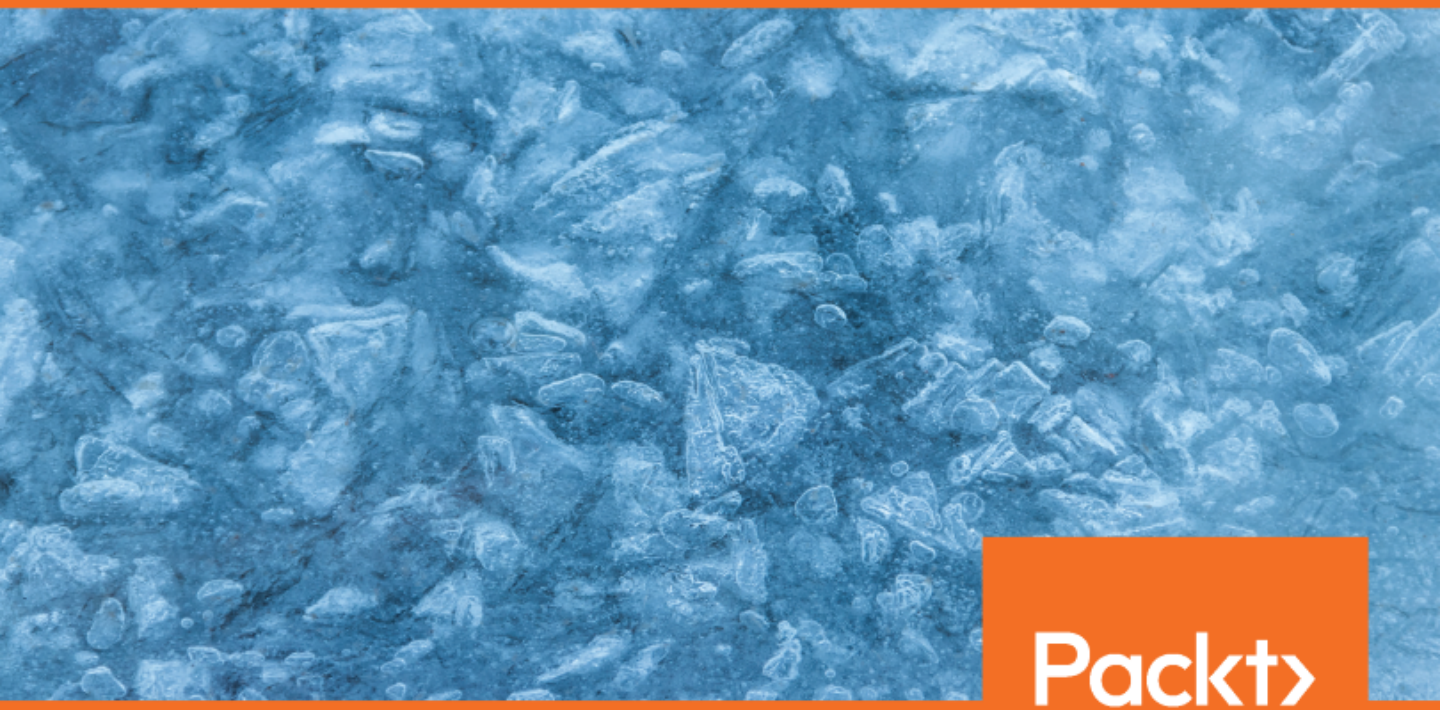
Hands-On Network Programming with C

Learn socket programming in C and write secure and optimized network code



Lewis Van Winkle

Packt
www.packt.com

Hands-On Network Programming with C

Learn socket programming in C and write secure and optimized network code

Lewis Van Winkle

BIRMINGHAM - MUMBAI

Hands-On Network Programming with C

Commissioning Editor: Richa Tripathi
Acquisition Editor: Shriram Shekhar
Content Development Editor: Digvijay Bagul
Technical Editor: Abin Sebastian
Copy Editor: Safis Editing
Project Coordinator: Prajakta Naik
Proofreader: Safis Editing
Indexer: Tejal Daruwale Soni
Graphics Coordinator: Tom Scaria
Production Coordinator: Aparna Bhagat

First published: May 2019

Production reference: 1100519

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

ISBN 978-1-78934-986-3

www.packtpub.com

For Doogie

– Lewis Van Winkle

mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

Packt.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Lewis Van Winkle is a software programming consultant, entrepreneur, and founder of a successful IoT company. He has over 20 years of programming experience after publishing his first successful software product at the age of 12. He has over 15 years of programming experience with the C programming language on a variety of operating systems and platforms. He is active in the open source community and has published several popular open source programs and libraries—many of them in C. Today, Lewis spends much of his time consulting, where he loves taking on difficult projects that other programmers have given up on. He specializes in network systems, financial systems, machine learning, and interoperation between different programming languages.

I would like to thank the publisher, Packt. This book wouldn't exist without their encouragement and backing. I would also like to extend a special thank you to my reviewer, Daniele Lacamera, for the careful work he carried out. This book improved significantly as a result of his valuable feedback. I also want to acknowledge the patience and support that my friends and family have shown over the last year while I've been away writing.

About the reviewer

Daniele Lacamera is a software technologist and researcher with vast experience in software design and development on embedded systems for different industries. He is currently working as freelance software developer and trainer. He is a worldwide expert in TCP/IP and transport protocol design and optimization, with more than 20 academic publications on the topic. He supports free software by contributing to several projects, including the Linux kernel, and is involved within a number of communities and organizations that promote the use of free and open source software in the IoT.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface

Packt first contacted me about writing this book nearly a year ago. It's been a long journey, harder than I anticipated at times, and I've learned a lot. The book you hold now is the culmination of many long days, and I'm proud to finally present it.

