Go to next chapter ›

8 Networking

175

This chapter covers

- Implementing a networking stack
- Handling multiple error types within local scope
- When to use trait objects
- Implementing state machines in Rust

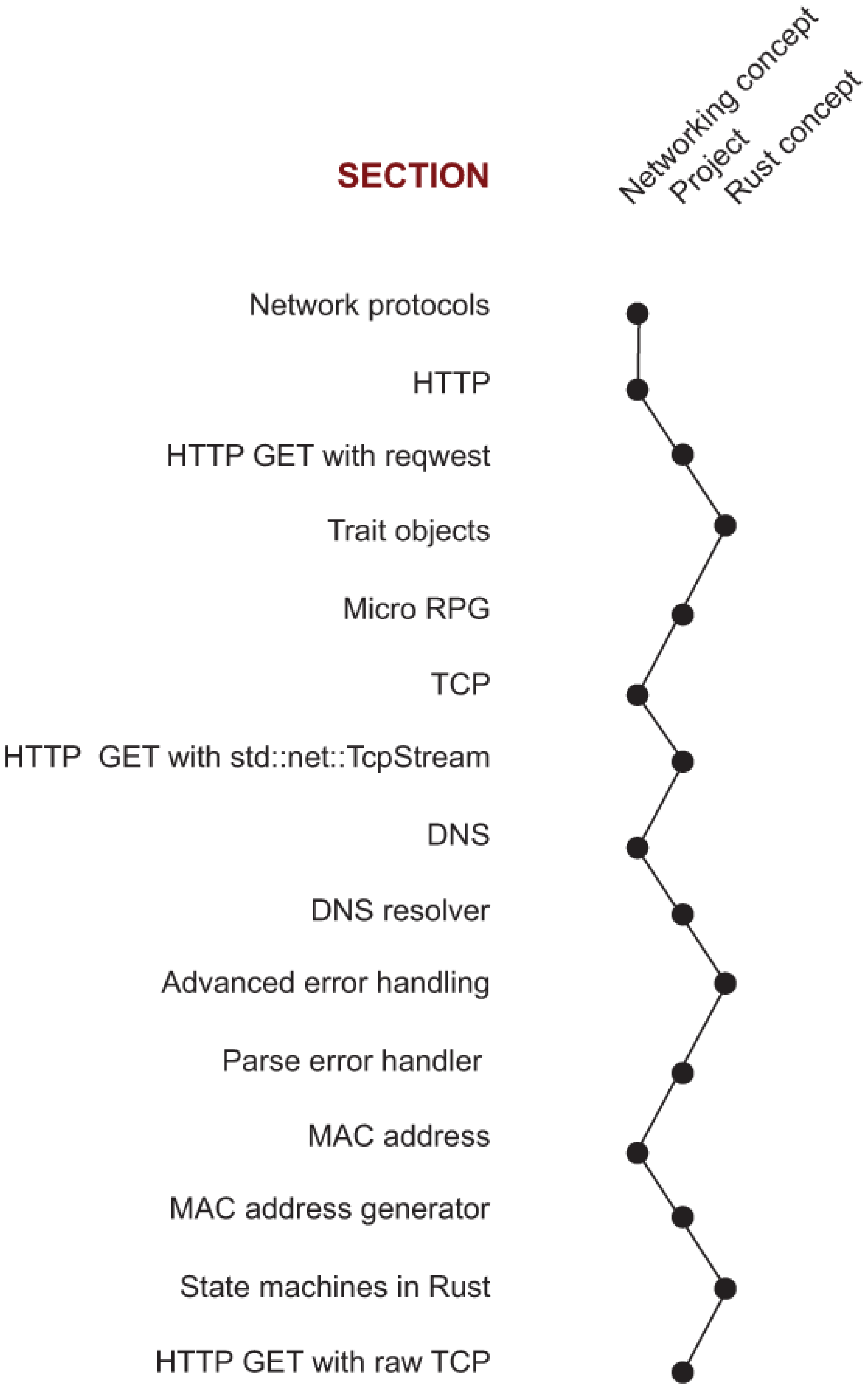
This chapter describes how to make HTTP requests multiple times, stripping away a layer of abstraction each time. We start by using a user-friendly library, then boil that away until we're left with manipulating raw TCP packets. When we're finished, you'll be able to distinguish an IP address from a MAC address. And you'll learn why we went straight from IPv4 to IPv6.

You'll also learn lots of Rust in this chapter, most of it related to advanced error handling techniques that become essential for incorporating upstream crates. Several pages are devoted to error handling. This includes a thorough introduction to trait objects.

Networking is a difficult subject to cover in a single chapter. Each layer is a fractal of complexity. Networking experts will hopefully overlook my lack of depth in treating such a diverse topic.

Figure 8.1 provides an overview of the topics that the chapter covers. Some of the projects that we cover include implementing DNS resolution and generating standards-compliant MAC addresses, including multiple examples of generating HTTP requests. A bit of a role-playing game is added for light relief.

Figure 8.1 Networking chapter map. The chapter incorporates a healthy mix of theory and practical exercises.



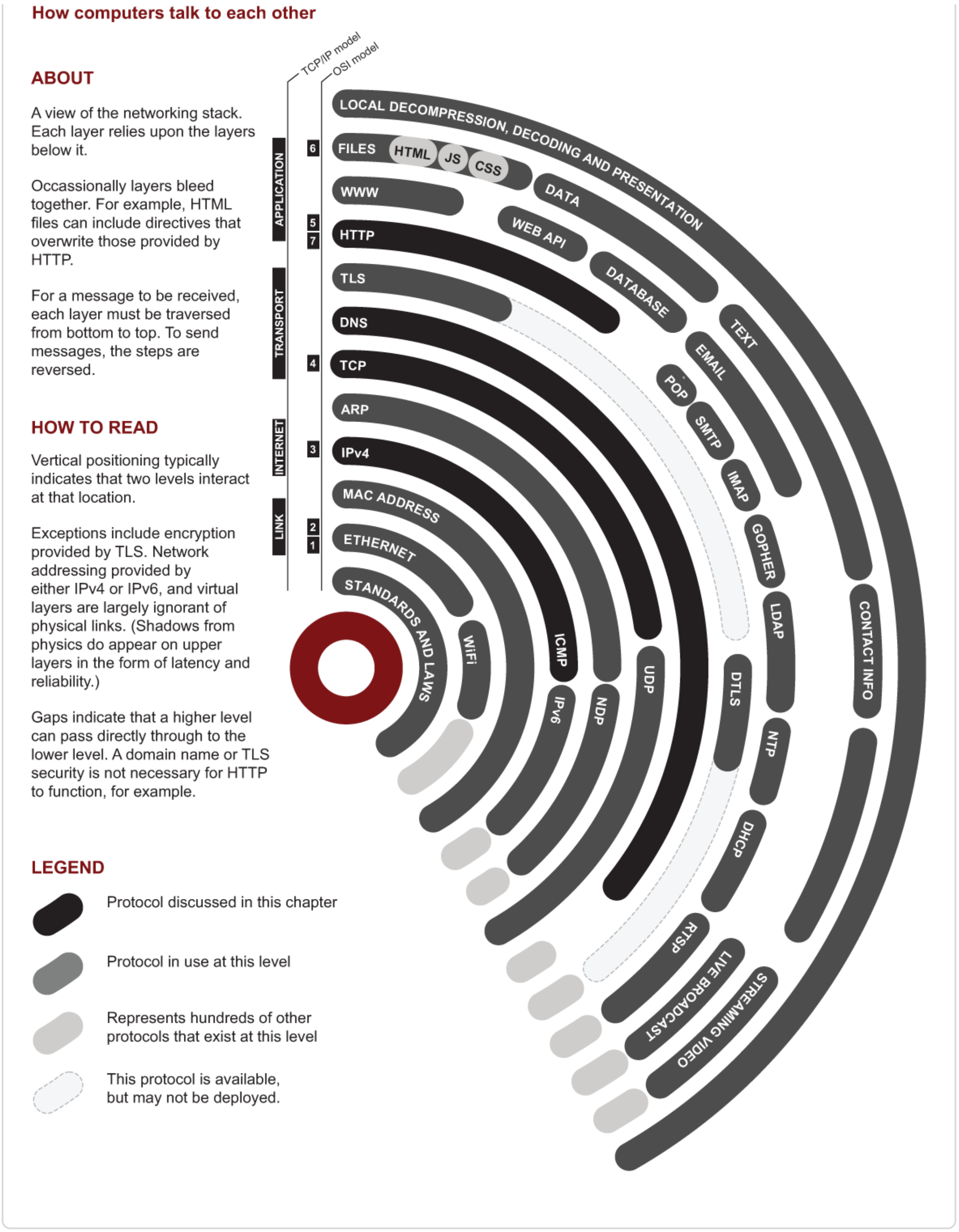
livebook features:

8.1 All of networking in seven paragraphs

Rather than trying to learn the whole networking stack, let's focus on something that's of practical use. Most readers of this book will have encountered web programming. Most web programming involves interacting with some sort of framework. Let's look there.

HTTP is the protocol that web frameworks understand. Learning more about HTTP enables us to extract the most performance out of our web frameworks. It can also help us to more easily diagnose any problems that occur. Figure 8.2 shows networking protocols for content delivery over the internet.

Figure 8.2 Several layers of networking protocols involved with delivering content over the internet. The figure compares some common models, including the seven-layer OSI model and the four-layer TCP/IP model.


How computers talk to each other

ABOUT

A view of the networking stack. Each layer relies upon the layers below it.

Occassionally layers bleed together. For example, HTML files can include directives that overwrite those provided by HTTP.

For a message to be received, each layer must be traversed from bottom to top. To send messages, the steps are reversed.

HOW TO READ

Vertical positioning typically indicates that two levels interact at that location.

Exceptions include encryption provided by TLS. Network addressing provided by either IPv4 or IPv6, and virtual layers are largely ignorant of physical links. (Shadows from physics do appear on upper layers in the form of latency and reliability.)

