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.

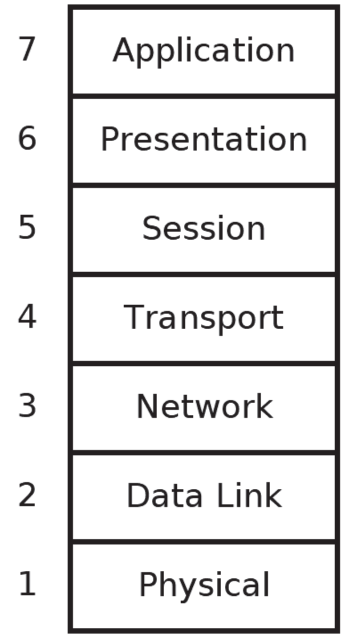
Networking encompasses concerns at many different abstraction levels. The concerns your web browser has with formatting a web page are much different than the concerns your router has with forwarding network packets. For this reason, it is useful to have a theoretical model that helps us to understand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

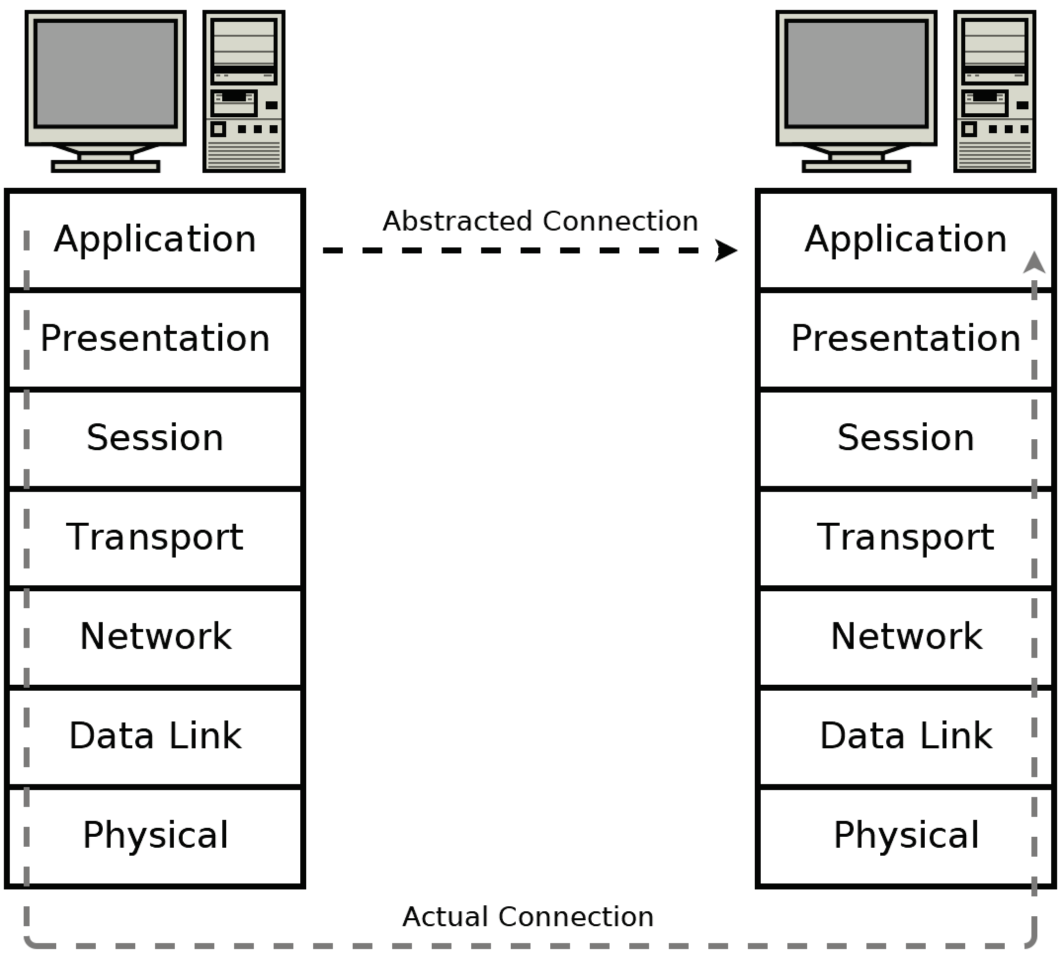


Let's understand these layers one by one:

- **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:



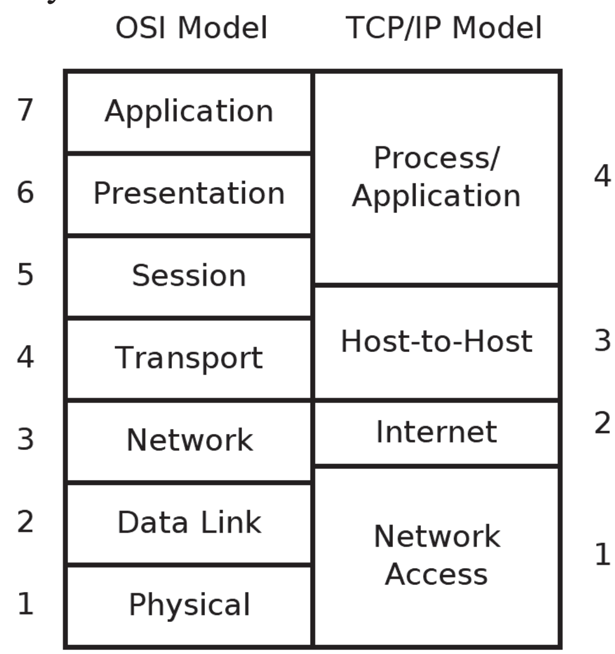
It should be noted that chunks of data are often referred to by different names depending on the OSI layer they're on. A data unit on layer 2 is called a **frame**, since layer 2 is responsible for framing messages. A data unit on layer 3 is referred to as a **packet**, while a data unit on layer 4 is 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:

