

# 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.

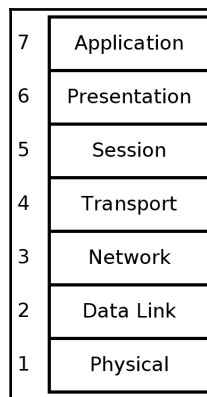
## OSI layer model

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.

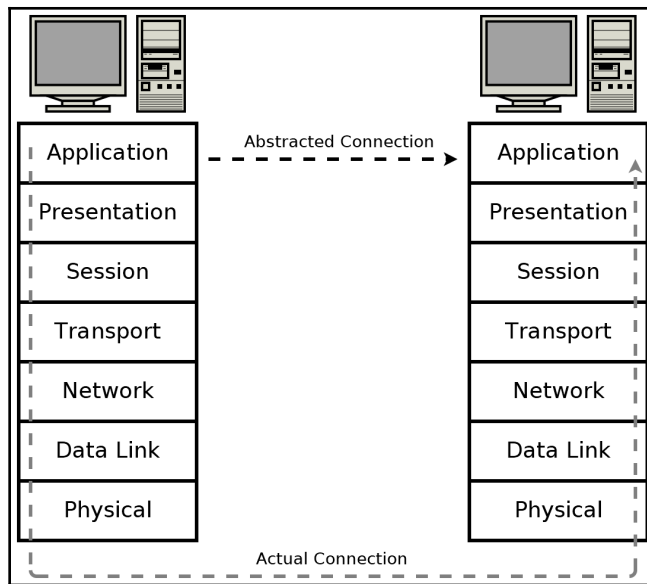
The most popular layer system for networking is called the **Open Systems Interconnection** model (**OSI** model). It was standardized in 1977 and is published as ISO 7498. It has seven layers:



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.

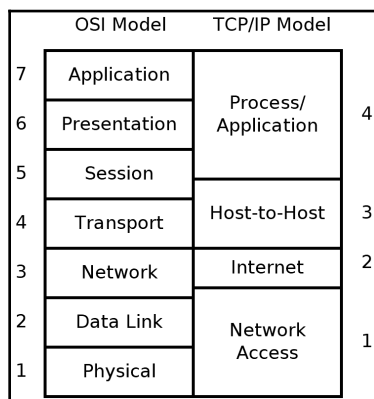
In this book, we often use the term packet as a generic term to refer to a data unit on any layer. However, segment will only be used in the context of a TCP connection, and datagram will only refer to UDP datagrams.

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:

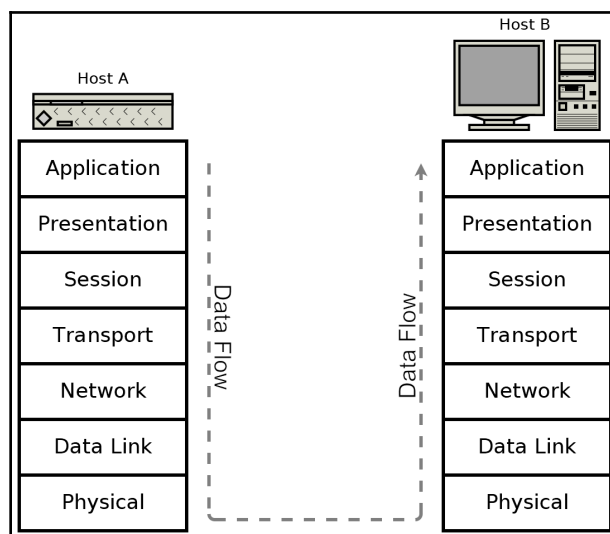
- **Network Access layer (1):** On this layer, physical connections and data framing happen. Sending an Ethernet or Wi-Fi packet are examples of layer 1 concerns.
- **Internet layer (2):** This layer deals with the concerns of addressing packets and routing them over multiple interconnection networks.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

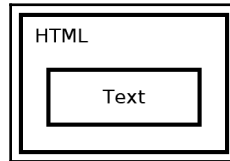
## Data encapsulation

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).

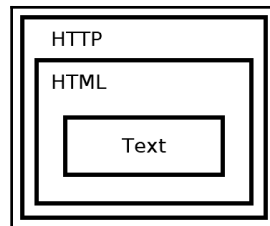
When communicating over a network, data must be processed down through the layers at the sender and up again through the layers at the receiver. For example, if we have a web server, **Host A**, which is transmitting a web page to the receiver, **Host B**, it may look like this:



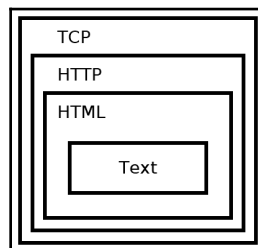
The web page contains a few paragraphs of text, but the web server doesn't only send the text by itself. For the text to be rendered correctly, it must be encoded in an **HTML** structure:



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**:

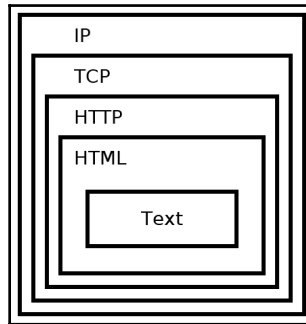


The **HTTP** is transmitted as part of a **TCP** session. This isn't done explicitly by the web server, but is taken care of by the operating system's TCP/IP stack:

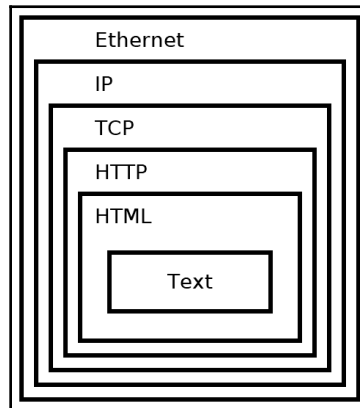




The **TCP** packet is routed by an **IP** packet:



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  $3.4 \times 10^{38}$  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
<code>::ffff:10.0.0.0</code>	<code>10.0.0.0</code>
<code>::ffff:172.16.0.5</code>	<code>172.16.0.5</code>
<code>::ffff:192.168.0.1</code>	<code>192.168.0.1</code>
<code>::ffff:192.168.50.1</code>	<code>192.168.50.1</code>

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 access 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.

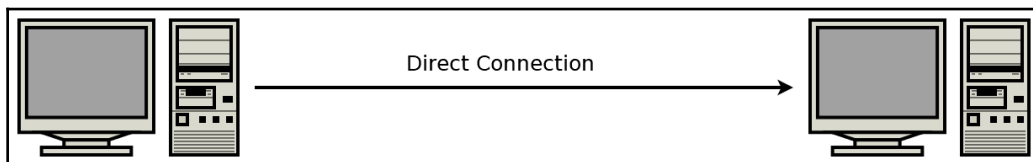
It is also possible, and common, for multiple records of the same type to exist, each pointing to a different address. This is useful for redundancy in the case where multiple servers can provide the same service.

We will see a lot more about DNS queries in [Chapter 5, Hostname Resolution and DNS](#).

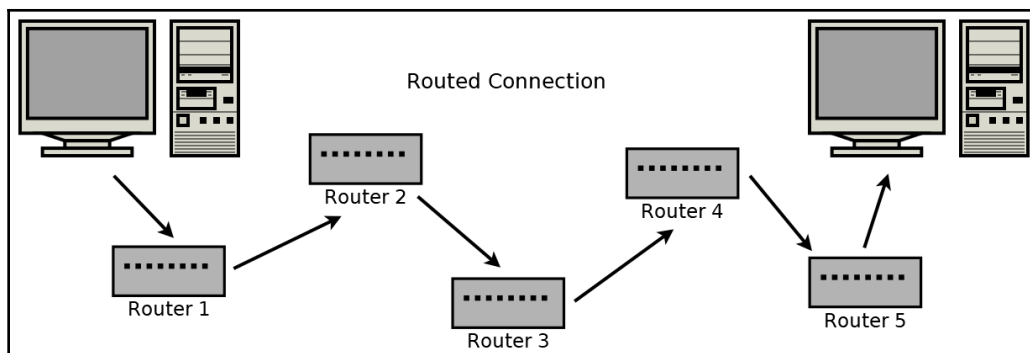
Now that we have a basic understanding of IP addresses and names, let's look into detail of how IP packets are routed over the internet.

## 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`:

```

Windows PowerShell
PS C:\> tracert example.com

Tracing route to example.com [93.184.216.34]
over a maximum of 30 hops:

  1  <1 ms  <1 ms  <1 ms  192.168.50.1
  2  *      *      *      Request timed out.
  3  *      *      *      Request timed out.
  4  2 ms   2 ms   1 ms   my.jetpack [192.168.1.1]
  5  119 ms 47 ms  41 ms  172.26.96.169
  6  66 ms  39 ms  38 ms  107.79.227.124
  7  *      *      *      Request timed out.
  8  58 ms  79 ms  70 ms  12.83.186.145
  9  61 ms  40 ms  41 ms  cgcil403igs.ip.att.net [12.122.133.33]
 10  78 ms  38 ms  39 ms  dcr1-so-4-0-0.atlanta.savvis.net [192.205.32.118]
 11  116 ms 198 ms 47 ms  192.229.225.133
 12  76 ms  40 ms  37 ms  93.184.216.34

Trace complete.
PS C:\>

```

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.