Gaps indicate that a higher level can pass directly through to the lower level. A domain name or TLS security is not necessary for HTTP to function, for example.

LEGEND

Protocol discussed in this chapter

Protocol in use at this level

Represents hundreds of other protocols that exist at this level

This protocol is available, but may not be deployed.

Networking is comprised of layers. If you’re new to the field, don’t be intimidated by a flood of acronyms. The most important thing to remember is that lower levels are unaware of what’s happening above them, and higher levels are agnostic to what’s happening below them. Lower levels receive a stream of bytes and pass it on. Higher levels don’t care how messages are sent; they just want them sent.

Let’s consider one example: HTTP. HTTP is known as an *application-level protocol*. Its job is to transport content like HTML, CSS, JavaScript, WebAssembly modules, images, video, and other formats.

These formats often include other embedded formats via compression and encoding standards. HTTP itself often redundantly includes information provided by one of the layers below it, TCP. Between HTTP and TCP sits TLS. TLS (Transport Layer Security), which has replaced SSL (Secure Sockets Layer), adds the S to HTTPS.

TLS provides encrypted messaging over an unencrypted connection. TLS is implemented on top of TCP. TCP sits upon many other protocols. These go all the way down to specifying how voltages should be interpreted as 0s and 1s. And yet, as complicated as this story is so far, it gets worse. These layers, as you have probably seen in your dealings with those as a computer user, bleed together like watercolor paint.

HTML includes a mechanism to supplement or overwrite directives omitted or specified within HTTP: the `<meta>` tag's `http-equiv` attribute. HTTP can make adjustments downwards to TCP. The "Connection: keep-alive" HTTP header instructs TCP to maintain its connection after this HTTP message has been received. These sorts of interactions occur all through the stack. Figure 8.2 provides one view of the networking stack. It is more complicated than most attempts. And even that complicated picture is highly simplified.

Despite all of that, we're going to try to implement as many layers as possible within a single chapter. By the end of it, you will be sending HTTP requests with a virtual networking device and a minimal TCP implementation that you created yourself, using a DNS resolver that you also created yourself.

8.2 Generating an HTTP GET request with reqwest

Our first implementation will be with a high-level library that is focused on HTTP. We'll use the reqwest library because its focus is primarily on making it easy for Rust programmers to create an HTTP request.

Although it's the shortest, the reqwest implementation is the most feature-complete. As well as being able to correctly interpret HTTP headers, it also handles cases like content redirects. Most importantly, it understands how to handle TLS properly.

In addition to expanded networking capabilities, reqwest also validates the content's encoding and ensures that it is sent to your application as a valid `String`. None of our lower-level implementations do any of that. The following shows the project structure for listing 8.2:

```
ch8-simple/
├── src
│   └── main.rs
└── Cargo.toml
```

copy

The following listing shows the metadata for listing 8.2. The source code for this listing is in ch8/ch8-simple/Cargo.toml.

Listing 8.1 Crate metadata for listing 8.2

```
[package]
name = "ch8-simple"
version = "0.1.0"
authors = ["Tim McNamara <author@rustinaction.com>"]
edition = "2018"

[dependencies]
reqwest = "0.9"
```

copy

The following listing illustrates how to make an HTTP request with the reqwest library. You'll find the source in ch8/ch8-simple/src/main.rs.

Listing 8.2 Making an HTTP request with `reqwest`

```rust
use std::error::Error;

use reqwest;

fn main() -> Result<(), Box<dyn Error>> {                    1
  let url = "http:/ /www.rustinaction.com/";
  let mut response = reqwest::get(url)?;

  let content = response.text()?;
  print!("{}", content);

  Ok(())
}
```

copy

If you've ever done any web programming, listing 8.2 should be straightforward. `reqwest::get()` issues an HTTP GET request to the URL represented by `url`. The `response` variable holds a struct representing the server's response. The `response .text()` method returns a `Result` that provides access to the HTTP body after validating that the contents are a legal `String`.

One question, though: What on earth is the error side of the `Result` return type `Box<dyn std::error::Error>`? This is an example of a trait object that enables Rust to support polymorphism at runtime. *Trait objects* are proxies for concrete types. The syntax `Box<dyn std::error::Error>` means a `Box` (a pointer) to any type that implements `std::error:Error`'s.

Using a library that knows about HTTP allows our programs to omit many details. For example

- *Knowing when to close the connection.* HTTP has rules for telling each of the parties when the connection ends. This isn't available to us when manually making requests. Instead, we keep the connection open for as long as possible and hope that the server will close.
- *Converting the byte stream to content.* Rules for translating the message body from `[u8]` to `String` (or perhaps an image, video, or some other content) are handled as part of the protocol. This can be tedious to handle manually as HTTP allows content to be compressed into several methods and encoded into several plain text formats.
- *Inserting or omitting port numbers.* HTTP defaults to port 80. A library that is tailored for HTTP, such as reqwest, allows you to omit port numbers. When we're building requests by hand with generic TCP crates, however, we need to be explicit.
- *Resolving the IP addresses.* The TCP protocol doesn't actually know about domain names like www.rustinaction.com, for example. The library resolves the IP address for www.rustinaction.com on our behalf.

8.3 Trait objects

This section describes trait objects in detail. You will also develop the world's next best-selling fantasy role-playing game—the rpg project. If you would like to focus on networking, feel free to skip ahead to section 8.4.

There is a reasonable amount of jargon in the next several paragraphs. Brace yourself. You'll do fine. Let's start by introducing trait objects by what they achieve and what they do, rather than focusing on what they are.

8.3.1 What do trait objects enable?

While trait objects have several uses, they are immediately helpful by allowing you to create containers of multiple types. Although players of our role-playing game can choose different races, and each race is defined in its own `struct`, you'll want to treat those as a single type. A `Vec<T>` won't work here because we can't easily have types `T`, `U`, and `V` wedged into `Vec<T>` without introducing some type of wrapper object.

8.3.2 What is a trait object?

Trait objects add a form of ***polymorphism***—the ability to share an interface between types—to Rust via ***dynamic dispatch***. Trait objects are similar to generic objects. Generics offer polymorphism via ***static dispatch***. Choosing between generics and type objects typically involves a trade off between disk space and time:

- Generics use more disk space with faster runtimes.
- Trait objects use less disk space but incur a small runtime overhead caused by pointer indirection.

Trait objects are ***dynamically-sized types***, which means that these are always seen in the wild behind a pointer. Trait objects appear in three forms: `&dyn Trait`, `&mut dyn Trait`, and `Box<dyn Trait>`.[1] The primary difference between the three forms is that `Box<dyn Trait>` is an owned trait object, whereas the other two are borrowed.

[1] In old Rust code, you may see `&Trait`, and `Box<Trait>`. While legal syntax, these are officially deprecated. Adding `dyn` keyword is strongly encouraged.

8.3.3 Creating a tiny role-playing game: The rpg project

Listing 8.4 is the start of our game. Characters in the game can be one of three races: humans, elves, and dwarves. These are represented by the `Human`, `Elf`, and `Dwarf` structs, respectively.

Characters interact with things. Things are represented by the `Thing` type.[2] `Thing` is an enum that currently represents swords and trinkets. There's only one form of interaction right now: enchantment. Enchanting a thing involves calling the `enchant()` method:

[2] Naming is hard.

```
character.enchant(&mut thing)
```

When enchantment is successful, `thing` glows brightly. When a mistake occurs, `thing` is transformed into a trinket. Within listing 8.4, we create a party of characters with the following syntax:

```
let d = Dwarf {};
let e = Elf {};
let h = Human {};

let party: Vec<&dyn Enchanter> = vec![&d, &h, &e];     #1
```

Casting the spell involves choosing a spellcaster. We make use of the rand crate for that:

```
let spellcaster = party.choose(&mut rand::thread_rng()).unwrap
spellcaster.enchant(&mut it)
```

copy

The `choose()` method originates from the `rand::seq::SliceRandom` trait that is brought into scope in listing 8.4. One of the party is chosen at random. The party then attempts to enchant the object `it`. Compiling and running listing 8.4 results in a variation of this:

```
$ cargo run
...
   Compiling rpg v0.1.0 (/rust-in-action/code/ch8/ch8-rpg)
    Finished dev [unoptimized + debuginfo] target(s) in 2.13s
     Running `target/debug/rpg`
Human mutters incoherently. The Sword glows brightly.

$ target/debug/rpg
Elf mutters incoherently. The Sword fizzes, then turns into a wor
```

copy

The following listing shows the metadata for our fantasy role-playing game. The source code for the rpg project is in ch8/ch8-rpg/Cargo.toml.

Listing 8.3 Crate metadata for the rpg project

```
[package]
name = "rpg"
version = "0.1.0"
authors = ["Tim McNamara <author@rustinaction.com>"]
edition = "2018"
```

```
[dependencies]
rand = "0.7"
```

copy

Listing 8.4 provides an example of using a trait object to enable a container to hold several types. You'll find its source in ch8/ch8-rpg/src/main.rs.