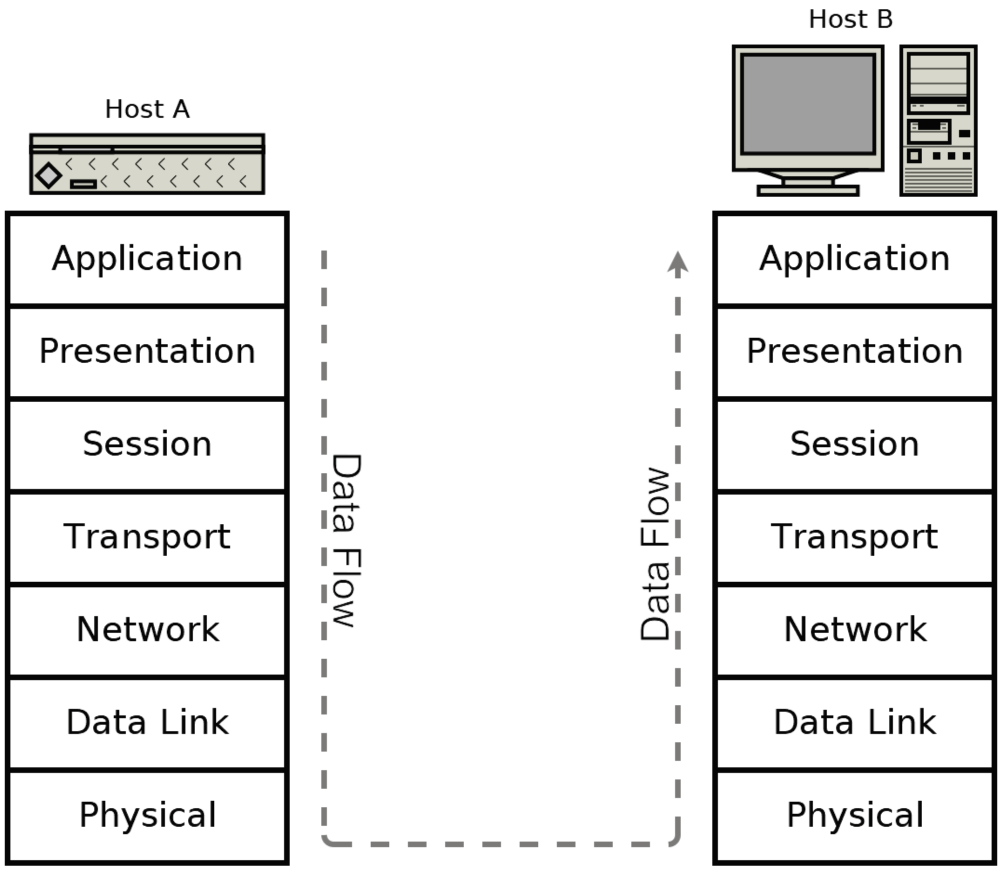
- **Network Access layer** (1): On this layer, physical connections and data framing happen. Sending an Ethernet or Wi-Fi packet are examples of layer 1 concerns.
- **Internet layer** (2): This layer deals with the concerns of addressing packets and routing them over multiple interconnection networks. 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.

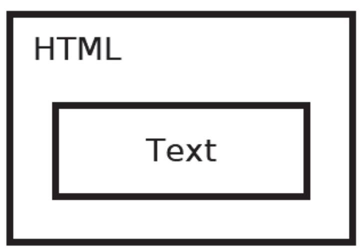
Data encapsulation

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).

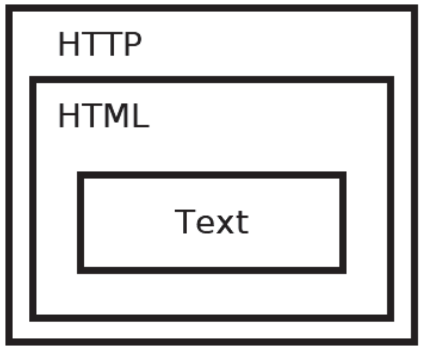
When communicating over a network, data must be processed down through the layers at the sender and up again through the layers at the receiver. For example, if we have a web server, **Host A**, which is transmitting a web page to the receiver, **Host B**, it may look like this:



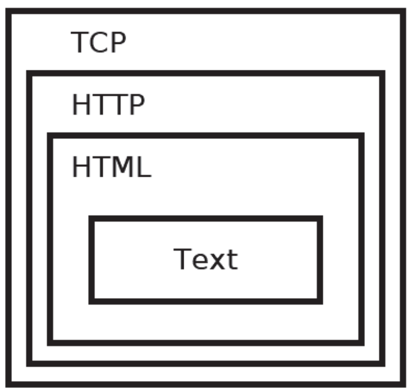
The web page contains a few paragraphs of text, but the web server doesn't only send the text by itself. For the text to be rendered correctly, it must be encoded in an **HTML** structure:



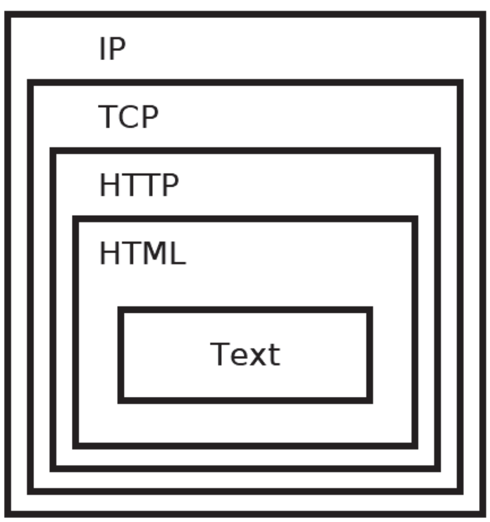
In some cases, the text is already preformatted into **HTML** and saved that way but, in this example, we are considering a web application that dynamically generates the **HTML**, which is the most common paradigm for dynamic web pages. As the text cannot be transmitted directly, neither can the **HTML**. 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**:



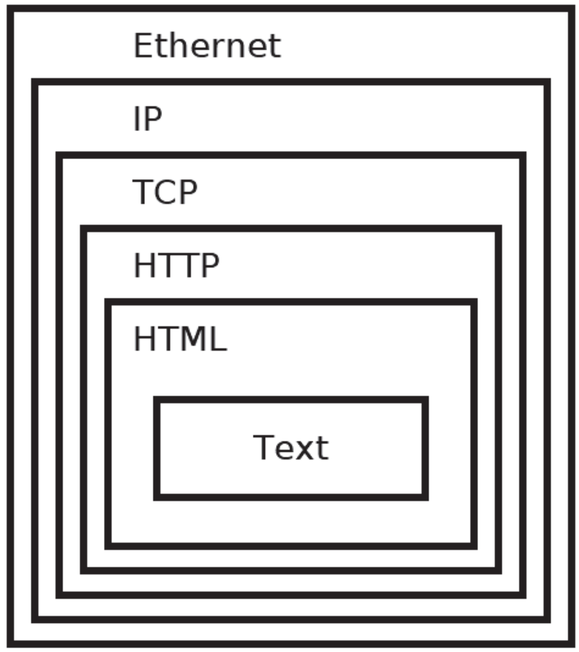
The **HTTP** is transmitted as part of a **TCP** session. This isn't done explicitly by the web server, but is taken care of by the operating system's TCP/IP stack:



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 x 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.