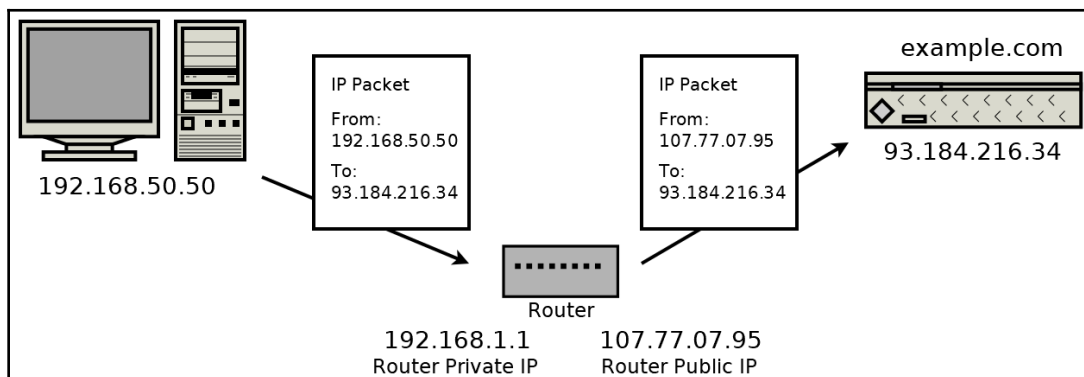
These reserved private ranges are as follows:

- `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`

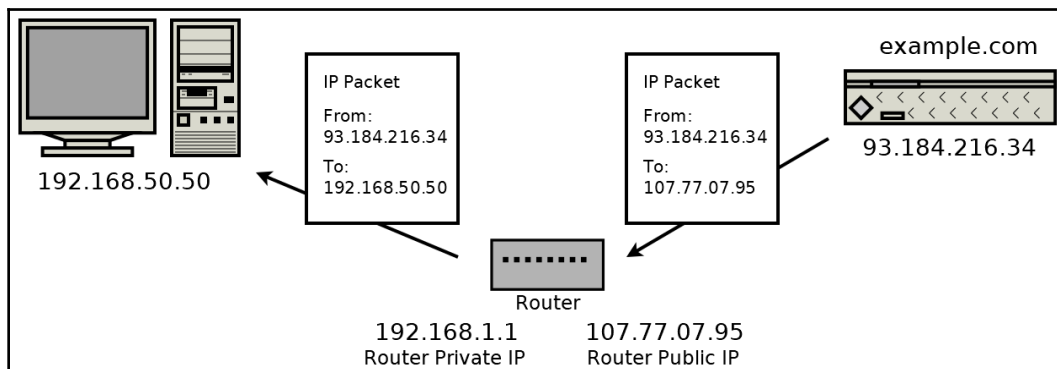
When a packet originates from a device on an IPv4 local network, it must undergo **Network Address Translation (NAT)** before being routed on the internet. A router that implements NAT remembers which local address a connection is established from.



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 (DNS)	Chapter 5, Hostname 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)

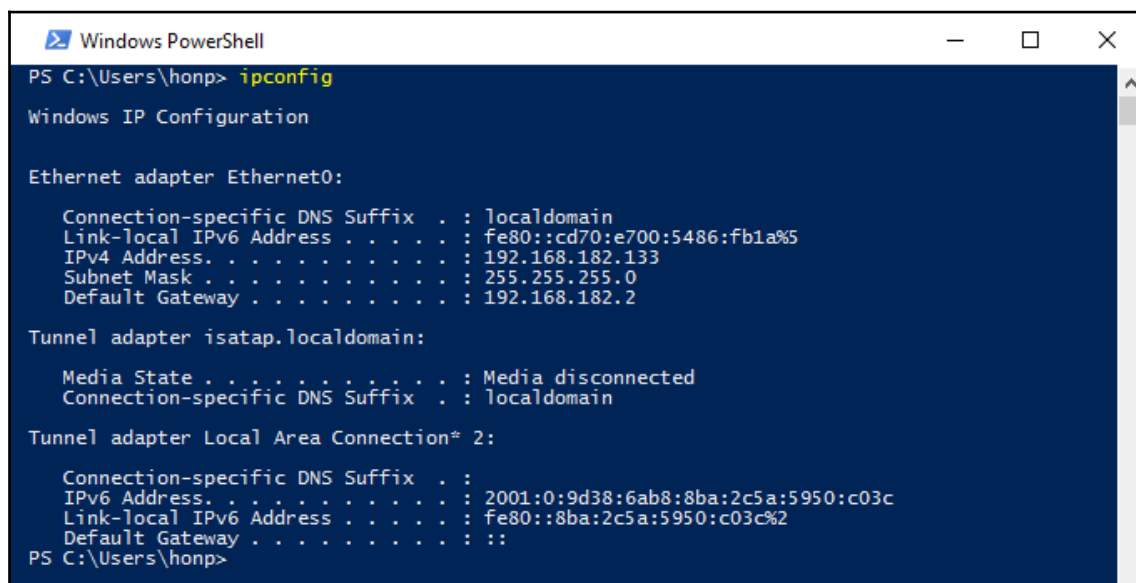
This 5-tuple is important, as it is how your operating system knows which application is responsible for any packets received. For example, if you use two web browsers to establish two simultaneous connections to `example.com` on port 80, then your operating system keeps the connections separate by looking 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:

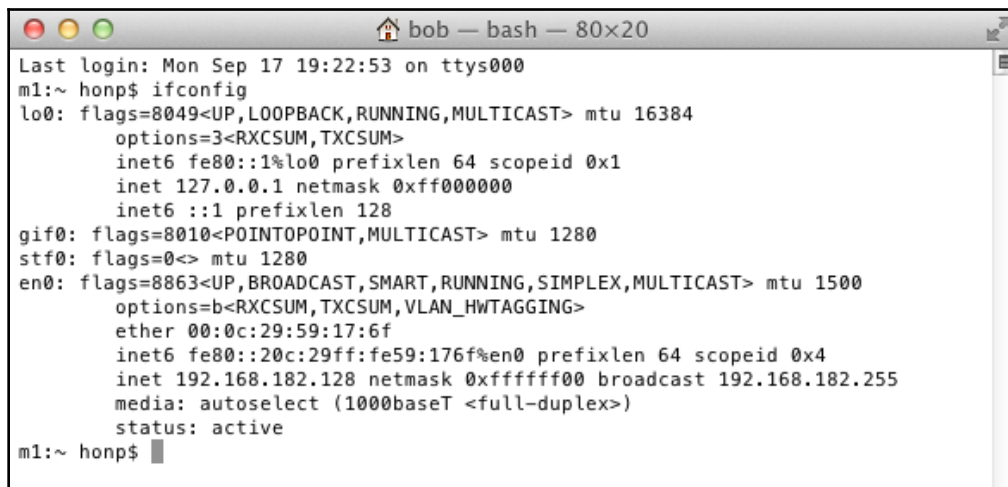
    Connection-specific DNS Suffix  . : 
    IPv6 Address. . . . . : 2001:0:9d38:6ab8:8ba:2c5a:5950:c03c
    Link-local IPv6 Address . . . . . : fe80::8ba:2c5a:5950:c03c%2
    Default Gateway . . . . . : ::

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.

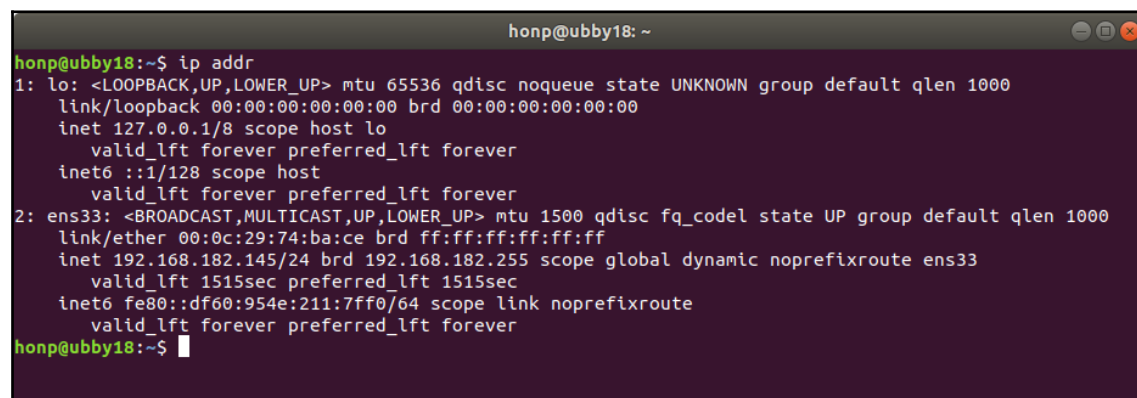
Using the `ifconfig` command from a macOS terminal looks like this:



```
bob — bash — 80x20
Last login: Mon Sep 17 19:22:53 on ttys000
ml:~ honp$ ifconfig
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 16384
    options=3<RXCSUM,TXCSUM>
    inet6 fe80::1%lo0 prefixlen 64 scopeid 0x1
    inet 127.0.0.1 netmask 0xff000000
    inet6 ::1 prefixlen 128
gif0: flags=8010<POINTOPOINT,MULTICAST> mtu 1280
stf0: flags=0<> mtu 1280
en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
    options=b<RXCSUM,TXCSUM,VLAN_HWTAGGING>
    ether 00:0c:29:59:17:6f
    inet6 fe80::20c:29ff:fe59:176f%en0 prefixlen 64 scopeid 0x4
    inet 192.168.182.128 netmask 0xffffffff00 broadcast 192.168.182.255
    media: autoselect (1000baseT <full-duplex>)
    status: active
ml:~ honp$
```

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:



```
honp@ubby18: ~
honp@ubby18:~$ ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 00:0c:29:74:ba:ce brd ff:ff:ff:ff:ff:ff
    inet 192.168.182.145/24 brd 192.168.182.255 scope global dynamic noprefixroute ens33
        valid_lft 1515sec preferred_lft 1515sec
    inet6 fe80::df60:954e:211:7ff0/64 scope link noprefixroute
        valid_lft forever preferred_lft forever
honp@ubby18:~$
```

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.

