self.inner.call(req).and_then(move |resp| Ok(resp)))
    }
}

// Everything server side
pub struct CollatzCodec;
pub struct CollatzProto;

// Represents a frame that has a RequestId and the actual data (String)
type CollatzFrame = (RequestId, String);

impl Decoder for CollatzCodec {
    type Item = CollatzFrame;
    type Error = io::Error;

    fn decode(&mut self, buf: &mut BytesMut) ->
    Result<Option<CollatzFrame>, io::Error> {
        // Do not proceed if we haven't received at least 6 bytes yet
        // 4 bytes for the RequestId + data + 1 byte for newline
        if buf.len() < 5 {
            return Ok(None);
        }
        let newline = buf[4..].iter().position(|b| *b == b'\n');
        if let Some(n) = newline {
            let line = buf.split_to(n + 4);
            buf.split_to(1);
            let request_id = io::Cursor::new(&line[0..4]).get_u32:
            :<BigEndian>();
            return match str::from_utf8(&line.as_ref()[4..]) {
                Ok(s) => Ok(Some((u64::from(request_id),
                s.to_string()))),
                Err(_) => Err(io::Error::new(io::ErrorKind::Other,
                "invalid string")),
            };
        }
        // Frame is not complete if it does not have a newline at the
        end
        Ok(None)
    }
}

impl Encoder for CollatzCodec {
    type Item = CollatzFrame;
    type Error = io::Error;

    fn encode(&mut self, msg: CollatzFrame, buf: &mut BytesMut) ->
    io::Result<()> {
        // Calculate final message length first
        let len = 4 + msg.1.len() + 1;
        buf.reserve(len);

        let (request_id, msg) = msg;

        buf.put_u32::<BigEndian>(request_id as u32);
        buf.put_slice(msg.as_bytes());
        buf.put_u8(b'\n');

        Ok(())
    }
}

impl<T: AsyncRead + AsyncWrite + 'static> ClientProto<T> for CollatzProto {
    type Request = String;
    type Response = String;

    type Transport = Framed<T, CollatzCodec>;
    type BindTransport = Result<Self::Transport, io::Error>;

    fn bind_transport(&self, io: T) -> Self::BindTransport {
        Ok(io.framed(CollatzCodec))
    }
}

impl<T: AsyncRead + AsyncWrite + 'static> ServerProto<T> for CollatzProto {
    type Request = String;
    type Response = String;

    type Transport = Framed<T, CollatzCodec>;
    type BindTransport = Result<Self::Transport, io::Error>;

    fn bind_transport(&self, io: T) -> Self::BindTransport {
        Ok(io.framed(CollatzCodec))
    }
}

pub struct CollatzService;

fn get_sequence(mut n: u64) -> Vec<u64> {
    let mut result = vec![];
    result.push(n);
    while n > 1 {
        if n % 2 == 0 {
            n /= 2;
        } else {
            n = 3 * n + 1;
        }
        result.push(n);
    }
    result
}

impl Service for CollatzService {
    type Request = String;
    type Response = String;
    type Error = io::Error;
    type Future = Box<Future<Item = Self::Response, Error = Self::Error>>;

    fn call(&self, req: Self::Request) -> Self::Future {
        match req.trim().parse::<u64>() {
            Ok(num) => {
                let res = get_sequence(num);
                Box::new(future::ok(format!("{:?}", res)))
            }
            Err(_) => Box::new(future::ok("Could not parse input as an
            u64".to_owned())),
        }
    }
}
```

Tokio provides a built-in type called `RequestId` to represent unique IDs for incoming requests, all states associated with it being managed internally by tokio. We define a custom data type called `CollatzFrame` for our frame; this has the `RequestId` and a `String` for our data. We move on to implementing `Decoder` and `Encoder` for `CollatzCodec` like last time. But, in both of these cases, we have to take into account the request ID in the header and the trailing newline. Because the `RequestId` type is a `u64` under the hood, it will always be four bytes and one extra byte for a newline. Thus, if we have received fewer than 5 bytes, we know that the whole frame has not been received yet. Note that this is not an error case, the frame is still being transmitted, so we return an `Ok(None)`. We then check whether the buffer has a newline (in compliance with our protocol). If everything looks good, we parse the request ID from the first 4 bytes (note that this will be in network-byte order). We then construct an instance of `CollatzFrame` and return it. The encoder implementation is the inverse; we just need to put the request ID back in, then the actual data, and end with a newline.

The next steps are to implement `ServerProto` and `ClientProto` for `CollatzProto`; both of these are boilerplates that bind the codec with the transport. Like last time, the last step is to implement the service. This step does not change at all. Note that we do not need to care about dealing with the request ID after implementing the codec because later stages do not see it at all. The codec deals with and manages it while passing on the actual data to later layers.

Here is what our frame looks like:



Our request frame with the RequestId as header and a trailing newline

This time, our client will be based on tokio too. Our `Client` struct wraps an instance of `ClientService`, which takes in the underlying TCP stream and the protocol implementation to use. We have a convenience function called `connect` for the `Client` type, which connects to a given server and returns a future. Lastly, we implement `Service` for `Client` in which the `call` method returns a future. We run the server and client as examples by putting them in a directory called `examples`. This way, cargo knows that those should be run as associated examples with this crate. The server looks like this:

```
// ch7/collatz-multiplexed/examples/server.rs

extern crate collatz_multiplexed as collatz;
extern crate tokio_proto;

use tokio_proto::TcpServer;
use collatz::{CollatzService, CollatzProto};

fn main() {
    let addr = "0.0.0.0:9999".parse().unwrap();
    TcpServer::new(CollatzProto, addr).serve(|| Ok(CollatzService));
}
```

This is pretty much the same as last time, just in a different file. We have to declare our parent crate as an external dependency so that Cargo can link everything properly. This is how the client looks:

```
// ch7/collatz-multiplexed/examples/client.rs

extern crate collatz_multiplexed as collatz;

extern crate futures;
extern crate tokio_core;
extern crate tokio_service;

use futures::Future;
use tokio_core::reactor::Core;
use tokio_service::Service;

pub fn main() {
    let addr = "127.0.0.1:9999".parse().unwrap();
    let mut core = Core::new().unwrap();
    let handle = core.handle();

    core.run(
        collatz::Client::connect(&addr, &handle)
            .and_then(|client| {
                client.call("110".to_string())
                    .and_then(move |response| {
                        println!("We got back: {:?}", response);
                        Ok(())
                    })
            })
    ).unwrap();
}
```

We run our client in an event loop, using tokio-core. We use the connect method defined on the client to get a future wrapping the connection. We use the `and_then` combinator and use the call method to send a string to the server. As this method returns a future as well, we can use the `and_then` combinator on the inner future to extract the response and then resolve it by returning an `Ok(())`. This also resolves the outer future.

Now, if we open two terminals and run the server in one and the client in another, here is what we should see in the client. Note that as we do not have sophisticated retries and error handling, the server should be run before the client:

```
$ cargo run --example client
   Compiling collatz-multiplexed v0.1.0 (file:///src/ch7/collatz-multiplexed)
    Finished dev [unoptimized + debuginfo] target(s) in 0.93 secs
     Running `target/debug/examples/client`
We got back: "[110, 55, 166, 83, 250, 125, 376, 188, 94, 47, 142, 71, 214, 107,
322, 161, 484, 242, 121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466, 233,
700, 350, 175, 526, 263, 790, 395, 1186, 593, 1780, 890, 445, 1336, 668, 334, 167,
502, 251, 754, 377, 1132, 566, 283, 850, 425, 1276, 638, 319, 958, 479, 1438, 719,
2158, 1079, 3238, 1619, 4858, 2429, 7288, 3644, 1822, 911, 2734, 1367, 4102, 2051,
6154, 3077, 9232, 4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976, 488,
244, 122, 61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4,
2, 1]"
```

And, as expected, this output matches what we got before.

Writing streaming protocols

In a number of cases, a protocol has a data unit with a header attached to it. The server typically reads the header first and, based on that, decides how to process the data. In some cases, the server may be able to do some processing based on the header. One such example is IP, which has a header that has the destination address. A server may start running a longest prefix match based on that information, before reading the body. In this section, we will look into using tokio to write such servers. We will expand our toy collatz protocol to include a header and some data body, and work from there. Let us start with an example, and our project setup will be exactly the same, setting it up using Cargo:

```
$ cargo new collatz-streaming
```

Cargo config does not change much:

```
[package]
name = "collatz-streaming"
version = "0.1.0"
authors = ["Foo <foo@bar.com>"]

[dependencies]
bytes = "0.4"
futures = "0.1"
tokio-io = "0.1"
tokio-core = "0.1"
tokio-proto = "0.1"
tokio-service = "0.1"
```

As this example is large, we have broken this down into constituent pieces, as follows. The first part shows setting up the client:

```
// ch7/collatz-streaming/src/lib.rs

extern crate bytes;
extern crate futures;
extern crate tokio_core;
extern crate tokio_io;
extern crate tokio_proto;
extern crate tokio_service;

use futures::{future, Future, Poll, Stream};
use futures::sync::mpsc;
use tokio_io::{AsyncRead, AsyncWrite};
use tokio_io::codec::{Decoder, Encoder, Framed};
use tokio_core::reactor::Handle;
use tokio_proto::TcpClient;
use tokio_proto::streaming::{Body, Message};
use tokio_proto::streaming::pipeline::{ClientProto, Frame, ServerProto};
use tokio_proto::util::client_proxy::ClientProxy;
use tokio_service::Service;
use std::str::FromStr;
use bytes::{BufMut, BytesMut};
use std::{io, str};
use std::net::SocketAddr;

// Everything about clients
type CollatzMessage = Message<String, Body<String, io::Error>>;

#[derive(Debug)]
pub enum CollatzInput {
    Once(String),
    Stream(CollatzStream),
}

pub struct CollatzProto;

pub struct Client {
    inner: ClientProxy<CollatzMessage, CollatzMessage, io::Error>,
}

impl Client {
    pub fn connect(
        addr: &SocketAddr,
        handle: &Handle,
    ) -> Box<Future<Item = Client, Error = io::Error>> {
        let ret = TcpClient::new(CollatzProto)
            .connect(addr, handle)
            .map(|cp| Client { inner: cp });

        Box::new(ret)
    }
}

impl Service for Client {
    type Request = CollatzInput;
    type Response = CollatzInput;
    type Error = io::Error;
    type Future = Box<Future<Item = Self::Response, Error =
    io::Error>>;

    fn call(&self, req: CollatzInput) -> Self::Future {
        Box::new(self.inner.call(req.into()).map(CollatzInput::from))
    }
}
```

As always, the first step is to set up extern crates and include everything required. We define a few types that we will use. A `CollatzMessage` is a message that our protocol receives; it has a header and a body both of type `String`. A `CollatzInput` is an input stream for the protocol, which is an enum with two variants: `Once` represents the case where we have received data in a non-streaming way, and `Stream` is for the second case. The protocol implementation is a unit struct called `CollatzProto`. We then define a struct for the client, which has an inner instance of `ClientProxy`, the actual client implementation. This takes in three types, the first two being request and response for the whole server, and the last one being for errors. We then implement a connect method for the `Client` struct that connects using `CollatzProto`, and this returns a future with the connection. The last step is to implement `Service` for `Client`, and both the input and output for this is of type `CollatzInput`, thus we have to transform the output to that type using map on the future. Let us move on to the server; it looks like this:

```
//ch7/collatz-streaming/src/lib.rs

// Everything about server
#[derive(Debug)]
pub struct CollatzStream {
    inner: Body<String, io::Error>,
}

impl CollatzStream {
    pub fn pair() -> (mpsc::Sender<Result<String, io::Error>>,
    CollatzStream) {
        let (tx, rx) = Body::pair();
        (tx, CollatzStream { inner: rx })
    }
}

impl Stream for CollatzStream {
    type Item = String;
    type Error = io::Error;

    fn poll(&mut self) -> Poll<Option<String>, io::Error> {
        self.inner.poll()
    }
}

pub struct CollatzCodec {
    decoding_head: bool,
}

// Decodes a frame to a byte slice
impl Decoder for CollatzCodec {
    type Item = Frame<String, String, io::Error>;
    type Error = io::Error;

    fn decode(&mut self, buf: &mut BytesMut) -> Result<Option<Self::Item>,
io::Error> {
        if let Some(n) = buf.as_ref().iter().position(|b| *b == b'\n') {
            let line = buf.split_to(n);

            buf.split_to(1);
            return match str::from_utf8(line.as_ref()) {
                Ok(s) => {
                    if s == "" {
                        let decoding_head = self.decoding_head;
                        self.decoding_head = !decoding_head;