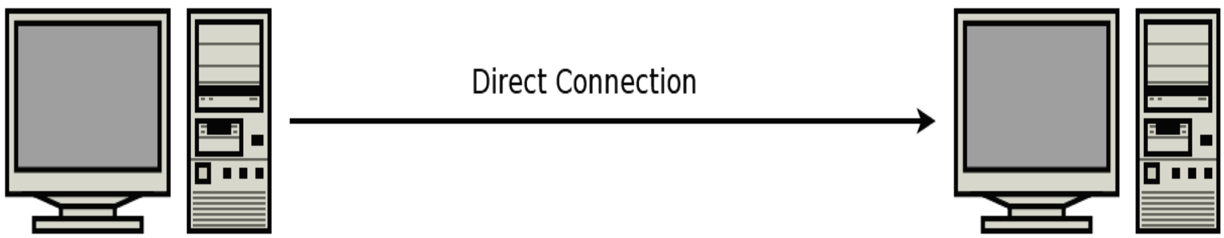
It is also possible, and common, for multiple records of the same type to exist, each pointing to a different address. This is useful for redundancy in the case where multiple servers can provide the same service.

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

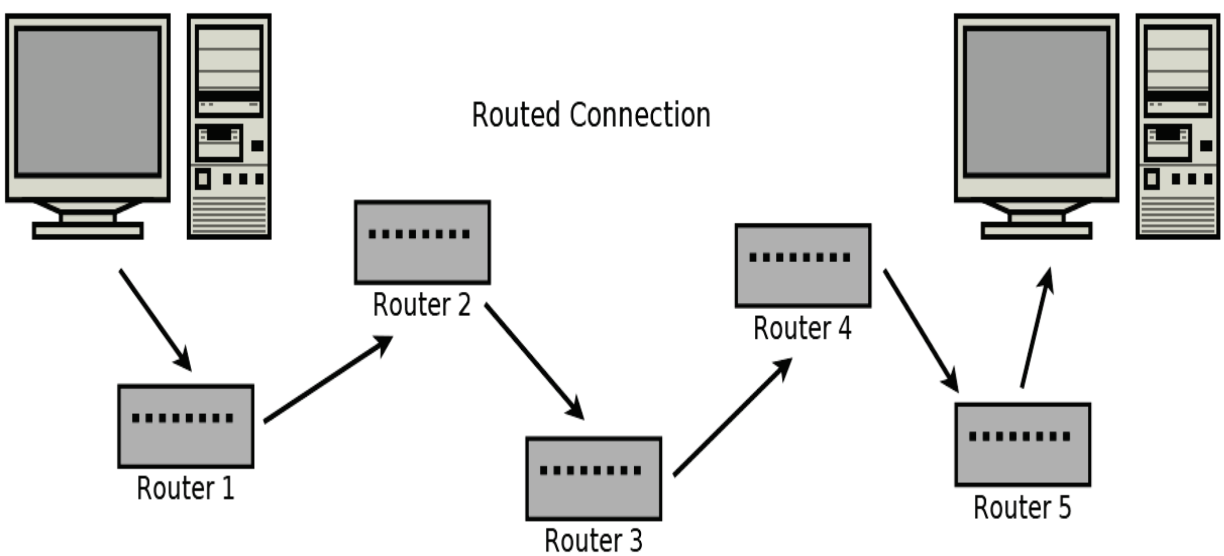
Now that we have a basic understanding of IP addresses and names, let's look into detail of how IP packets are routed over the internet.

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:



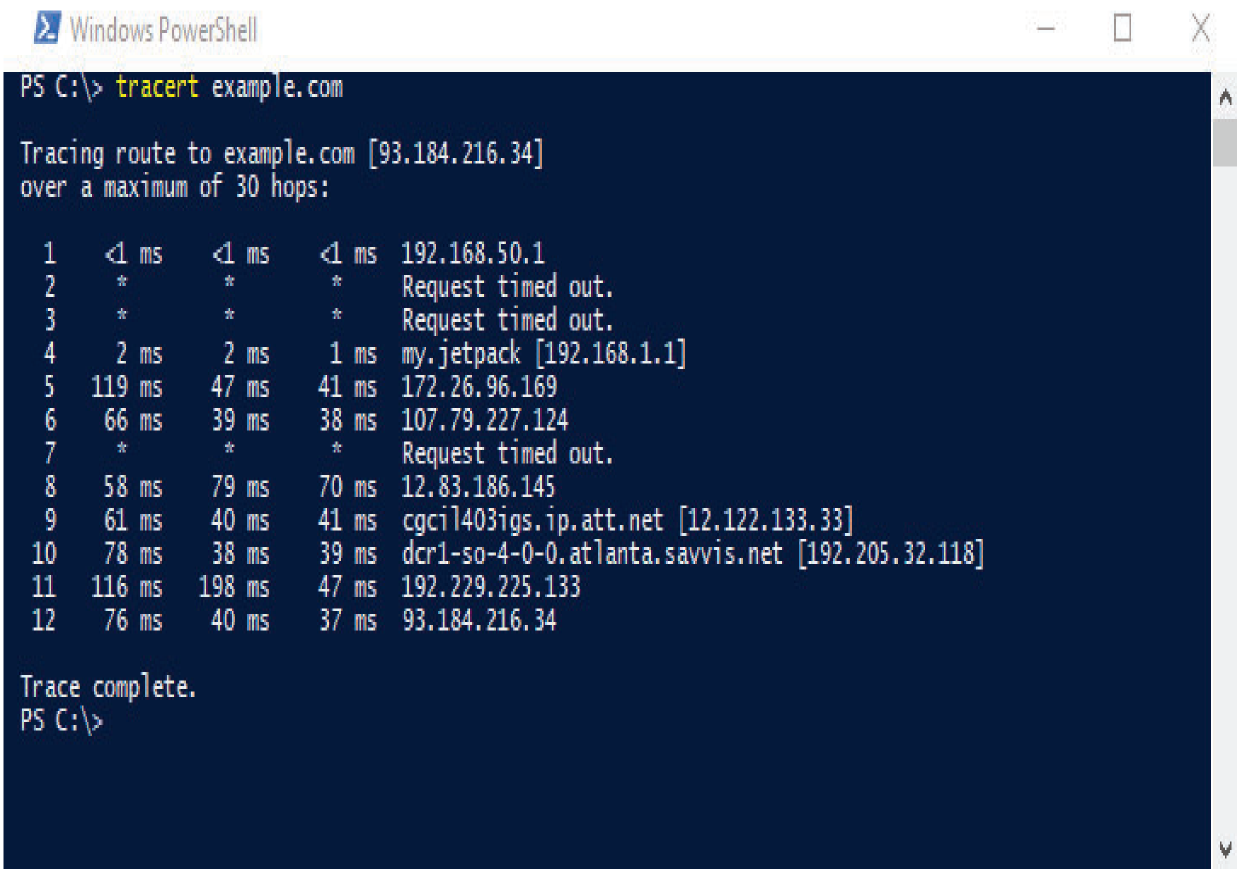
The internet today has an estimated 20 billion devices connected. When you make a connection over the internet, your data first transmits to your local router. From there, it is transmitted to another router, which is connected to another router, and so on. Eventually, your data reaches a router that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