I think C is a beautiful programming language. No other language in everyday use gets you as close to the machine as C does. I've used C to program 8-bit microcontrollers with only 16 bytes of RAM, just the same as I've used it to program modern desktops with multi-core, multi-GHz processors. It's truly remarkable that C works efficiently in both contexts.

Network programming is a fun topic, but it's also a very deep one; a lot is going on at many levels. Some programming languages hide these abstractions. In the Python programming language, for example, you can download an entire web page using only one line of code. This isn't the case in C! In C, if you want to download a web page, you have to know how everything works. You need to know sockets, you need to know **Transfer Control Protocol** (**TCP**), and you need to know HTTP. In C network programming, nothing is hidden.

C is a great language to learn network programming in. This is not only because we get to see all the details, but also because the popular operating systems all use kernels written in C. No other language gives you the same first-class access as C does. In C, everything is under your control – you can lay out your data structures exactly how you want, manage memory precisely as you please, and even shoot yourself in the foot just the way you want.

When I first began writing this book, I surveyed other resources related to learning network programming with C. I found much misinformation – not only on the web, but even in print. There is a lot of C networking code that is done wrong. Internet tutorials about C sockets often use deprecated functions and ignore memory safety completely. When it comes to network programming, you can't take the *it works so it's good enough programming-by-coincidence* approach. You have to use reasoning.

In this book, I take care to approach network programming in a modern and safe way. The example programs are carefully designed to work with both IPv4 and IPv6, and they are all written in a portable, operating system-independent way, whenever possible. Wherever there is an opportunity for memory errors, I try to take notice and point out these concerns. Security is too often left as an afterthought. I believe security is important, and it should be planned in the system from the beginning. Therefore, in addition to teaching network basics, this book spends a lot of time working with secure protocols, such as TLS.

I hope you enjoy reading this book as much as I enjoyed writing it.

Who this book is for

This book is for the C or C++ programmer who wants to add networking features to their software. It is also designed for the student or professional who simply wants to learn about network programming and common network protocols.

It is assumed that the reader already has some familiarity with the C programming language. This includes a basic proficiency with pointers, basic data structures, and manual memory management.

What this book covers

`Chapter 1`, *Introducing Networks and Protocols*, introduces the important concepts related to networking. This chapter includes example programs to determine your IP address pragmatically.

`Chapter 2`, *Getting to Grips with Socket APIs*, introduces socket programming APIs and has you build your first networked program—a tiny web server.

`Chapter 3`, *An In-Depth Overview of TCP Connections*, focuses on programming TCP sockets. In this chapter, example programs are developed for both the client and server sides.

`Chapter 4`, *Establishing UDP Connections*, covers programming with **User Datagram Protocol** (**UDP**) sockets.

`Chapter 5`, *Hostname Resolution and DNS*, explains how hostnames are translated into IP addresses. In this chapter, we build an example program to perform manual DNS query lookups using UDP.

`Chapter 6`, *Building a Simple Web Client*, introduces HTTP—the protocol that powers websites. We dive right in and build an HTTP client in C.

`Chapter 7`, *Building a Simple Web Server*, describes how to construct a fully functional web server in C. This program is able to serve a static website to any modern web browser.

`Chapter 8`, *Making Your Program Send Email*, describes **Simple Mail Transfer Protocol** (**SMTP**)—the protocol that is powering email. In this chapter, we develop a program that can send email over the internet.

`Chapter 9`, *Loading Secure Web Pages with HTTPS and OpenSSL*, explores TLS—the protocol that secures web pages. In this chapter, we develop an HTTPS client that is capable of downloading web pages securely.

`Chapter 10`, *Implementing a Secure Web Server*, continues on the security theme and explores the construction of a secure HTTPS web server.

`Chapter 11`, *Establishing SSH Connections with libssh*, continues with the secure protocol theme. The use of **Secure Shell** (**SSH**) is covered to connect to a remote server, execute commands, and download files securely.

`Chapter 12`, *Network Monitoring and Security*, discusses the tools and techniques used to test network functionality, troubleshoot problems, and eavesdrop on insecure communication protocols.

`Chapter 13`, *Socket Programming Tips and Pitfalls*, goes into detail about TCP and addresses many important edge cases that appear in socket programming. The techniques covered are invaluable for creating robust network programs.

`Chapter 14`, *Web Programming for the Internet of Things*, gives an overview of the design and programming for **Internet of Things** (**IoT**) applications.

`Appendix A`, *Answers to Questions*, provides answers to the comprehension questions given at the end of each chapter.

`Appendix B`, *Setting Up Your C Compiler on Windows*, gives instructions for setting up a development environment on Windows that is needed for compiling all of the example programs in this book.

`Appendix C`, *Setting Up Your C Compiler on Linux*, provides the setup instructions for preparing your Linux computer to be capable of compiling all of the example programs in this book.

`Appendix D`, *Setting Up Your C Compiler on macOS*, gives step-by-step instructions for configuring your macOS system to be capable of compiling all of the example programs in this book.

`Appendix E`, *Example Programs*, lists each example program, by chapter, included in this book's code repository.

To get the most out of this book

The reader is expected to be proficient in the C programming language. This includes a familiarity with memory management, the use of pointers, and basic data structures.

A Windows, Linux, or macOS development machine is recommended; you can refer to the appendices for setup instructions.

This book takes a hands-on approach to learning and includes 44 example programs. Working through these examples as you read the book will help enforce the concepts.

The code for this book is released under the MIT open source license. The reader is encouraged to use, modify, improve, and even publish their changes to these example programs.

Download the example code files

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

The code bundle for the book is also publicly hosted on GitHub at `https://github.com/codeplea/hands-on-network-programming-with-c`. In case there's an update to the code, it will be updated on that GitHub repository. Each chapter that introduces example programs begins with the commands needed to download the book's code.