                        if decoding_head {
                            Ok(Some(Frame::Message {
                                message: s.to_string(),
                                body: true,
                            }))
                        } else {
                            Ok(Some(Frame::Body { chunk: None }))
                        }
                    } else {
                        if self.decoding_head {
                            Ok(Some(Frame::Message {
                                message: s.to_string(),
                                body: false,
                            }))
                        } else {
                            Ok(Some(Frame::Body {
                                chunk: Some(s.to_string()),
                            }))
                        }
                    }
                }
                Err(_) => Err(io::Error::new(io::ErrorKind::Other,
                "invalid string")),
            };
        }

        Ok(None)
    }
}

// Encodes a given byte slice to a frame
impl Encoder for CollatzCodec {
    type Item = Frame<String, String, io::Error>;
    type Error = io::Error;

    fn encode(&mut self, msg: Self::Item, buf: &mut BytesMut) ->
    io::Result<()> {
        match msg {
            Frame::Message { message, body } => {
                buf.reserve(message.len());
                buf.extend(message.as_bytes());
            }
            Frame::Body { chunk } => {
                if let Some(chunk) = chunk {
                    buf.reserve(chunk.len());
                    buf.extend(chunk.as_bytes());
                }
            }
            Frame::Error { error } => {
                return Err(error);
            }
        }
        buf.put_u8(b'\n');
        Ok(())
    }
}

impl<T: AsyncRead + AsyncWrite + 'static> ClientProto<T> for CollatzProto {
    type Request = String;
    type RequestBody = String;
    type Response = String;
    type ResponseBody = String;
    type Error = io::Error;

    type Transport = Framed<T, CollatzCodec>;
    type BindTransport = Result<Self::Transport, io::Error>;

    fn bind_transport(&self, io: T) -> Self::BindTransport {
        let codec = CollatzCodec {
            decoding_head: true,
        };

        Ok(io.framed(codec))
    }
}

impl<T: AsyncRead + AsyncWrite + 'static> ServerProto<T> for CollatzProto {
    type Request = String;
    type RequestBody = String;
    type Response = String;
    type ResponseBody = String;
    type Error = io::Error;

    type Transport = Framed<T, CollatzCodec>;
    type BindTransport = Result<Self::Transport, io::Error>;

    fn bind_transport(&self, io: T) -> Self::BindTransport {
        let codec = CollatzCodec {
            decoding_head: true,
        };

        Ok(io.framed(codec))
    }
}
```

As expected, a `CollatzStream` has a body that is either a string or has resulted in an error. Now, for a streaming protocol, we need to provide an implementation of a function that returns the sender half of the stream; we do this in the `pair` function for `CollatzStream`. Next, we implement the `Stream` trait for our custom stream; the `poll` method in this case simply polls the inner `Body` for more data. Having set up the stream, we can implement the codec. In here, we will need to maintain a way to know what part of the frame we are processing at this moment. This is done using a boolean called `decoding_head` that we flip as we need. We need to implement `Decoder` for our codec, and this is pretty much the same as the last few times; just note that we need to keep track of the streaming and non-streaming cases and the boolean defined previously. The `Encoder` implementation is the reverse. We also need to bind the protocol implementation to the codec; this is done by implementing `ClientProto` and `ServerProto` for `CollatzProto`. In both cases, we set the boolean to true, as the first thing to be read after receiving the message is the header.

The last step in the stack is to implement the services by implementing the `Service` trait for `CollatzService`. In that, we read the header and try to parse it as a `u64`. If that works fine, we move on to calculating the collatz sequence of that `u64` and return the result as a `CollatzInput::Once` in a leaf future. In the other case, we iterate over the body and print it on the console. Finally, we return a fixed string to the client. Here is what this looks like:

```
//ch7/collatz-streaming/src/lib.rs

pub struct CollatzService;

// Given an u64, returns it's collatz sequence
fn get_sequence(mut n: u64) -> Vec<u64> {
    let mut result = vec![];
    result.push(n);
    while n > 1 {
        if n % 2 == 0 {
            n /= 2;
        } else {
            n = 3 * n + 1;
        }
        result.push(n);
    }
    result
}

// Removes leading and trailing whitespaces from a given line
// and tries to parse it as a u64
fn clean_line(line: &str) -> Result<u64, <u64 as FromStr>::Err> {
    line.trim().parse::<u64>()
}

impl Service for CollatzService {
    type Request = CollatzInput;
    type Response = CollatzInput;
    type Error = io::Error;
    type Future = Box<Future<Item = Self::Response, Error = Self::Error>>;

    fn call(&self, req: Self::Request) -> Self::Future {
        match req {
            CollatzInput::Once(line) => {
                println!("Server got: {}", line);
                let res = get_sequence(clean_line(&line).unwrap());
                Box::new(future::done(Ok(CollatzInput::Once
                (format!("{:?}", res)))))
            }
            CollatzInput::Stream(body) => {
                let resp = body.for_each(|line| {
                    println!("{}", line);
                    Ok(())
                }).map(|_| CollatzInput::Once("Foo".to_string()));

                Box::new(resp) as Box<Future<Item = Self::
                Response, Error = io::Error>>
            }
        }
    }
}
```

We have also written two conversion helpers from `CollatzMessage` to `CollatzInput`, and vice versa, by implementing the `From` trait accordingly. Like everything else, we will have to deal with the two cases: when the message has a body and when it does not (in other words, the header has arrived but not the rest of the message). Here are those:

```
// ch7/collatz-streaming/src/lib.rs

// Converts a CollatzMessage to a CollatzInput
impl From<CollatzMessage> for CollatzInput {
    fn from(src: CollatzMessage) -> CollatzInput {
        match src {
            Message::WithoutBody(line) => CollatzInput::Once(line),
            Message::WithBody(_, body) => CollatzInput::Stream(CollatzStream {
inner: body }),
        }
    }
}

// Converts a CollatzInput to a Message<String, Body>
impl From<CollatzInput> for Message<String, Body<String, io::Error>> {
    fn from(src: CollatzInput) -> Self {
        match src {
            CollatzInput::Once(line) => Message::WithoutBody(line),
            CollatzInput::Stream(body) => {
                let CollatzStream { inner } = body;
                Message::WithBody("".to_string(), inner)
            }
        }
    }
}
```

Having set up the server and the client, we will implement our tests as examples, like last time. Here is what they look like:

```
// ch7/collatz-streaming/examples/server.rs

extern crate collatz_streaming as collatz;
extern crate futures;
extern crate tokio_proto;

use tokio_proto::TcpServer;
use collatz::{CollatzProto, CollatzService};

fn main() {
    let addr = "0.0.0.0:9999".parse().unwrap();

    TcpServer::new(CollatzProto, addr).serve(|| Ok(CollatzService));
}
```

Here is what the client looks like. There is a bit to digest here, compared to the server:

```
// ch7/collatz-streaming/examples/client.rs

extern crate collatz_streaming as collatz;

extern crate futures;
extern crate tokio_core;
extern crate tokio_service;

use collatz::{CollatzInput, CollatzStream};
use std::thread;
use futures::Sink;
use futures::Future;
use tokio_core::reactor::Core;
use tokio_service::Service;

pub fn main() {
    let addr = "127.0.0.1:9999".parse().unwrap();
    let mut core = Core::new().unwrap();
    let handle = core.handle();

    // Run the client in the event loop
    core.run(
        collatz::Client::connect(&addr, &handle)
            .and_then(|client| {
                client.call(CollatzInput::Once("10".to_string()))
                    .and_then(move |response| {
                        println!("Response: {:?}", response);

                        let (mut tx, rx) = CollatzStream::pair();

                        thread::spawn(move || {
                            for msg in &["Hello", "world", "!"] {
                                tx =
                                tx.send(Ok(msg.to_string()))
                                .wait().unwrap();
                            }
                        });

                        client.call(CollatzInput::Stream(rx))
                    })
                    .and_then(|response| {
                        println!("Response: {:?}", response);
                        Ok(())
                    })
            })
    ).unwrap();
}
```

We use the `connect` method defined earlier to set up connection to the server on a known address and port. We use the `and_then` combinator to send a fixed string to the server, and we print the response. At this point, we have transmitted our header and we move on to transmitting the body. This is done by splitting the stream in two halves and using the sender to send a number of strings. A final combinator prints the response and resolves the future. Everything previously is run in an event loop.

This is what a session looks like for the server:

```
$ cargo run --example server
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/examples/server`
Server got: 10
Hello
world
!
```

And this is how it looks like for the client:

```
$ cargo run --example client
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/examples/client`
Response: Once("[10, 5, 16, 8, 4, 2, 1]")
Response: Once("Foo")
```

As expected, the server processed the header before it got the actual body of the request (we know that because we sent the body after sending the header).

Other than what we discussed here, tokio supports a bunch of other features as well. For instance, you can implement a protocol handshake by changing the `ServerProto` and `ClientProto` implementations to exchange setup messages before constructing the `BindTransport` future. This is extremely important, as a lot of network protocols need some form of handshaking to set up a shared state to work with. Note that it is perfectly possible for a protocol be streaming and pipelined, or streaming and multiplexed. For these, an implementation needs to substitute traits from the `streaming::pipeline` or `streaming::multiplex` namespace respectively.

The larger tokio ecosystem

Let us take a look at the current state of the tokio ecosystem. Here are the commonly useful crates in the `tokio-rs` GitHub organization, at the time of writing:

Crate	**Function**
`tokio-minihttp`	Simple HTTP server implementation in tokio; should not be used in production.
`tokio-core`	Future-aware networking implementations; the core event loop.
`tokio-io`	IO primitives for tokio.
`tokio-curl`	A `libcurl`-based HTTP client implementation using tokio.
`tokio-uds`	Non-blocking unix domain sockets using tokio.
`tokio-tls`	TLS and SSL implementation based on tokio.
`tokio-service`	Provides the `Service` trait that we have used extensively.
`tokio-proto`	Provides a framework for building network protocols using tokio; we have used this one extensively.
`tokio-socks5`	A SOCKS5 server using tokio, not production-ready.
`tokio-middleware`	Collection of middlewares for tokio services; lacks essential services at the moment.
`tokio-times`	Timer-related functionality based on tokio.
`tokio-line`	Sample-line protocol for demonstrating tokio.
`tokio-redis`	Proof-of-concept redis client based on tokio; should not be used in production.
`service-fn`	Provides a function that implements the Service trait for a given closure.

Note that a number of these are either not updated in a long time or proof-of-concept implementations that should not be used in anything useful. But that is not a problem. A large number of independent utilities have adopted tokio since it was launched, resulting in a vibrant ecosystem. And, in our opinion, that is the true sign of success of any open source project.

Let us look at some commonly used libraries in the preceding list , starting with `tokio-curl`. For our example, we will simply download a single file from a known location, write it to local disk, and print out the headers we got back from the server. Because this is a binary, we will set the project up like this:

```
$ cargo new --bin tokio-curl
```

Here is what the Cargo setup looks like:

```
[package]
name = "tokio-curl"
version = "0.1.0"
authors = ["Foo <foo@bar.com>"]

[dependencies]
tokio-curl = "0.1"
tokio-core = "0.1"
curl = "0.4.8"
```

As the `tokio-curl` library is a wrapper around the Rust `curl` library, we will need to include that as well. Here is the main file:

```
// ch7/toki-curl/src/main.rs

extern crate curl;
extern crate tokio_core;
extern crate tokio_curl;

use curl::easy::Easy;
use tokio_core::reactor::Core;
use tokio_curl::Session;
use std::io::Write;
use std::fs::File;

fn main() {
    let mut core = Core::new().unwrap();
    let session = Session::new(core.handle());

    let mut handle = Easy::new();
    let mut file = File::create("foo.zip").unwrap();
    handle.get(true).unwrap();
    handle.url("http://ipv4.download.thinkbroadband.com/5MB.zip").unwrap();
    handle.header_function(|header| {
        print!("{}", std::str::from_utf8(header).unwrap());
        true
    }).unwrap();
    handle.write_function(move |data| {
        file.write_all(data).unwrap();
        Ok(data.len())
    }).unwrap();

    let request = session.perform(handle);

    let mut response = core.run(request).unwrap();
    println!("{:?}", response.response_code());
}
```

We will use the `Easy` API from `curl` crate. We start with creating our event loop and HTTP session. We then create a handle that `libcurl` will use to process our request. We call the `get` method with a bool to indicate that we are interested in doing an HTTP GET. We then pass the URL to the handle. Next, we set two callbacks passed as closures. The first one is called the `header_function`; this one shows each of the client-side headers. The second one is called the `write_function`, which writes the data we got to our file. Finally, we create a request by calling the `perform` function for our session. Lastly, we run the request in our event loop and print out the status code we got back.

Running this does the following:

```
$ cargo run
   Compiling tokio-curl v0.1.0 (file:///src/ch7/tokio-curl)
    Finished dev [unoptimized + debuginfo] target(s) in 0.97 secs
     Running `target/debug/tokio-curl`
HTTP/1.1 200 OK
Server: nginx
Date: Mon, 25 Dec 2017 20:16:12 GMT
Content-Type: application/zip
Content-Length: 5242880
Last-Modified: Mon, 02 Jun 2008 15:30:42 GMT
Connection: keep-alive
ETag: "48441222-500000"
Access-Control-Allow-Origin: *
Accept-Ranges: bytes