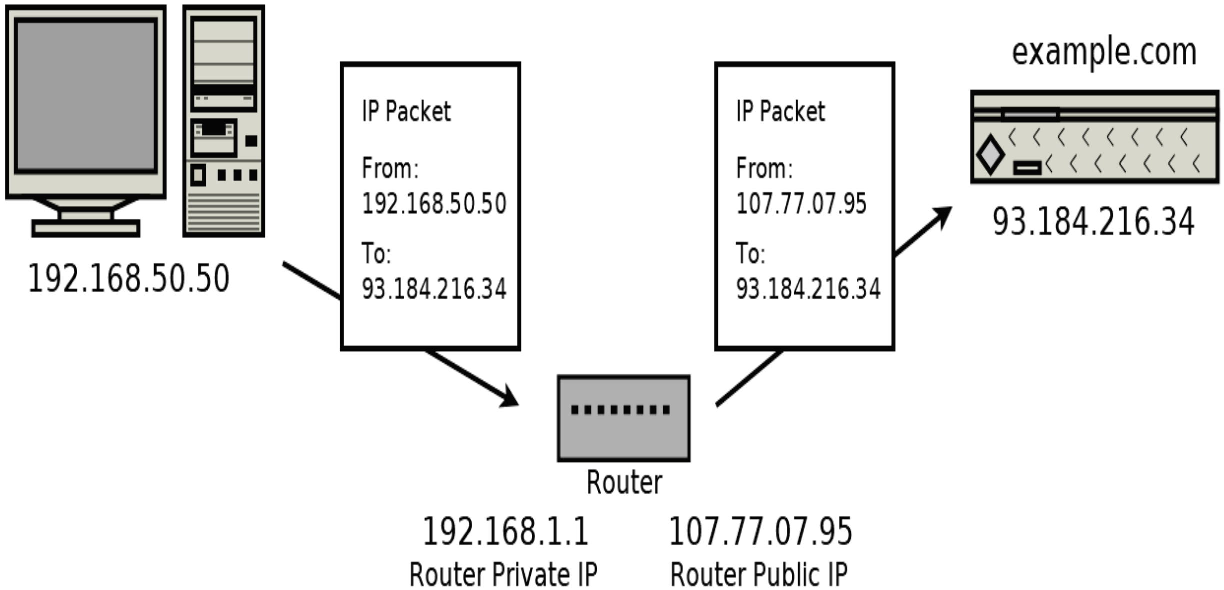
It's common for households and organizations to have small **Local Area Networks** (**LANs**). As mentioned previously, there are IPv4 addresses ranges reserved for use in these small local networks.

These reserved private ranges are as follows:

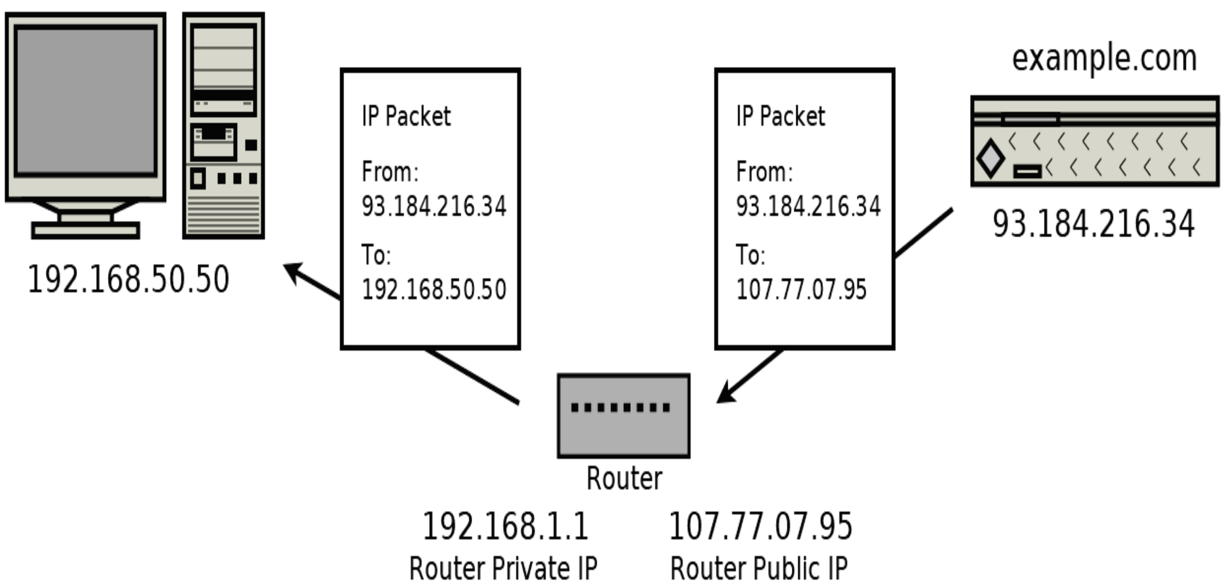
- `10.0.0.0` to `10.255.255.255`
- `172.16.0.0` to `172.31.255.255`
- `192.168.0.0` to `192.168.255.255`

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.

Multicast, broadcast, and anycast

When a packet is routed from one sender to one receiver, it uses **unicast** addressing. This is the simplest and most common type of addressing. All of the protocols we deal with in this book use unicast addressing.

Broadcast addressing allows a single sender to address a packet to all recipients simultaneously. It is typically used to deliver a packet to every receiver on an entire subnet.

If a broadcast is a one-to-all communication, then **multicast** is a one-to-many communication. Multicast involves some group management, and a message is addressed and delivered to members of a group.

Anycast addressed packets are used to deliver a message to one recipient when you don't care who that recipient is. This is useful if you have several servers that provide the same functionality, and you simply want one of them (you don't care which) to handle your request.

IPv4 and lower network levels support local broadcast addressing. IPv4 provides some optional (but commonly implemented) support for multicasting. IPv6 mandates multicasting support while providing additional features over IPv4's multicasting. Though IPv6 is not considered to broadcast, its multicasting functionality can essentially emulate it.