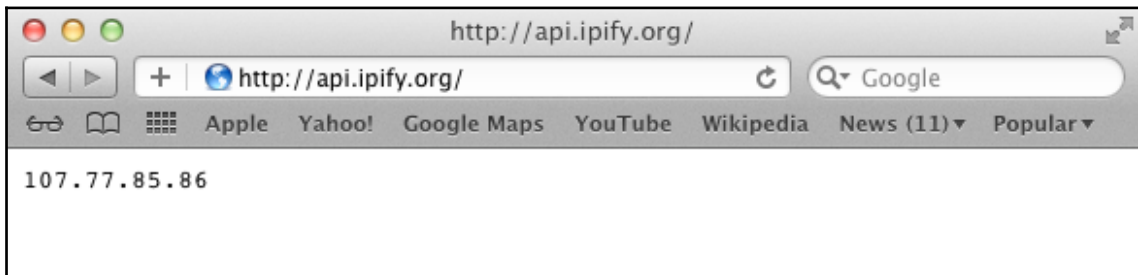
These commands, `ifconfig`, `ip addr`, and `ipconfig`, show the IP address or addresses for each adapter on your computer. You may have several. If you are on a local network, the IP addresses you see will be your local private network IP addresses.

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*/

#ifdef _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*.



# 2

## Getting to Grips with Socket APIs

In this chapter, we will begin to really start working with network programming. We will introduce the concept of sockets, and explain a bit of the history behind them. We will cover the important differences between the socket APIs provided by Windows and Unix-like operating systems, and we will review the common functions that are used in socket programming. This chapter ends with a concrete example of turning a simple console program into a networked program you can access through your web browser.

The following topics are covered in this chapter:

- What are sockets?
- Which header files are used with socket programming?
- How to compile a socket program on Windows, Linux, and macOS
- Connection-oriented and connectionless sockets
- TCP and UDP protocols
- Common socket functions
- Building a simple console program into a web server

## Technical requirements

The example programs in this chapter can be compiled with any modern 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 here: <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/chap02
```

Each example program in this chapter is standalone, and each example runs on Windows, Linux, and macOS. When compiling for Windows, keep in mind that most of the example programs require linking with the Winsock library.

This is accomplished by passing the `-lws2_32` option to `gcc`. We provide the exact commands needed to compile each example as they are introduced.

## What are sockets?

A socket is one endpoint of a communication link between systems. Your application sends and receives all of its network data through a socket.

There are a few different socket **application programming interfaces (APIs)**. The first were Berkeley sockets, which were released in 1983 with 4.3BSD Unix. The Berkeley socket API was widely successful and quickly evolved into a de facto standard. From there, it was adopted as a POSIX standard with little modification. The terms Berkeley sockets, BSD sockets, Unix sockets, and **Portable Operating System Interface (POSIX)** sockets are often used interchangeably.

If you're using Linux or macOS, then your operating system provides a proper implementation of Berkeley sockets.

Windows' socket API is called **Winsock**. It was created to be largely compatible with Berkeley sockets. In this book, we strive to create cross-platform code that is valid for both Berkeley sockets and Winsock.

Historically, sockets were used for **inter-process communication (IPC)** as well as various network protocols. In this book, we use sockets only for communication with TCP and UDP.

Before we can start using sockets, we need to do a bit of setup. Let's dive right in!

## Socket setup

Before we can use the socket API, we need to include the socket API header files. These files vary depending on whether we are using Berkeley sockets or Winsock. Additionally, Winsock requires initialization before use. It also requires that a cleanup function is called when we are finished. These initialization and cleanup steps are not used with Berkeley sockets.

We will use the C preprocessor to run the proper code on Windows compared to Berkeley socket systems. By using the preprocessor statement, `#if defined(_WIN32)`, we can include code in our program that will only be compiled on Windows.

Here is a complete program that includes the needed socket API headers for each platform and properly initializes Winsock on Windows:

```
/*sock_init.c*/

#if defined(_WIN32)
#ifndef _WIN32_WINNT
#define _WIN32_WINNT 0x0600
#endif
#include <winsock2.h>
#include <ws2tcpip.h>
#pragma comment(lib, "ws2_32.lib")

#else
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
#include <errno.h>

#endif

#include <stdio.h>

int main() {

    #if defined(_WIN32)
        WSADATA d;
        if (WSAStartup(MAKEWORD(2, 2), &d)) {
            fprintf(stderr, "Failed to initialize.\n");
            return 1;
        }
    #endif
}
```

```
#endif

    printf("Ready to use socket API.\n");

    #if defined(_WIN32)
        WSACleanup();
    #endif

    return 0;
}
```

The first part includes `winsock.h` and `ws2tcpip.h` on Windows. `_WIN32_WINNT` must be defined for the Winsock headers to provide all the functions we need. We also include the `#pragma comment(lib, "ws2_32.lib")` pragma statement. This tells the Microsoft Visual C compiler to link your program against the Winsock library, `ws2_32.lib`. If you're using MinGW as your compiler, then `#pragma` is ignored. In this case, you need to tell the compiler to link in `ws2_32.lib` on the command line using the `-lws2_32` option.

If the program is not compiled on Windows, then the section after `#else` will compile. This section includes the various Berkeley socket API headers and other headers we need on these platforms.

In the `main()` function, we call `WSAStartup()` on Windows to initialize Winsock. The `MAKEWORD` macro allows us to request Winsock version 2.2. If our program is unable to initialize Winsock, it prints an error message and aborts.

When using Berkeley sockets, no special initialization is needed, and the socket API is always ready to use.

Before our program finishes, `WSACleanup()` is called if we're compiling for Winsock on Windows. This function allows the Windows operating system to do additional cleanup.

Compiling and running this program on Linux or macOS is done with the following command:

```
gcc sock_init.c -o sock_init
./sock_init
```

Compiling on Windows using MinGW can be done with the following command:

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

Notice that the `-lws2_32` flag is needed with MinGW to tell the compiler to link in the Winsock library, `ws2_32.lib`.

Now that we've done the necessary setup to begin using the socket APIs, let's take a closer look at what we will be using these sockets for.

## Two types of sockets

Sockets come in two basic types—**connection-oriented** and **connectionless**. These terms refer to types of protocols. Beginners sometimes get confused with the term **connectionless**. Of course, two systems communicating over a network are in some sense connected. Keep in mind that these terms are used with special meanings, which we will cover shortly, and should not imply that some protocols manage to send data without a connection.

The two protocols that are used today are **Transmission Control Protocol (TCP)** and **User Datagram Protocol (UDP)**. TCP is a connection-oriented protocol, and UDP is a connectionless protocol.

The socket APIs also support other less-common or outdated protocols, which we do not cover in this book.

In a connectionless protocol, such as UDP, each data packet is addressed individually. From the protocol's perspective, each data packet is completely independent and unrelated to any packets coming before or after it.

A good analogy for UDP is **postcards**. When you send a postcard, there is no guarantee that it will arrive. There is also no way to know if it did arrive. If you send many postcards at once, there is no way to predict what order they will arrive in. It is entirely possible that the first postcard you send gets delayed and arrives weeks after the last postcard was sent.

With UDP, these same caveats apply. UDP makes no guarantee that a packet will arrive. UDP doesn't generally provide a method to know if a packet did not arrive, and UDP does not guarantee that the packets will arrive in the same order they were sent. As you can see, UDP is no more reliable than postcards. In fact, you may consider it less reliable, because with UDP, it is possible that a single packet may arrive twice!

If you need reliable communication, you may be tempted to develop a scheme where you number each packet that's sent. For the first packet sent, you number it one, the second packet sent is numbered two, and so on. You could also request that the receiver send an acknowledgment for each packet. When the receiver gets packet one, it sends a return message, **packet one received**. In this way, the receiver can be sure that received packets are in the proper order. If the same packet arrives twice, the receiver can just ignore the redundant copy. If a packet isn't received at all, the sender knows from the missing acknowledgment and can resend it.

This scheme is essentially what connection-oriented protocols, such as TCP, do. TCP guarantees that data arrives in the same order it is sent. It prevents duplicate data from arriving twice, and it retries sending missing data. It also provides additional features such as notifications when a connection is terminated and algorithms to mitigate network congestion. Furthermore, TCP implements these features with an efficiency that is not achievable by piggybacking a custom reliability scheme on top of UDP.

For these reasons, TCP is used by many protocols. HTTP (for severing web pages), FTP (for transferring files), SSH (for remote administration), and SMTP (for delivering email) all use TCP. We will cover HTTP, SSH, and SMTP in the coming chapters.

UDP is used by DNS (for resolving domain names). It is suitable for this purpose because an entire request and response can fit in a single packet.

UDP is also commonly used in real-time applications, such as audio streaming, video streaming, and multiplayer video games. In real-time applications, there is often no reason to retry sending dropped packets, so TCP's guarantees are unnecessary. For example, if you are streaming live video and a few packets get dropped, the video simply resumes when the next packet arrives. There is no reason to resend (or even detect) the dropped packet, as the video has already progressed past that point.

UDP also has the advantage in cases where you want to send a message without expecting a response from the other end. This makes it useful when using IP broadcast or multicast. TCP, on the other hand, requires bidirectional communication to provide its guarantees, and TCP does not work with IP multicast or broadcast.

If the guarantees that TCP provides are not needed, then UDP can achieve greater efficiency. This is because TCP adds some additional overhead by numbering packets. TCP must also delay packets that arrive out of order, which can cause unnecessary delays in real-time applications. If you do need the guarantees provided by TCP, however, it is almost always preferable to use TCP instead of trying to add those mechanisms to UDP.

Now that we have an idea of the communication models we use sockets for, let's look at the actual functions that are used in socket programming.

## Socket functions

The socket APIs provide many functions for use in network programming. Here are the common socket functions that we use in this book:

- `socket()` creates and initializes a new socket.
- `bind()` associates a socket with a particular local IP address and port number.
- `listen()` is used on the `server` to cause a TCP socket to listen for new connections.
- `connect()` is used on the `client` to set the remote address and port. In the case of TCP, it also establishes a connection.
- `accept()` is used on the `server` to create a new socket for an incoming TCP connection.
- `send()` and `recv()` are used to send and receive data with a socket.
- `sendto()` and `recvfrom()` are used to send and receive data from sockets without a bound remote address.
- `close()` (**Berkeley sockets**) and `closesocket()` (Winsock sockets) are used to close a socket. In the case of TCP, this also terminates the connection.
- `shutdown()` is used to close one side of a TCP connection. It is useful to ensure an orderly connection teardown.
- `select()` is used to wait for an event on one or more sockets.
- `getnameinfo()` and `getaddrinfo()` provide a protocol-independent manner of working with hostnames and addresses.
- `setsockopt()` is used to change some socket options.
- `fcntl()` (**Berkeley sockets**) and `ioctlsocket()` (Winsock sockets) are also used to get and set some socket options.

You may see `some Berkeley socket networking programs using read() and write()`. These functions don't port to Winsock, so we prefer `send()` and `recv()` here. `Some other common functions that are used with Berkeley sockets are poll() and dup()`. We will avoid these in order to keep our programs portable.

Other differences between Berkeley sockets and Winsock sockets are addressed later in this chapter.

Now that we have an idea of the functions involved, let's consider program design and flow next.

## Anatomy of a socket program

As we mentioned in Chapter 1, *An Introduction to Networks and Protocols*, network programming is usually done using a client-server paradigm. In this paradigm, a server listens for new connections at a published address. The client, knowing the server's address, is the one to establish the connection initially. Once the connection is established, the client and the server can both send and receive data. This can continue until either the client or the server terminates the connection.

A traditional client-server model usually implies different behaviors for the client and server. The way web browsing works, for example, is that the server resides at a known address, waiting for connections. A client (web browser) establishes a connection and sends a request that includes which web page or resource it wants to download. The server then checks that it knows what to do with this request and responds appropriately (by sending the web page).

An alternative paradigm is the peer-to-peer model. For example, this model is used by the BitTorrent protocol. In the peer-to-peer model, each peer has essentially the same responsibilities. While a web server is optimized to send requested data from the server to the client, a peer-to-peer protocol is balanced in that data is exchanged somewhat evenly between peers. However, even in the peer-to-peer model, the underlying sockets that are using TCP or UDP aren't created equal. That is, for each peer-to-peer connection, one peer was listening and the other connecting. BitTorrent works by having a central server (called a **tracker**) that stores a list of peer IP addresses. Each of the peers on that list has agreed to behave like a server and listen for new connections. When a new peer wants to join the swarm, it requests a list of peers from the central server, and then tries to establish a connection to peers on that list while simultaneously listening for new connections from other peers. In summary, a peer-to-peer protocol doesn't so much replace the client-server model; it is just expected that each peer be a client and a server both.

Another common protocol that pushes the boundary of the client-server paradigm is FTP. The FTP server listens for connections until the FTP client connects. After the initial connection, the FTP client issues commands to the server. If the FTP client requests a file from the server, the server will attempt to establish a new connection to the FTP client to transfer the file over. So, for this reason, the FTP client first establishes a connection as a TCP client, but later accepts connections like a TCP server.



Network programs can usually be described as one of four types—a TCP server, a TCP client, a UDP server, or a UDP client. Some protocols call for a program to implement two, or even all four types, but it is useful for us to consider each of the four types separately.

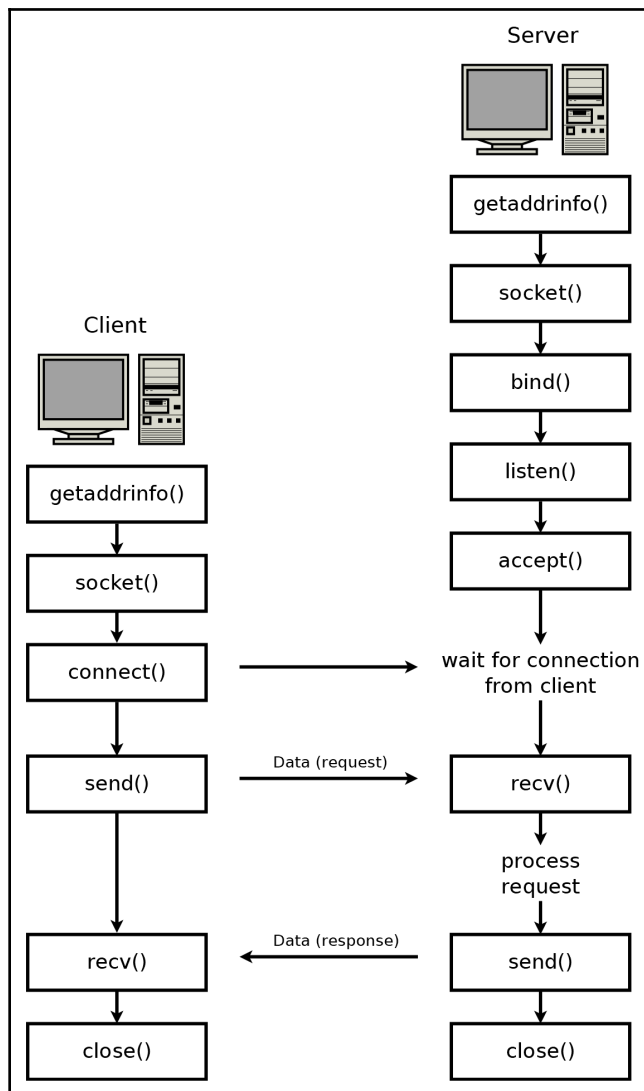
## TCP program flow

A TCP client program must first know the TCP server's address. This is often input by a user. In the case of a web browser, the server address is either input directly by the user into the address bar, or is known from the user clicking on a link. The TCP client takes this address (for example, `http://example.com`) and uses the `getaddrinfo()` function to resolve it into a `struct addrinfo` structure. The client then creates a socket using a call to `socket()`. The client then establishes the new TCP connection by calling `connect()`. At this point, the client can freely exchange data using `send()` and `recv()`.

A TCP server listens for connections at a particular port number on a particular interface. The program must first initialize a `struct addrinfo` structure with the proper listening IP address and port number. The `getaddrinfo()` function is helpful so that you can do this in an IPv4/IPv6 independent way. The server then creates the socket with a call to `socket()`. The socket must be bound to the listening IP address and port. This is accomplished with a call to `bind()`.

The server program then calls `listen()`, which puts the socket in a state where it listens for new connections. The server can then call `accept()`, which will wait until a client establishes a connection to the server. When the new connection has been established, `accept()` returns a new socket. This new socket can be used to exchange data with the client using `send()` and `recv()`. Meanwhile, the first socket remains listening for new connections, and repeated calls to `accept()` allow the server to handle multiple clients.

Graphically, the program flow of a TCP client and server looks like this:



The program flow given here should serve as a good example of how basic client-server TCP programs interact. That said, considerable variation on this basic program flow is possible. There is also no rule about which side calls `send()` or `recv()` first, or how many times. Both sides could call `send()` as soon as the connection is established.

Also, note that the TCP client could call `bind()` before `connect()` if it is particular about which network interface is being used to connect with. This is sometimes important on servers that have multiple network interfaces. It's often not important for general purpose software.

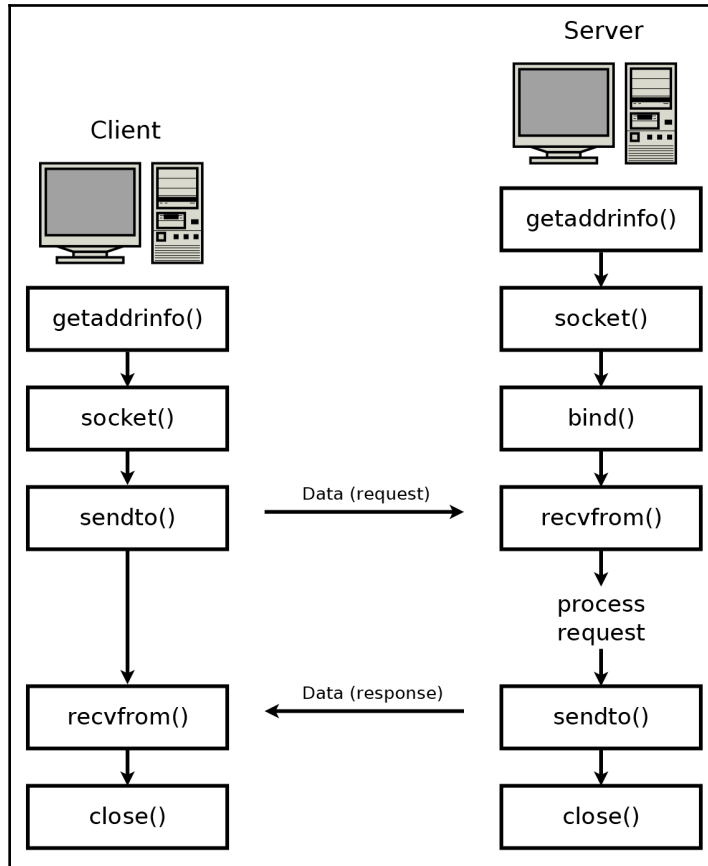
Many other variations of TCP operation are possible too, and we will look at some in Chapter 3, *An In-Depth Overview of TCP Connections*.

## UDP program flow

A UDP client must know the address of the remote UDP peer in order to send the first packet. The UDP client uses the `getaddrinfo()` function to resolve the address into a `struct addrinfo` structure. Once this is done, the client creates a socket of the proper type. The client can then call `sendto()` on the socket to send the first packet. The client can continue to call `sendto()` and `recvfrom()` on the socket to send and receive additional packets. Note that the client must send the first packet with `sendto()`. The UDP client cannot receive data first, as the remote peer would have no way of knowing where to send data without it first receiving data from the client. This is different from TCP, where a connection is first established with a handshake. In TCP, either the client or server can send the first application data.

A UDP server listens for connections from a UDP client. This server should initialize `struct addrinfo` structure with the proper listening IP address and port number. The `getaddrinfo()` function can be used to do this in a protocol-independent way. The server then creates a new socket with `socket()` and binds it to the listening IP address and port number using `bind()`. At this point, the server can call `recvfrom()`, which causes it to block until it receives data from a UDP client. After the first data is received, the server can reply with `sendto()` or listen for more data (from the first client or any new client) with `recvfrom()`.

Graphically, the program flow of a UDP client and server looks like this:



We cover some variations of this example program flow in Chapter 4, *Establishing UDP Connections*.

We're almost ready to begin implementing our first networked program, but before we begin, we should take care of some cross-platform concerns. Let's work on this now.

## Berkeley sockets versus Winsock sockets

As we stated earlier, Winsock sockets were modeled on Berkeley sockets. Therefore, there are many similarities between them. However, there are also many differences we need to be aware of.

In this book, we will try to create each program so that it can run on both Windows and Unix-based operating systems. This is made much easier by defining a few C macros to help us with this.

## Header files

As we mentioned earlier, the needed header files differ between implementations. We've already seen how these header file discrepancies can be easily overcome with a preprocessor statement.

## Socket data type

In UNIX, a socket descriptor is represented by a standard file descriptor. This means you can use any of the standard UNIX file I/O functions on sockets. This isn't true on Windows, so we simply avoid these functions to maintain portability.

Additionally, in UNIX, all file descriptors (and therefore socket descriptors) are small, non-negative integers. In Windows, a socket handle can be anything. Furthermore, in UNIX, the `socket()` function returns an `int`, whereas in Windows it returns a `SOCKET`. `SOCKET` is a typedef for an unsigned `int` in the Winsock headers. As a workaround, I find it useful to either typedef `int SOCKET` or `#define SOCKET int` on non-Windows platforms. That way, you can store a socket descriptor as a `SOCKET` type on all platforms:

```
#if !defined(_WIN32)
#define SOCKET int
#endif
```

## Invalid sockets

On Windows, `socket()` returns `INVALID_SOCKET` if it fails. On Unix, `socket()` returns a negative number on failure. This is particularly problematic as the Windows `SOCKET` type is unsigned. I find it useful to define a macro to indicate if a socket descriptor is valid or not:

```
#if defined(_WIN32)
#define ISVALIDSOCKET(s) ((s) != INVALID_SOCKET)
#else
#define ISVALIDSOCKET(s) ((s) >= 0)
#endif
```

## Closing sockets

All sockets on Unix systems are also standard file descriptors. For this reason, sockets on Unix systems can be closed using the standard `close()` function. On Windows, a special close function is used instead—`closesocket()`. It's useful to abstract out this difference with a macro:

```
#if defined(_WIN32)
#define CLOSESOCKET(s) closesocket(s)
#else
#define CLOSESOCKET(s) close(s)
#endif
```

## Error handling

When a socket function, such as `socket()`, `bind()`, `accept()`, and so on, has an error on a Unix platform, the error number gets stored in the thread-global `errno` variable. On Windows, the error number can be retrieved by calling `WSAGetLastError()` instead. Again, we can abstract out this difference using a macro:

```
#if defined(_WIN32)
#define GETSOCKETERRNO() (WSAGetLastError())
#else
#define GETSOCKETERRNO() (errno)
#endif
```

In addition to obtaining an error code, it is often useful to retrieve a text description of the error condition. Please refer to Chapter 13, *Socket Programming Tips and Pitfalls*, for a technique for this.

With these helper macros out of the way, let's dive into our first real socket program.

## Our first program

Now that we have a basic idea of socket APIs and the structure of networked programs, we are ready to begin our first program. By building an actual real-world program, we will learn the useful details of how socket programming actually works.

As an example task, we are going to build a web server that tells you what time it is right now. This could be a useful resource for anybody with a smartphone or web browser that needs to know what time it is right now. They can simply navigate to our web page and find out. This is a good first example because it does something useful but still trivial enough that it won't distract from what we are trying to learn—network programming.

## A motivating example

Before we begin the networked program, it is useful to solve our problem with a simple console program first. In general, it is a good idea to work out your program's functionality locally before adding in networked features.

The local, console version of our time-telling program is as follows:

```
/*time_console.c*/

#include <stdio.h>
#include <time.h>

int main()
{
    time_t timer;
    time(&timer);

    printf ("Local time is: %s", ctime(&timer));

    return 0;
}
```

You can compile and run it like this:

```
$ gcc time_console.c -o time_console
$ ./time_console
Local time is: Fri Oct 19 08:42:05 2018
```

The program works by getting the time with the built-in C `time()` function. It then converts it into a string with the `ctime()` function.

## Making it networked

Now that we've worked out our program's functionality, we can begin on the networked version of the same program.

To begin with, we **include the needed headers**:

```
/*time_server.c*/

#ifdef _WIN32
#ifndef _WIN32_WINNT
#define _WIN32_WINNT 0x0600
#endif
#include <winsock2.h>
#include <ws2tcpip.h>
#pragma comment(lib, "ws2_32.lib")

#else
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
#include <errno.h>

#endif
```

As we discussed earlier, this detects if the compiler is running on Windows or not and includes the proper headers for the platform it is running on.

We also **define some macros**, which abstract out some of the difference between the Berkeley socket and Winsock APIs:

```
/*time_server.c continued*/

#ifdef _WIN32
#define ISVALIDSOCKET(s) ((s) != INVALID_SOCKET)
#define Closesocket(s) closesocket(s)
#define GETSOCKETERRNO() (WSAGetLastError())

#else
#define ISVALIDSOCKET(s) ((s) >= 0)
#define Closesocket(s) close(s)
#define SOCKET int
#define GETSOCKETERRNO() (errno)
#endif
```



We need a couple of standard C headers, hopefully for obvious reasons:

```
/*time_server.c continued*/  
  
#include <stdio.h>  
#include <string.h>  
#include <time.h>
```

Now, we are ready to begin the `main()` function. The first thing the `main()` function will do is initialize Winsock if we are compiling on Windows:

```
/*time_server.c continued*/  
  
int main() {  
  
    #if defined(_WIN32)  
        WSADATA d;  
        if (WSAStartup(MAKEWORD(2, 2), &d)) {  
            fprintf(stderr, "Failed to initialize.\n");  
            return 1;  
        }  
    #endif
```

We must now figure out the local address that our web server should bind to:

```
/*time_server.c continued*/  
  
    printf("Configuring local address...\n");  
    struct addrinfo hints;  
    memset(&hints, 0, sizeof(hints));  
    hints.ai_family = AF_INET;  
    hints.ai_socktype = SOCK_STREAM;  
    hints.ai_flags = AI_PASSIVE;  
  
    struct addrinfo *bind_address;  
    getaddrinfo(0, "8080", &hints, &bind_address);
```

We use `getaddrinfo()` to fill in a `struct addrinfo` structure with the needed information. `getaddrinfo()` takes a `hints` parameter, which tells it what we're looking for. In this case, we've zeroed out `hints` using `memset()` first. Then, we set `ai_family = AF_INET`. `AF_INET` specifies that we are looking for an IPv4 address. We could use `AF_INET6` to make our web server listen on an IPv6 address instead (more on this later).

Next, we set `ai_socktype = SOCK_STREAM`. This indicates that we're going to be using TCP. `SOCK_DGRAM` would be used if we were doing a UDP server instead. Finally, `ai_flags = AI_PASSIVE` is set. This is telling `getaddrinfo()` that we want it to bind to the wildcard address. That is, we are asking `getaddrinfo()` to set up the address, so we listen on any available network interface.

Once `hints` is set up properly, we declare a pointer to a `struct addrinfo` structure, which holds the return information from `getaddrinfo()`. We then call the `getaddrinfo()` function. The `getaddrinfo()` function has many uses, but for our purpose, it generates an address that's suitable for `bind()`. To make it generate this, we must pass in the first parameter as `NULL` and have the `AI_PASSIVE` flag set in `hints.ai_flags`.

The second parameter to `getaddrinfo()` is the port we listen for connections on. A standard HTTP server would use port 80. However, only privileged users on Unix-like operating systems can bind to ports 0 through 1023. The choice of port number here is arbitrary, but we use 8080 to avoid issues. If you are running with superuser privileges, feel free to change the port number to 80 if you like. Keep in mind that only one program can bind to a particular port at a time. If you try to use a port that is already in use, then the call to `bind()` fails. In this case, just change the port number to something else and try again.

It is common to see programs that don't use `getaddrinfo()` here. Instead, they fill in a `struct addrinfo` structure directly. The advantage to using `getaddrinfo()` is that it is protocol-independent. Using `getaddrinfo()` makes it very easy to convert our program from IPv4 to IPv6. In fact, we only need to change `AF_INET` to `AF_INET6`, and our program will work on IPv6. If we filled in the `struct addrinfo` structure directly, we would need to make many tedious changes to convert our program into IPv6.

Now that we've figured out our local address info, we can create the socket:

```
/*time_server.c continued*/

printf("Creating socket...\n");
SOCKET socket_listen;
socket_listen = socket(bind_address->ai_family,
                      bind_address->ai_socktype, bind_address->ai_protocol);
```

Here, we define `socket_listen` as a `SOCKET` type. Recall that `SOCKET` is a Winsock type on Windows, and that we have a macro defining it as `int` on other platforms. We call the `socket()` function to generate the actual socket. `socket()` takes three parameters: the socket family, the socket type, and the socket protocol. The reason we used `getaddrinfo()` before calling `socket()` is that we can now pass in parts of `bind_address` as the arguments to `socket()`. Again, this makes it very easy to change our program's protocol without needing a major rewrite.

It is common to see programs written so that they call `socket()` first. The problem with this is that it makes the program more complicated as the socket family, type, and protocol must be entered multiple times. Structuring our program as we have here is better.

We should check that the call to `socket()` was successful:

```
/*time_server.c continued*/

    if (!ISVALIDSOCKET(socket_listen)) {
        fprintf(stderr, "socket() failed. (%d)\n", GETSOCKETERRNO());
        return 1;
    }
```

We can check that `socket_listen` is valid using the `ISVALIDSOCKET()` macro we defined earlier. If the socket is not valid, we print an error message. Our `GETSOCKETERRNO()` macro is used to retrieve the error number in a cross-platform way.

After the socket has been created successfully, we can call `bind()` to associate it with our address from `getaddrinfo()`:

```
/*time_server.c continued*/

    printf("Binding socket to local address...\n");
    if (bind(socket_listen,
            bind_address->ai_addr, bind_address->ai_addrlen)) {
        fprintf(stderr, "bind() failed. (%d)\n", GETSOCKETERRNO());
        return 1;
    }
    freeaddrinfo(bind_address);
```

`bind()` returns 0 on success and non-zero on failure. If it fails, we print the error number much like we did for the error handling on `socket()`. `bind()` fails if the port we are binding to is already in use. In that case, either close the program using that port or change your program to use a different port.

After we have bound to `bind_address`, we can call the `freeaddrinfo()` function to release the address memory.

Once the socket has been created and bound to a local address, we can cause it to **start listening for connections with the `listen()` function:**

```
/*time_server.c continued*/

    printf("Listening...\n");
    if (listen(socket_listen, 10) < 0) {
        fprintf(stderr, "listen() failed. (%d)\n", GETSOCKETERRNO());
        return 1;
    }
```

The second argument to `listen()`, which is 10 in this case, tells `listen()` how many connections it is allowed to queue up. If many clients are connecting to our server all at once, and we aren't dealing with them fast enough, then the operating system begins to queue up these incoming connections. If 10 connections become queued up, then the operating system will reject new connections until we remove one from the existing queue.

Error handling for `listen()` is done the same way as we did for `bind()` and `socket()`.

After the socket has begun listening for connections, we can **accept any incoming connection with the `accept()` function:**

```
/*time_server.c continued*/

    printf("Waiting for connection...\n");
    struct sockaddr_storage client_address;
    socklen_t client_len = sizeof(client_address);
    SOCKET socket_client = accept(socket_listen,
        (struct sockaddr*) &client_address, &client_len);
    if (!ISVALIDSOCKET(socket_client)) {
        fprintf(stderr, "accept() failed. (%d)\n", GETSOCKETERRNO());
        return 1;
    }
```

`accept()` has a few functions. First, when it's called, it will block your program until a new connection is made. In other words, your program will sleep until a connection is made to the listening socket. When the new connection is made, `accept()` will create a new socket for it. Your original socket continues to listen for new connections, but the new socket returned by `accept()` can be used to send and receive data over the newly established connection. `accept()` also fills in address info of the client that connected.

Before calling `accept()`, we must declare a new `struct sockaddr_storage` variable to store the address info for the connecting client. The `struct sockaddr_storage` type is guaranteed to be large enough to hold the largest supported address on your system. We must also tell `accept()` the size of the address buffer we're passing in. When `accept()` returns, it will have filled in `client_address` with the connected client's address and `client_len` with the length of that address. `client_len` differs, depending on whether the connection is using IPv4 or IPv6.

We store the return value of `accept()` in `socket_client`. We check for errors by detecting if `client_socket` is a valid socket. This is done in exactly the same way as we did for `socket()`.

At this point, a TCP connection has been established to a remote client. We can print the client's address to the console:

```
/*time_server.c continued*/

printf("Client is connected... ");
char address_buffer[100];
getnameinfo((struct sockaddr*)&client_address,
            client_len, address_buffer, sizeof(address_buffer), 0, 0,
            NI_NUMERICHOST);
printf("%s\n", address_buffer);
```

This step is completely optional, but it is good practice to log network connections somewhere.

`getnameinfo()` takes the client's address and address length. The address length is needed because `getnameinfo()` can work with both IPv4 and IPv6 addresses. We then pass in an output buffer and buffer length. This is the buffer that `getnameinfo()` writes its hostname output to. The next two arguments specify a second buffer and its length. `getnameinfo()` outputs the service name to this buffer. We don't care about that, so we've passed in 0 for those two parameters. Finally, we pass in the `NI_NUMERICHOST` flag, which specifies that we want to see the hostname as an IP address.

As we are programming a web server, we expect the client (for example, a web browser) to send us an HTTP request. We read this request using the `recv()` function:

```
/*time_server.c continued*/

printf("Reading request...\n");
char request[1024];
int bytes_received = recv(socket_client, request, 1024, 0);
printf("Received %d bytes.\n", bytes_received);
```

We define a request buffer, so that we can store the browser's HTTP request. In this case, we allocate 1,024 bytes to it, which should be enough for this application. `recv()` is then called with the client's socket, the request buffer, and the request buffer size. `recv()` returns the number of bytes that are received. If nothing has been received yet, `recv()` blocks until it has something. If the connection is terminated by the client, `recv()` returns 0 or -1, depending on the circumstance. We are ignoring that case here for simplicity, but you should always check that `recv() > 0` in production. The last parameter to `recv()` is for flags. Since we are not doing anything special, we simply pass in 0.

The request received from our client should follow the proper HTTP protocol. We will go into detail about HTTP in Chapter 6, *Building a Simple Web Client*, and Chapter 7, *Building a Simple Web Server*, where we will work on web clients and servers. A real web server would need to parse the request and look at which resource the web browser is requesting. Our web server only has one function—to tell us what time it is. So, for now, we just ignore the request altogether.

If you want to print the browser's request to the console, you can do it like this:

```
printf("%.s", bytes_received, request);
```

Note that we use the `printf()` format string, `"%.s"`. This tells `printf()` that we want to print a specific number of characters—`bytes_received`. It is a common mistake to try printing data that's received from `recv()` directly as a C string. There is no guarantee that the data received from `recv()` is null terminated! If you try to print it with `printf(request)` or `printf("%s", request)`, you will likely receive a segmentation fault error (or at best it will print some garbage).

Now that the web browser has sent its request, we can send our response back:

```
/*time_server.c continued*/
```

```
printf("Sending response...\n");
const char *response =
    "HTTP/1.1 200 OK\r\n"
    "Connection: close\r\n"
    "Content-Type: text/plain\r\n\r\n"
    "Local time is: ";
int bytes_sent = send(socket_client, response, strlen(response), 0);
printf("Sent %d of %d bytes.\n", bytes_sent, (int)strlen(response));
```

To begin with, we set `char *response` to a standard HTTP response header and the beginning of our message (`Local time is:`). We will discuss HTTP in detail in Chapter 6, *Building a Simple Web Client*, and Chapter 7, *Building a Simple Web Server*. For now, know that this response tells the browser three things—your request is OK; the server will close the connection when all the data is sent and the data you receive will be plain text.

The HTTP response header ends with a blank line. HTTP requires line endings to take the form of a carriage return character, followed by a newline character. So, a blank line in our response is `\r\n`. The part of the string that comes after the blank line, `Local time is:`, is treated by the browsers as plain text.

We send the data to the client using the `send()` function. This function takes the client's socket, a pointer to the data to be sent, and the length of the data to send. The last parameter to `send()` is flags. We don't need to do anything special, so we pass in 0.

`send()` returns the number of bytes sent. You should generally check that the number of bytes sent was as expected, and you should attempt to send the rest if it's not. We are ignoring that detail here for simplicity. (Also, we are only attempting to send a few bytes; if `send()` can't handle that, then something is probably very broken, and resending won't help.)

After the HTTP header and the beginning of our message is sent, we can send the actual time. We get the local time the same way we did in `time_console.c`, and we send it using `send()`:

```
/*time_server.c continued*/

time_t timer;
time(&timer);
char *time_msg = ctime(&timer);
bytes_sent = send(socket_client, time_msg, strlen(time_msg), 0);
printf("Sent %d of %d bytes.\n", bytes_sent, (int)strlen(time_msg));
```

We must then **close the client connection to indicate** to the browser that we've sent all of our data:

```
/*time_server.c continued*/

printf("Closing connection...\n");
CLOSESOCKET(socket_client);
```

If we don't close the connection, the browser will just wait for more data until it times out.

At this point, we could call `accept()` on `socket_listen` to accept additional connections. That is exactly what a real server would do. However, as this is just a quick example program, we will instead **close the listening socket too** and terminate the program:

```
/*time_server.c continued*/

    printf("Closing listening socket...\n");
    CLOSESOCKET(socket_listen);

#ifdef _WIN32
    WSACleanup();
#endif

    printf("Finished.\n");

    return 0;
}
```

That's the complete program. After you compile and run it, you can navigate a web browser to it, and it'll display the current time.

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

```
gcc time_server.c -o time_server
./time_server
```

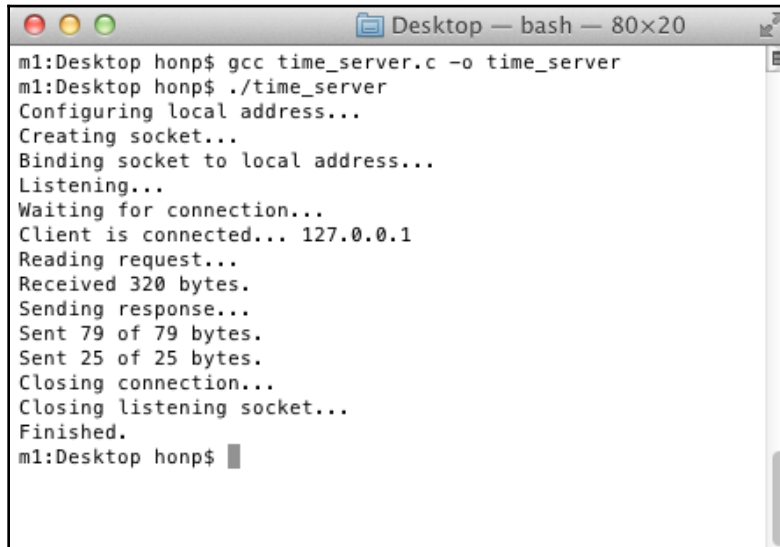
On Windows, you can compile and run with MinGW using these commands:

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

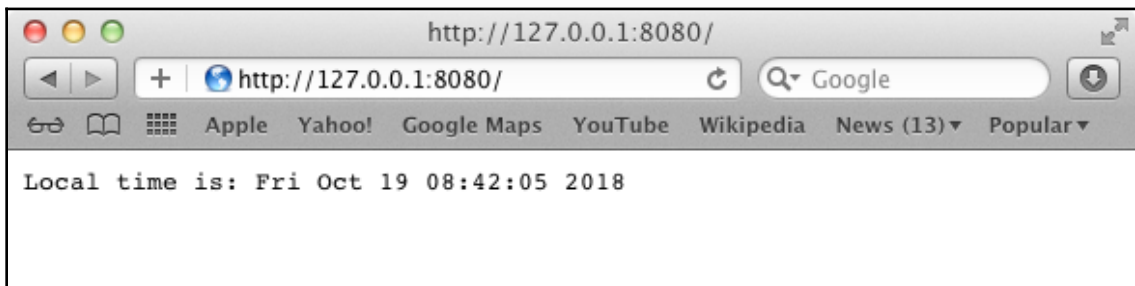
When you run the program, it waits for a connection. You can open a web browser and navigate to `http://127.0.0.1:8080` to load the web page. Recall that `127.0.0.1` is the IPv4 loopback address, which connects to the same machine it's running on. The `:8080` part of the URL specifies the port number to connect to. If it were left out, your browser would default to port 80, which is the standard for HTTP connections.



Here is what you should see if you compile and run the program, and then connect a web browser to it on the same computer:

A terminal window titled "Desktop — bash — 80x20" showing the execution of a C program. The user runs 'gcc time\_server.c -o time\_server' and then './time\_server'. The program outputs status messages: 'Configuring local address...', 'Creating socket...', 'Binding socket to local address...', 'Listening...', 'Waiting for connection...', 'Client is connected... 127.0.0.1', 'Reading request...', 'Received 320 bytes.', 'Sending response...', 'Sent 79 of 79 bytes.', 'Sent 25 of 25 bytes.', 'Closing connection...', 'Closing listening socket...', and 'Finished.' The prompt returns to 'm1:Desktop honp\$'.

Here is the web browser connected to our `time_server` program on port 8080:

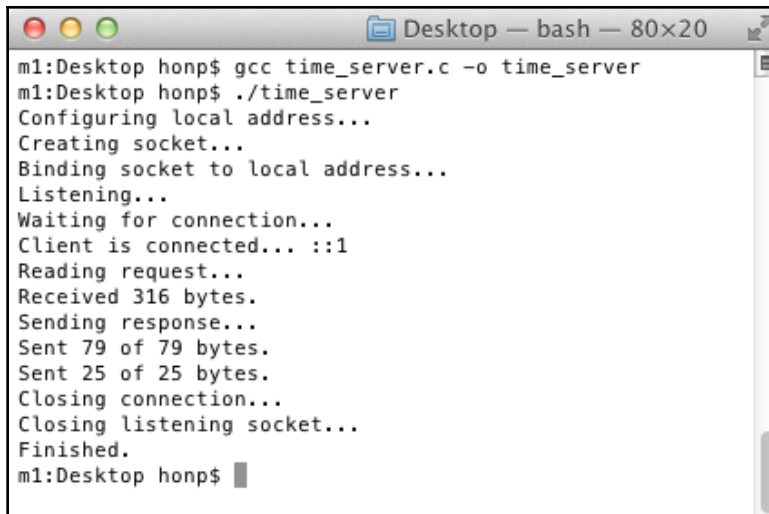


## Working with IPv6

Please recall the `hints.ai_family = AF_INET` part of `time_server.c` near the beginning of the `main()` function. If this line is changed to `hints.ai_family = AF_INET6`, then your web server listens for IPv6 connections instead of IPv4 connections. This modified file is included in the GitHub repository as `time_server_ipv6.c`.

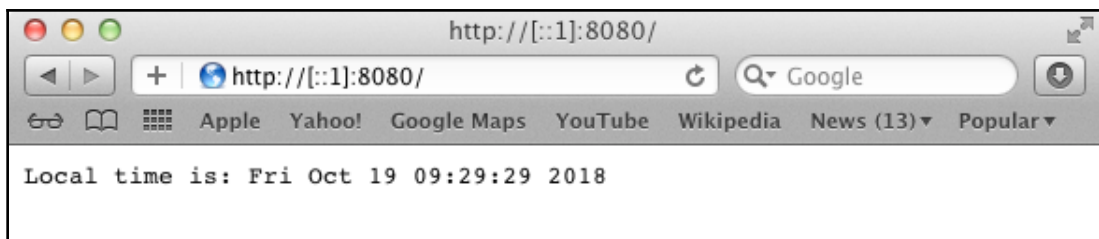
In this case, you should navigate your web browser to `http://[::1]:8080` to see the web page. `::1` is the IPv6 loopback address, which tells the web browser to connect to the same machine it's running on. In order to use IPv6 addresses in URLs, you need to put them in square brackets, `[]. :8080` specifies the port number in the same way that we did for the IPv4 example.

Here is what you should see when compiling, running, and connecting a web browser to our `time_server_ipv6` program:

A terminal window titled "Desktop — bash — 80x20" showing the execution of a C program. The user runs `gcc time_server.c -o time_server` and then `./time_server`. The program outputs status messages: "Configuring local address...", "Creating socket...", "Binding socket to local address...", "Listening...", "Waiting for connection...", "Client is connected... ::1", "Reading request...", "Received 316 bytes.", "Sending response...", "Sent 79 of 79 bytes.", "Sent 25 of 25 bytes.", "Closing connection...", "Closing listening socket...", and "Finished." The prompt returns to `m1:Desktop honp$`.