Ok(200)
```

This will also produce a file called `foo.zip` in the current directory. You can use a regular file to download the file and compare the SHA sums of both files, to verify that they are indeed the same.

Conclusion

This chapter was an introduction to one of the most exciting components of the larger Rust ecosystem. Futures and the Tokio ecosystem provide powerful primitives that can be widely used in applications, including networking software. In itself, Futures can be used to model any computation that is otherwise slow and/or depends on an external resource. Coupled with tokio, it can be used to model complex protocol behaviors that are pipelined, or multiplexed, and so on.

Some major drawbacks of using these center around lack of proper documentation and examples. Also, error messages from these applications are often heavily templated and, hence, verbose. Because the Rust compiler itself does not know of the abstractions as such, it often complains about type mismatches, and it is up to the user to reason through deeply nested types. Somewhere down the line, it may make sense to implement a compiler plugin for futures that can translate these errors in more intuitive forms.

In the following chapter, we will look at implementing common security-related primitives in Rust.

Security

Security is often looked at as an afterthought in systems design. That is evident in common protocols; security related RFCs has historically been proposed after the main protocol. Notice that any communication over a public medium (like the internet) is vulnerable to man-in-the-middle attacks. An adversary might hijack the communication by carefully inspecting input packets from both sides. In light of that, some security related questions are reasonable: When a client connects to a server, how does it verify that the server is the one it claims to be? How do they decide on a shared secret key to use for encryption? In this chapter, we will see how these questions are commonly addressed.

We will cover the following topics:

- Securing web-based applications using certificates
- Key exchange using the Diffie-Hellman method

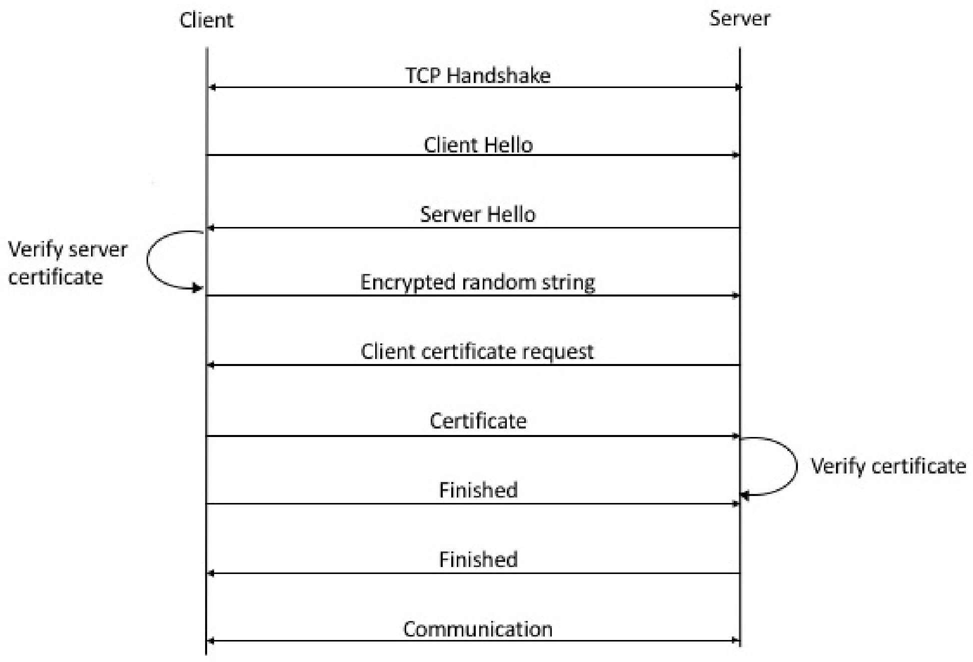
Securing the web

In a previous chapter, we studied HTTP. We noted how important it has been in making our lives easier. However, HTTP is vulnerable to a range of attacks that might result in leaking the payload. Thus, it was necessary to add some form of security between parties using HTTP to communicate. RFC 2818 proposed HTTPS (HTTP Secure) as a version of HTTP that uses a secure streaming protocol underneath. Initially, this was **Secure Socket Layer** (**SSL**), and later evolved into **Transport Layer Security** (**TLS**).

The basic scheme of things goes like this:

- The **Client** and **Server** establish a TCP connection.
- The **Client** and **Server** agree upon a cipher and hash function to use throughout the connection. For this, the client sends a list of ciphers and hash functions. The **Server** picks one from that list and lets the **Client** know.
- The **Server** sends a certificate to the **Client**. The **Client** validates this against a list of certificate authorities that it has locally.
- Both agree on a session key to be used to encrypt data during the connection.
- At this point, a regular HTTP session can begin.

The following image illustrates the steps for this:



Client-server communication over SSL

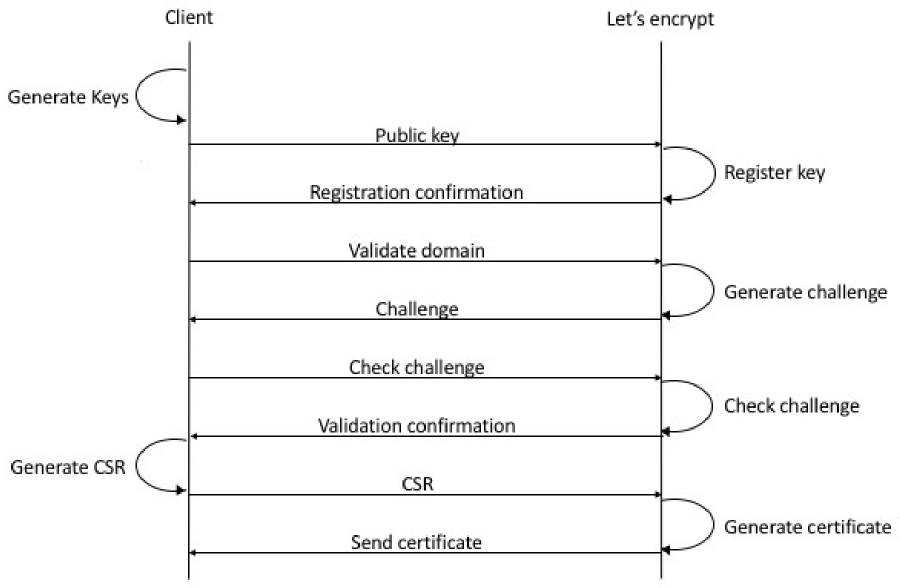
One of the most important steps here is verifying the server’s identity. This answers a fundamental question: W*hen the client talks to a server, how does it know the server is actually the server it wanted?* In practice, this is achieved using certificates. A certificate is a document that is signed by a certificate authority, a trusted provider who is legally allowed to vouch for others. A client has a list of trusted CAs, and if the CA which issued a given certificate is on that list, a client can trust the server that presents that certificate. In practice, there are often a chain of certificates issued as a chain of trust relationships, going back to a root CA.

In recent years, search engines put a heavy emphasis on having HTTPS on websites, often ranking them higher in that case. However, issuing a certificate for a website has traditionally been a tedious process. A website owner will have to log in to a CA’s website and provide some form of identification. Issuing the certificate often took a few hours and was very costly for small business owners. In 2015, Let’s Encrypt launched as a non-profit CA with the goal of providing free, short-lived certificates to all websites on the internet. They automate the validation process by issuing a challenge to the server administrator. This typically involves either placing a given file in a known location on the website or creating a DNS record with a given content. Once `letsencrypt` has validated the server, it issues a certificate valid for 90 days. Thus, the certificate needs to be renewed periodically.

More recently, `letsencrypt` standardized the challenge response protocol as JSON over HTTPS and named it ACME. Here is how this works:

- The local client generates a private-public keypair and contacts the letsencrypt server with the **Public** key.
- The server creates an account for the given key and registers it.
- Based on challenge preferences, the **Client** will present a list of challenges that can be fulfilled and ask the server to validate the domain. Currently, two challenges are supported: an HTTP based challenge where a file is placed on a known location and the server will read it to validate, or a DNS based challenge where the operator has to create a TXT record on the domain with a given content.
- The server generates the challenge and sends it back.
- At this point, the client will poll the server for confirmation.
- When the server returns an OK, the client can proceed to generating a **certificate signing request** (**CSR**) for the server and send it across.
- The server then generates a certificate and sends it back.

The following diagram illustrates individual steps in the ACME protocol:



ACME protocol operation

Letsencrypt using Rust

Currently, there is one crate that allows access to `letsencrypt` using Rust. The CLI tool called `acme-client` can interact with the API to obtain or revoke a certificate or run validations of ownership. The binary is backed by a crate called acme-client that enables programmatic interaction with the API. Let's see how this can be used to secure an HTTP server running on Rocket. Remember, for this to work, `letsencrypt` will need to reach the server. Thus, this needs to be publicly accessible over the internet.

The first step is to install the CLI tool using Cargo:

```
$ cargo install acme-client
```

For our example, we will run our rocket blog over TLS. While Rocket does support TLS out of the box, it is not enabled by default. We will need to change the `Cargo.toml` file to add TLS as a feature flag. It should look like this:

```
[package]
authors = ["Foo <foo@bar.com>"]
name = "rocket-blog"
version = "0.1.0"

[dependencies]
rocket = { version="0.3.5", features = ["tls"] }
rocket_codegen = "0.3.5"
rocket_contrib = "0.3.5"
diesel = { version = "0.16.0", features = ["sqlite"] }
diesel_codegen = { version = "0.16.0", features = ["sqlite"] }
dotenv = "0.10.1"
serde = "1.0.21"
serde_json = "1.0.6"
serde_derive = "1.0.21"
lazy_static = "0.2.11"
r2d2 = "0.7.4"
r2d2-diesel = "0.16.0"
```

We will also run Rocket on the public interface. For this, we will place a config file called `Rocket.toml` in the root of the repository. This is how it looks; everything else is left at the defaults:

```
$ cat Rocket.toml
[global]
address = "0.0.0.0"
```

Like we did before, we can run our server using Cargo:

```
$ DATABASE_URL=./db.sql cargo run
```

Letsencrypt also requires all servers to have a domain name. Thus, we will need to create a record for our server in a DNS provider. In our example, that DNS name is `my.domain.io`, which we will use in subsequent steps. Once that record has propagated everywhere, we can move on to the next step: generating certificates. Here is how we will do that using the CLI:

```
$ acme-client -vvvvv sign --dns -D my.domain.io -P /var/www -o domain.crt
INFO:acme_client: Registering account
DEBUG:acme_client: User successfully registered
INFO:acme_client: Sending identifier authorization request for foo.datasine.com
Please create a TXT record for _acme-challenge.my.domain.io:
MFfatN9I3UFQk2WP_f1uRWi4rLnr4qMVaI
Press enter to continue

INFO:acme_client: Triggering dns-01 validation
DEBUG:acme_client: Status is pending, trying again...
INFO:acme_client: Signing certificate
DEBUG:acme_client: Certificate successfully signed
```

We will have to use the DNS based validation here since we will not be able to serve back challenges. The CLI asked us to create a `TXT` record with a given name and content. Once we create the record, we will need to wait for some time so that it can propagate before we move forward. It is a good idea to check if the record is updated using `dig` before moving forward. Here is how the output of `dig` should look:

```
$ dig _acme-challenge.my.domain.io TXT

; <<>> DiG 9.9.7-P3 <<>> _acme-challenge.my.domain.io TXT
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 49153
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 8192
;; QUESTION SECTION:
;_acme-challenge.my.domain.io. IN TXT

;; ANSWER SECTION:
_acme-challenge.my.domain.io. 60 IN TXT "MFfatN9I3UFQk2WP_f1uRWi4rLnr4qMVaI"

;; Query time: 213 msec
;; SERVER: 192.168.0.1#53(192.168.0.1)
;; WHEN: Fri Jan 05 18:04:01 GMT 2018
;; MSG SIZE rcvd: 117
```

When the output from `dig` says that we have the correct `TXT` record, we can move ahead with installing the certificate. `letsencrypt` will then query the `TXT` record and run domain validation. When everything looks fine, we will get a new certificate named `domain.crt`. Let's inspect the certificate to make sure everything looks okay. The subject should match our domain name and the issuer should always be *Let's Encrypt*, as shown in the following code snippet:

```
$ openssl x509 -in domain.crt -subject -issuer -noout
subject= /CN=my.domain.io
issuer= /C=US/O=Let's Encrypt/CN=Let's Encrypt Authority X3
```

Now we are ready to use this certificate in our Rocket application. We will need to place it at a location where the running Rocket has permission to read it. Now, if we restart Rocket and use `curl` to access the endpoint over HTTPS, it should work exactly like last time:

```
$ curl -sS -D - https://my.domain.io:8000/posts
HTTP/1.1 200 OK
Content-Type: application/json
Server: Rocket
Content-Length: 992
Date: Fri, 05 Jan 2018 18:37:58 GMT

[{"id":1,"title":"test","body":"test body","pinned":true},{"id":2,"title":"Hello
Rust!","body":"Rust is awesome!!","pinned":true},{"id":3,"title":"Testing
this","body":"Try to write something","pinned":true},{"id":4,"title":"Testing
this","body":"Try to write something","pinned":true},{"id":5,"title":"Testing
this","body":"Try to write something","pinned":true},{"id":6,"title":"Testing
this","body":"Try to write something","pinned":true},{"id":7,"title":"Testing
this","body":"Try to write something","pinned":true},{"id":8,"title":"Testing
this","body":"Try to write something","pinned":true},{"id":9,"title":"Testing
this","body":"Try to write something","pinned":true},{"id":10,"title":"Testing
this","body":"Try to write something","pinned":true},{"id":11,"title":"Testing
this","body":"Try to write something","pinned":true},{"id":12,"title":"Testing
this","body":"Try to write something","pinned":true},{"id":13,"title":"Testing
this","body":"Try to write something","pinned":true}]
```

As of the time of writing, Let’s Encrypt has a limit of five certificates for a given domain per week. For testing, it is possible to reach that limit pretty quickly. If that happens, the client will just say Acme server error.

Let’s write a simple client for our server here using `rustls`. We set up the project using Cargo:

```
$ cargo new --bin rustls-client
```

We then add `rustls` to our project as a dependency :

```
[package]
name = "rustls-client"
version = "0.1.0"
authors = ["Foo <foo@bar.com>"]

[dependencies]
rustls = "0.12.0"
webpki = "0.18.0-alpha"
webpki-roots = "0.14.0"
```

Here is how the client looks; note that the `webpki` crate does DNS resolution, which is then used by `rustls`:

```
// ch8/rustls-client/src/main.rs

use std::sync::Arc;

use std::net::TcpStream;
use std::io::{Read, Write};

extern crate rustls;
extern crate webpki;
extern crate webpki_roots;

fn main() {
    let mut tls = rustls::ClientConfig::new();
    tls.root_store.add_server_trust_anchors
    (&webpki_roots::TLS_SERVER_ROOTS);

    let name = webpki::DNSNameRef::try_from_ascii_str("my.domain.io")
    .expect("Could not resolve name");
    let mut sess = rustls::ClientSession::new(&Arc::new(tls), name);
    let mut conn = TcpStream::connect("my.domain.io:8000").unwrap();
    let mut stream = rustls::Stream::new(&mut sess, &mut conn);
    stream.write(concat!("GET /posts HTTP/1.1\r\n",
                         "Connection: close\r\n",
                         "\r\n")
              .as_bytes())
        .expect("Could not write request");
    let mut plaintext = Vec::new();
    stream.read_to_end(&mut plaintext).expect("Could not read");
    println!("{}", String::from_utf8(plaintext)
    .expect("Could not print output"));
}
```

Here, we are connecting to our server and running the same HTTP request. We import all three required crates. The first step is to initialize a TLS session and add the root certificates to it. We then resolve our given server to a DNS name reference and establish a TLS session with it. Having set this up, we can set up a TCP session by using `connect`. Finally, a `rustls Stream` is a combination of the SSL session and the TCP session. Once all those are working, we can write HTTP queries by hand. Here we are running a `GET` on the `/posts` endpoint. Later, we read the response and print it out. The output should be exactly the same as using the other clients:

```
HTTP/1.1 200 OK
Content-Type: application/json
Server: Rocket
Content-Length: 992
Date: Fri, 05 Jan 2018 18:37:58 GMT

[{"id":1,"title":"test","body":"test body","pinned":true},{"id":2,"title":"Hello
Rust!","body":"Rust is awesome!!","pinned":true},{"id":3,"title":"Testing
this","body":"Try to write something","pinned":true},{"id":4,"title":"Testing
this","body":"Try to write something","pinned":true},{"id":5,"title":"Testing
this","body":"Try to write something","pinned":true},{"id":6,"title":"Testing
this","body":"Try to write something","pinned":true},{"id":7,"title":"Testing
this","body":"Try to write something","pinned":true},{"id":8,"title":"Testing
this","body":"Try to write something","pinned":true},{"id":9,"title":"Testing
this","body":"Try to write something","pinned":true},{"id":10,"title":"Testing
this","body":"Try to write something","pinned":true},{"id":11,"title":"Testing
this","body":"Try to write something","pinned":true},{"id":12,"title":"Testing
this","body":"Try to write something","pinned":true},{"id":13,"title":"Testing
this","body":"Try to write something","pinned":true}]
```

OpenSSL using Rust

The OpenSSL library and CLI is a complete set of tools to work with SSL (and TLS) objects. It is an open source project and is widely used by companies all around. As one would expect, Rust has bindings for using as a library in Rust projects. In this discussion, we will take a closer look at the certificates we saw in the last section.

These are commonly defined by a standard called X.509 (defined in RFC 5280) and have the following fields:

- **Version number**: Almost always set to 2, corresponding to version 3 (since the first version is 0). According to the standard, this field can be omitted and should be assumed to be version 1 (value set to 0).
- **Serial number**: A 20 octet identifier that is unique for the CA which signed this certificate.
- **Signature**: A unique ID that identifies the algorithm used to sign this certificate. This is usually a string defined in subsequent RFCs, an example being `sha1WithRSAEncryption`.
- **Issuer name**: Identifies the CA that signed the certificate. This must have at least one **Distinguished name** (**DN**) composed of a number of components, including a **Common Name** (**CN**), **State** (**ST**), **Country** (**C**), and so on.
- **Validity**: Defines when the certificate should be valid. This has two sub-fields; `notBefore` denotes when this starts to be valid and `notAfter` denotes when it expires.
- **Subject name**: Identifies the entity this certificate certifies. For a root certificate, this will be the same as the issuer. This has the same format as the issuer name and should at least have a DN.
- **Subject public key info**: Information about the subject’s encrypted public key. This has two sub-fields; the first one is an ID for the encryption algorithm (as in the signature field), and the second is a bit stream that has the encrypted public key.
- **Issuer unique ID**: An optional field that can be used to uniquely identify the issuer.
- **Subject unique ID**: An optional field that can be used to identify the subject.
- **Extensions**: This field is only applicable if the version is set to 3.

Denotes a number of optional fields that can be used to attach additional information to certificates.

- **Certificate signature algorithm**: Algorithm used to sign this certificate; must be the same as the one used in the signature attribute previously.
- **Certificate signature value**: A bit string that has the actual signature for verification.

In the following example, we will use `rust-openssl` to generate and inspect a certificate from scratch. Installing the library, however, can be a little complicated. Since it is a wrapper around `libopenssl`, it needs to link against the native library. Thus, the library has to be installed. The crate's documentation has instructions to get this set up. The project setup is routine:

```
$ cargo new --bin openssl-example
```

Here is the `Cargo.toml` file:

```
[package]
name = "openssl-example"
version = "0.1.0"
authors = ["Foo <foo@bar.com>"]

[dependencies]
openssl = { git = "https://github.com/sfackler/rust-openssl" }
```

Also, for this example, we will use the master branch of the repository, since we need a few features that are not released yet. For that, we will need to specify the repository link, as shown previously. Here is the main file:

```
// ch8/openssl-example/src/main.rs

extern crate openssl;

use std::env;
use std::fs::File;
use std::io::Write;

use openssl::x509::{X509, X509Name};
use openssl::nid::Nid;
use openssl::pkey::{PKey, Private};
use openssl::rsa::Rsa;
use openssl::error::ErrorStack;
use openssl::asn1::Asn1Time;
use openssl::bn::{BigNum, MsbOption};
use openssl::hash::MessageDigest;

fn create_cert() -> Result<(X509, PKey<Private>), ErrorStack> {
    let mut cert_builder = X509::builder()?;
    cert_builder.set_version(2)?;

    let serial_number = {
        let mut serial = BigNum::new()?;
        serial.rand(160, MsbOption::MAYBE_ZERO, false)?;
        serial.to_asn1_integer()?
    };
    cert_builder.set_serial_number(&serial_number)?;

    let mut name = X509Name::builder()?;
    name.append_entry_by_text("C", "UK")?;
    name.append_entry_by_text("CN", "Our common name")?;
    let cert_name = name.build();
    cert_builder.set_issuer_name(&cert_name)?;

    let not_before = Asn1Time::days_from_now(0)?;
    cert_builder.set_not_before(&not_before)?;

    let not_after = Asn1Time::days_from_now(365)?;
    cert_builder.set_not_after(&not_after)?;

    cert_builder.set_subject_name(&cert_name)?;

    let private_key = PKey::from_rsa(Rsa::generate(3072)?)?;
    cert_builder.set_pubkey(&private_key)?;

    cert_builder.sign(&private_key, MessageDigest::sha512())?;
    let cert = cert_builder.build();

    Ok((cert, private_key))
}

fn main() {
    if let Some(arg) = env::args().nth(1) {
        let (cert, _key) = create_cert().expect("could not create
        cert");
        let cert_data = cert.to_pem().expect("could not convert cert
        to pem");

        let mut cert_file = File::create(arg)
        .expect("could not create cert file");
        cert_file
            .write_all(&cert_data)
            .expect("failed to write cert");

        let subject = cert.subject_name();
        let cn = subject
            .entries_by_nid(Nid::COMMONNAME)
            .next()
            .expect("failed to get subject");
        println!("{}",String::from_utf8(cn.data()
        .as_slice().to_vec()).unwrap()
        );
    } else {
        eprintln!("Expected at least one argument");
        std::process::exit(1);
    }
}
```

Here, we take in a filename as a command-line parameter to write the certificate to. Certificate creation is offloaded to a helper function called `create_cert` that returns either a tuple having the generated certificate and private key or a list of errors as a `Result`.

The first step is to initialize a certificate builder object that we will add on to, and finally, build our certificate. We use the `set_version` method to set the version to 3 (numerically set to `2`). Now we need to generate a serial number and set it. We generate that by randomly sampling 160 bits (20 octets of 8 bits each). We use the `set_serial_number` method to set the serial number. The next step is to generate a name by using a name builder. The `append_entry_by_text` method is then used to add a country name and a common name to our name builder. We use `set_issuer_name` to append the `name` object to the certificate. We set the expiry date to 365 days after the current date and use `set_not_before` and `set_not_after` to set those two. The subject name is set to the same name object using `set_subject_name`. Finally, we need to generate a private key, which we generate using the `Rsa` module, and we set the private key on the certificate. Now, the certificate needs to be signed with the private key using SHA512. Once done, we can use the `build` method to create the certificate. At the end of the function, we return the certificate and private key to the caller.

In our `main` function, we call the `helper` function. For our example, we will ignore the key, but a real application does need to save it for verification later. We convert the `certificate` object to PEM encoding and write it to a file on the disk. The next step is reading the subject name programmatically. For this, we use the `subject_name` method on the `certificate` object and print it out as a string. This should match the subject name we set earlier.

Here is how we can run this using Cargo. Note that this creates a certificate named `bar.crt` in the current directory:

```
$ cargo run bar.crt
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/openssl-example bar.crt`
Our common name
$ ls -la bar.crt
-rw-r--r--+ 1 Abhishek staff 1399 19 Feb 22:02 bar.crt
```

Securing tokio applications

A common problem in implementing Tokio based protocols is around securing them. Luckily, the Tokio ecosystem has `tokio-tls` for this. Let's look at an example of using this to secure our hyper example from a previous chapter. Here is our setup:

```
$ cargo new --bin tokio-tls-example
```

The Cargo manifest should look like this:

```
[package]
name = "tokio-tls-example"
version = "0.1.0"
authors = ["Foo <foo@bar.com>"]

[dependencies]
hyper = "0.11.7"
futures = "0.1.17"
net2 = "0.2.31"
tokio-core = "0.1.10"
num_cpus = "1.0"
native-tls = "*"
tokio-service = "*"
tokio-proto = "*"
tokio-tls = { version = "0.1", features = ["tokio-proto"] }
```

We will need to use the `tokio-proto` feature on `tokio-tls` to enable integration with `tokio-proto`. The next step is to generate a self-signed certificate for our server. `tokio-tls` uses the `native-tls` library underneath, and that does not support constructing acceptors from X509 certificates at the time of this writing. Thus, we will need to use PKCS12 certificates. The command shown ahead generates a self signed certificate, valid for 365 days, in PEM format. This will ask for a pass-phrase for the certificate. In our case, we used `foobar`. Please ensure that this command is run in the `tokio-tls-example` directory so that our code can read the certificate:

```
$ openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -days 365
```

The following command converts the given certificate to PKCS12 format. This should generate a file called `cert.pfx` in the current directory, which we will use in the code:

```
$ openssl pkcs12 -export -out cert.pfx -inkey key.pem -in cert.pem
```

Here is our main file, with some changes for enabling SSL:

```
// ch8/tokio-tls-example/src/main.rs

extern crate futures;
extern crate hyper;
extern crate native_tls;
extern crate tokio_proto;
extern crate tokio_service;
extern crate tokio_tls;

use std::io;
use std::{thread, time};
use futures::future::{ok, Future};
use hyper::server::Http;
use hyper::header::ContentLength;
use hyper::{Request, Response, StatusCode};
use native_tls::{Pkcs12, TlsAcceptor};
use tokio_proto::TcpServer;
use tokio_service::Service;
use tokio_tls::proto;

fn heavy_work() -> String {
    let duration = time::Duration::from_millis(100);
    thread::sleep(duration);
    "done".to_string()
}

struct SlowMo;

impl Service for SlowMo {
    type Request = Request;
    type Response = Response;
    type Error = io::Error;
    type Future = Box<Future<Item = Response, Error = io::Error>>;

    fn call(&self, req: Request) -> Self::Future {
        let b = heavy_work().into_bytes();
        println!("Request: {:?}", req);
        Box::new(ok(Response::new()
            .with_status(StatusCode::Ok)
            .with_header(ContentLength(b.len() as u64))
            .with_body(b)))
    }
}

fn main() {
    let raw_cert = include_bytes!("../cert.pfx");
    let cert = Pkcs12::from_der(raw_cert, "foobar").unwrap();
    let acceptor = TlsAcceptor::builder(cert).unwrap().build().unwrap();
    let proto = proto::Server::new(Http::new(), acceptor);
    let addr = "0.0.0.0:9999".parse().unwrap();
    let srv = TcpServer::new(proto, addr);
    println!("Listening on {}", addr);
    srv.serve(|| Ok(SlowMo));
}
```

The major changes here are that in the main function, we use the `include_bytes` macro to read the certificate as raw bytes. We then construct a `Pkcs12` object using `from_der` by passing the certificate bytes and the pass-phrase that we used while creating the certificate. The next step is to create a `TlsAcceptor` object using the given `Pkcs12 certificate` object. We will then need to wrap the `acceptor` object and the `hyper protocol` object into a `Server`. This is passed to the `TcpServer` constructor, which we then start.

Here is what a session looks like from a client's perspective:

```
$ curl -k https://localhost:9999
done
```

Here is what the server prints; this originates from the `println!` macro in the call function:

```
$ cargo run
   Compiling rustls-example v0.1.0 (file:///src/ch8/rustls-example)
    Finished dev [unoptimized + debuginfo] target(s) in 3.14 secs
     Running `target/debug/rustls-example`
Listening on 0.0.0.0:9999
Request: Request { method: Get, uri: "/", version: Http11, remote_addr: None,
headers: {"Host": "localhost:9999", "User-Agent": "curl/7.57.0", "Accept": "*/*"} }
^C
```

Interestingly, the `openssl` command-line tool has a TCP client that can be used to test SSL connections as well. Here is how to use it to test our server:

```
$ openssl s_client -connect 127.0.0.1:9999
CONNECTED(00000003)
depth=0 C = UK, ST = Scotland, O = Internet Widgits Pty Ltd
verify error:num=18:self signed certificate
verify return:1
depth=0 C = UK, ST = Scotland, O = Internet Widgits Pty Ltd
verify return:1
---
Certificate chain
 0 s:/C=UK/ST=Scotland/O=Internet Widgits Pty Ltd
   i:/C=UK/ST=Scotland/O=Internet Widgits Pty Ltd
---
Server certificate
-----BEGIN CERTIFICATE-----
...
-----END CERTIFICATE-----
subject=/C=UK/ST=Scotland/O=Internet Widgits Pty Ltd
issuer=/C=UK/ST=Scotland/O=Internet Widgits Pty Ltd
---
No client certificate CA names sent
Peer signing digest: SHA256
Server Temp Key: ECDH, P-256, 256 bits
---
SSL handshake has read 2128 bytes and written 433 bytes
---
New, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES256-GCM-SHA384
Server public key is 4096 bit
Secure Renegotiation IS supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
SSL-Session:
    Protocol : TLSv1.2
    Cipher : ECDHE-RSA-AES256-GCM-SHA384
    Session-ID: 9FAB89D29DB02891EF52C825AC23E3A658FDCE228A1A7E4FD97652AC3A5E24F3
    Session-ID-ctx:
    Master-Key:
940EF0C4FA1A929133C2D273739C8042FAF1BD5057E793ED1D7A0F0187F0236EF9E43D236DF8C17D663D
    Key-Arg : None
    PSK identity: None
    PSK identity hint: None
    SRP username: None
    Start Time: 1515350045
    Timeout : 300 (sec)
    Verify return code: 18 (self signed certificate)
---
GET / HTTP/1.1
host: foobar

HTTP/1.1 200 OK
Content-Length: 4
```

```
Date: Sun, 07 Jan 2018 18:34:52 GMT

done
```

This tool negotiates an SSL session and dumps the server certificate as shown previously (we have replaced the actual certificate for brevity). Notice that it correctly detects that the given server is using a self-signed certificate. Finally, it starts a TCP session. Since this is bare TCP, we will need to hand craft our HTTP requests. If we use a simple `GET` request on `/`, we get back a response of `200 OK`, and the string is done. On the other side, this is what the server prints:

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/tokio-tls-example`
Listening on 0.0.0.0:9999
Request: Request { method: Get, uri: "/", version: Http11, remote_addr: None,
headers: {"Host": "foobar"} }
```

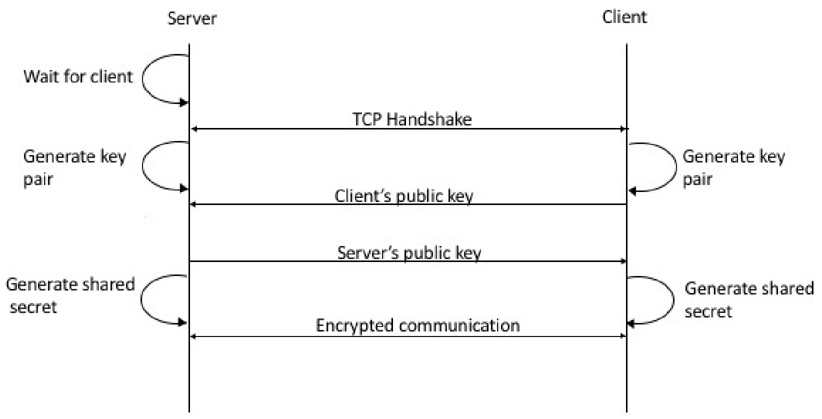
Notice that this prints the `Host` header, set to the string `foobar`, as we wrote on the client side.

Cryptography using ring

A commonly used crate for cryptography is called `ring`. This crate supports a number of lower-level crypto primitives, like random number generation, key exchanges, and so on. In this section, we will take key exchange as an example and see how this crate can be used in client-server applications.

A common problem in communication is that of encrypting information so that a third-party cannot decipher it. In a private key system, both the client and the server will need to agree on a key to use for this to work. Now, this key cannot be transmitted in plain text over an insecure connection. The Diffie-Hellman key exchange method defines a mechanism where two parties talking over a secure link can negotiate a key that is shared between them, but has not been transmitted over the connection. This method has a number of implementations on many platforms, including the crate in question.

The following diagram shows how the protocol works:



Diffie-Hellman key exchange in action

Initially, the server will be listening for incoming clients. When a client connects, this is the sequence of events:

1. A TCP session is first set up.
2. Both the server and the client generate private and public keys.
3. The client then sends its public key to the server.
4. The server responds by sending the public key that it generated.
5. At this point, both of the parties can generate the shared secret key using their private keys and the public key that they received.
6. Further communication can be encrypted using the shared secret key.

We will create an empty library project for this example. We will then create an example directory and place the two files shown later in there. Project setup will go like this:

```
$ cargo new key-exchange
```

Also, here is what `Cargo.toml` should look like:

```
[package]
name = "key-exchange"
version = "0.1.0"
authors = ["Foo <foo@bar.com>"]

[dependencies]
ring = "0.12.1"
untrusted = "0.5.1"
```

Let's look at the client first. This is heavily borrowed from the simple TCP servers we wrote in the second chapter. This is completely synchronous and blocking:

```
// ch8/key-exchange/examples/client.rs

extern crate ring;
extern crate untrusted;

use std::net::TcpStream;
use std::io::{BufRead, BufReader, Write};

use ring::{agreement, rand};
use untrusted::Input;

fn main() {
    let mut stream = TcpStream::connect("127.0.0.1:8888")
    .expect("Could not connect to server");

    let rng = rand::SystemRandom::new();

    // Generate the client's private key
    let client_private_key =
    agreement::EphemeralPrivateKey::generate(&agreement::X25519, &rng)
                        .expect("Failed to generate key");
    let mut client_public_key = [0u8; agreement::PUBLIC_KEY_MAX_LEN];
    let client_public_key = &mut
    client_public_key[..client_private_key.public_key_len()];

    // Generate the client's public key
    client_private_key.compute_public_key
    (client_public_key).expect("Failed to
    generate key");

    // Send client's public key to server
    stream.write(client_public_key)
    .expect("Failed to write to server");

    let mut buffer: Vec<u8> = Vec::new();
    let mut reader = BufReader::new(&stream);

    // Read server's public key
    reader.read_until(b'\n', &mut buffer)
    .expect("Could not read into buffer");
    let peer_public_key = Input::from(&buffer);

    println!("Received: {:?}", peer_public_key);
    // Generate shared secret key
    let res = agreement::agree_ephemeral
    (client_private_key, &agreement::X25519,
    peer_public_key,
    ring::error::Unspecified,
    |key_material| {
        let mut key = Vec::new();
        key.extend_from_slice(key_material);
        Ok(key)
    });

    println!("{:?}", res.unwrap());
}
```

We of course need the `ring` crate as an external dependency. The other crate, called `untrusted`, is a helper for taking in data from untrusted sources to `ring` as input. We then initialize ring's random number generator; this will be used to generate keys later. We then generate a private key for the client using the `generate` method. The client's public key is generated based on the private key using `compute_public_key`. At this point, the client is ready to send the public key to the server. It writes the key as a byte stream to the connection created earlier. Ideally, the server should send out its public key at this point, which the client needs to read off the same connection. This is done by the `read_until` call that places the data received in the buffer. The incoming data is then passed through the untrusted crate so that `ring` can consume it. Finally, the client-side key is generated using `agree_ephemeral`, which takes in the two keys collected (the client's private key and the server's public key), an error value, and a closure to be used on the generated byte stream. In the closure, we collect all the data in a vector and return it. The last step is to print that vector.

The server is similar and looks like the following code snippet:

```
// ch8/key-exchange/src/examples/server.rs

extern crate ring;
extern crate untrusted;

use std::net::{TcpListener, TcpStream};
use std::thread;
use std::io::{Read, Write};

use ring::{agreement, rand};
use untrusted::Input;
use ring::error::Unspecified;

fn handle_client(mut stream: TcpStream) -> Result<(), Unspecified> {
    let rng = rand::SystemRandom::new();

    // Generate server's private key
    let server_private_key =
    agreement::EphemeralPrivateKey::generate
    (&agreement::X25519, &rng)?;
    let mut server_public_key = [0u8; agreement::PUBLIC_KEY_MAX_LEN];
    let server_public_key = &mut
    server_public_key[..server_private_key.public_key_len()];

    // Generate server's public key
    server_private_key.compute_public_key(server_public_key)?;
    let mut peer_public_key_buf = [0u8; 32];

    // Read client's public key
    stream.read(&mut peer_public_key_buf).expect("Failed to read");
    let peer_public_key = Input::from(&peer_public_key_buf);

    println!("Received: {:?}", peer_public_key);

    // Send server's public key
    stream.write(&server_public_key)
    .expect("Failed to send server public key");

    // Generate shared secret key
    let res = agreement::agree_ephemeral(server_private_key,
                                         &agreement::X25519,
                                         peer_public_key,
                                         ring::error::Unspecified,
                                         |key_material| {
        let mut key = Vec::new();
        key.extend_from_slice(key_material);
        Ok(key)
    });

    println!("{:?}", res.unwrap());

    Ok(())
}

fn main() {
    let listener = TcpListener::bind("0.0.0.0:8888").expect("Could not bind");
    for stream in listener.incoming() {
        match stream {
            Err(e) => { eprintln!("failed: {}", e) }
            Ok(stream) => {
                thread::spawn(move || {
                    handle_client(stream)
                    .unwrap_or_else(|error| eprintln!(
                    "{:?}", error));
                });
            }
        }
    }
}
```

Like we did last time, we handle each client in a new thread. The semantics in the `handle_client` function is similar to that of the client; we start with generating a private and a public key. The next step is to read the public key that the client sent, followed by sending the public key of the server. Once settled, we can then use `agree_ephemeral` to generate the shared secret key, which should match the one generated on the client side.