It's worth noting that these alternative addressing methods don't generally work over the broader internet. Imagine if one peer was able to broadcast a packet to every connected internet device. It would be a mess!

If you can use IP multicasting on your local network, though, it is worthwhile to implement it. Sending one IP level multicast conserves bandwidth compared to sending the same unicast message multiple times.

However, multicasting is often done at the application level. That is, when the application wants to deliver the same message to several recipients, it sends the message multiple times – once to each recipient. In `Chapter 3`, *An In-Depth Overview of TCP Connections*, we build a chat room. This chat room could be said to use application-level multicasting, but it does not take advantage of IP multicasting.

We've covered how messages are routed through a network. Now, let's see how a message knows which application is responsible for it once it arrives at a specific system.

Port numbers

An IP address alone isn't quite enough. We need port numbers. To return to the telephone analogy, if IP addresses are phone numbers, then port numbers are like phone extensions.

Generally, an IP address gets a packet routed to a specific system, but a port number is used to route the packet to a specific application on that system.

For example, on your system, you may be running multiple web browsers, an email client, and a video-conferencing client. When your computer receives a TCP segment or UDP datagram, your operating system looks at the destination port number in that packet. That port number is used to look up which application should handle it.

Port numbers are stored as unsigned 16-bit integers. This means that they are between `0` and `65,535` inclusive.

Some port numbers for common protocols are as follows:

Port Number		**Protocol**	
`20`, `21`	TCP	**File Transfer Protocol (FTP)**	
`22`	TCP	**Secure Shell (SSH)**	`Chapter 11`, *Establishing SSH Connections with libssh*
`23`	TCP	**Telnet**	
`25`	TCP	**Simple Mail Transfer Protocol (SMTP)**	`Chapter 8`, *Making Your Program Send Email*
`53`	UDP	**Domain Name System**	`Chapter 5`, *Hostname*

		(DNS)	*Resolution and DNS*
80	TCP	**Hypertext Transfer Protocol (HTTP)**	Chapter 6, *Building a Simple Web Client* Chapter 7, *Building a Simple Web Server*
110	TCP	**Post Office Protocol, Version 3 (POP3)**	
143	TCP	**Internet Message Access Protocol (IMAP)**	
194	TCP	**Internet Relay Chat (IRC)**	
443	TCP	**HTTP over TLS/SSL (HTTPS)**	Chapter 9, *Loading Secure Web Pages with HTTPS and OpenSSL* Chapter 10, *Implementing a Secure Web Server*
993	TCP	**IMAP over TLS/SSL (IMAPS)**	
995	TCP	**POP3 over TLS/SSL (POP3S)**	

Each of these listed port numbers is assigned by the **Internet Assigned Numbers Authority** (**IANA**). They are responsible for the official assignments of port numbers for specific protocols. Unofficial port usage is very common for applications implementing custom protocols. In this case, the application should try to choose a port number that is not in common use to avoid conflict.

Clients and servers

In the telephone analogy, a call must be initiated first by one party. The initiating party dials the number for the receiving party, and the receiving party answers.

This is also a common paradigm in networking called the **client-server** model. In this model, a server listens for connections. The client, knowing the address and port number that the server is listening on, establishes the connection by sending the first packet.

For example, the web server at `example.com` listens on port `80` (HTTP) and port `443` (HTTPS). A web browser (client) must establish the connection by sending the first packet to the web server address and port.

Putting it together

A socket is one end-point of a communication link between systems. It's an abstraction in which your application can send and receive data over the network, in much the same way that your application can read and write to a file using a file handle.

An open socket is uniquely defined by a 5-tuple consisting of the following:

- Local IP address
- Local port
- Remote IP address
- Remote port
- Protocol (UDP or TCP)

This 5-tuple is important, as it is how your operating system knows which application is responsible for any packets received. For example, if you use two web browsers to establish two simultaneous connections to `example.com` on port `80`, then your operating system keeps the connections separate by looking at the local IP address, local port, remote IP address, remote port, and protocol. In this case, the local IP addresses, remote IP addresses, remote port (`80`), and protocol (TCP) are identical.

The deciding factor then is the local port (also called the **ephemeral port**), which will have been chosen to be different by the operating system for connection. This 5-tuple is also important to understand how NAT works. A private network may have many systems accessing the same outside resource, and the router NAT must store this five tuple for each connection in order to know how to route received packets back into the private network.

What's your address?

You can find your IP address using the `ipconfig` command on Windows, or the `ifconfig` command on Unix-based systems (such as Linux and macOS).

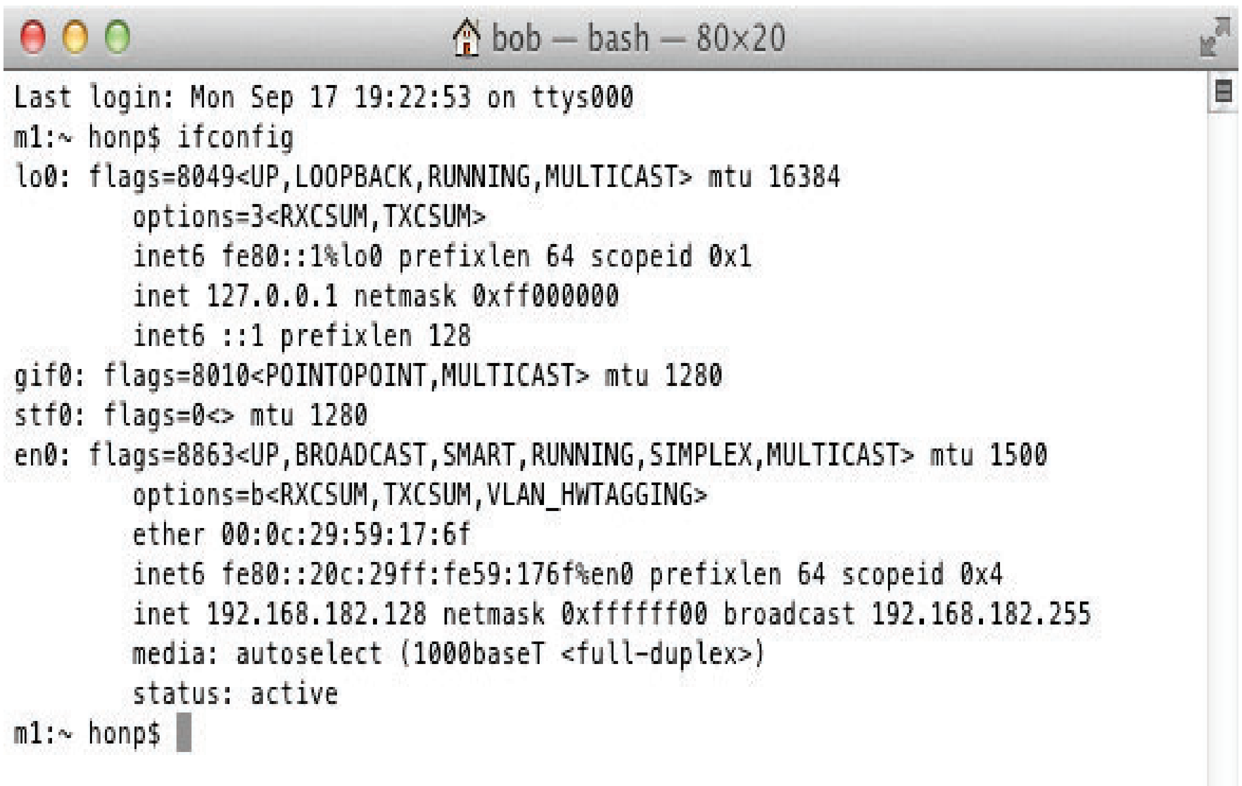
Using the `ipconfig` command from Windows PowerShell looks like this:

```
Windows PowerShell

PS C:\Users\honp> ipconfig

Windows IP Configuration


Ethernet adapter Ethernet0:

   Connection-specific DNS Suffix  . : localdomain
   Link-local IPv6 Address . . . . . : fe80::cd70:e700:5486:fb1a%5
   IPv4 Address. . . . . . . . . . . : 192.168.182.133
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.182.2

Tunnel adapter isatap.localdomain:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . : localdomain

Tunnel adapter Local Area Connection* 2:

   Connection-specific DNS Suffix  . :
   IPv6 Address. . . . . . . . . . . : 2001:0:9d38:6ab8:8ba:2c5a:5950:c03c
   Link-local IPv6 Address . . . . . : fe80::8ba:2c5a:5950:c03c%2
   Default Gateway . . . . . . . . . : ::
PS C:\Users\honp>
```