Listing 8.4 Using the trait object `&dyn Enchanter`

```
use rand;
use rand::seq::SliceRandom;
use rand::Rng;

#[derive(Debug)]
struct Dwarf {}

#[derive(Debug)]
struct Elf {}

#[derive(Debug)]
struct Human {}

#[derive(Debug)]
enum Thing {
  Sword,
  Trinket,
}

trait Enchanter: std::fmt::Debug {
  fn competency(&self) -> f64;

  fn enchant(&self, thing: &mut Thing) {
    let probability_of_success = self.competency();
    let spell_is_successful = rand::thread_rng()
      .gen_bool(probability_of_success);

    print!("{:?} mutters incoherently. ", self);
    if spell_is_successful {
      println!("The {:?} glows brightly.", thing);
    } else {
      println!("The {:?} fizzes, \
             then turns into a worthless trinket.", thing);
      *thing = Thing::Trinket {};
    }
  }
}

impl Enchanter for Dwarf {
  fn competency(&self) -> f64 {
    0.5
  }
}
impl Enchanter for Elf {
  fn competency(&self) -> f64 {
    0.95
  }
}
impl Enchanter for Human {
  fn competency(&self) -> f64 {
    0.8
  }
}

fn main() {
  let mut it = Thing::Sword;

  let d = Dwarf {};
  let e = Elf {};
  let h = Human {};

  let party: Vec<&dyn Enchanter> = vec![&d, &h, &e];
  let spellcaster = party.choose(&mut rand::thread_rng()).unwra

  spellcaster.enchant(&mut it);
}
```

copy

Trait objects are a powerful construct in the language. In a sense, they provide a way to navigate Rust's rigid type system. As you learn about this feature in more detail, you will encounter some jargon. For example, trait objects are a form of ***type erasure***. The compiler does not have access to the original type during the call to `enchant()`.

TRAIT VS. TYPE

One of the frustrating things about Rust's syntax for beginners is that trait objects and type parameters look similar. But types and traits are used in different places. For example, consider these two lines:

```
use rand::Rng;
use rand::rngs::ThreadRng;
```

copy

Although these both have something to do with random number generators, they're quite different. `rand::Rng` is a trait; `rand::rngs::ThreadRng` is a struct. Trait objects make this distinction harder.

When used as a function argument and in similar places, the form `&dyn Rng` is a reference to something that implements the `Rng` trait, whereas `&ThreadRng` is a reference to a value of `ThreadRng`. With time, the distinction between traits and types becomes easier to grasp. Here's some common use cases for trait objects:

- Creating collections of heterogeneous objects.
- Returning a value. Trait objects enable functions to return multiple concrete types.
- Supporting dynamic dispatch, whereby the function that is called is determined at runtime rather than at compile time.

Before the Rust 2018 edition, the situation was even more confusing. The `dyn` keyword did not exist. This meant that context was needed to decide between `&Rng` and `&ThreadRng`.

Trait objects are not objects in the sense that an object-oriented programmer would understand. They're perhaps closer to a mixin class. Trait objects don't exist on their own; they are agents of some other type.

An alternative analogy would be a singleton object that is delegated with some authority by another concrete type. In listing 8.4, the `&Enchanter` is delegated to act on behalf of three concrete types.

view how

You missed out on some activities - why not try them now?

8.4 TCP

Dropping down from HTTP, we encounter TCP (Transmission Control Protocol). Rust's standard library provides us with cross-platform tools for making TCP requests. Let's use those. The file structure for listing 8.6, which creates an HTTP GET request, is provided here:

```
ch8-stdlib
├── src
│   └── main.rs
└── Cargo.toml
```

copy

The following listing shows the metadata for listing 8.6. You'll find the source for this listing in ch8/ch8-stdlib/Cargo.toml.

Listing 8.5 Project metadata for listing 8.6

```toml
[package]
name = "ch8-stdlib"
version = "0.1.0"
authors = ["Tim McNamara <author@rustinaction.com>"]
edition = "2018"

[dependencies]
```

copy

The next listing shows how to use the Rust standard library to construct an HTTP GET request with `std::net::TcpStream`. The source for this listing is in ch8/ch8-stdlib/src/main.rs.

Listing 8.6 Constructing an HTTP GET request

```
1  1 use std::io::prelude::*;
2   2 use std::net::TcpStream;
3   3
4   4 fn main() -> std::io::Result<()> {
5   5   let host = "www.rustinaction.com:80";                    1
6   6
7   7   let mut conn =
8   8     TcpStream::connect(host)?;
9   9
10 10   conn.write_all(b"GET / HTTP/1.0")?;
11 11   conn.write_all(b"\r\n")?;                                2
12 12
13 13   conn.write_all(b"Host: www.rustinaction.com")?;
14 14   conn.write_all(b"\r\n\r\n")?;                            3
15 15
16 16   std::io::copy(
17 17     &mut conn,
18 18     &mut std::io::stdout()                                4
19 19   )?;
20 20
21 21   Ok(())
22 22 }
```

copy

Some remarks about listing 8.6:

- On line 10, we specify HTTP 1.0. Using this version of HTTP ensures that the connection is closed when the server sends its response. HTTP 1.0, however, does not support "keep alive" requests. Specifying HTTP 1.1 actually confuses this code as the server will refuse to close the connection until it has received another request, and the client will never send one.
- On line 13, we include the hostname. This may feel redundant given that we used that exact hostname when we connected on lines 7–8. However, one should remembers that the connection is established over IP, which does not have host names. When `TcpStream::connect()` connects to the server, it only uses an IP address. Adding the Host

HTTP header allows us to inject that information back into the context.

You missed out on some activities – why not try them now?

8.4.1 What is a port number?

Port numbers are purely virtual. They are simply `u16` values. Port numbers allow a single IP address to host multiple services.

8.4.2 Converting a hostname to an IP address

So far, we've provided the hostname www.rustinaction.com to Rust. But to send messages over the internet, the IP (internet protocol) address is required. TCP knows nothing about domain names. To convert a domain name to an IP address, we rely on the Domain Name System (DNS) and its process called ***domain name resolution***.

We're able to resolve names by asking a server, which can recursively ask other servers. DNS requests can be made over TCP, including encryption with TLS, but are also sent over UDP (User Datagram Protocol). We'll use DNS here because it's more useful for learning purposes.

To explain how the translation from a domain name to an IP address works, we'll create a small application that does the translation. We'll call the application ***resolve***. You'll find its source code in listing 8.9. The application makes use of public DNS services, but you can easily add your own with the `-s` argument.

PUBLIC DNS PROVIDERS

At the time of writing, several companies provide DNS servers for public use. Any of the IP addresses listed here should offer roughly equivalent service:

- 1.1.1.1 and 1.0.0.1 by Cloudflare
- 8.8.8.8 and 8.8.4.4. by Google

- 9.9.9.9 by Quad9 (founded by IBM)
- .6.64.6 and 64.6.65.6 by VeriSign

Our resolve application only understands a small portion of DNS protocol, but that portion is sufficient for our purposes. The project makes use of an external crate, trust-dns, to perform the hard work. The trust-dns crate implements RFC 1035, which defines DNS and several later RFCs quite faithfully using terminology derived from it. Table 8.1 outlines some of the terms that are useful to understand.

Table 8.1 Terms that are used in RFC 1035, the trust_dns crate, and listing 8.9, and how these interlink (view table figure)

Term	Definition	Representation in code

Domain name	A domain name is almost what you probably think of when you use the term ***domain name*** in your everyday language.
	
	The technical definition includes some special cases such as the ***root*** domain, which is encoded as a single dot, and domain names that need to be case-insensitive.
		`Defined in trust_dns::domain::Name`
		`pub struct Name {`
		`    is_fqdn: bool,       ①`
		`    labels: Vec<Label>,`
		`}`
		① fqdn stands for fully-qualified domain name.

Message	A message is a container for both requests	`Defined in trust_dns::domain::Name`
		`struct Message {`
		`  header: Header,`
		`  queries: Vec<Query>,`
		`  answers: Vec<Record>,`
		`  name_servers: Vec<Record>,`

	to DNS servers (called ***queries***) and responses back to clients (called ***answers***).
	
	Messages must contain a header, but other fields are not required. A `Message` struct represents this and includes several `Vec<T>` fields. These do not need to be wrapped in `Option` to represent missing values as their length can be 0.
		```
		  additionals: Vec<Record>,
		  sig0: Vec<Record>,       ①
		  edns: Option<Edns>,      ②

		}
		```
		① sig0, a cryptographically signed record, verifies the message's integrity. It is defined in RFC 2535.
		
		② edns indicates whether the message includes extended DNS.

Message type	A message type identifies the message as a query or as an answer. Queries can also be updates, which are functionality that our code ignores.	
		```
		Defined in trust_dns::op::MessageType
		pub enum MessageType {
		    Query,
		    Response,
		}
		```

Message	A number that	`u16`

ID	is used for senders to link queries and answers.	

Resource record type	The resource record type refers to the DNS codes that you've probably encountered if you've ever configured a domain name.
	
	Of note is how trust_dns handles invalid codes. The `RecordType` enum contains an `Unknown(u16)` variant that can be used for codes that it doesn't understand.
		```
		Defined in trust_dns::rr::record_type::RecordType
		pub enum RecordType {
		    A,
		    AAAA,
		    ANAME,
		    ANY,
		    // ...
		    Unknown(u16),
		    ZERO,
		}
		```

Query	A `Query` struct holds the domain name and the record type that we're seeking the DNS details for.
	
	These traits also describe the DNS class and allow
		```
		Defined in trust_dns::op::Query
		pub struct Query {
		    name: Name,
		    query_type: RecordType,
		    query_class: DNSClass,
		}
		```

	queries to distinguish between messages sent over the internet from other transport protocols.	

Opcode	An `OpCode` enum is, in some sense, a subtype of `MessageType`. This is an extensibility mechanism that allows future functionality. For example, RFC 1035 defines the `Query` and `Status` opcodes but others were defined later. The `Notify` and `Update` opcodes are defined by RFC 1996 and RFC 2136, respectively.	`Defined in trust_dns::op::OpCode`
		`pub enum OpCode {`
		`    Query,`
		`    Status,`
		`    Notify,`
		`    Update,`
		`}`

An unfortunate consequence of the protocol, which I suppose is a consequence of reality, is that there are many options, types, and subtypes involved. Listing 8.7, an excerpt from listing 8.9, shows the process of constructing a message that asks, “Dear DNS server, what is the IPv4 address for `domain_name`?” The listing constructs the DNS message, whereas the trust-dns crate requests an IPv4 address for `domain_name`.