Here is a sample run on the server side:

```
$ cargo run --example server
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/examples/server`
Received: Input { value: Slice { bytes: [60, 110, 82, 192, 131, 173, 255, 92, 134,
0, 185, 186, 87, 178, 51, 71, 136, 201, 15, 179, 204, 137, 125, 32, 87, 94, 227,
209, 47, 243, 75, 73] } }
Generated: [184, 9, 123, 15, 139, 191, 170, 9, 133, 143, 81, 45, 254, 15, 234, 12,
223, 57, 131, 145, 127, 231, 93, 101, 92, 251, 163, 179, 219, 24, 81, 111]
```

And here is one on the client side:

```
$ cargo run --example client
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/examples/client`
Received: Input { value: Slice { bytes: [83, 44, 93, 28, 132, 238, 70, 152, 163,
73, 185, 146, 142, 5, 172, 255, 219, 52, 51, 151, 99, 134, 35, 98, 154, 192, 210,
137, 141, 167, 60, 67] } }
Generated: [184, 9, 123, 15, 139, 191, 170, 9, 133, 143, 81, 45, 254, 15, 234, 12,
223, 57, 131, 145, 127, 231, 93, 101, 92, 251, 163, 179, 219, 24, 81, 111]
```

Note that the generated key is the same for both the server and the client, which is what we expect.

At the time of writing, the ring crate does not work on the latest nightly. To make it work, run these in the project directory:

```
$ rustup component add rustfmt-preview --toolchain nightly-2017-12-21
$ rustup override set nightly-2017-12-21
```

Summary

In this chapter, we took a quick look at securing communication over public networks. We started with an overview of certificates and how they are used for identifying servers on the web. We looked at using `letsencrypt` and `openssl` in Rust. We then moved on to securing Tokio applications using the same techniques. Finally, we took a quick look at doing key exchanges using the DH method.

The following section is the appendix; there, we will look at some extra crates and techniques that are becoming popular in the community.

Appendix

Rust is an open source project with a large number of contributors from all over the world. As with any such project, there are often multiple solutions to the same problems. The crate ecosystem makes this easier, since people can publish multiple crates that propose multiple ways of solving the same set of problems. This approach fosters a healthy sense of competition in true open source spirit. In this book, we have covered a number of different topics in the ecosystem. This chapter will be an addendum to those topics. We will discuss:

- Coroutine and generator-based approaches to concurrency
- The async/await abstraction
- Data parallelism
- Parsing using Pest
- Miscellaneous utilities in the ecosystem

Introduction to coroutines and generators

We looked into the Tokio ecosystem earlier. We saw how it is very common to chain futures in Tokio, yielding a larger task that can then be scheduled as necessary. In practice, the following often looks like the pseudocode:

```
fn download_parse(url: &str) {
    let task = download(url)
               .and_then(|data| parse_html(data))
               .and_then(|link| process_link(link))
               .map_err(|e| OurErr(e));
    Core.run(task);
}
```

Here, our function takes in a URL and recursively downloads raw HTML. It then parses and collects links in the document. Our task is run in an event loop. Arguably, the control flow here is harder to follow, due to all the callbacks and how they interact. This becomes more complex with larger tasks and more conditional branches.

The idea of coroutines helps us to better reason about non-linearity in these examples. A coroutine (also known as a **generator**) is a generalization of a function that can suspend or resume itself at will, yielding control to another entity. Since they do not require preemption by a supersizing entity, they are often easier to work with. Note that we always assume a cooperative model where a coroutine yields when it is waiting for a computation to finish or I/O, and we ignore cases where a coroutine might be malicious in some other way. When a coroutine starts execution again, it resumes from the point where it left off, providing a form of continuity. They generally have multiple entry and exit points.

Having set that up, a subroutine (function) becomes approximately a special case of coroutine, which has exactly one entry and exit point, and cannot be externally suspended. Computer science literature also distinguishes between generators and coroutines, arguing that the former cannot control where execution continues after they are suspended, while the later can. However, in our current context, we will use generator and coroutines interchangeably.

Coroutines can be of two broad types: stackless and stackful. Stackless coroutines do not maintain a stack when they are suspended. As a result, they cannot resume at arbitrary locations. Stackful coroutines, on the other hand, always maintain a small stack. This enables them to suspend and resume from any arbitrary point in execution. They always preserve the complete state when they suspend. Thus, from a caller's point of view, they behave like any regular function that can run independent of current execution. In practice, stackful coroutines are generally more resource heavy but easier to work with (for the caller). Note that all coroutines are resource light compared to threads and processes.

In the recent past, Rust has been experimenting with a generator implementation in the standard library. These are located in `std::ops`, and like all new features, this is behind multiple feature gates: `generators` and `generator_trait`. There are a few parts to this implementation. Firstly, there is a new yield keyword for yielding from a generator. Generators are defined by overloading the closure syntax. Secondly, there are a few items defined as follows:

```
pub trait Generator {
    type Yield;
    type Return;
    fn resume(&mut self) -> GeneratorState<Self::Yield, Self::Return>;
}
```

Since these are generators (not coroutines), they can have only one `yield` and only one `return`.

The `Generator` trait has two types: one for the `yield` case and one for the `return` case. The `resume` function resumes execution from the last point. The return value for resume is `GeneratorState`, which is an enum, and looks like this:

```
pub enum GeneratorState<Y, R> {
    Yielded(Y),
    Complete(R)
}
```

There are two variants; `Yielded` represents the variant for the `yield` statement, and `Complete` represents the variant for the `return` statement. Also, the `Yielded` variant represents that the generator can continue again from the last yield statement. `Complete` represents that the generator has finished executing.

Let's look at an example of using this to generate the Collatz sequence:

```
// appendix/collatz-generator/src/main.rs

#![feature(generators, generator_trait)]

use std::ops::{Generator, GeneratorState};
use std::env;

fn main() {
    let input = env::args()
        .nth(1)
        .expect("Please provide only one argument")
        .parse::<u64>()
        .expect("Could not convert input to integer");

    // The given expression will evaluate to our generator
    let mut generator = || {
        let end = 1u64;
        let mut current: u64 = input;
        while current != end {
            yield current;
            if current % 2 == 0 {
                current /= 2;
            } else {
                current = 3 * current + 1;
            }
        }
        return end;
    };

    loop {
        // The resume call can have two results. If we have an
        // yielded value, we print it. If we have a completed
        // value, we print it and break from the loop (this is
        // the return case)
        match generator.resume() {
            GeneratorState::Yielded(el) => println!("{}", el),
            GeneratorState::Complete(el) => {
                println!("{}", el);
                break;
            }
        }
    }
}
```

Since these are in the standard library, we do not need external dependencies. We start with the feature gates and imports. In our main function, we define a generator using the closure syntax. Notice how it captures the variable called `input` from the enclosing scope. Instead of returning the current position at the sequence, we `yield` it. And when we are done, we `return` from the generator. Now we need to call `resume` on the generator to actually run it. We do that in an infinite loop, since we do not know in advance how many times we will need to iterate. In that, we `match` on the `resume` call; in both arms, we print out the value that we have. Additionally, in the `Complete` arm, we need to `break` away from the loop.

Note that we did not use the implicit return syntax here; rather, we did an explicit `return end;`. This was not necessary, but this makes the code a little bit easier to read in this case.

This is what it produces:

```
$ cargo run 10
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/collatz-generator 10`
10
5
16
8
4
2
1
```

Generators are only available in nightly Rust, for now. Their syntax is expected to change over time, possibly by a huge amount.

How May handles coroutines

The Rust ecosystem has a number of coroutine implementations, while the core team is working on in-language implementations. Of these, one of the widely used is called *May*, which is a standalone stackful coroutine library based on generators. May strives to be user-friendly enough so that a function can be asynchronously invoked using a simple macro that takes in the function. In feature parity with the Go programming language, this macro is called `go!`. Let’s look at a small example of using May.

We will use our good ol’ friend, Collatz sequence, for this example; this will show us multiple ways of achieving the same goal. Let’s start with setting up our project using Cargo:

```
[package]
name = "collatz-may"
version = "0.1.0"
authors = ["Foo <foo@bar.com>"]

[dependencies]
may = "0.2.0"
generator = "0.6"
```

The following is the main file. There are two examples here; one uses the generator crate to yield numbers in the collatz sequence, acting as a coroutine. The other one is a regular function, which is being run as a coroutine using the `go!` macro:

```
// appendix/collatz-may/src/main.rs

#![feature(conservative_impl_trait)]

#[macro_use]
extern crate generator;
#[macro_use]
extern crate may;

use std::env;
use generator::Gn;

// Returns a generator as an iterator
fn collatz_generator(start: u64) -> impl Iterator<Item = u64> {
    Gn::new_scoped(move |mut s| {
        let end = 1u64;
        let mut current: u64 = start;
        while current != end {
            s.yield_(current);
            if current % 2 == 0 {
                current /= 2;
            } else {
                current = 3 * current + 1;
            }
        }
        s.yield_(end);
        done!();
    })
}

// A regular function returning result in a vector
fn collatz(start: u64) -> Vec<u64> {
    let end = 1u64;
    let mut current: u64 = start;
    let mut result = Vec::new();
    while current != end {
        result.push(current);
        if current % 2 == 0 {
            current /= 2;
        } else {
            current = 3 * current + 1;
        }
    }
    result.push(end);
    result
}

fn main() {
    let input = env::args()
        .nth(1)
        .expect("Please provide only one argument")
        .parse::<u64>()
        .expect("Could not convert input to integer");

    // Using the go! macro
    go!(move || {
        println!("{:?}", collatz(input));
    }).join()
        .unwrap();

    // results is a generator expression that we can
    // iterate over
    let results = collatz_generator(input);
    for result in results {
        println!("{}", result);
    }
}
```

Let's start with the `collatz_generator` function that takes in an input to start from and returns an iterator of type `u64`. To be able to specify this, we will need to activate the `conservative_impl_trait` feature. We create a scoped generator using `Gn::new_scoped` from the generator crate. It takes in a closure that actually does the computation. We yield the current value using the `yield_` function and signal the end of the computation using the `done!` macro.

Our second example is a regular function that returns a vector of numbers in the Collatz sequence. It collects intermediate results in a vector and finally returns it once the sequence reaches `1`. In our main function, we parse and sanitize input as we always do. We then call our non-generator function asynchronously in a coroutine using the `go!` macro. However, `collatz_generator` returns an iterator, which we can iterate over in a loop while printing out the numbers.

As one would expect, this is how the output looks:

```
$ cargo run 10
```

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/collatz-may 10`
[10, 5, 16, 8, 4, 2, 1]
10
5
16
8
4
2
1
```

May also includes an asynchronous network stack implementation (like Tokio), and over the last few months, it has gathered a mini ecosystem of a few dependent crates. Apart from the generator and coroutine implementations, there is an HTTP library based on those, an RPC library, and a crate that supports actor-based programming. Let’s look at an example where we write our hyper HTTP using May. Here is what a Cargo config looks like:

```
[package]
name = "may-http"
version = "0.1.0"
authors = ["Foo <foo@bar.com>"]

[dependencies]
may_minihttp = { git = "https://github.com/Xudong-Huang/may_minihttp.git" }
may = "0.2.0"
num_cpus = "1.7.0"
```

At the time of writing, `may_minihttp` is not published in `crates.io` yet, so we will need to use the repository to build. Here is the main file:

```
// appendix/may-http/src/main.rs

extern crate may;
extern crate may_minihttp;
extern crate num_cpus;

use std::io;
use may_minihttp::{HttpServer, HttpService, Request, Response};
use std::{thread, time};

fn heavy_work() -> String {
    let duration = time::Duration::from_millis(100);
    thread::sleep(duration);
    "done".to_string()
}

#[derive(Clone, Copy)]
struct Echo;

// Implementation of HttpService for our service struct Echo
// This implementation defines how content should be served
impl HttpService for Echo {
    fn call(&self, req: Request) -> io::Result<Response> {
        let mut resp = Response::new();
        match (req.method(), req.path()) {
            ("GET", "/data") => {
                let b = heavy_work();
                resp.body(&b).status_code(200, "OK");
            }
            (&_, _) => {
                resp.status_code(404, "Not found");
            }
        }
        Ok(resp)
    }
}

fn main() {
    // Set the number of IO workers to the number of cores
    may::config().set_io_workers(num_cpus::get());
    let server = HttpServer(Echo).start("0.0.0.0:3000").unwrap();
    server.join().unwrap();
}
```

This time, our code is much shorter than with Hyper. This is because May nicely abstracts away a lot of things, while letting us have a similar set of functionalities. Like the `Service` trait earlier, the `HttpService` trait is what defines a server. Functionality is defined by the `call` function. There are some minor differences in function calls and how responses are constructed. Another advantage here is that this does not expose futures and works with regular `Result`. Arguably, this model is easier to understand and follow. In the `main` function, we set the number of I/O workers to the number of cores we have. We then start the server on port `3000` and wait for it to exit. According to some rough benchmarks on the GitHub page, the `may` based HTTP server is slightly faster than a Tokio implementation.

Here is what we see upon running the server. In this particular run, it got two `GET` requests:

```
$ cargo run
   Compiling may-http v0.1.0 (file:///Users/Abhishek/Desktop/rust-
book/src/appendix/may-http)
    Finished dev [unoptimized + debuginfo] target(s) in 1.57 secs
     Running `target/debug/may-http`
Incoming request <HTTP Request GET /data>
Incoming request <HTTP Request GET /data>
^C
```

Our client side is just `curl`, and we see `done` being printed for each request. Note that since our server and the client are on the same physical box, we can use `127.0.0.1` as the server’s address. If that is not the case, the actual address should be used:

```
$ curl http://127.0.0.1:3000/data
done
```

Awaiting the future

In the last section, we saw how tasks composed of multiple futures are often difficult to write and debug. One attempt at remedying this is using a crate that wraps futures and associated error handling, yielding a more linear flow of code. This crate is called *futures-await* and is under active development.

This crate provides two primary mechanisms of dealing with futures:

- The `#[async]` attribute that can be applied to functions, marking them as asynchronous. These functions must return the `Result` of their computation as a future.
- The `await!` macro that can be used with async functions to consume the future, returning a `Result`.

Given these constructions, our earlier example download will look like this:

```
#[async]
fn download(url: &str) -> io::Result<Data> {
    ...
}

#[async]
fn parse_html(data: &Data) -> io::Result<Links> {
    ...
}

#[async]
fn process_links(links: &Links) -> io::Result<bool> {
    ...
}

#[async]
fn download_parse(url: &str) -> io::Result<bool> {
    let data = await!(download(url));
    let links = await!(parse_html(data));
    let result = await!(process_links(links));
    Ok(result)
}

fn main() {
    let task = download_parse("foo.bar");
    Core.run(task).unwrap();
}
```

This is arguably easier to read than the example with futures. Internally, the compiler translates this to code which looks like our earlier example. Additionally, since each of the steps returns a `Result`, we can use the `?` operator to nicely bubble up errors. The final task can then be run in an event loop, like always.

Let’s look at a more concrete example, rewriting our hyper server project using this crate. In this case, our Cargo setup looks like this:

```
[package]
name = "hyper-async-await"
version = "0.1.0"
authors = ["Foo <foo@bar.com>"]

[dependencies]
hyper = "0.11.7"
futures = "0.1.17"
net2 = "0.2.31"
tokio-core = "0.1.10"
num_cpus = "1.0"
futures-await = "0.1.0"
```

Here is our code, as shown in the following code snippet. Notice that we have not used types from the futures crate like last time. Instead, we have used types re-exported from futures-await, which are wrapped versions of those types:

```
// appendix/hyper-async-await/src/main.rs

#![feature(proc_macro, conservative_impl_trait, generators)]

extern crate futures_await as futures;
extern crate hyper;
extern crate net2;
extern crate tokio_core;
extern crate num_cpus;

use futures::prelude::*;
use net2::unix::UnixTcpBuilderExt;
use tokio_core::reactor::Core;
use tokio_core::net::TcpListener;
use std::{thread, time};
use std::net::SocketAddr;
use std::sync::Arc;
use hyper::{Get, StatusCode};
use hyper::header::ContentLength;
use hyper::server::{Http, Service, Request, Response};
use futures::future::FutureResult;
use std::io;

// Our blocking function that waits for a random
// amount of time and then returns a fixed string
fn heavy_work() -> String {
    let duration = time::Duration::from_millis(100);
    thread::sleep(duration);
    "done".to_string()
}

#[derive(Clone, Copy)]
struct Echo;

// Service implementation for the Echo struct
impl Service for Echo {
    type Request = Request;
    type Response = Response;
    type Error = hyper::Error;
    type Future = FutureResult<Response, hyper::Error>;

    fn call(&self, req: Request) -> Self::Future {
        futures::future::ok(match (req.method(), req.path()) {
            (&Get, "/data") => {
                let b = heavy_work().into_bytes();
                Response::new()
                    .with_header(ContentLength(b.len() as u64))
                    .with_body(b)
            }
            _ => Response::new().with_status(StatusCode::NotFound),
        })
    }
}

// Sets up everything and runs the event loop
fn serve(addr: &SocketAddr, protocol: &Http) {
    let mut core = Core::new().unwrap();
    let handle = core.handle();
    let listener = net2::TcpBuilder::new_v4()
        .unwrap()
        .reuse_port(true)
        .unwrap()
        .bind(addr)
        .unwrap()
        .listen(128)
        .unwrap();
    let listener = TcpListener::from_listener
    (listener, addr, &handle).unwrap();
    let server = async_block! {
        #[async]
        for (socket, addr) in listener.incoming() {
            protocol.bind_connection(&handle, socket, addr, Echo);
        }
        Ok::<(), io::Error>(())
    };
    core.run(server).unwrap();
}

fn start_server(num: usize, addr: &str) {
    let addr = addr.parse().unwrap();

    let protocol = Arc::new(Http::new());
    {
        for _ in 0..num - 1 {
            let protocol = Arc::clone(&protocol);
            thread::spawn(move || serve(&addr, &protocol));
        }
    }
    serve(&addr, &protocol);
}


fn main() {
    start_server(num_cpus::get(), "0.0.0.0:3000");
}
```

The `async_block!` macro takes in a closure and converts that to an `async` function. Thus, our server here is an `async` function. We also use an asynchronous `for` loop (a `for` loop marked by `#[async]`) to asynchronously iterate over the stream of connections. The rest of the code is exactly the same as last time. Running the server is simple; we will use Cargo:

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/hyper-server-faster`
```

On the client side, we can use curl:

```
$ curl http://127.0.0.1:3000/data
```

```
done$ curl http://127.0.0.1:3000/data
done$
```

At the time of writing, running this example will produce a warning about using bind_connection. Since there is no clear deprecation timeline for this API, we will ignore the warning for now.

Data parallelism

Data parallelism is a way of speeding up computation by making data a central entity. This is in contrast to the coroutine and thread-based parallelism that we have seen so far. In those cases, we first determine tasks that can be run independently. We then distribute available data to those tasks as needed. This approach is often called **task parallelism**. Our topic of discussion in this section is data parallelism. In this case, we need to figure out what parts of the input data can be used independently; then multiple tasks can be assigned to individual parts. This also conforms to the divide and conquer approach, one strong example being `mergesort`.

The Rust ecosystem has a library called **Rayon** that provides simple APIs for writing data parallel code. Let's look at a simple example of using Rayon for a binary search on a given slice. We start with setting up our project using `cargo`:

```
$ cargo new --bin rayon-search
```

Let's look at the Cargo configuration file:

```
[package]
name = "rayon-search"
version = "0.1.0"
authors = ["Foo <foo@bar.com>"]

[dependencies]
rayon = "0.9.0"
```

In our code, we have two implementations of the binary search function, both being recursive. The naive implementation is called `binary_search_recursive` and does not do any data parallelism. The other version, called `binary_search_rayon`, computes the two cases in parallel. Both functions take in a slice of type `T` that implements a number of traits. They also take an element of the same type. The functions will look for the given element in the slice and return `true` if it exists, and `false` otherwise. Let's look at the code now:

```
// appendix/rayon-search/src/main.rs

extern crate rayon;

use std::fmt::Debug;
use rayon::scope;

// Parallel binary search, searches the two halves in parallel
fn binary_search_rayon<T: Ord + Send + Copy + Sync + Debug>(src: &mut [T], el: T) -
> bool {
    src.sort();
    let mid = src.len() / 2;
    let srcmid = src[mid];
    if src.len() == 1 && src[0] != el {
        return false;
    }
    if el == srcmid {
        true
    } else {
        let mut left_result = false;
        let mut right_result = false;
        let (left, right) = src.split_at_mut(mid);
        scope(|s| if el < srcmid {
            s.spawn(|_| left_result = binary_search_rayon(left, el))
        } else {
            s.spawn(|_| right_result = binary_search_rayon(right, el))
        });
        left_result || right_result
    }
}

// Non-parallel binary search, goes like this:
// 1. Sort input
// 2. Find middle point, return if middle point is target
// 3. If not, recursively search left and right halves
fn binary_search_recursive<T: Ord + Send + Copy>(src: &mut [T], el: T) -> bool {
    src.sort();
    let mid = src.len() / 2;
    let srcmid = src[mid];
    if src.len() == 1 && src[0] != el {
        return false;
    }
    if el == srcmid {
        true
    } else {
        let (left, right) = src.split_at_mut(mid);
        if el < srcmid {
            binary_search_recursive(left, el)
        } else {
            binary_search_recursive(right, el)
        }
    }
}

fn main() {
    let mut v = vec![100, 12, 121, 1, 23, 35];
    println!("{}", binary_search_recursive(&mut v, 5));
    println!("{}", binary_search_rayon(&mut v, 5));
    println!("{}", binary_search_rayon(&mut v, 100));
}
```

In both cases, the first thing to do is to sort the input slice, since a binary search requires input to be sorted. `binary_search_recursive` is straightforward; we compute the middle point, and if the element there is the one we want, we return a `true`. We also include a case checking if there is only one element left in the slice and if that is the element we want. We return `true` or `false`, accordingly. This case forms the base case for our recursion. We can then check if our desired element is less than or greater than the current mid-point. We recursively search either side, based on that check.

The Rayon case is mostly the same, the only difference being how we recurse. We use a `scope` to spawn the two cases in parallel and collect their results. The scope takes in a closure and invokes that in reference to the named scope, `s`. The scope also ensures that each of the tasks is completed before it exits. We collect the results from each half in two variables, and finally, we return a logical OR of those, since we care about finding the element in either of the two halves. Here is how a sample run looks:

```
$ cargo run
   Compiling rayon-search v0.1.0 (file:///Users/Abhishek/Desktop/rust-
book/src/appendix/rayon-search)
    Finished dev [unoptimized + debuginfo] target(s) in 0.88 secs
     Running `target/debug/rayon-search`
false
false
true
```

Rayon also provides a parallel iterator, an iterator that has the same semantics as the one in the standard library, but where elements might be accessed in parallel. This construct is useful in situations where each unit of data can be processed completely independently, without any form of synchronization between each processing task. Let's look at an example of how to use these iterators, starting with the project setup using Cargo:

```
$ cargo new --bin rayon-parallel
```

In this case, we will compare the performance of a regular iterator to that of a parallel iterator using Rayon. For that, we will need to use the `rustc-test` crate. Here is how the Cargo setup looks:

```
[package]
name = "rayon-parallel"
version = "0.1.0"
authors = ["Foo <foo@bar.com>"]

[dependencies]
rayon = "0.9.0"
rustc-test = "0.3.0"
```

Here is the code, as shown in the following code snippet. We have two functions doing the exact same thing. Both of them receive a vector of integers, they then iterate over that vector and filter out even integers. Finally, they return squares of odd integers:

```
// appendix/rayon-parallel/src/main.rs

#![feature(test)]

extern crate rayon;
extern crate test;

use rayon::prelude::*;

// This function uses a parallel iterator to
// iterate over the input vector, filtering
// elements that are even and then squares the remaining
fn filter_parallel(src: Vec<u64>) -> Vec<u64> {
    src.par_iter().filter(|x| *x % 2 != 0)
    .map(|x| x * x)
    .collect()
}

// This function does exactly the same operation
// but uses a regular, sequential iterator
fn filter_sequential(src: Vec<u64>) -> Vec<u64> {
    src.iter().filter(|x| *x % 2 != 0)
    .map(|x| x * x)
    .collect()
}

fn main() {
    let nums_one = (1..10).collect();
    println!("{:?}", filter_sequential(nums_one));

    let nums_two = (1..10).collect();
    println!("{:?}", filter_parallel(nums_two));
}

#[cfg(test)]
mod tests {
    use super::*;
    use test::Bencher;

    #[bench]
    fn bench_filter_sequential(b: &mut Bencher) {
        b.iter(|| filter_sequential((1..1000).collect::<Vec<u64>>()));
    }

    #[bench]
    fn bench_filter_parallel(b: &mut Bencher) {
        b.iter(|| filter_parallel((1..1000).collect::<Vec<u64>>()));
    }
}
```

We start with importing everything from Rayon. In `filter_parallel`, we use `par_iter` to get a parallel iterator. `filter_sequential` is the same, the only difference being that it uses the `iter` function to get a regular iterator. In our main function, we create two sequences and pass those to our functions while printing the outputs. Here is what we should see:

```
$ cargo run
   Compiling rayon-parallel v0.1.0 (file:///Users/Abhishek/Desktop/rust-
book/src/appendix/rayon-parallel)
    Finished dev [unoptimized + debuginfo] target(s) in 1.65 secs
     Running `target/debug/rayon-parallel`
[1, 9, 25, 49, 81]
[1, 9, 25, 49, 81]
```

Not surprisingly, both return the same result. The most important part of this example is the benchmark. For this to work, we will need to activate the test feature using `#![feature(test)]` and declare a new test module. In that, we import everything from the top-level module, which is the main file in this case. We also import `test::Bencher`, which will be used for running the benchmarks. The benchmarks are defined by the `#[bench]` attributes, which are applied to functions that take in an object which is a mutable reference to a `Bencher` type. We pass the functions that we need to benchmark to the bencher, which takes care of running those and printing results.