```
m1:Desktop honp$ gcc time_server.c -o time_server
m1:Desktop honp$ ./time_server
Configuring local address...
Creating socket...
Binding socket to local address...
Listening...
Waiting for connection...
Client is connected... ::1
Reading request...
Received 316 bytes.
Sending response...
Sent 79 of 79 bytes.
Sent 25 of 25 bytes.
Closing connection...
Closing listening socket...
Finished.
m1:Desktop honp$
```

Here is the web browser that's connected to our server using an IPv6 socket:



See `time_server_ipv6.c` for the complete program.

## Supporting both IPv4 and IPv6

It is also possible for the listening IPv6 socket to accept IPv4 connections with a dual-stack socket. Not all operating systems support dual-stack sockets. With Linux in particular, support varies between distros. If your operating system does support dual-stack sockets, then I highly recommend implementing your server programs using this feature. It allows your programs to communicate with both IPv4 and IPv6 peers while requiring no extra work on your part.

We can modify `time_server_ipv6.c` to use dual-stack sockets with only a minor addition. After the call to `socket()` and before the call to `bind()`, we must clear the `IPV6_V6ONLY` flag on the socket. This is done with the `setsockopt()` function:

```
/*time_server_dual.c excerpt*/

int option = 0;
if (setsockopt(socket_listen, IPPROTO_IPV6, IPV6_V6ONLY,
(void*)&option, sizeof(option)) {
    fprintf(stderr, "setsockopt() failed. (%d)\n", GETSOCKETERRNO());
    return 1;
}
```

We first declare `option` as an integer and set it to 0. `IPV6_V6ONLY` is enabled by default, so we clear it by setting it to 0. `setsockopt()` is called on the listening socket. We pass in `IPPROTO_IPV6` to tell it what part of the socket we're operating on, and we pass in `IPV6_V6ONLY` to tell it which flag we are setting. We then pass in a pointer to our option and its length. `setsockopt()` returns 0 on success.

Windows Vista and later supports dual-stack sockets. However, many Windows headers are missing the definitions for `IPV6_V6ONLY`. For this reason, it might make sense to include the following code snippet at the top of the file:

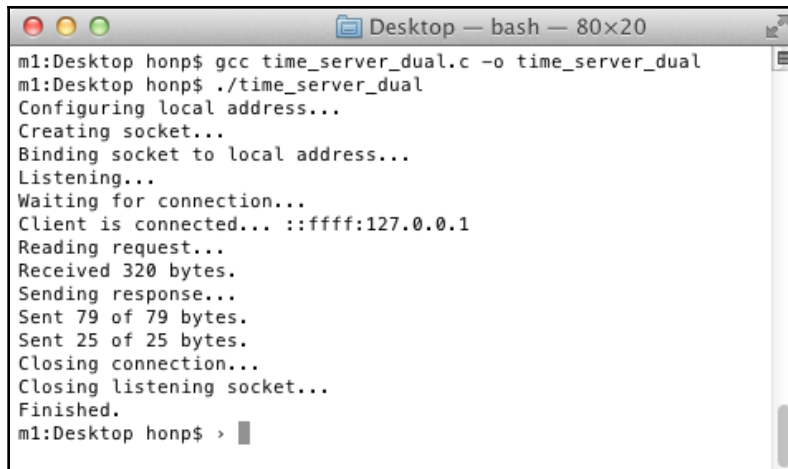
```
/*time_server_dual.c excerpt*/

#ifdef _WIN32
#define IPV6_V6ONLY 27
#endif
```

Keep in mind that the socket needs to be initially created as an IPv6 socket. This is accomplished with the `hints.ai_family = AF_INET6` line in our code.

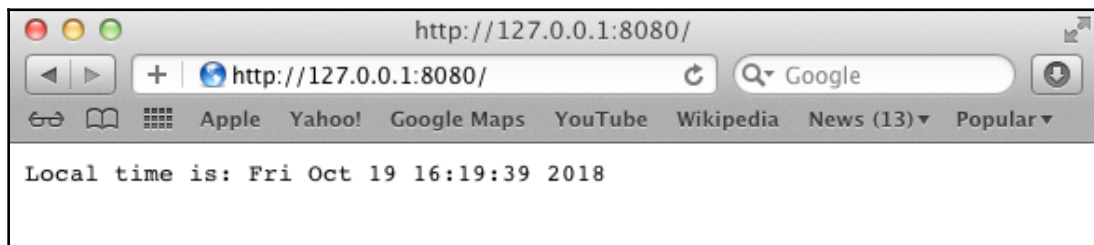
When an IPv4 peer connects to our dual-stack server, the connection is remapped to an IPv6 connection. This happens automatically and is taken care of by the operating system. When your program sees the client IP address, it will still be presented as a special IPv6 address. These are represented by IPv6 addresses where the first 96 bits consist of the prefix—`0:0:0:0:0:ffff`. The last 32 bits of the address are used to store the IPv4 address. For example, if a client connects with the IPv4 address `192.168.2.107`, then your dual-stack server sees it as the IPv6 address `::ffff.192.168.2.107`.

Here is what it looks like to compile, run, and connect to `time_server_dual`:

A terminal window titled "Desktop — bash — 80x20" showing the execution of a C program. The user runs `gcc time_server_dual.c -o time_server_dual` and then `./time_server_dual`. The program outputs status messages: "Configuring local address...", "Creating socket...", "Binding socket to local address...", "Listening...", "Waiting for connection...", "Client is connected... ::ffff:127.0.0.1", "Reading request...", "Received 320 bytes.", "Sending response...", "Sent 79 of 79 bytes.", "Sent 25 of 25 bytes.", "Closing connection...", "Closing listening socket...", "Finished." The prompt returns to `m1:Desktop honp$`.

```
m1:Desktop honp$ gcc time_server_dual.c -o time_server_dual
m1:Desktop honp$ ./time_server_dual
Configuring local address...
Creating socket...
Binding socket to local address...
Listening...
Waiting for connection...
Client is connected... ::ffff:127.0.0.1
Reading request...
Received 320 bytes.
Sending response...
Sent 79 of 79 bytes.
Sent 25 of 25 bytes.
Closing connection...
Closing listening socket...
Finished.
m1:Desktop honp$ >
```

Here is a web browser connected to our `time_server_dual` program using the loopback IPv4 address:



Notice that the browser is navigating to the IPv4 address `127.0.0.1`, but we can see on the console that the server sees the connection as coming from the IPv6 address `::ffff:127.0.0.1`.

See `time_server_dual.c` for the complete dual-stack socket server.

## Networking with *inetd*

On Unix-like systems, such as Linux or macOS, a service called *inetd* can be used to turn console-only applications into networked ones. You can configure *inetd* (with `/etc/inetd.conf`) with your program's location, port number, protocol (TCP or UDP), and the user you want it to run as. *inetd* will then listen for connections on your desired port. After an incoming connection is accepted by *inetd*, it will start your program and redirect all socket input/output through `stdin` and `stdout`.

Using *inetd*, we could have `time_console.c` behave like `time_server.c` with very minimal changes. We would only need to add in an extra `printf()` function with the HTTP response header, read from `stdin`, and configure *inetd*.

You may be able to use *inetd* on Windows through Cygwin or the Windows Subsystem for Linux.

## Summary

In this chapter, we learned about the basics of using sockets for network programming. Although there are many differences between Berkeley sockets (used on Unix-like operating systems) and Winsock sockets (used on Windows), we mitigated those differences with preprocessor statements. In this way, it was possible to write one program that compiles cleanly on Windows, Linux, and macOS.

We covered how the UDP protocol is connectionless and what that means. We learned that TCP, being a connection-oriented protocol, gives some reliability guarantees, such as automatically detecting and resending lost packets. We also saw that UDP is often used for simple protocols (for example, DNS) and for real-time streaming applications. TCP is used for most other protocols.

After that, we worked through a real example by converting a console application into a web server. We learned how to write the program using the `getaddrinfo()` function, and why that matters for making the program IPv4/IPv6-agnostic. We used `bind()`, `listen()`, and `accept()` on the server to wait for an incoming connection from the web browser. Data was then read from the client using `recv()`, and a reply was sent using `send()`. Finally, we terminated the connection with `close()` (`closesocket()` on Windows).

When we built the web server, `time_server.c`, we covered much ground. It's OK if you didn't understand all of it. We will revisit many of these functions again throughout Chapter 3, *An In-Depth Overview of TCP Connections*, and the rest of this book.

In the next chapter, Chapter 3, *An In-Depth Overview of TCP Connections*, we will consider programming for TCP connections in more depth.

## Questions

Try these questions to test your knowledge on this chapter:

1. What is a socket?
2. What is a connectionless protocol? What is a connection-oriented protocol?
3. Is UDP a connectionless or connection-oriented protocol?
4. Is TCP a connectionless or connection-oriented protocol?
5. What types of applications generally benefit from using the UDP protocol?
6. What types of applications generally benefit from using the TCP protocol?
7. Does TCP guarantee that data will be transmitted successfully?
8. What are some of the main differences between Berkeley sockets and Winsock sockets?
9. What does the `bind()` function do?
10. What does the `accept()` function do?
11. In a TCP connection, does the client or the server send application data first?

Answers are in Appendix A, *Answers to Questions*.