Listing 8.7 Constructing a DNS message in Rust

```
let mut msg = Message::new();                        #1
msg
  .set_id(rand::random::<u16>())                     #2
  .set_message_type(MessageType::Query)
  .add_query(                                        #3
      Query::query(domain_name, RecordType::A)       #4
  )
  .set_op_code(OpCode::Query)
  .set_recursion_desired(true);                      #5
```

copy

We’re now in a position where we can meaningfully inspect the code. It has the following structure:

- Parses command-line arguments
- Builds a DNS message using trust_dns types
- Converts the structured data into a stream of bytes
- Sends those bytes across the wire

After that, we need to accept the response from the server, decode the incoming bytes, and print the result. Error handling remains relatively ugly, with many calls to `unwrap()` and `expect()`. We’ll address that problem shortly in section 8.5. The end process is a command-line application that’s quite simple.

Running our resolve application involves little ceremony. Given a domain name, it provides an IP address:

```
$ resolve www.rustinaction.com 35.185.44.232
```

copy

Listings 8.8 and 8.9 are the project's source code. While you are experimenting with the project, you may want to use some features of `cargo run` to speed up your process:

```
$ cargo run -q -- www.rustinaction.com       #1
.185.44.232
```

copy

To compile the resolve application from the official source code repository, execute these commands in the console:

```
$ git clone https:/ /github.com/rust-in-action/code rust-in-actio
Cloning into 'rust-in-action'...

$ cd rust-in-action/ch8/ch8-resolve

$ cargo run -q -- www.rustinaction.com      #1
.185.44.232
```

copy

To compile and build from scratch, follow these instructions to establish the project structure:

1. At the command-line, enter these commands:

```
$ cargo new resolve
     Created binary (application) `resolve` package

$ cargo install cargo-edit
...

$ cd resolve
$ cargo add rand@0.6
    Updating 'https:/ /github.com/rust-lang/crates.io-index' inde
      Adding rand v0.6 to dependencies

$ cargo add clap@2
    Updating 'https:/ /github.com/rust-lang/crates.io-index' inde
      Adding rand v2 to dependencies

$ cargo add trust-dns@0.16 --no-default-features
    Updating 'https:/ /github.com/rust-lang/crates.io-index' inde
      Adding trust-dns v0.16 to dependencies
```

copy

2. Once the structure has been established, you check that your Cargo.toml matches listing 8.8, available in ch8/ch8-resolve/Cargo.toml.
3. Replace the contents of src/main.rs with listing 8.9. It is available from ch8/ch8-resolve/src/main.rs.

The following snippet provides a view of how the files of the project and the listings are interlinked:

```
ch8-resolve
├── Cargo.toml      #1
└── src
    └── main.rs     #2
```

copy

Listing 8.8 Crate metadata for the resolve app

```
[package]
name = "resolve"
version = "0.1.0"
authors = ["Tim McNamara <author@rustinaction.com>"]
edition = "2018"

[dependencies]
rand = "0.6"
clap = "2.33"
trust-dns = { version = "0.16", default-features = false }
```

copy

Listing 8.9 A command-line utility to resolve IP addresses from hostnames

```
use std::net::{SocketAddr, UdpSocket};
use std::time::Duration;

use clap::{App, Arg};
use rand;
use trust_dns::op::{Message, MessageType, OpCode, Query};
use trust_dns::rr::domain::Name;
use trust_dns::rr::record_type::RecordType;
use trust_dns::serialize::binary::*;

fn main() {
  let app = App::new("resolve")
    .about("A simple to use DNS resolver")
    .arg(Arg::with_name("dns-server").short("s").default_value
    .arg(Arg::with_name("domain-name").required(true))
    .get_matches();

  let domain_name_raw = app                              #1
    .value_of("domain-name").unwrap();                   #1
  let domain_name =                                      #1
    Name::from_ascii(&domain_name_raw).unwrap();         #1

  let dns_server_raw = app                               #2
    .value_of("dns-server").unwrap();                    #2
  let dns_server: SocketAddr =                           #2
    format!("{}:53", dns_server_raw)                     #2
    .parse()                                             #2
    .expect("invalid address");                          #2

  let mut request_as_bytes: Vec<u8> =                    #3
    Vec::with_capacity(512);                             #3
  let mut response_as_bytes: Vec<u8> =                   #3
    vec![0; 512];                                        #3

  let mut msg = Message::new();                          #4
  msg
    .set_id(rand::random::<u16>())
    .set_message_type(MessageType::Query)                #5
    .add_query(Query::query(domain_name, RecordType::A))
    .set_op_code(OpCode::Query)
    .set_recursion_desired(true);

  let mut encoder =
    BinEncoder::new(&mut request_as_bytes);              #6
  msg.emit(&mut encoder).unwrap();

  let localhost = UdpSocket::bind("0.0.0.0:0")           #7
    .expect("cannot bind to local socket");
  let timeout = Duration::from_secs(3);
  localhost.set_read_timeout(Some(timeout)).unwrap();
  localhost.set_nonblocking(false).unwrap();

  let _amt = localhost
    .send_to(&request_as_bytes, dns_server)
    .expect("socket misconfigured");

  let (_amt, _remote) = localhost
    .recv_from(&mut response_as_bytes)
    .expect("timeout reached");

  let dns_message = Message::from_vec(&response_as_bytes)
    .expect("unable to parse response");

  for answer in dns_message.answers() {
    if answer.record_type() == RecordType::A {
      let resource = answer.rdata();
      let ip = resource
        .to_ip_addr()
        .expect("invalid IP address received");
      println!("{}", ip.to_string());
    }
  }
}
```

copy

Listing 8.9 includes some business logic that deserves explaining. Lines 30–33, repeated here, use two forms of initializing a `Vec<u8>`. Why?

```
let mut request_as_bytes: Vec<u8> =
     Vec::with_capacity(512);
   let mut response_as_bytes: Vec<u8> =
     vec![0; 512];
```

copy

Each form creates a subtly different outcome:

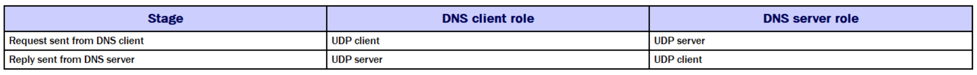
- `Vec::with_capacity(512)` creates a `Vec<T>` with length 0 and capacity 512.
- `vec![0; 512]` creates a `Vec<T>` with length 512 and capacity 512.

The underlying array looks the same, but the difference in length is significant. Within the call to `recv_from()` at line 58, the trust-dns crate includes a check that `response_as_bytes` has sufficient space. That check uses the length field, which results in a crash. Knowing how to wriggle around with initialization can be handy for satisfying an APIs' expectations.

HOW DNS SUPPORTS CONNECTIONS WITHIN UDP

UDP does not have a notion of long-lived connections. Unlike TCP, all messages are short-lived and one-way. Put another way, UDP does not support two-way (***duplex***) communications. But DNS requires a response to be sent from the DNS server back to the client.

To enable two-way communications within UDP, both parties must act as clients and servers, depending on context. That context is defined by the protocol built on top of UDP. Within DNS, the client becomes a DNS server to receive the server's reply. The following table provides a flow chart of the process.



Stage	DNS client role	DNS server role
Request sent from DNS client	UDP client	UDP server
Reply sent from DNS server	UDP server	UDP client

It's time to recap. Our overall task in this section was to make HTTP requests. HTTP is built on TCP. Because we only had a domain name (www.rustinaction.com) when we made the request, we needed to use DNS. DNS is primarily delivered over UDP, so we needed to take a diversion and learn about UDP.

Now it's almost time to return to TCP. Before we're able to do that, though, we need to learn how to combine error types that emerge from multiple dependencies.

8.5 Ergonomic error handling for libraries

Rust's error handling is safe and sophisticated. However, it offers a few challenges. When a function incorporates `Result` types from two upstream crates, the `?` operator no longer works because it only understands a single type. This proves to be important when we refactor our domain resolution code to work alongside our TCP code. This section discusses some of those challenges as well as strategies for managing them.

8.5.1 Issue: Unable to return multiple error types

Returning a `Result<T, E>` works great when there is a single error type `E`. But things become more complicated when we want to work with multiple error types.

TIP

For single files, compile the code with `rustc <filename>` rather than using `cargo build`. For example, if a file is named io-error.rs, then the shell command is `rustc io-error.rs && ./io-error[.exe]`.

To start, let's look at a small example that covers the easy case of a single error type. We'll try to open a file that does not exist. When run, listing 8.10 prints a short message in Rust syntax:

```
$ rustc ch8/misc/io-error.rs && ./io-error
Error: Os { code: 2, kind: NotFound, message: "No such file or di
```

copy

We won't win any awards for user experience here, but we get a chance to learn a new language feature. The following listing provides the code that produces a single error type. You'll find its source in ch8/misc/io-error.rs.

Listing 8.10 A Rust program that always produces an I/O error

```
use std::fs::File;

fn main() -> Result<(), std::io::Error> {
    let _f = File::open("invisible.txt")?;

    Ok(())
}
```

copy

Now, let's introduce another error type into `main()`. The next listing produces a compiler error, but we'll work through some options to get the code to compile. The code for this listing is in ch8/misc/multierror.rs.

Listing 8.11 A function that attempts to return multiple `Result` types

```
use std::fs::File;
use std::net::Ipv6Addr;

fn main() -> Result<(), std::io::Error> {
  let _f = File::open("invisible.txt")?;    #1

  let _localhost = "::1"                    #2
    .parse::<Ipv6Addr>()?;                  #2

  Ok(())
}
```

copy

To compile listing 8.11, enter the ch8/misc directory and use rustc. This produces quite a stern, yet helpful, error message:

```
$ rustc multierror.rs
error[E0277]: `?` couldn't convert the error to `std::io::Error`
 --> multierror.rs:8:25
  |
4 | fn main() -> Result<(), std::io::Error> {
  |              -------------------------- expected `std::io::Er
                                            because of this
...
8 |     .parse::<Ipv6Addr>()?;
  |                         ^ the trait `From<AddrParseError>`
                              is not implemented for `std::io::Er
  |
  = note: the question mark operation (`?`) implicitly performs a
          conversion on the error value using the `From` trait
  = help: the following implementations were found:
            <std::io::Error as From<ErrorKind>>
            <std::io::Error as From<IntoInnerError<W>>>
            <std::io::Error as From<NulError>>
  = note: required by `from`

error: aborting due to previous error

For more information about this error, try `rustc --explain E0277
```

copy

The error message can be difficult to interpret if you don’t know what the question mark operator (`?`) is doing. Why are there multiple messages about `std::convert::From`? Well, the `?` operator is syntactic sugar for the `try!` macro. `try!` performs two functions:

- When it detects `Ok(value)`, the expression evaluates to `value`.
- When `Err(err)` occurs, `try!`/`?` returns early after attempting to convert `err` to the error type defined in the calling function.