In this example, you can find that the IPv4 address is listed under `Ethernet adapter Ethernet0`. Your system may have more network adapters, and each will have its own IP address. We can tell that this computer is on a local network because the IP address, `192.168.182.133`, is in the private IP address range.

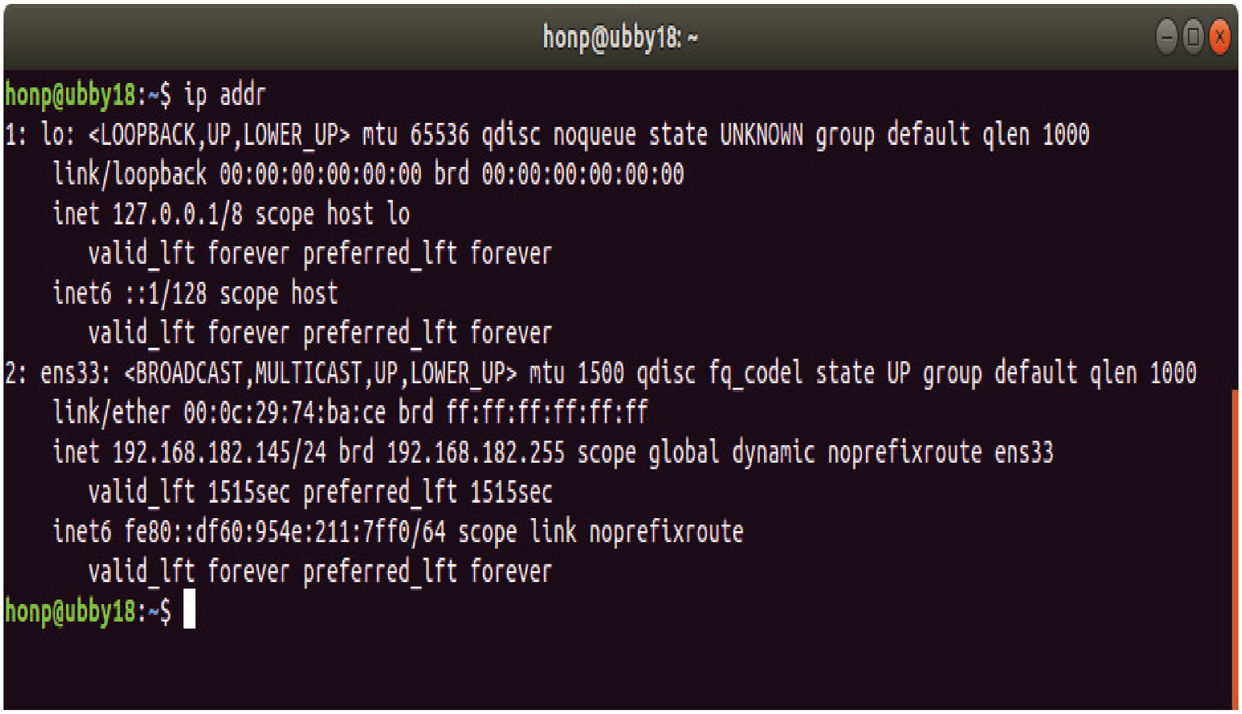
On Unix-based systems, we use either the `ifconfig` or `ip addr` commands. The `ifconfig` command is the old way and is now deprecated on some systems. The `ip addr` command is the new way, but not all systems support it yet.

Using the `ifconfig` command from a macOS terminal looks like this:



The IPv4 address is listed next to `inet`. In this case, we can see that it's `192.168.182.128`. Again, we see that this computer is on a local network because of the IP address range. The same adapter has an IPv6 address listed next to `inet6`.

The following screenshot shows using the `ip addr` command on Ubuntu Linux:



The preceding screenshot shows the local IPv4 address as `192.168.182.145`. We can also see that the link-local IPv6 address is `fe80::df60:954e:211:7ff0`.

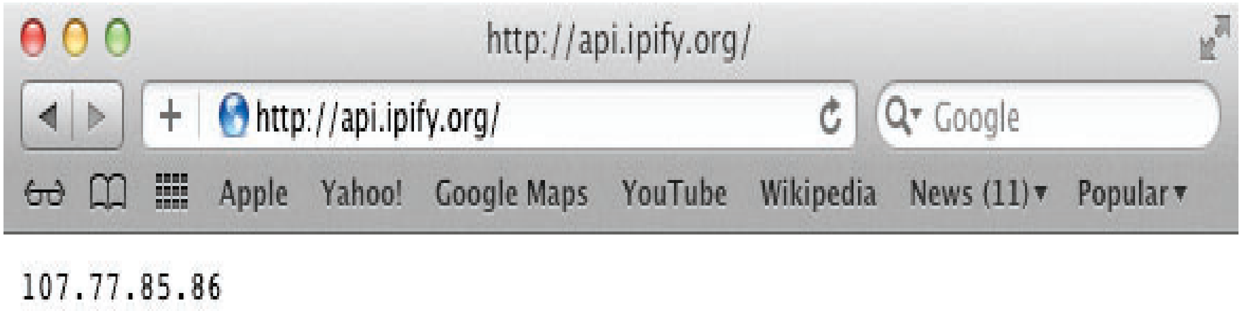
These commands, `ifconfig`, `ip addr`, and `ipconfig`, show the IP address or addresses for each adapter on your computer. You may have several. If you are on a local network, the IP addresses you see will be your local private network IP addresses.