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `http://www.packtpub.com/sites/default/files/downloads/9781789349863_ColorImages.pdf`.

Conventions used

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

`CodeInText`: Indicates code words in text, variable names, function names, directory names, filenames, file extensions, pathnames, URLs, and user input. Here is an example: "Use the `select()` function to wait for network data."

A block of code is set as follows:

```
/* example program */

#include <stdio.h>
int main() {
    printf("Hello World!\n");
    return 0;
}
```

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

```
gcc hello.c –o hello
./hello
```

Bold: Indicates a new term, an important word, or words that you see on screen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select **System info** from the **Administration** panel."

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packt.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

1
Section 1 - Getting Started with Network Programming

This section will get the reader up and running with the basics of networking, the relevant network protocols, and basic socket programming.

The following chapters are in this section:

`Chapter 1`, *An Introduction to Networks and Protocols*

`Chapter 2`, *Getting to Grips with Socket APIs*

`Chapter 3`, *An In-Depth Overview of TCP Connections*

`Chapter 4`, *Establishing UDP Connections*

`Chapter 5`, *Hostname Resolution and DNS*

1
Introducing Networks and Protocols

In this chapter, we will review the fundamentals of computer networking. We'll look at abstract models that attempt to explain the main concerns of networking, and we'll explain the operation of the primary network protocol, the Internet Protocol. We'll look at address families and end with writing programs to list your computer's local IP addresses.

The following topics are covered in this chapter:

- Network programming and C
- OSI layer model
- TCP/IP reference model
- The Internet Protocol
- IPv4 addresses and IPv6 addresses
- Domain names
- Internet protocol routing
- Network address translation
- The client-server paradigm
- Listing your IP addresses programmatically from C

Technical requirements

Most of this chapter focuses on theory and concepts. However, we do introduce some sample programs near the end. To compile these programs, you will need a good C compiler. We recommend MinGW on Windows and GCC on Linux and macOS. See `Appendix B`, *Setting Up Your C Compiler On Windows*, `Appendix C`, *Setting Up Your C Compiler On Linux*, and `Appendix D`, *Setting Up Your C Compiler On macOS*, for compiler setup.

The code for this book can be found at: `https://github.com/codeplea/Hands-On-Network-Programming-with-C`.

From the command line, you can download the code for this chapter with the following command:

```
git clone https://github.com/codeplea/Hands-On-Network-Programming-with-C
cd Hands-On-Network-Programming-with-C/chap01
```

On Windows, using MinGW, you can use the following command to compile and run code:

```
gcc win_list.c -o win_list.exe -liphlpapi -lws2_32
win_list
```

On Linux and macOS, you can use the following command:

```
gcc unix_list.c -o unix_list
./unix_list
```

The internet and C

Today, the internet needs no introduction. Certainly, millions of desktops, laptops, routers, and servers are connected to the internet and have been for decades. However, billions of additional devices are now connected as well—mobile phones, tablets, gaming systems, vehicles, refrigerators, television sets, industrial machinery, surveillance systems, doorbells, and even light bulbs. The new **Internet of Things** (**IoT**) trend has people rushing to connect even more unlikely devices every day.

Over 20 billion devices are estimated to be connected to the internet now. These devices use a wide variety of hardware. They connect over an Ethernet connection, Wi-Fi, cellular, a phone line, fiber optics, and other media, but they likely have one thing in common; they likely use **C**.

The use of the C programming language is ubiquitous. Almost every network stack is programmed in C. This is true for Windows, Linux, and macOS. If your mobile phone uses Android or iOS, then even though the apps for these were programmed in a different language (Java and Objective C), the kernel and networking code was written in C. It is very likely that the network routers that your internet data goes through are programmed in C. Even if the user interface and higher-level functions of your modem or router are programmed in another language, the networking drivers are still probably implemented in C.

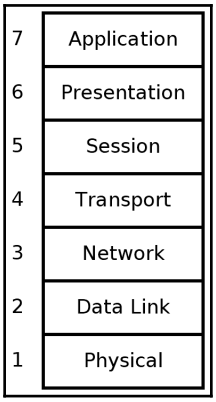
Networking encompasses concerns at many different abstraction levels. The concerns your web browser has with formatting a web page are much different than the concerns your router has with forwarding network packets. For this reason, it is useful to have a theoretical model that helps us to understand communications at these different levels of abstraction. Let's look at these models now.

OSI layer model

It's clear that if all of the disparate devices composing the internet are going to communicate seamlessly, there must be agreed-upon standards that define their communications. These standards are called **protocols**. Protocols define everything from the voltage levels on an Ethernet cable to how a JPEG image is compressed on a web page. It's clear that, when we talk about the voltage on an Ethernet cable, we are at a much different level of abstraction compared to talking about the JPEG image format. If you're programming a website, you don't want to think about Ethernet cables or Wi-Fi frequencies. Likewise, if you're programming an internet router, you don't want to have to worry about how JPEG images are compressed. For this reason, we break the problem down into many smaller pieces.

One common method of breaking down the problem is to place levels of concern into layers. Each layer then provides services for the layer on top of it, and each upper layer can rely on the layers underneath it without concern for how they work.

