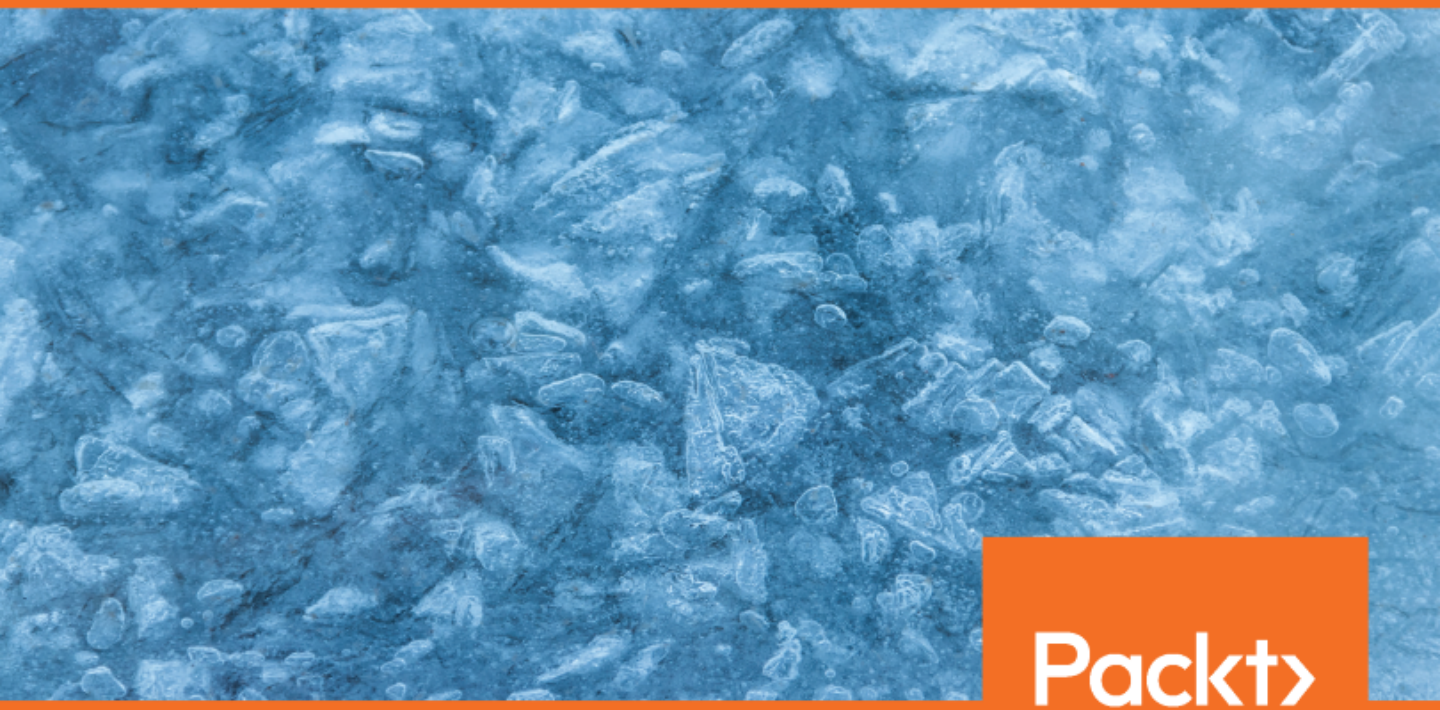
Hands-On Network Programming with C

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



Lewis Van Winkle

Packt
www.packt.com

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

    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.