In Rust-like pseudocode, the `try!` macro could be defined as

```
macro try {
  match expression {
    Result::Ok(val) => val,                       #1
    Result::Err(err) => {
      let converted = convert::From::from(err);    #2
      return Result::Err(converted);               #3
    }
  });
}
```

Looking at listing 8.11 again, we can see the `try!` macro in action as `?`:

```
fn main() -> Result<(), std::io::Error> {
  let _f = File::open("invisible.txt")?;     #1

  let _localhost = "::1"                     #2
    .parse::<Ipv6Addr>()?;                   #2

  Ok(())
}
```

In addition to saving you from needing to use explicit pattern matching to extract the value or return an error, the `?` operator also attempts to convert its argument into an error type if required. Because the signature of main is `main() → Result<(), std::io ::Error>`, Rust attempts to convert the `std::net::AddrParseError` produced by `parse::<Ipv6Addr>()` into a `std::io::Error`. Don't worry, though; we can fix this! Earlier, in section 8.3, we introduced trait objects. Now we'll be able to put those to good use.

Using `Box<dyn Error>` as the error variant in the `main()` function allows us to progress. The `dyn` keyword is short for *dynamic*, implying that there is a runtime cost for this flexibility. Running listing 8.12 produces this output:

```
$ rustc ch8/misc/traiterror.rs && ./traiterror
Error: Os { code: 2, kind: NotFound, message: "No such file or di
```

copy

I suppose it’s a limited form of progress, but progress nonetheless. We’ve circled back to the error we started with. But we’ve passed through the compiler error, which is what we wanted.

Going forward, let’s look at listing 8.12. It implements a trait object in a return value to simplify error handling when errors originate from multiple upstream crates. You can find the source for this listing in ch8/misc/traiterror.rs.

Listing 8.12 Using a trait object in a return value

```
use std::fs::File;
use std::error::Error;
use std::net::Ipv6Addr;

fn main() -> Result<(), Box<dyn Error>> {      #1

  let _f = File::open("invisible.txt")?;      #2

  let _localhost = "::1"
    .parse::<Ipv6Addr>()?                      #3

  Ok(())
}
```

copy

Wrapping trait objects in `Box` is necessary because their size (in bytes on the stack) is unknown at compile time. In the case of listing 8.12, the trait object might originate from either `File::open()` or `"::1".parse()`. What actually happens depends on the circumstances encountered at runtime. A `Box` has a known size on the stack. Its raison d’être is to point to things that don’t, such as trait objects.

8.5.2 Wrapping downstream errors by defining our own error type

The problem that we are attempting to solve is that each of our dependencies defines its own error type. Multiple error types in one function prevent returning `Result`. The first strategy we looked at was to use trait objects, but trait objects have a potentially significant downside.

Using trait objects is also known as ***type erasure***. Rust is no longer aware that an error has originated upstream. Using `Box<dyn Error>` as the error variant of a `Result` means that the upstream error types are, in a sense, lost. The original errors are now converted to exactly the same type.

It is possible to retain the upstream errors, but this requires more work on our behalf. We need to bundle upstream errors in our own type. When the upstream errors are needed later (say, for reporting errors to the user), it's possible to extract these with pattern matching. Here is the process:

1. Define an enum that includes the upstream errors as variants.
2. Annotate the enum with `#[derive(Debug)]`.
3. Implement `Display`.
4. Implement `Error`, which almost comes for free because we have implemented `Debug` and `Display`.
5. Use `map_err()` in your code to convert the upstream error to your omnibus error type.

NOTE

You haven't previously encountered the `map_err()` function. We'll explain what it does when we get there later in this section.

It’s possible to stop with the previous steps, but there’s an optional extra step that improves the ergonomics. We need to implement `std::convert::From` to remove the need to call `map_err()`. To begin, let’s start back with listing 8.11, where we know that the code fails:

```
use std::fs::File;
use std::net::Ipv6Addr;

fn main() -> Result<(), std::io::Error> {
  let _f = File::open("invisible.txt")?;

  let _localhost = "::1"
    .parse::<Ipv6Addr>()?;

  Ok(())
}
```

This code fails because `"".parse::<Ipv6Addr>()` does not return a `std::io::Error`. What we want to end up with is code that looks a little more like the following listing.

Listing 8.13 Hypothetical example of the kind of code we want to write

```
use std::fs::File;
use std::io::Error;                                   1
use std::net::AddrParseError;
use std::net::Ipv6Addr;

enum UpstreamError{
  IO(std::io::Error),
  Parsing(AddrParseError),
}

fn main() -> Result<(), UpstreamError> {
  let _f = File::open("invisible.txt")?
    .maybe_convert_to(UpstreamError);

  let _localhost = "::1"
    .parse::<Ipv6Addr>()?
    .maybe_convert_to(UpstreamError);

  Ok(())
}
```

copy

You missed out on some activities - why not try them now?

Define an enum that includes the upstream errors as variants

The first thing to do is to return a type that can hold the upstream error types. In Rust, an enum works well. Listing 8.13 does not compile, but does do this step. We'll tidy up the imports slightly, though:

```rust
use std::io;
use std::net;

enum UpstreamError{
  IO(io::Error),
  Parsing(net::AddrParseError),
}
```

copy

Annotate the enum with #[derive(Debug)]

The next change is easy. It's a single-line change—the best kind of change. To annotate the enum, we'll add `#[derive(Debug)]`, as the following shows:

```rust
use std::io;
use std::net;

#[derive(Debug)]
enum UpstreamError{
  IO(io::Error),
  Parsing(net::AddrParseError),
}
```

copy

Implement std::fmt::Display

We’ll cheat slightly and implement `Display` by simply using `Debug`. We know that this is available to us because errors must define `Debug`. Here’s the updated code:

```
use std::fmt;
use std::io;
use std::net;

#[derive(Debug)]
enum UpstreamError{
  IO(io::Error),
  Parsing(net::AddrParseError),
}

impl fmt::Display for UpstreamError {
  fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
    write!(f, "{:?}", self)                                    1
  }
}
```

copy

You missed out on some activities – why not try them now?

Implement std::error::Error

Here’s another easy change. To end up with the kind of code that we’d like to write, let’s make the following change:

```
use std::error;                                                1
use std::fmt;
use std::io;
use std::net;

#[derive(Debug)]
enum UpstreamError{
  IO(io::Error),
  Parsing(net::AddrParseError),
}

impl fmt::Display for UpstreamError {
  fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
    write!(f, "{:?}", self)
  }
}

impl error::Error for UpstreamError { }
```

2

copy

The `impl` block is—well, we can rely on default implementations provided by the compiler—especially terse. Because there are default implementations of every method defined by `std::error::Error`, we can ask the compiler to do all of the work for us.

You missed out on some activities – why not try them now?

Use map_err()

The next fix is to add `map_err()` to our code to convert the upstream error to the omnibus error type. Back at listing 8.13, we wanted to have a `main()` that looks like this:

```
fn main() -> Result<(), UpstreamError> {
  let _f = File::open("invisible.txt")?
    .maybe_convert_to(UpstreamError);

  let _localhost = "::1"
    .parse::<Ipv6Addr>()?
    .maybe_convert_to(UpstreamError);

  Ok(())
}
```

copy

I can’t offer you that. I can, however, give you this:

```
fn main() -> Result<(), UpstreamError> {
  let _f = File::open("invisible.txt")
```

```
      .map_err(UpstreamError::IO)?;

    let _localhost = "::1"
      .parse::<Ipv6Addr>()
      .map_err(UpstreamError::Parsing)?;

    Ok(())
}
```

copy

This new code works! Here’s how. The `map_err()` function maps an error to a function. (Variants of our `UpstreamError` enum can be used as functions here.) Note that the `?` operator needs to be at the end. Otherwise, the function can return before the code has a chance to convert the error.

Listing 8.14 provides the new code. When run, it produces this message to the console:

```
$ rustc ch8/misc/wraperror.rs && ./wraperror
Error: IO(Os { code: 2, kind: NotFound, message: "No such file c
```

copy

To retain type safety, we can use the new code in the following listing. You’ll find its source in ch8/misc/wraperror.rs.

Listing 8.14 Wrapping upstream errors in our own type

```
 1 use std::io;
  2 use std::fmt;
  3 use std::net;
  4 use std::fs::File;
  5 use std::net::Ipv6Addr;
  6
  7 #[derive(Debug)]
  8 enum UpstreamError{
  9   IO(io::Error),
 10   Parsing(net::AddrParseError),
 11 }
 12
```

```rust
impl fmt::Display for UpstreamError {
  fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
    write!(f, "{:?}", self)
  }
}

impl error::Error for UpstreamError { }

fn main() -> Result<(), UpstreamError> {
  let _f = File::open("invisible.txt")
    .map_err(UpstreamError::IO)?;

  let _localhost = "::1"
    .parse::<Ipv6Addr>()
    .map_err(UpstreamError::Parsing)?;

  Ok(())
}
```

copy

It's also possible to remove the calls to `map_err()`. But to enable that, we need to implement `From`.

You missed out on some activities – why not try them now?