If you are behind a NAT, there is often no good way to know your public IP address. Usually, the only resort is to contact an internet server that provides an API that informs you of your IP address.

A few free and public APIs for this are as follows:

- `http://api.ipify.org/`
- `http://helloacm.com/api/what-is-my-ip-address/`
- `http://icanhazip.com/`
- `http://ifconfig.me/ip`

You can test out these APIs in a web browser:



Each of these listed web pages should return your public IP address and not much else. These sites are useful for when you need to determine your public IP address from behind an NAT programmatically. We look at writing a small HTTP client capable of downloading these web pages and others in Chapter 6, *Building a Simple Web Client*.

Now that we've seen the built-in utilities for determining our local IP addresses, let's next look at how to accomplish this from C.

Listing network adapters from C

Sometimes, it is useful for your C programs to know what your local address is. For most of this book, we are able to write code that works both on Windows and Unix-based (Linux and macOS) systems. However, the API for listing local addresses is very different between systems. For this reason, we split this program into two: one for Windows and one for Unix-based systems.

We will address the Windows case first.

Listing network adapters on Windows

The Windows networking API is called **Winsock**, and we will go into much more detail about it in the next chapter.

Whenever we are using Winsock, the first thing we must do is initialize it. This is done with a call to `WSAStartup()`. Here is a small C program, `win_init.c`, showing the initialization and cleanup of Winsock:

```
/*win_init.c*/

#include <stdio.h>
#include <winsock2.h>
#pragma comment(lib, "ws2_32.lib")

int main() {
    WSADATA d;

    if (WSAStartup(MAKEWORD(2, 2), &d)) {
        printf("Failed to initialize.\n");
        return -1;
    }

    WSACleanup();
    printf("Ok.\n");
    return 0;
}
```

The `WSAStartup()` function is called with the requested version, Winsock 2.2 in this case, and a `WSADATA` structure. The `WSADATA` structure will be filled in by `WSAStartup()` with details about the Windows Sockets implementation. The `WSAStartup()` function returns `0` upon success, and non-zero upon failure.

When a Winsock program is finished, it should call `WSACleanup()`.

If you are using Microsoft Visual C as your compiler, then `#pragma comment(lib, "ws2_32.lib")` tells Microsoft Visual C to link the executable with the Winsock library, `ws2_32.lib`.

If you are using MinGW as your compiler, the pragma is ignored. You need to explicitly tell the compiler to link in the library by adding the command-line option, `-lws2_32`. For example, you can compile this program using MinGW with the following command:

```
gcc win_init.c -o win_init.exe -lws2_32
```

We will cover Winsock initialization and usage in more detail in Chapter 2, *Getting to Grips with Socket APIs*.

Now that we know how to initialize Winsock, we will begin work on the complete program to list network adapters on Windows. Please refer to the `win_list.c` file to follow along.

To begin with, we need to define `_WIN32_WINNT` and include the needed headers:

```
/*win_list.c*/

#ifndef _WIN32_WINNT
#define _WIN32_WINNT 0x0600
#endif

#include <winsock2.h>
#include <iphlpapi.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>
```

The `_WIN32_WINNT` macro must be defined first so that the proper version of the Windows headers are included. `winsock2.h`, `iphlpapi.h`, and `ws2tcpip.h` are the Windows headers we need in order to list network adapters. We need `stdio.h` for the `printf()` function and `stdlib.h` for memory allocation.

Next, we include the following pragmas to tell Microsoft Visual C which libraries must be linked with the executable:

```
/*win_list.c continued*/

#pragma comment(lib, "ws2_32.lib")
#pragma comment(lib, "iphlpapi.lib")
```

If you're compiling with MinGW, these lines will have no effect. You will need to link to these libraries explicitly on the command line, for example, `gcc win_list.c -o win_list.exe -liphlpapi -lws2_32`.

We then enter the `main()` function and initialize Winsock 2.2 using `WSAStartup()` as described earlier. We check its return value to detect any errors:

```
/*win_list.c continued*/

int main() {

    WSADATA d;
    if (WSAStartup(MAKEWORD(2, 2), &d)) {
        printf("Failed to initialize.\n");
        return -1;
    }
```

Next, we allocate memory for the adapters, and we request the adapters' addresses from Windows using the `GetAdapterAddresses()` function:

```
/*win_list.c continued*/

    DWORD asize = 20000;
    PIP_ADAPTER_ADDRESSES adapters;
    do {
        adapters = (PIP_ADAPTER_ADDRESSES)malloc(asize);

        if (!adapters) {
            printf("Couldn't allocate %ld bytes for adapters.\n", asize);
            WSACleanup();
            return -1;
        }

        int r = GetAdaptersAddresses(AF_UNSPEC, GAA_FLAG_INCLUDE_PREFIX, 0,
                adapters, &asize);
        if (r == ERROR_BUFFER_OVERFLOW) {
            printf("GetAdaptersAddresses wants %ld bytes.\n", asize);
            free(adapters);
        } else if (r == ERROR_SUCCESS) {
            break;
        } else {
            printf("Error from GetAdaptersAddresses: %d\n", r);
            free(adapters);
            WSACleanup();
            return -1;
        }
    } while (!adapters);
```

The `asize` variable will store the size of our adapters' address buffer. To begin with, we set it to `20000` and allocate 20,000 bytes to `adapters` using the `malloc()` function. The `malloc()` function will return `0` on failure, so we test for that and display an error message if allocation failed.

Next, we call `GetAdapterAddresses()`. The first parameter, `AF_UNSPEC`, tells Windows that we want both IPv4 and IPv6 addresses. You can pass in `AF_INET` or `AF_INET6` to request only IPv4 or only IPv6 addresses. The second parameter, `GAA_FLAG_INCLUDE_PREFIX`, is required to request a list of addresses. The next parameter is reserved and should be passed in as `0` or `NULL`. Finally, we pass in our buffer, `adapters`, and a pointer to its size, `asize`.