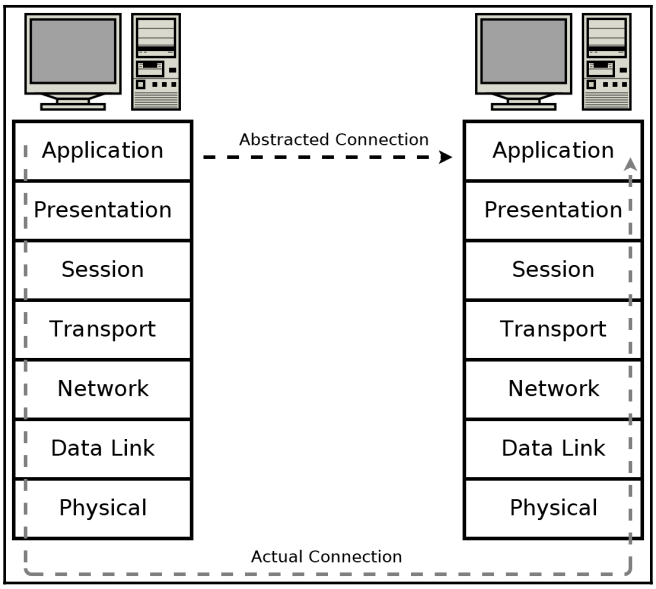
The most popular layer system for networking is called the **Open Systems Interconnection** model (**OSI** model). It was standardized in 1977 and is published as ISO 7498. It has seven layers:



Let's understand these layers one by one:

- **Physical** (1): This is the level of physical communication in the real world. At this level, we have specifications for things such as the voltage levels on an Ethernet cable, what each pin on a connector is for, the radio frequency of Wi-Fi, and the light flashes over an optic fiber.
- **Data Link** (2): This level builds on the physical layer. It deals with protocols for directly communicating between two nodes. It defines how a direct message between nodes starts and ends (framing), error detection and correction, and flow control.
- **Network layer** (3): The network layer provides the methods to transmit data sequences (called packets) between nodes in different networks. It provides methods to route packets from one node to another (without a direct physical connection) by transferring through many intermediate nodes. This is the layer that the Internet Protocol is defined on, which we will go into in some depth later.
- **Transport layer** (4): At this layer, we have methods to reliably deliver variable length data between hosts. These methods deal with splitting up data, recombining it, ensuring data arrives in order, and so on. The **Transmission Control Protocol** (**TCP**) and **User Datagram Protocol** (**UDP**) are commonly said to exist on this layer.
- **Session layer** (5): This layer builds on the transport layer by adding methods to establish, checkpoint, suspend, resume, and terminate dialogs.
- **Presentation layer** (6): This is the lowest layer at which data structure and presentation for an application are defined. Concerns such as data encoding, serialization, and encryption are handled here.
- **Application layer** (7): The applications that the user interfaces with (for example, web browsers and email clients) exist here. These applications make use of the services provided by the six lower layers.

In the OSI model, an application, such as a web browser, exists in the application layer (layer 7). A protocol from this layer, such as HTTP used to transmit web pages, doesn't have to concern itself with how the data is being transmitted. It can rely on services provided by the layer underneath it to effectively transmit data. This is illustrated in the following diagram:



It should be noted that chunks of data are often referred to by different names depending on the OSI layer they're on. A data unit on layer 2 is called a **frame**, since layer 2 is responsible for framing messages. A data unit on layer 3 is referred to as a **packet**, while a data unit on layer 4 is a **segment** if it is part of a TCP connection or a **datagram** if it is a UDP message.

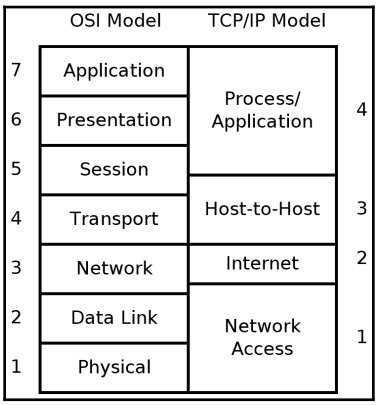
In this book, we often use the term packet as a generic term to refer to a data unit on any layer. However, segment will only be used in the context of a TCP connection, and datagram will only refer to UDP datagrams.

As we will see in the next section, the OSI model doesn't fit precisely with the common protocols in use today. However, it is still a handy model to explain networking concerns, and it is still in widespread use for that purpose today.

TCP/IP layer model

The **TCP/IP protocol suite** is the most common network communication model in use today. The TCP/IP reference model differs a bit from the OSI model, as it has only four layers instead of seven.

The following diagram illustrates how the four layers of the TCP/IP model line up to the seven layers of the OSI model:



Notably, the TCP/IP model doesn't match up exactly with the layers in the OSI model. That's OK. In both models, the same functions are performed; they are just divided differently.

The TCP/IP reference model was developed after the TCP/IP protocol was already in common use. It differs from the OSI model by subscribing a less rigid, although still hierarchical, model. For this reason, the OSI model is sometimes better for understanding and reasoning about networking concerns, but the TCP/IP model reflects a more realistic view of how networking is commonly implemented today.

The four layers of the TCP/IP model are as follows:

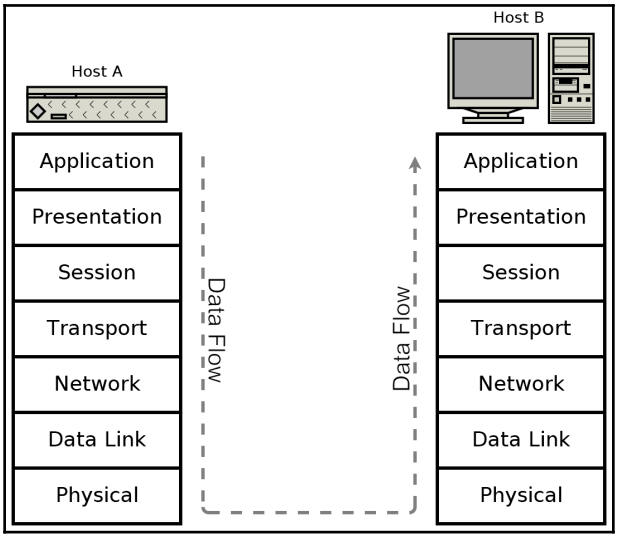
- **Network Access layer** (1): On this layer, physical connections and data framing happen. Sending an Ethernet or Wi-Fi packet are examples of layer 1 concerns.
- **Internet layer** (2): This layer deals with the concerns of addressing packets and routing them over multiple interconnection networks. It's at this layer that an IP address is defined.
- **Host-to-Host layer** (3): The host-to-host layer provides two protocols, TCP and UDP, which we will discuss in the next few chapters. These protocols address concerns such as data order, data segmentation, network congestion, and error correction.
- **Process/Application layer** (4): The process/application layer is where protocols such as HTTP, SMTP, and FTP are implemented. Most of the programs that feature in this book could be considered to take place on this layer while consuming functionality provided by our operating system's implementation of the lower layers.