Implement std::convert::From to remove the need to call map_err()

The `std::convert::From` trait has a single required method, `from()`. We need two `impl` blocks to enable our two upstream error types to be convertible. The following snippet shows how:

```rust
impl From<io::Error> for UpstreamError {
  fn from(error: io::Error) -> Self {
    UpstreamError::IO(error)
  }
}

impl From<net::AddrParseError> for UpstreamError {
  fn from(error: net::AddrParseError) -> Self {
    UpstreamError::Parsing(error)
  }
}
```

copy

Now the `main()` function returns to a simple form of itself:

```
fn main() -> Result<(), UpstreamError> {
  let _f = File::open("invisible.txt")?;
  let _localhost = "::1".parse::<Ipv6Addr>()?;

  Ok(())
}
```

copy

The full code listing is provided in listing 8.15. Implementing `From` places the burden of extra syntax on the library writer. It results in a much easier experience when using your crate, simplifying its use by downstream programmers. You'll find the source for this listing in ch8/misc/wraperror2.rs.

Listing 8.15 Implementing `std::convert::From` for our wrapper error type

```
use std::io;
use std::fmt;
use std::net;
use std::fs::File;
use std::net::Ipv6Addr;

#[derive(Debug)]
enum UpstreamError{
  IO(io::Error),
  Parsing(net::AddrParseError),
}

impl fmt::Display for UpstreamError {
  fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
    write!(f, "{:?}", self)
  }
}

impl error::Error for UpstreamError { }

impl From<io::Error> for UpstreamError {
  fn from(error: io::Error) -> Self {
    UpstreamError::IO(error)
  }
}
```

```

impl From<net::AddrParseError> for UpstreamError {
  fn from(error: net::AddrParseError) -> Self {
    UpstreamError::Parsing(error)
  }
}

fn main() -> Result<(), UpstreamError> {
  let _f = File::open("invisible.txt")?;
  let _localhost = "::1".parse::<Ipv6Addr>()?;

  Ok(())
}
```

copy

You missed out on some activities – why not try them now?

8.5.3 Cheating with unwrap() and expect()

The final approach for dealing with multiple error types is to use `unwrap()` and `expect()`. Now that we have the tools to handle multiple error types in a function, we can continue our journey.

NOTE

This is a reasonable approach when writing a `main()` function, but it isn't recommended for library authors. Your users don't want their programs to crash because of things outside of their control.

8.6 MAC addresses

Several pages ago in listing 8.9, you implemented a DNS resolver. That enabled conversions from a host name such as www.rustinaction.com to an IP address. Now we have an IP address to connect to.

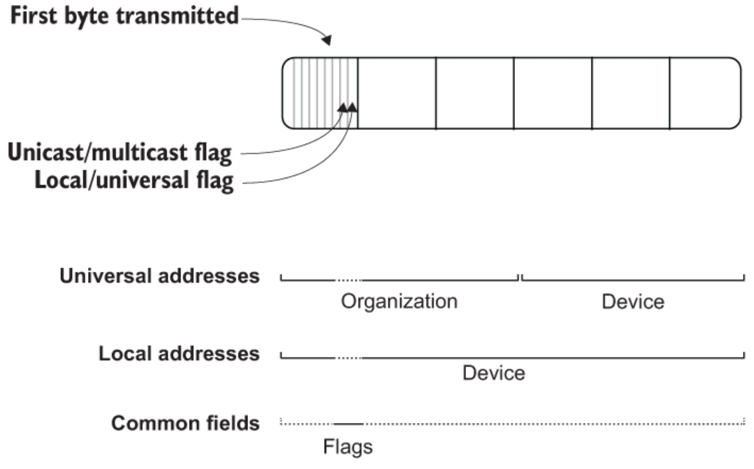
The internet protocol enables devices to contact each other via their IP addresses. But that's not all. Every hardware device also includes a unique identifier that's independent of the network it's connected to. Why a second number? The answer is partially technical and partially historical.

Ethernet networking and the internet started life independently. Ethernet's focus was on local area networks (LANs). The internet was developed to enable communication between networks, and Ethernet is the addressing system understood by devices that share a physical link (or a radio link in the case of WiFi, Bluetooth, and other wireless technologies).

Perhaps a better way to express this is that MAC (short for ***media access control***) addresses are used by devices that share electrons (figure 8.3). But there are a few differences:

- *IP addresses are hierarchical, but MAC addresses are not.* Addresses appearing close together numerically are not close together physically, or organizationally.
- *MAC addresses are 48 bits (6 bytes) wide.* IP addresses are 32 bits (4 bytes) wide for IPv4 and 128 bits (16 bytes) for IPv6.

Figure 8.3 In-memory layout for MAC addresses

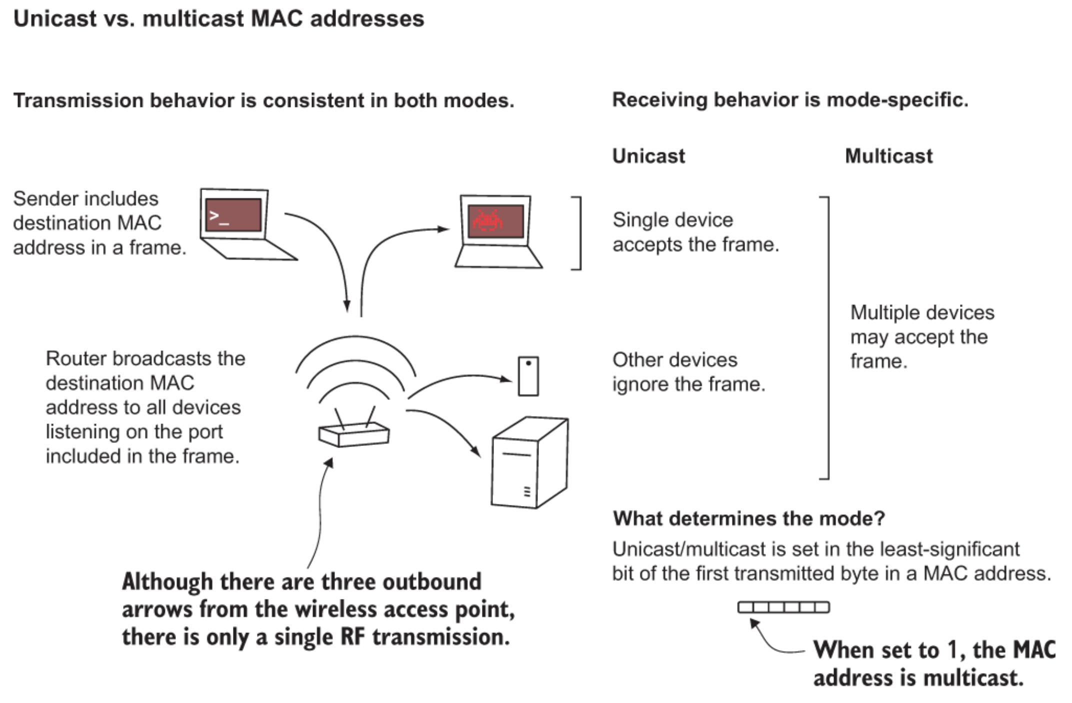

First byte transmitted
In Rust syntax, the layout of a MAC address is specified as [u8; 6].
Unicast/multicast flag
Local/universal flag
Universal addresses
Organization
Device
The role of specific bits changes according to the local/universal flag.
Local addresses
Device
Common fields
Flags

There are two forms of MAC addresses:

- *Universally administered (or universal) addresses are set when devices are manufactured.* Manufacturers use a prefix assigned by the IEEE Registration Authority and a scheme of their choosing for the remaining bits.
- *Locally administered (or local) addresses allow devices to create their own MAC addresses without registration.* When setting a device's MAC address yourself in software, you should make sure that your address is set to the local form.

MAC addresses have two modes: *unicast* and *multicast*. The transmission behavior for these forms is identical. The distinction is made when a device makes a decision about whether to accept a frame. A *frame* is a term used by the Ethernet protocol for a byte slice at this level. Analogies to frame include a packet, wrapper, and envelope. Figure 8.4 shows this distinction.

Figure 8.4 The differences between multicast and unicast MAC addresses


Unicast vs. multicast MAC addresses
Transmission behavior is consistent in both modes.
Receiving behavior is mode-specific.
Unicast
Multicast
Sender includes destination MAC address in a frame.
Single device accepts the frame.
Multiple devices may accept the frame.
Router broadcasts the destination MAC address to all devices listening on the port included in the frame.
Other devices ignore the frame.
What determines the mode?
Unicast/multicast is set in the least-significant bit of the first transmitted byte in a MAC address.
Although there are three outbound arrows from the wireless access point, there is only a single RF transmission.
When set to 1, the MAC address is multicast.

Unicast addresses are intended to transport information between two points that are in direct contact (say, between a laptop and a router). Wireless access points complicate matters somewhat but don't change the fundamentals. A multicast address can be accepted by multiple recipients, whereas unicast has a single recipient. The term *unicast* is somewhat misleading, though. Sending an Ethernet packet involves more than two devices. Using a unicast address alters what devices do when they receive packets but not which data is transmitted over the wire (or through the radio waves).

8.6.1 Generating MAC addresses

When we begin talking about raw TCP in section 8.8, we'll create a virtual hardware device in listing 8.22. To convince anything to talk to us, we need to learn how to assign our virtual device a MAC address. The macgen project in listing 8.17 generates the MAC addresses for us. The following listing shows the metadata for that project. You can find its source in ch8/ch8-mac/Cargo.toml.

Listing 8.16 Crate metadata for the macgen project

```
[package]
name = "ch8-macgen"
version = "0.1.0"
authors = ["Tim McNamara <author@rustinaction.com>"]
edition = "2018"
```

```
[dependencies]
rand = "0.7"
```

copy

The following listing shows the macgen project, our MAC address generator. The source code for this project is in the ch8/ch8-mac/src/main.rs file.