If our buffer is not big enough to store all of the addresses, then `GetAdapterAddresses()` returns `ERROR_BUFFER_OVERFLOW` and sets `asize` to the required buffer size. In this case, we free our `adapters` buffer and try the call again with a larger buffer.

On success, `GetAdapterAddresses()` returns `ERROR_SUCCESS`, in which case, we break from the loop and continue. Any other return value is an error.

When `GetAdapterAddresses()` returns successfully, it will have written a linked list into `adapters` with each adapter's address information. Our next step is to loop through this linked list and print information for each adapter and address:

```
/*win_list.c continued*/

    PIP_ADAPTER_ADDRESSES adapter = adapters;
    while (adapter) {
        printf("\nAdapter name: %S\n", adapter->FriendlyName);

        PIP_ADAPTER_UNICAST_ADDRESS address = adapter->FirstUnicastAddress;
        while (address) {
            printf("\t%s",
                    address->Address.lpSockaddr->sa_family == AF_INET ?
                    "IPv4" : "IPv6");

            char ap[100];

            getnameinfo(address->Address.lpSockaddr,
                    address->Address.iSockaddrLength,
                    ap, sizeof(ap), 0, 0, NI_NUMERICHOST);
            printf("\t%s\n", ap);

            address = address->Next;
        }

        adapter = adapter->Next;
    }
```

We first define a new variable, `adapter`, which we use to walk through the linked list of adapters. The first adapter is at the beginning of `adapters`, so we initially set `adapter` to `adapters`. At the end of each loop, we set `adapter = adapter->Next;` to get the next adapter. The loop aborts when `adapter` is `0`, which means we've reached the end of the list.

We get the adapter name from `adapter->FriendlyName`, which we then print using `printf()`.

The first address for each adapter is in `adapter->FirstUnicastAddress`. We define a second pointer, `address`, and set it to this address. Addresses are also stored as a linked list, so we begin an inner loop that walks through the addresses.

The `address->Address.lpSockaddr->sa_family` variable stores the address family type. If it is set to `AF_INET`, then we know this is an IPv4 address. Otherwise, we assume it is an IPv6 address (in which case the family is `AF_INET6`).

Next, we allocate a buffer, `ap`, to store the text representation of the address. The `getnameinfo()` function is called to convert the address into a standard notation address. We'll cover more about `getnameinfo()` in the next chapter.

Finally, we can print the address from our buffer, `ap`, using `printf()`.

We finish the program by freeing the allocated memory and calling `WSACleanup()`:

```
/*win_list.c continued*/

    free(adapters);
    WSACleanup();
    return 0;
}
```

On Windows, using MinGW, you can compile and run the program with the following:

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

It should list each of your adapter's names and addresses.

Now that we can list local IP addresses on Windows, let's consider the same task for Unix-based systems.

Listing network adapters on Linux and macOS

Listing local network addresses is somewhat easier on a Unix-based system, compared to Windows. Load up `unix_list.c` to follow along.

To begin with, we include the necessary system headers:

```
/*unix_list.c*/

#include <sys/socket.h>
#include <netdb.h>
#include <ifaddrs.h>
#include <stdio.h>
#include <stdlib.h>
```

We then enter the `main` function:

```
/*unix_list.c continued*/

int main() {

    struct ifaddrs *addresses;

    if (getifaddrs(&addresses) == -1) {
        printf("getifaddrs call failed\n");
        return -1;
    }
```

We declare a variable, `addresses`, which stores the addresses. A call to the `getifaddrs()` function allocates memory and fills in a linked list of addresses. This function returns `0` on success or `-1` on failure.

Next, we use a new pointer, `address`, to walk through the linked list of addresses. After considering each address, we set `address = address->ifa_next` to get the next address. We stop the loop when `address == 0`, which happens at the end of the linked list:

```
/*unix_list.c continued*/

    struct ifaddrs *address = addresses;
```

```
    while(address) {
        int family = address->ifa_addr->sa_family;
        if (family == AF_INET || family == AF_INET6) {

            printf("%s\t", address->ifa_name);
            printf("%s\t", family == AF_INET ? "IPv4" : "IPv6");

            char ap[100];
            const int family_size = family == AF_INET ?
                sizeof(struct sockaddr_in) : sizeof(struct sockaddr_in6);
            getnameinfo(address->ifa_addr,
                    family_size, ap, sizeof(ap), 0, 0, NI_NUMERICHOST);
            printf("\t%s\n", ap);

        }
        address = address->ifa_next;
    }
```

For each address, we identify the address family. We are interested in `AF_INET` (IPv4 addresses) and `AF_INET6` (IPv6 addresses). The `getifaddrs()` function can return other types, so we skip those.

For each address, we then continue to print its adapter name and its address type, IPv4 or IPv6.

We then define a buffer, `ap`, to store the textual address. A call to the `getnameinfo()` function fills in this buffer, which we can then print. We cover the `getnameinfo()` function in more detail in the next chapter, Chapter 2, *Getting to Grips with Socket APIs*.

Finally, we free the memory allocated by `getifaddrs()` and we have finished:

```
/*unix_list.c continued*/

    freeifaddrs(addresses);
    return 0;
}
```

On Linux and macOS, you can compile and run this program with the following:

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

It should list each of your adapter's names and addresses.

Summary

In this chapter, we looked briefly at how internet traffic is routed. We learned that there are two Internet Protocol versions, IPv4 and IPv6. IPv4 has a limited number of addresses, and these addresses are running out. One of IPv6's main advantages is that it has enough address space for every system to have its own unique publicly-routable address. The limited address space of IPv4 is largely mitigated by network address translation performed by routers. We also looked at how to detect your local IP address using both utilities and APIs provided by the operating system.

We saw that the APIs provided for listing local IP addresses differ quite a bit between Windows and Unix-based operating systems. In future chapters, we will see that most other networking functions are similar between operating systems, and we can write one portable program that works between operating systems.

It's OK if you didn't pick up all of the details in this chapter. Most of this information is a helpful background, but it's not essential to most network application programming. Details such as network address translation are handled by the network, and these details will not usually need to be explicitly addressed by your programs.

In the next chapter, we will reinforce the ideas covered here by introducing socket-programming APIs.

Questions

Try these questions to test your knowledge from this chapter:

1. What are the key differences between IPv4 and IPv6?
2. Are the IP addresses given by the `ipconfig` and `ifconfig` commands the same IP addresses that a remote web server sees if you connect to it?
3. What is the IPv4 loopback address?
4. What is the IPv6 loopback address?
5. How are domain names (for example, `example.com`) resolved into IP addresses?
6. How can you find your public IP address?
7. How does an operating system know which application is responsible for an incoming packet?

The answers are in `Appendix A`, *Answers to Questions*.