Regardless of your chosen abstraction model, real-world protocols do work at many levels. Lower levels are responsible for handling data for the higher levels. These lower-level data structures must, therefore, encapsulate data from the higher levels. Let's look at encapsulating data now.

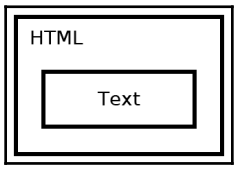
Data encapsulation

The advantage of these abstractions is that, when programming an application, we only need to consider the highest-level protocol. For example, a web browser needs only to implement the protocols dealing specifically with websites—HTTP, HTML, CSS, and so on. It does not need to bother with implementing TCP/IP, and it certainly doesn't have to understand how an Ethernet or Wi-Fi packet is encoded. It can rely on ready-made implementations of the lower layers for these tasks. These implementations are provided by the operating system (for example, Windows, Linux, and macOS).

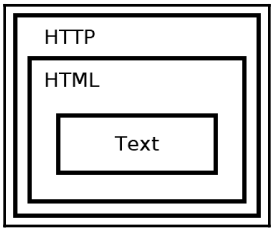
When communicating over a network, data must be processed down through the layers at the sender and up again through the layers at the receiver. For example, if we have a web server, **Host A**, which is transmitting a web page to the receiver, **Host B**, it may look like this:



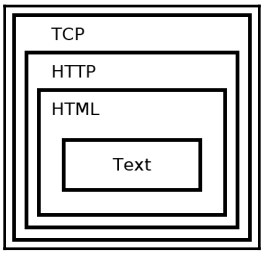
The web page contains a few paragraphs of text, but the web server doesn't only send the text by itself. For the text to be rendered correctly, it must be encoded in an **HTML** structure:



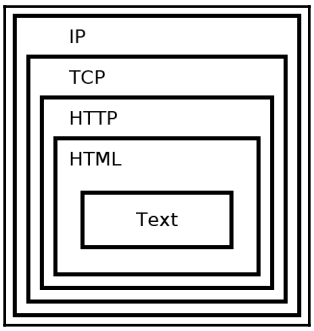
In some cases, the text is already preformatted into **HTML** and saved that way but, in this example, we are considering a web application that dynamically generates the **HTML**, which is the most common paradigm for dynamic web pages. As the text cannot be transmitted directly, neither can the **HTML**. It instead must be transmitted as part of an **HTTP** response. The web server does this by applying the appropriate **HTTP** response header to the **HTML**:



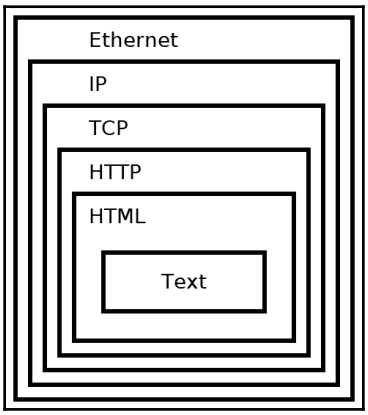
The **HTTP** is transmitted as part of a **TCP** session. This isn't done explicitly by the web server, but is taken care of by the operating system's TCP/IP stack:



The **TCP** packet is routed by an **IP** packet:



This is transmitted over the wire in an **Ethernet** packet (or another protocol):



Luckily for us, the lower-level concerns are handled automatically when we use the socket APIs for network programming. It is still useful to know what happens behind the scenes. Without this knowledge, dealing with failures or optimizing for performance is difficult if not impossible.

With some of the theory out of the way, let's dive into the actual protocols powering modern networking.

Internet Protocol

Twenty years ago, there were many competing networking protocols. Today, one protocol is overwhelmingly common—the Internet Protocol. It comes in two versions—IPv4 and IPv6. IPv4 is completely ubiquitous and deployed everywhere. If you're deploying network code today, you must support IPv4 or risk that a significant portion of your users won't be able to connect.

IPv4 uses 32-bit addresses, which limits it to addressing no more than 2^{32} or 4,294,967,296 systems. However, these 4.3 billion addresses were not initially assigned efficiently, and now many **Internet Service Providers** (**ISPs**) are forced to ration IPv4 addresses.

IPv6 was designed to replace IPv4 and has been standardized by the **Internet Engineering Task Force** (**IETF**) since 1998. It uses a 128-bit address, which allows it to address a theoretical 2^{128} = 340,282,366,920,938,463,463,374,607,431,768,211,456, or about a 3.4×10^{38} addresses.

Today, every major desktop and smartphone operating system supports both IPv4 and IPv6 in what is called a **dual-stack configuration**. However, many applications, servers, and networks are still only configured to use IPv4. From a practical standpoint, this means that you need to support IPv4 in order to access much of the internet. However, you should also support IPv6 to be future-proof and to help the world to transition away from IPv4.

What is an address?

All Internet Protocol traffic routes to an address. This is similar to how phone calls must be dialed to phone numbers. IPv4 addresses are 32 bits long. They are commonly divided into four 8-bit sections. Each section is displayed as a decimal number between `0` and `255` inclusive and is delineated by a period.