Listing 8.17 Creating macgen, a MAC address generator

```
extern crate rand;

use rand::RngCore;
use std::fmt;
use std::fmt::Display;

#[derive(Debug)]
struct MacAddress([u8; 6]);                           #1

impl Display for MacAddress {
  fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
    let octet = &self.0;
    write!(
      f,
      "{:02x}:{:02x}:{:02x}:{:02x}:{:02x}:{:02x}",    #2
      octet[0], octet[1], octet[2],                   #2
      octet[3], octet[4], octet[5]                    #2
    )
  }
}

impl MacAddress {
  fn new() -> MacAddress {
    let mut octets: [u8; 6] = [0; 6];
    rand::thread_rng().fill_bytes(&mut octets);
    octets[0] |= 0b_0000_0011;                        #3
    MacAddress { 0: octets }
  }

  fn is_local(&self) -> bool {
    (self.0[0] & 0b_0000_0010) == 0b_0000_0010
  }

  fn is_unicast(&self) -> bool {
    (self.0[0] & 0b_0000_0001) == 0b_0000_0001
  }
}

fn main() {
  let mac = MacAddress::new();
  assert!(mac.is_local());
  assert!(mac.is_unicast());
  println!("mac: {}", mac);
}
```

copy

The code from listing 8.17 should feel legible. Line 25 contains some relatively obscure syntax, though. `octets[0] |= 0b_0000_0011` coerces the two flag bits described at figure 8.3 to a state of `1`. That designates every MAC address we generate as locally assigned and unicast.

8.7 Implementing state machines with Rust's enums

Another prerequisite for handling network messages is being able to define a state machine. Our code needs to adapt to changes in connectivity.

Listing 8.22 contains a state machine, implemented with a `loop`, a `match`, and a Rust enum. Because of Rust's expression-based nature, control flow operators also return values. Every time around the loop, the state is mutated in place. The following listing shows the pseudocode for how a repeated `match` on a `enum` works together.

Listing 8.18 Pseudocode for a state machine implementation

```
enum HttpState {
    Connect,
    Request,
    Response,
}

loop {
    state = match state {
        HttpState::Connect if !socket.is_active() => {
            socket.connect();
            HttpState::Request
        }

        HttpState::Request if socket.may_send() => {
            socket.send(data);
            HttpState::Response
        }

        HttpState::Response if socket.can_recv() => {
            received = socket.recv();
            HttpState::Response
        }

        HttpState::Response if !socket.may_recv() => {
            break;
        }
        _ => state,
    }
}
```

copy

More advanced methods to implement finite state machines do exist. This is the simplest, however. We'll make use of it in listing 8.22. Making use of an enum embeds the state machine's transitions into the type system itself.

But we're still at a level that is far too high! To dig deeper, we're going to need to get some assistance from the OS.

8.8 Raw TCP

Integrating with the raw TCP packets typically requires root/superuser access. The OS starts to get quite grumpy when an unauthorized user asks to make raw network requests. We can get around this (on Linux) by creating a proxy device that non-super users are allowed to communicate with directly.

DON’T HAVE LINUX?

If you’re running another OS, there are many virtualization options available. Here are a few:

- The Multipass project (https://multipass.run/) provides fast Ubuntu virtual machines on macOS and Windows hosts.
- WSL, the Windows Subsystem for Linux (https://docs.microsoft.com/en-us/windows/wsl/about), is another option to look into.
- Oracle VirtualBox (https://www.virtualbox.org/) is an open source project with excellent support for many host operating systems.

8.9 Creating a virtual networking device

To proceed with this section, you will need to create virtual networking hardware. Using virtual hardware provides more control to freely assign IP and MAC addresses. It also avoids changing your hardware settings, which could affect its ability to connect to the network. To create a TAP device called tap-rust, execute the following command in your Linux console:

```
$ sudo \                 #1
>  ip tuntap \           #2
>    add \               #3
>    mode tap \          #4
>    name tap-rust \     #5
>    user $USER          #6
```

copy

When successful, `ip` prints no output. To confirm that our tap-rust device was added, we can use the `ip tuntap list` subcommand as in the following snippet. When executed, you should see the tap-rust device in the list of devices in the output:

```
$ ip tuntap list
tap-rust: tap persist user
```

copy

Now that we have created a networking device, we also need to allocate an IP address for it and tell our system to forward packets to it. The following shows the commands to enable this functionality:

```
$ sudo ip link set tap-rust up                        #1
$ sudo ip addr add 192.168.42.100/24 dev tap-rust     #2

$ sudo iptables \                                     #3
>   -t nat\                                           #3
>   -A POSTROUTING \                                  #3
>   -s 192.168.42.0/24 \                              #3
>   -j MASQUERADE                                     #3


$ sudo sysctl net.ipv4.ip_forward=1                   #4
```

copy

The following shows how to remove the device (once you have completed this chapter) by using `del` rather than `add`:

```
$ sudo ip tuntap del mode tap name tap-rust
```

copy

8.10 “Raw” HTTP

We should now have all the knowledge we need to take on the challenge of using HTTP at the TCP level. The mget project (mget is short for *manually get*) spans listings 8.20–8.23. It is a large project, but you’ll find it immensely satisfying to understand and build. Each file provides a different role:

- *main.rs (listing 8.20)*—Handles command-line parsing and weaves together the functionality provided by its peer files. This is where we combine the error types using the process outlined in section 8.5.2.
- *ethernet.rs (listing 8.21)*—Generates a MAC address using the logic from listing 8.17 and converts between MAC address types (defined by the smoltcp crate) and our own.
- *http.rs (listing 8.22)*—Carries out the work of interacting with the server to make the HTTP request.
- *dns.rs (listing 8.23)*—Performs DNS resolution, which converts a domain name to an IP address.

NOTE

The source code for these listings (and every code listing in the book) is available from https://github.com/rust-in-action/code or https://www.manning.com/books/rust-in-action.

It's important to acknowledge that listing 8.22 was derived from the HTTP client example within the smoltcp crate itself. whitequark (https://whitequark.org/) has built an absolutely fantastic networking library. Here's the file structure for the mget project:

```
ch8-mget
├── Cargo.toml          #1
└── src
    ├── main.rs         #2
    ├── ethernet.rs     #3
    ├── http.rs         #4
    └── dns.rs          #5
```

copy

To download and run the mget project from source control, execute these commands at the command line:

```
$ git clone https:/ /github.com/rust-in-action/code rust-in-actio
Cloning into 'rust-in-action'...

$ cd rust-in-action/ch8/ch8-mget
```

copy

Here are the project setup instructions for those readers who enjoy doing things step by step (with the output omitted).

1. Enter these commands at the command-line:

```
$ cargo new mget

$ cd mget

$ cargo install cargo-edit

$ cargo add clap@2

$ cargo add url@02

$ cargo add rand@0.7

$ cargo add trust-dns@0.16 --no-default-features

$ cargo add smoltcp@0.6 --features='proto-igmp proto-ipv4 verbose
```

copy

2. Check that your project's Cargo.toml matches listing 8.19.
3. Within the src directory, listing 8.20 becomes main.rs, listing 8.21 becomes ethernet.rs, listing 8.22 becomes http.rs, and listing 8.23 becomes dns.rs.

The following listing shows the metadata for mget. You'll find its source code in the ch8/ch8-mget/Cargo.toml file.

Listing 8.19 Crate metadata for mget

```
[package]
name = "mget"
version = "0.1.0"
authors = ["Tim McNamara <author@rustinaction.com>"]
edition = "2018"

[dependencies]
clap = "2"                          #1
rand = "0.7"                        #2
smoltcp = {                         #3
  version = "0.6",
  features = ["proto-igmp", "proto-ipv4", "verbose", "log"]
}
trust-dns = {                       #4
  version = "0.16",
  default-features = false
}
url = "2"                           #5
```

copy

The following listing shows the command-line parsing for our project. You'll find this source in ch8/ch8-mget/src/main.rs.

Listing 8.20 mget command-line parsing and overall coordination

```
use clap::{App, Arg};
use smoltcp::phy::TapInterface;
use url::Url;

mod dns;
mod ethernet;
mod http;

fn main() {
  let app = App::new("mget")
    .about("GET a webpage, manually")
    .arg(Arg::with_name("url").required(true))          #1
    .arg(Arg::with_name("tap-device").required(true))   #2
    .arg(
      Arg::with_name("dns-server")
        .default_value("1.1.1.1"),                      #3
    )
    .get_matches();                                     #4

  let url_text = app.value_of("url").unwrap();
  let dns_server_text =
    app.value_of("dns-server").unwrap();
  let tap_text = app.value_of("tap-device").unwrap();

  let url = Url::parse(url_text)                        #5
    .expect("error: unable to parse <url> as a URL");

  if url.scheme() != "http" {                           #5
    eprintln!("error: only HTTP protocol supported");
    return;
  }

  let tap = TapInterface::new(&tap_text)                #5
    .expect(
      "error: unable to use <tap-device> as a \
       network interface",
    );

  let domain_name =
    url.host_str()                                      #5
      .expect("domain name required");

  let _dns_server: std::net::Ipv4Addr =
    dns_server_text
      .parse()                                          #5
      .expect(
        "error: unable to parse <dns-server> as an \
         IPv4 address",
      );

  let addr =
    dns::resolve(dns_server_text, domain_name)           #6
      .unwrap()
      .unwrap();

  let mac = ethernet::MacAddress::new().into();          #7

  http::get(tap, mac, addr, url).unwrap();               #8

}
```

copy

The following listing generates our MAC address and converts between MAC address types defined by the smoltcp crate and our own. The code for this listing is in ch8/ch8-mget/src/ethernet.rs.