The benchmarks can be run using Cargo:

```
$ cargo bench
    Finished release [optimized] target(s) in 0.0 secs
```

```
     Running target/release/deps/rayon_parallel-333850e4b1422ead

running 2 tests
test tests::bench_filter_parallel ... bench: 92,630 ns/iter (+/- 4,942)
test tests::bench_filter_sequential ... bench: 1,587 ns/iter (+/- 269)

test result: ok. 0 passed; 0 failed; 0 ignored; 2 measured; 0 filtered out
```

This output shows the two functions and how much time it took to execute an iteration of each. The number in the bracket indicates the confidence interval for the given measurement. While the confidence interval for the parallel version is larger than the non-parallel one, it does perform 58 times more iterations than the non-parallel one. Thus, the parallel version is considerably faster.

Parsing using Pest

We studied different parsing techniques in Chapter 4, *Data serialization, De-Serialization, and Parsing*. We looked at using parser combinators using Nom, building a large parser from smaller parts. There is a completely different way of solving the same problem of parsing textual data, using **Parsing Expression Grammar (PEG)**. A PEG is a formal grammar that defines how a parser should behave. Thus, it includes a finite set of rules, from basic tokens to more complex structures. A library that can take in such grammar to produce a functional parser is Pest. Let's look at an example of rewriting our HTTP parsing example from Chapter 4, *Data Serialization, De-Serialization, and Parsing*, using Pest. Start with the Cargo project set up:

```
$ cargo new --bin pest-example
```

Like always, we will need to declare dependency on Pest components like this:

```
[package]
name = "pest-example"
version = "0.1.0"
authors = ["Foo <foo@bar.com>"]

[dependencies]
pest = "^1.0"
pest_derive = "^1.0"
```

The next step is to define our grammar, which is a linear collection of parsing rules. Like we did previously, we are interested in parsing `HTTP GET` or `POST` requests. Here is what the grammar looks like:

```
// src/appendix/pest-example/src/grammar.pest

newline = { "\n" }
carriage_return = { "\r" }
space = { " " }
get = { "GET" }
post = { "POST" }
sep = { "/" }
version = { "HTTP/1.1" }
chars = { 'a'..'z' | 'A'..'Z' }
request = { get | post }

ident_list = _{ request ~ space ~ sep ~ chars+ ~ sep ~ space ~ version ~
carriage_return ~ newline }
```

The first step is to define literal rules that are to be matched verbatim. These correspond to the leaf parsers in Nom. We define literals for newline, carriage return, space, the two request strings, the separator, and the fixed string for the HTTP version. We also define `request` as the logical OR of the two request literals. A list of characters is the logical OR of all lowercase letters and all uppercase letters. At this point, we have all we need to define the final rule. That is given by `ident_list` and consists of the request, followed by a single space, then a separator; then we indicate that our parser should accept one or more characters using `*`. The next valid input is again a separator, followed by a single space, the version string, a carriage return, and finally, a newline. Note that consecutive inputs are separated by the `~` character. The leading `_` in front indicates that this is a silent rule and should only be used at the top level, as we will see shortly.

The main file looks like this:

```
// src/appendix/pest-example/src/main.rs

extern crate pest;
#[macro_use]
extern crate pest_derive;

use pest::Parser;

#[derive(Parser)]
#[grammar = "grammar.pest"]
// Unit struct that will be used as the parser
struct RequestParser;

fn main() {
    let get = RequestParser::parse(Rule::ident_list, "GET /foobar/ HTTP/1.1\r\n")
        .unwrap_or_else(|e| panic!("{}", e));
    for pair in get {
        println!("Rule: {:?}", pair.as_rule());
        println!("Span: {:?}", pair.clone().into_span());
        println!("Text: {}", pair.clone().into_span().as_str());
    }

    let _ = RequestParser::parse(Rule::ident_list, "WRONG /foobar/
    HTTP/1.1\r\n")
        .unwrap_or_else(|e| panic!("{}", e));
}
```

The code is simple; the library provides one basic trait, called `Parser`. This can be custom derived for unit structures to produce a functional parser based on the grammar file, as input using an attribute called `grammar`. Notably, this library uses custom derivatives and custom attributes very efficiently to provide a nicer user experience. In our case, the unit structure is called `RequestParser`, which implements the `parse` method. In our main function, we call the that method, passing in the rule from which parsing should start (in our case, that happens to be the final top-level rule, called `ident_list`) and a string to parse. Errors are handled by aborting, since there is not much point in continuing if parsing failed.

Having set this structure up, we attempt to parse two strings. The first one is a normal HTTP request. The `parse` method returns an iterator over the stream of parsed tokens. We loop over them and print out the name of the rule that token matched with, the span in the input that has the token, and the literal text in that token. Later, we attempt to parse a string which does not have a valid HTTP request. Here is the output:

```
$ cargo run
   Compiling pest-example v0.1.0 (file:///Users/Abhishek/Desktop/rust-
book/src/chapter4/pest-example)
    Finished dev [unoptimized + debuginfo] target(s) in 1.0 secs
     Running `target/debug/pest-example`
Rule: request
Span: Span { start: 0, end: 3 }
Text: GET
Rule: space
Span: Span { start: 3, end: 4 }
Text:
Rule: sep
Span: Span { start: 4, end: 5 }
Text: /
Rule: chars
Span: Span { start: 5, end: 6 }
Text: f
Rule: chars
Span: Span { start: 6, end: 7 }
Text: o
Rule: chars
Span: Span { start: 7, end: 8 }
Text: o
Rule: chars
Span: Span { start: 8, end: 9 }
Text: b
Rule: chars
Span: Span { start: 9, end: 10 }
Text: a
Rule: chars
Span: Span { start: 10, end: 11 }
Text: r
Rule: sep
Span: Span { start: 11, end: 12 }
Text: /
Rule: space
Span: Span { start: 12, end: 13 }
Text:
Rule: version
Span: Span { start: 13, end: 21 }
Text: HTTP/1.1
Rule: carriage_return
Span: Span { start: 21, end: 22 }
Text:
Rule: newline
Span: Span { start: 22, end: 23 }
Text:

thread 'main' panicked at ' --> 1:1
  |
1 | WRONG /foobar/ HTTP/1.1
  | ^---
  |
  = expected request', src/main.rs:21:29
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

The first thing to notice is that parsing a wrong HTTP request failed. The error message is nice and clear, explaining exactly where it failed to parse. The correct request did parse successfully and printed out the tokens and all required details to further process those.

Miscellaneous utilities

In C and C++, a common workflow is to define a set of bits as flags. They are generally defined at powers of two, so the first flag will have the decimal value of one, the second one will have two, and so on. This helps in performing logical combinations of those flags. The Rust ecosystem has a crate to facilitate the same workflow. Let’s look at an example of using the bitflags crate for working with flags. Let’s start with initializing an empty project using Cargo:

```
$ cargo new --bin bitflags-example
```

We will set up our project manifest to add `bitflags` as a dependency:

```
$ cat Cargo.toml
[package]
name = "bitflags-example"
version = "0.1.0"
authors = ["Foo <foo@bar.com>"]

[dependencies]
bitflags = "1.0"
```

When all of that is ready, our main file will look like this:

```
// appendix/bitflags-example/src/main.rs

#[macro_use]
extern crate bitflags;

// This macro defines a struct that holds our
// flags. This also defines a number of convinience
// methods on the struct.
bitflags! {
    struct Flags: u32 {
        const X = 0b00000001;
        const Y = 0b00000010;
    }
}

// We define a custom trait to print a
// given bitflag in decimal
pub trait Format {
    fn decimal(&self);
}

// We implement our trait for the Flags struct
// which is defined by the bitflags! macro
impl Format for Flags {
    fn decimal(&self) {
        println!("Decimal: {}", self.bits());
    }
}

// Main driver function
fn main() {
    // A logical OR of two given bitflags
    let flags = Flags::X | Flags::Y;
    // Prints the decimal representation of
    // the logical OR
    flags.decimal();

    // Same as before
    (Flags::X | Flags::Y).decimal();

    // Prints one individual flag in decimal
    (Flags::Y).decimal();

    // Examples of the convenience methods mentioned
    // earlier. The all method gets the current state
    // as a human readable string. The contain method
    // returns a bool indicating if the given bitflag
    // has the other flag.
    println!("Current state: {:?}", Flags::all());
    println!("Contains X? {:?}", flags.contains(Flags::X));
}
```

We import our dependencies and then use the `bitflags!` macro to define a number of flags, as mentioned before, we set their values in powers of two. We also demonstrate attaching additional properties to the `bitflags` using the trait system. For this, we have a custom trait called `Format` that prints a given input as a decimal. The conversion is achieved using the `bits()` method that returns all the bits in the given input. The next step is to implement our trait for the `Flags` structure.

Once we have done that, we move on to the `main` function; in there, we construct a logical OR of two given flags. We use the `decimal` method to print out representations of the bitflags and ensure they are equal. Finally, we use the `all` function to display a human readable form of the flags. Here, the `contains` function returns `true`, since the flag `X` is indeed in the logical OR of `X` and `Y`.

Here is what we should see upon running this:

```
$ cargo run
   Compiling bitflags-example v0.1.0 (file:///Users/Abhishek/Desktop/rust-
book/src/appendix/bitflags-example)
    Finished dev [unoptimized + debuginfo] target(s) in 0.48 secs
     Running `target/debug/bitflags-example`
Decimal: 3
Decimal: 3
Decimal: 2
Current state: X | Y
Contains X? true
```

The values of individual flags should always be an integer type.

Another useful utility for network programming is the `url` crate. This crate provides a number of functionalities to parse parts of URLs, from links to web pages to relative addresses. Let’s look at a very simple example, starting with the project setup:

```
$ cargo new --bin url-example
```

The Cargo manifest should look like this:

```
$ cat Cargo.toml
[package]
name = "url-example"
version = "0.1.0"
authors = ["Foo <foo@bar.com>"]

[dependencies]
url = "1.6.0"
```

Let’s look at the main file. In this relatively short example, we are parsing a GitLab URL to extract a few important pieces of information:

```
// appendix/url-example/src/main.rs

extern crate url;

use url::Url;

fn main() {
    // We are parsing a gitlab URL. This one happens to be using
    // git and https, a given username/password and a fragment
    // pointing to one line in the source
    let url = Url::parse("git+https://foo:bar@gitlab.com/gitlab-org/gitlab-
ce/blob/master/config/routes/development.rb#L8").unwrap();

    // Prints the scheme
    println!("Scheme: {}", url.scheme());

    // Prints the username
    println!("Username: {}", url.username());

    // Prints the password
    println!("Password: {}", url.password().unwrap());

    // Prints the fragment (everything after the #)
    println!("Fragment: {}", url.fragment().unwrap());

    // Prints the host
    println!("Host: {:?}", url.host().unwrap());
}
```

This example URL contains a fragment, pointing to a one-line number in a file. The scheme is set to git, and there is a username and password set for HTTP based authentication. The URL crate provides a method call `parse` that takes in a string and returns a struct that has all required information. We can subsequently call individual methods on that variable to print out relevant information.

Here is what this code outputs, matching our expectation:

```
$ cargo run
   Compiling url-example v0.1.0 (file:///Users/Abhishek/Desktop/rust-
book/src/appendix/url-example)
    Finished dev [unoptimized + debuginfo] target(s) in 0.58 secs
     Running `target/debug/url-example`
Scheme: git+https
```

```
Username: foo
Password: bar
Fragment: L8
Host: Domain("gitlab.com")
```

Summary

This last chapter was a mix of a number of topics which we did not consider to be mainstream enough for other chapters. But we should remember that, in a large ecosystem like Rust has, things evolve very quickly. So some ideas which may not be mainstream today, might just be adopted in the community tomorrow.

Overall, Rust is a wonderful language with enormous potential. We earnestly hope that this book helped the reader get a sense of how to harness its power for network programming.

- Understand system programming language problems and see how Rust provides unique solutions
- Get to know the core concepts of Rust to develop fast and safe applications
- Explore the possibility of integrating Rust units into existing applications to make them more efficient
- Achieve better parallelism, security, and performance
- Explore ways to package your Rust application and ship it for deployment in a production environment
- Discover how to build web applications and services using Rust to provide high-performance to the end user