Here are some examples of IPv4 addresses:

- `0.0.0.0`
- `127.0.0.1`
- `10.0.0.0`
- `172.16.0.5`
- `192.168.0.1`
- `192.168.50.1`
- `255.255.255.255`

A special address, called the **loopback** address, is reserved at `127.0.0.1`. This address essentially means *establish a connection to myself*. Operating systems short-circuit this address so that packets to it never enter the network but instead stay local on the originating system.

IPv4 reserves some address ranges for private use. If you're using IPv4 through a router/NAT, then you are likely using an IP address in one of these ranges. These reserved private ranges are as follows:

- `10.0.0.0` to `10.255.255.255`
- `172.16.0.0` to `172.31.255.255`
- `192.168.0.0` to `192.168.255.255`

The concept of IP address ranges is a useful one that comes up many times in networking. It's probably not surprising then that there is a shorthand notation for writing them. Using **Classless Inter-Domain Routing** (**CIDR**) notation, we can write the three previous address ranges as follows:

- `10.0.0.0/8`
- `172.16.0.0/12`
- `192.168.0.0/16`

CIDR notation works by specifying the number of bits that are fixed. For example, `10.0.0.0/8` specifies that the first eight bits of the `10.0.0.0` address are fixed, the first eight bits being just the first `10.` part; the remaining `0.0.0` part of the address can be anything and still be on the `10.0.0.0/8` block. Therefore, `10.0.0.0/8` encompasses `10.0.0.0` through `10.255.255.255`.

IPv6 addresses are 128 bits long. They are written as eight groups of four hexadecimal characters delineated by colons. A hexadecimal character can be from 0-9 or from a-f. Here are some examples of IPv6 addresses:

- `0000:0000:0000:0000:0000:0000:0000:0001`
- `2001:0db8:0000:0000:0000:ff00:0042:8329`
- `fe80:0000:0000:0000:75f4:ac69:5fa7:67f9`
- `ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff`

Note that the standard is to use lowercase letters in IPv6 addresses. This is in contrast to many other uses of hexadecimal in computers.

There are a couple of rules for shortening IPv6 addresses to make them easier. Rule 1 allows for the leading zeros in each section to be omitted (for example, `0db8` = `db8`). Rule 2 allows for consecutive sections of zeros to be replaced with a double colon (`::`). Rule 2 may only be used once in each address; otherwise, the address would be ambiguous.

Applying both rules, the preceding addresses can be shortened as follows:

- `::1`
- `2001:db8::ff00:42:8329`
- `fe80::75f4:ac69:5fa7:67f9`
- `ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff`

Like IPv4, IPv6 also has a loopback address. It is `::1`.

Dual-stack implementations also recognize a special class of IPv6 address that map directly to an IPv4 address. These reserved addresses start with 80 zero bits, and then by 16 one bits, followed by the 32-bit IPv4 address. Using CIDR notation, this block of address is `::ffff:0:0/96`.

These mapped addresses are commonly written with the first 96 bits in IPv6 format followed by the remaining 32 bits in IPv4 format. Here are some examples:

IPv6 Address	**Mapped IPv4 Address**
`::ffff:10.0.0.0`	`10.0.0.0`
`::ffff:172.16.0.5`	`172.16.0.5`
`::ffff:192.168.0.1`	`192.168.0.1`
`::ffff:192.168.50.1`	`192.168.50.1`

You may also run into IPv6 **site-local addresses**. These site-local addresses are in the `fec0::/10` range and are for use on private local networks. Site-local addresses have now been deprecated and should not be used for new networks, but many existing implementations still use them.

Another address type that you should be familiar with are **link-local addresses**. Link-local addresses are usable only on the local link. Routers never forward packets from these addresses. They are useful for a system to accesses auto-configuration functions before having an assigned IP address. Link-local addresses are in the IPv4 `169.254.0.0/16` address block or the IPv6 `fe80::/10` address block.

It should be noted the IPv6 introduces many additional features over IPv4 besides just a greatly expanded address range. IPv6 addresses have new attributes, such as scope and lifetime, and it is normal for IPv6 network interfaces to have multiple IPv6 addresses. IPv6 addresses are used and managed differently than IPv4 addresses.

Regardless of these differences, in this book, we strive to write code that works well for both IPv4 and IPv6.

If you think that IPv4 addresses are difficult to memorize, and IPv6 addresses impossible, then you are not alone. Luckily, we have a system to assign names to specific addresses.

Domain names

The Internet Protocol can only route packets to an IP address, not a name. So, if you try to connect to a website, such as `example.com`, your system must first resolve that domain name, `example.com`, into an IP address for the server that hosts that website.

This is done by connecting to a **Domain Name System** (**DNS**) server. You connect to a domain name server by knowing in advance its IP address. The IP address for a domain name server is usually assigned by your ISP.

Many other domain name servers are made publicly available by different organizations. Here are a few free and public DNS servers:

DNS Provider	IPv4 Addresses	IPv6 Addresses
Cloudflare 1.1.1.1	`1.1.1.1`	`2606:4700:4700::1111`
	`1.0.0.1`	`2606:4700:4700::1001`
FreeDNS	`37.235.1.174`	
	`37.235.1.177`	
Google Public DNS	`8.8.8.8`	`2001:4860:4860::8888`
	`8.8.4.4`	`2001:4860:4860::8844`
OpenDNS	`208.67.222.222`	`2620:0:ccc::2`
	`208.67.220.220`	`2620:0:ccd::2`

To resolve a hostname, your computer sends a UDP message to your domain name server and asks it for an AAAA-type record for the domain you're trying to resolve. If this record exists, an IPv6 address is returned. You can then connect to a server at that address to load the website. If no AAAA record exists, then your computer queries the server again, but asks for an A record. If this record exists, you will receive an IPv4 address for the server. In many cases, a site will publish an A record and an AAAA record that route to the same server.