Listing 8.21 Ethernet type conversion and MAC address generation

```
use rand;
use std::fmt;
use std::fmt::Display;

use rand::RngCore;
use smoltcp::wire;

#[derive(Debug)]
pub struct MacAddress([u8; 6]);

impl Display for MacAddress {
  fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
    let octet = self.0;
    write!(
      f,
      "{:02x}:{:02x}:{:02x}:{:02x}:{:02x}:{:02x}",
      octet[0], octet[1], octet[2],
      octet[3], octet[4], octet[5]
    )
  }
}

impl MacAddress {
  pub fn new() -> MacAddress {
    let mut octets: [u8; 6] = [0; 6];
    rand::thread_rng().fill_bytes(&mut octets);    #1
    octets[0] |= 0b_0000_0010;                     #2
    octets[0] &= 0b_1111_1110;                     #3
    MacAddress { 0: octets }
  }
}
```

```

impl Into<wire::EthernetAddress> for MacAddress {
  fn into(self) -> wire::EthernetAddress {
    wire::EthernetAddress { 0: self.0 }
  }
}
```

copy

The following listing shows how to interact with the server to make the HTTP request. The code for this listing is in ch8/ch8-mget/src/http.rs.

Listing 8.22 Manually creating an HTTP request using TCP primitives

```
1 use std::collections::BTreeMap;
  2 use std::fmt;
  3 use std::net::IpAddr;
  4 use std::os::unix::io::AsRawFd;
  5
  6 use smoltcp::iface::{EthernetInterfaceBuilder, NeighborCac
  7 use smoltcp::phy::{wait as phy_wait, TapInterface};
  8 use smoltcp::socket::{SocketSet, TcpSocket, TcpSocketBuffe
  9 use smoltcp::time::Instant;
 10 use smoltcp::wire::{EthernetAddress, IpAddress, IpCidr, Ip
 11 use url::Url;
 12
 13 #[derive(Debug)]
 14 enum HttpState {
 15   Connect,
 16   Request,
 17   Response,
 18 }
 19
 20 #[derive(Debug)]
 21 pub enum UpstreamError {
 22   Network(smoltcp::Error),
 23   InvalidUrl,
 24   Content(std::str::Utf8Error),
 25 }
 26
 27 impl fmt::Display for UpstreamError {
 28   fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result
 29     write!(f, "{:?}", self)
 30   }
 31 }
 32
 33 impl From<smoltcp::Error> for UpstreamError {
 34   fn from(error: smoltcp::Error) -> Self {
 35     UpstreamError::Network(error)
 36   }
```

```
}

impl From<std::str::Utf8Error> for UpstreamError {
  fn from(error: std::str::Utf8Error) -> Self {
    UpstreamError::Content(error)
  }
}

fn random_port() -> u16 {
  49152 + rand::random::<u16>() % 16384
}

pub fn get(
  tap: TapInterface,
  mac: EthernetAddress,
  addr: IpAddr,
  url: Url,
) -> Result<(), UpstreamError> {
  let domain_name = url.host_str().ok_or(UpstreamError::Ir

  let neighbor_cache = NeighborCache::new(BTreeMap::new())

  let tcp_rx_buffer = TcpSocketBuffer::new(vec![0; 1024]);
  let tcp_tx_buffer = TcpSocketBuffer::new(vec![0; 1024]);
  let tcp_socket = TcpSocket::new(tcp_rx_buffer, tcp_tx_bu

  let ip_addrs = [IpCidr::new(IpAddress::v4(192, 168, 42,

  let fd = tap.as_raw_fd();
  let mut routes = Routes::new(BTreeMap::new());
  let default_gateway = Ipv4Address::new(192, 168, 42, 10
  routes.add_default_ipv4_route(default_gateway).unwrap();
  let mut iface = EthernetInterfaceBuilder::new(tap)
    .ethernet_addr(mac)
    .neighbor_cache(neighbor_cache)
    .ip_addrs(ip_addrs)
    .routes(routes)
    .finalize();

  let mut sockets = SocketSet::new(vec![]);
  let tcp_handle = sockets.add(tcp_socket);

  let http_header = format!(
    "GET {} HTTP/1.0\r\nHost: {}\r\nConnection: close\r\n\
    url.path(),
    domain_name,
  );

  let mut state = HttpState::Connect;
  'http: loop {
    let timestamp = Instant::now();
    match iface.poll(&mut sockets, timestamp) {
      Ok(_) => {}
      Err(smoltcp::Error::Unrecognized) => {}
      Err(e) => {
        eprintln!("error: {:?}", e);
      }
    }

    {
      let mut socket = sockets.get::<TcpSocket>(tcp_handle

      state = match state {
        HttpState::Connect if !socket.is_active() => {
         eprintln!("connecting");
         socket.connect((addr, 80), random_port())?;
         HttpState::Request
        }

        HttpState::Request if socket.may_send() => {
         eprintln!("sending request");
         socket.send_slice(http_header.as_ref())?;
         HttpState::Response
        }

        HttpState::Response if socket.can_recv() => {
         socket.recv(|raw_data| {
          let output = String::from_utf8_lossy(raw_data)
          println!("{}", output);
          (raw_data.len(), ())
         })?;
         HttpState::Response
        }

        HttpState::Response if !socket.may_recv() => {
         eprintln!("received complete response");
         break 'http;
        }
        _ => state,
      }
    }

    phy_wait(fd, iface.poll_delay(&sockets, timestamp))
      .expect("wait error");
  }

  Ok(())
}
```

And finally, the following listing performs the DNS resolution. The source for this listing is in ch8/ch8-mget/src/dns.rs.

Listing 8.23 Creating DNS queries to translate domain names to IP addresses

```
use std::error::Error;
use std::net::{SocketAddr, UdpSocket};
use std::time::Duration;

use trust_dns::op::{Message, MessageType, OpCode, Query};
use trust_dns::proto::error::ProtoError;
use trust_dns::rr::domain::Name;
use trust_dns::rr::record_type::RecordType;
use trust_dns::serialize::binary::*;

fn message_id() -> u16 {
  let candidate = rand::random();
  if candidate == 0 {
    return message_id();
  }
  candidate
}

#[derive(Debug)]
pub enum DnsError {
  ParseDomainName(ProtoError),
  ParseDnsServerAddress(std::net::AddrParseError),
  Encoding(ProtoError),
  Decoding(ProtoError),
  Network(std::io::Error),
  Sending(std::io::Error),
  Receving(std::io::Error),
}

impl std::fmt::Display for DnsError {
  fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::
    write!(f, "{:#?}", self)
  }
}

impl std::error::Error for DnsError {}                          1

pub fn resolve(
  dns_server_address: &str,
  domain_name: &str,
) -> Result<Option<std::net::IpAddr>, Box<dyn Error>> {
  let domain_name =
    Name::from_ascii(domain_name)
      .map_err(DnsError::ParseDomainName)?;

  let dns_server_address =                                      2
    format!("{}:53", dns_server_address);
  let dns_server: SocketAddr = dns_server_address
    .parse()
    .map_err(DnsError::ParseDnsServerAddress)?;

  let mut request_buffer: Vec<u8> =                             3
    Vec::with_capacity(64);
  let mut response_buffer: Vec<u8> =                            4
    vec![0; 512];

  let mut request = Message::new();
  request.add_query(                                            5
    Query::query(domain_name, RecordType::A)
  );

  request
    .set_id(message_id())
    .set_message_type(MessageType::Query)
    .set_op_code(OpCode::Query)                                 6
    .set_recursion_desired(true);

  let localhost =
    UdpSocket::bind("0.0.0.0:0").map_err(DnsError::Network

  let timeout = Duration::from_secs(5);
  localhost
    .set_read_timeout(Some(timeout))                         7
    .map_err(DnsError::Network)?;

  localhost
    .set_nonblocking(false)
    .map_err(DnsError::Network)?;

  let mut encoder = BinEncoder::new(&mut request_buffer);
  request.emit(&mut encoder).map_err(DnsError::Encoding)?;

  let _n_bytes_sent = localhost
    .send_to(&request_buffer, dns_server)
    .map_err(DnsError::Sending)?;
                                                             8
  loop {
    let (_b_bytes_recv, remote_port) = localhost
      .recv_from(&mut response_buffer)
      .map_err(DnsError::Receving)?;

    if remote_port == dns_server {
      break;
    }
  }

  let response =
    Message::from_vec(&response_buffer)
      .map_err(DnsError::Decoding)?;

  for answer in response.answers() {
    if answer.record_type() == RecordType::A {
      let resource = answer.rdata();
      let server_ip =
        resource.to_ip_addr().expect("invalid IP address r
      return Ok(Some(server_ip));
    }
  }

  Ok(None)
}
```

copy

mget is an ambitious project. It brings together all the threads from the chapter, is dozens of lines long, and yet is less capable than the `request::get(url)` call we made in listing 8.2. Hopefully it's revealed several interesting avenues for you to explore. Perhaps, surprisingly, there are several more networking layers to unwrap.

Well done for making your way through a lengthy and challenging chapter.

You missed out on some activities – why not try them now?

Summary

- Networking is complicated. Standard models such as OSIs are only partially accurate.
- Trait objects allow for runtime polymorphism. Typically, programmers prefer generics because trait objects incur a small runtime cost. However, this situation is not always clear-cut. Using trait objects can reduce space because only a single version of each function needs to be compiled. Fewer functions also benefits cache coherence.
- Networking protocols are particular about which bytes are used. In general, you should prefer using `&[u8]` literals (`b"..."`) over `&str` literals (`"..."`) to ensure that you retain full control.
- There are three main strategies for handling multiple upstream error types within a single scope:
  - Create an internal wrapper type and implement `From` for each of the upstream types
  - Change the return type to make use of a trait object that implements `std::error:Error`
  - Use `.unwrap()` and its cousin `.expect()`
- Finite state machines can be elegantly modeled in Rust with an enum and a loop. At each iteration, indicate the next state by returning the appropriate enum variant.
- To enable two-way communications in UDP, each side of the conversation must be able to act as a client and a server.

Up next...
9 Time and timekeeping

- Understanding how a computer keeps time
- How operating systems represent timestamps
- Synchronizing atomic clocks with the Network Time Protocol (NTP)