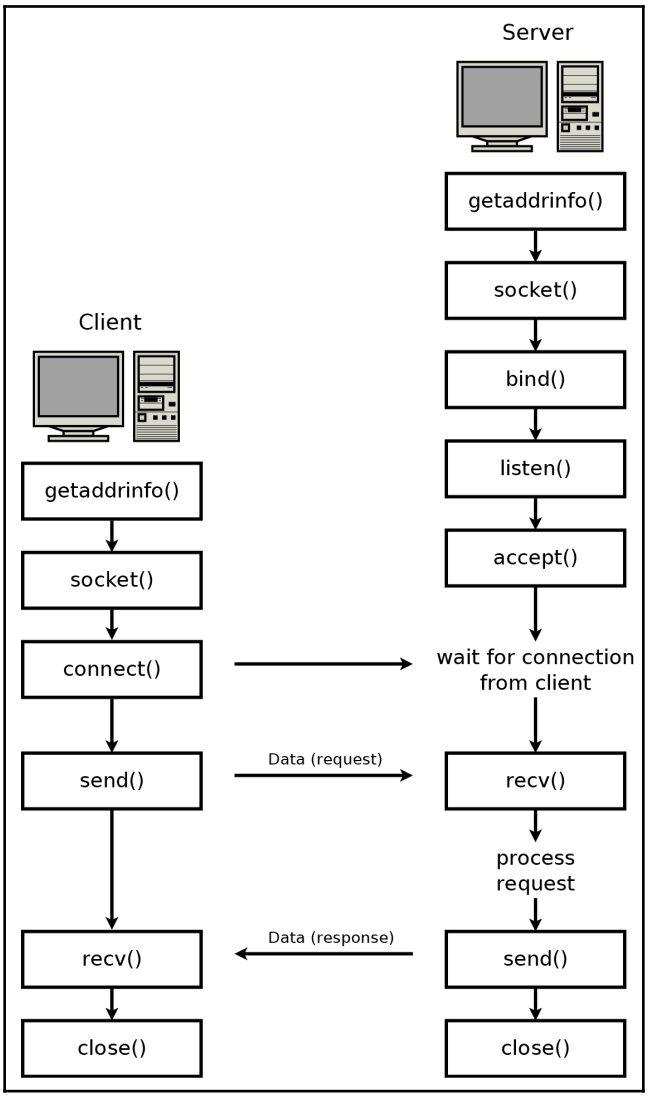
TCP program flow

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.

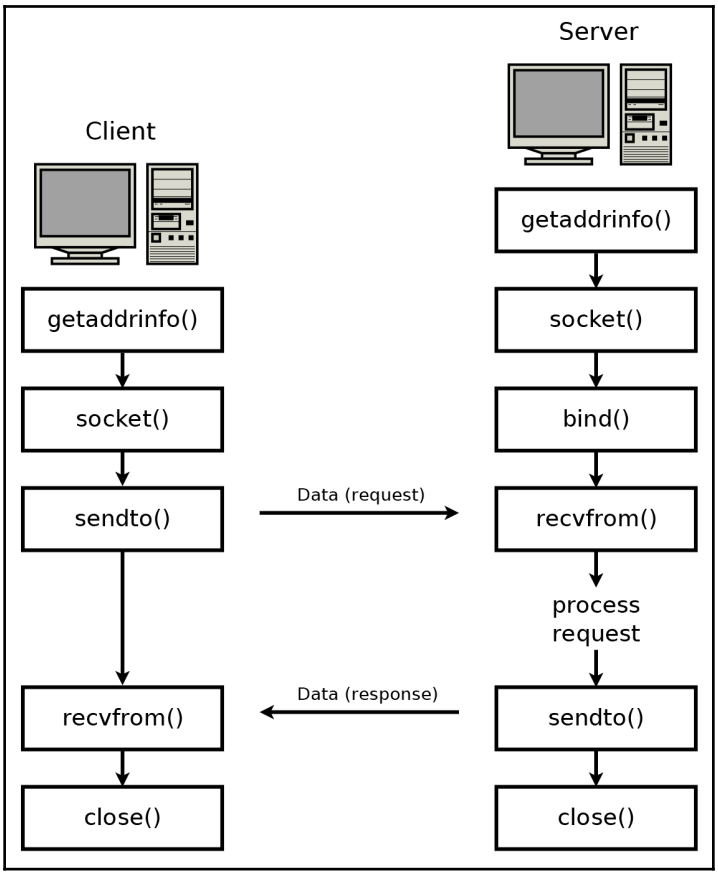
Many other variations of TCP operation are possible too, and we will look at some in `Chapter 3`, *An In-Depth Overview of TCP Connections*.