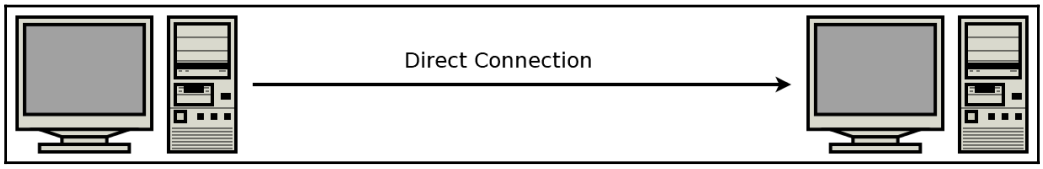
It is also possible, and common, for multiple records of the same type to exist, each pointing to a different address. This is useful for redundancy in the case where multiple servers can provide the same service.

We will see a lot more about DNS queries in `Chapter 5`, *Hostname Resolution and DNS*.

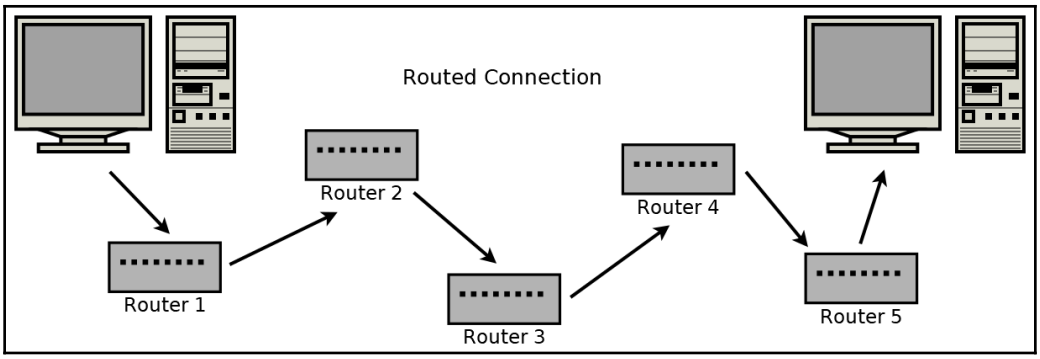
Now that we have a basic understanding of IP addresses and names, let's look into detail of how IP packets are routed over the internet.

Internet routing

If all networks contained only a maximum of only two devices, then there would be no need for routing. Computer A would just send its data directly over the wire, and computer B would receive it as the only possibility:



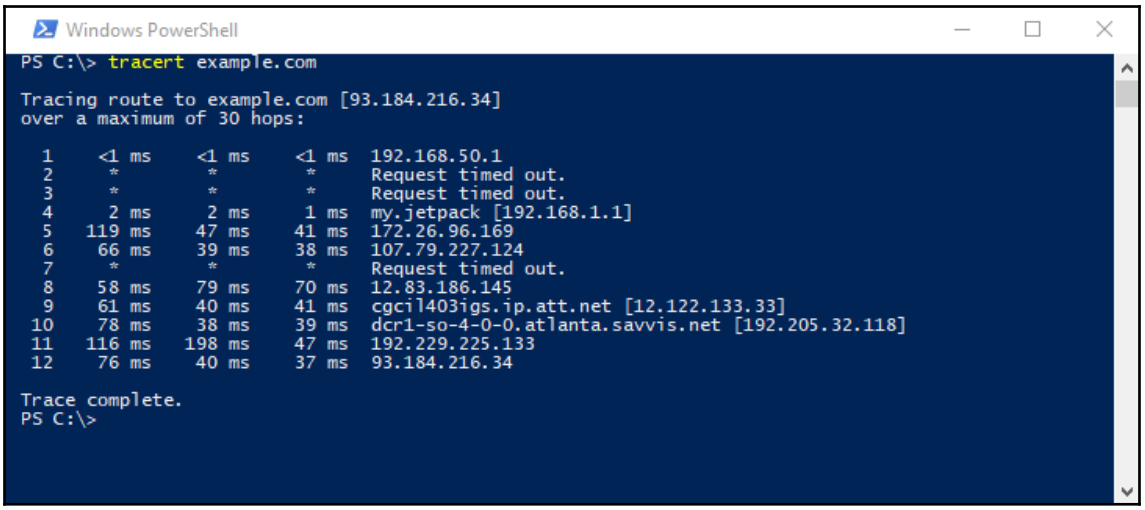
The internet today has an estimated 20 billion devices connected. When you make a connection over the internet, your data first transmits to your local router. From there, it is transmitted to another router, which is connected to another router, and so on. Eventually, your data reaches a router that is connected to the receiving device, at which point, the data has reached its destination:



Imagine that each router in the preceding diagram is connected to tens, hundreds, or even thousands of other routers and systems. It's an amazing feat that IP can discover the correct path and deliver traffic seamlessly.

Windows includes a utility, `tracert`, which lists the routers between your system and the destination system.

Here is an example of using the `tracert` command on Windows 10 to trace the route to `example.com`:



As you can see from the example, there are 11 hops between our system and the destination system (`example.com`, `93.184.216.34`). The IP addresses are listed for many of these intermediate routers, but a few are missing with the `Request timed out` message. This usually means that the system in question doesn't support the part of the **Internet Control Message Protocol** (**ICMP**) protocol needed. It's not unusual to see a few such systems when running `tracert`.

In Unix-based systems, the utility to trace routes is called `traceroute`. You would use it like `traceroute example.com`, for example, but the information obtained is essentially the same.

More information on `tracert` and `traceroute` can be found in `Chapter 12`, *Network Monitoring and Security*.

Sometimes, when IP packets are transferred between networks, their addresses must be translated. This is especially common when using IPv4. Let's look at the mechanism for this next.

Local networks and address translation

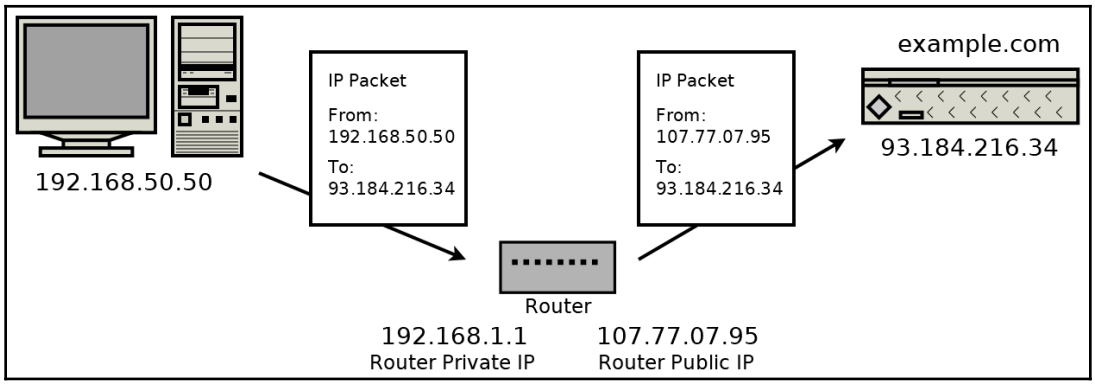
It's common for households and organizations to have small **Local Area Networks** (**LANs**). As mentioned previously, there are IPv4 addresses ranges reserved for use in these small local networks.

These reserved private ranges are as follows:

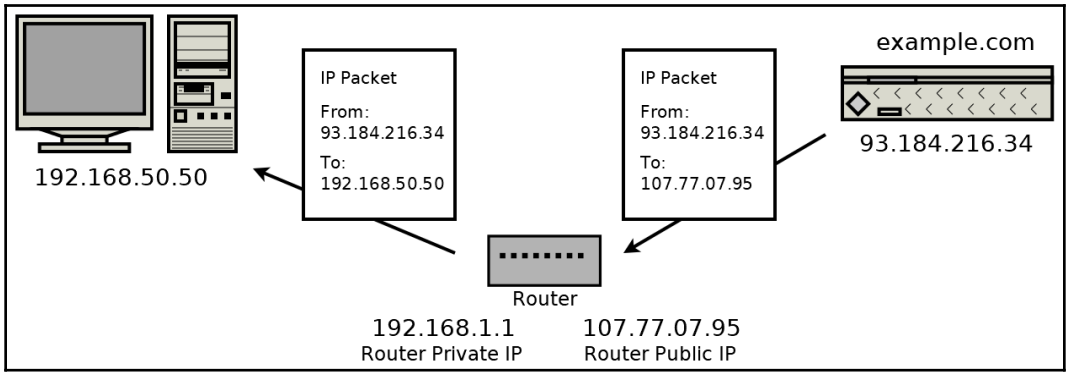
- `10.0.0.0` to `10.255.255.255`
- `172.16.0.0` to `172.31.255.255`
- `192.168.0.0` to `192.168.255.255`

When a packet originates from a device on an IPv4 local network, it must undergo **Network Address Translation** (**NAT**) before being routed on the internet. A router that implements NAT remembers which local address a connection is established from.

The devices on the same LAN can directly address one another by their local address. However, any traffic communicated to the internet must undergo address translation by the router. The router does this by modifying the source IP address from the original private LAN IP address to its public internet IP address:



Likewise, when the router receives the return communication, it must modify the destination address from its public IP to the private IP of the original sender. It knows the private IP address because it was stored in memory after the first outgoing packet:



Network address translation can be more complicated than it first appears. In addition to modifying the source IP address in the packet, it must also update the checksums in the packet. Otherwise, the packet would be detected as containing errors and discarded by the next router. The NAT router must also remember which private IP address sent the packet in order to route the reply. Without remembering the translation address, the NAT router wouldn't know where to send the reply to on the private network.

NATs will also modify the packet data in some cases. For example, in the **File Transfer Protocol** (**FTP**), some connection information is sent as part of the packet's data. In these cases, the NAT router will look at the packet's data in order to know how to forward future incoming packets. IPv6 largely avoids the need for NAT, as it is possible (and common) for each device to have its own publicly-addressable address.

You may be wondering how a router knows whether a message is locally deliverable or whether it must be forwarded. This is done using a netmask, subnet mask, or CIDR.

Subnetting and CIDR

IP addresses can be split into parts. The most significant bits are used to identify the network or subnetwork, and the least significant bits are used to identify the specific device on the network.

This is similar to how your home address can be split into parts. Your home address includes a house number, a street name, and a city. The city is analogous to the network part, the street name could be the subnetwork part, and your house number is the device part.

IPv4 traditionally uses a mask notation to identify the IP address parts. For example, consider a router on the `10.0.0.0` network with a subnet mask of `255.255.255.0`. This router can take any incoming packet and perform a bitwise `AND` operation with the subnet mask to determine whether the packet belongs on the local subnet or needs to be forwarded on. For example, this router receives a packet to be delivered to `10.0.0.105`. It does a bitwise `AND` operation on this address with the subnet mask of `255.255.255.0`, which produces `10.0.0.0`. That matches the subnet of the router, so the traffic is local. If, instead, we consider a packet destined for `10.0.15.22`, the result of the bitwise AND with the subnet mask is `10.0.15.0`. This address doesn't match the subnet the router is on, and so it must be forwarded.

IPv6 uses CIDR. Networks and subnetworks are specified using the CIDR notation we described earlier. For example, if the IPv6 subnet is `/112`, then the router knows that any address that matches on the first 112 bits is on the local subnet.

So far, we've covered only routing with one sender and one receiver. While this is the most common situation, let's consider alternative cases too.