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*/

#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
```

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*/

#if defined(_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);


#if defined(_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:

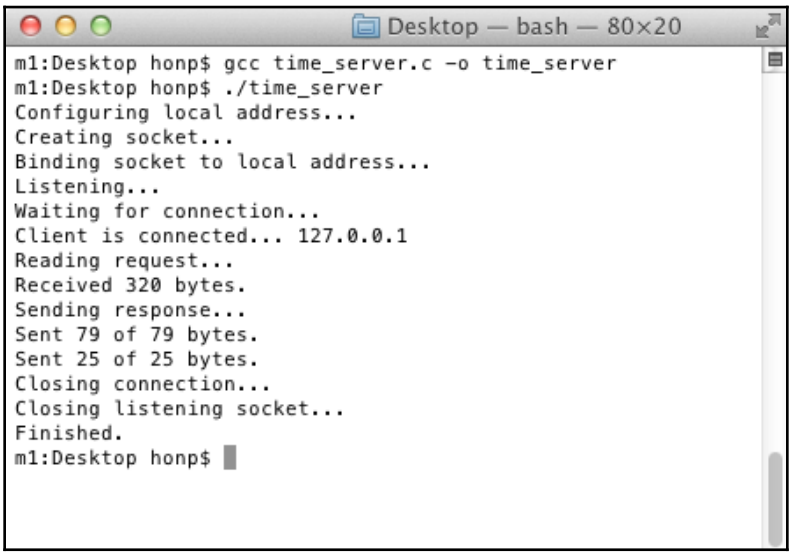
```
gcc time_server.c -o time_server
./time_server
```

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

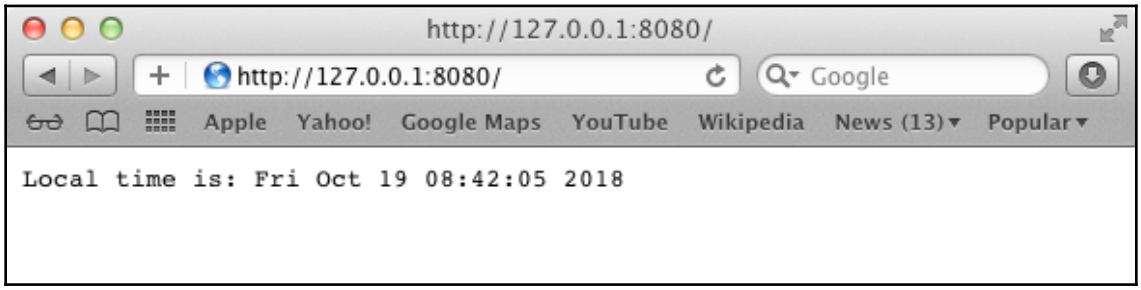
```
gcc time_server.c -o time_server.exe -lws2_32
time_server
```

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:



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

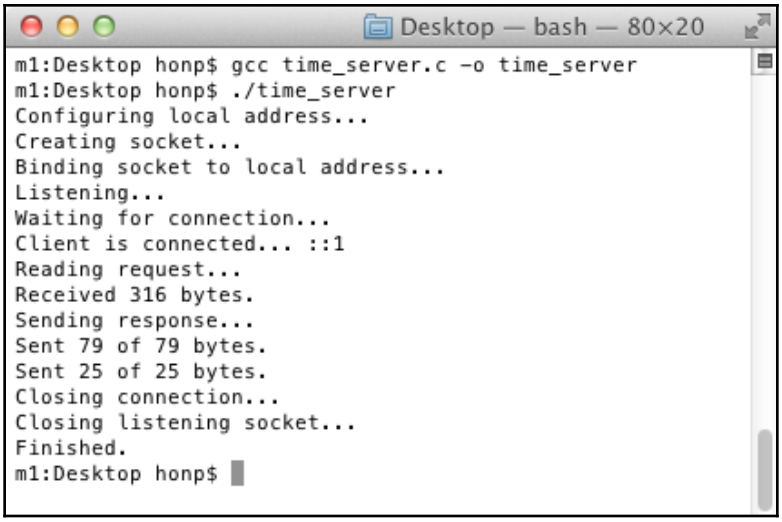


Working with IPv6

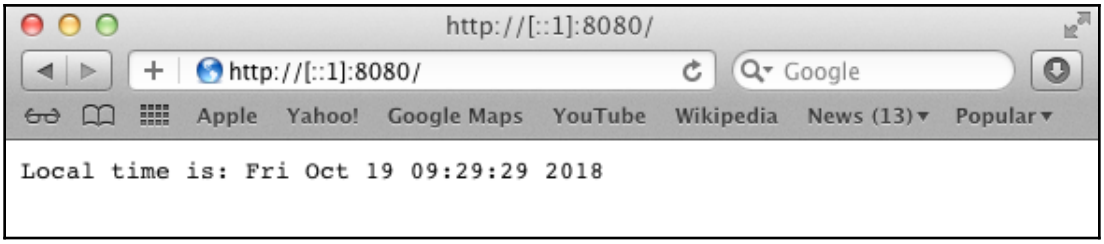
Please recall the `hints.ai_family = AF_INET` part of `time_server.c` near the beginning of the `main()` function. If this line is changed to `hints.ai_family = AF_INET6`, then your web server listens for IPv6 connections instead of IPv4 connections. This modified file is included in the GitHub repository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



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:

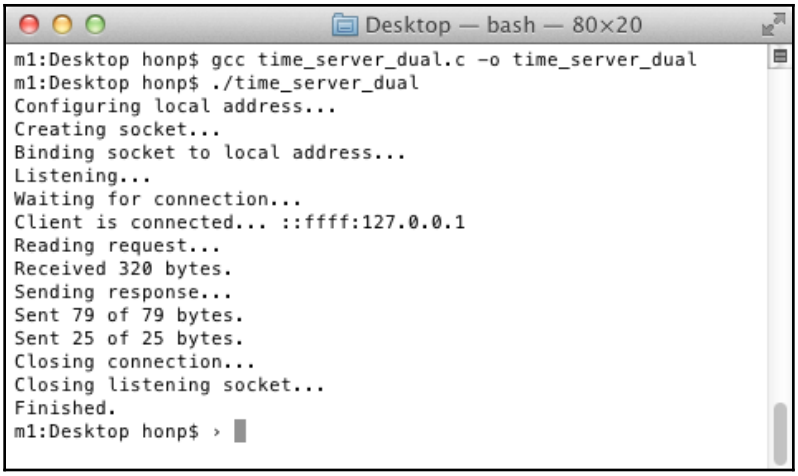
```
/*time_server_dual.c excerpt*/

#if !defined(IPV6_V6ONLY)
#define IPV6_V6ONLY 27
#endif
```

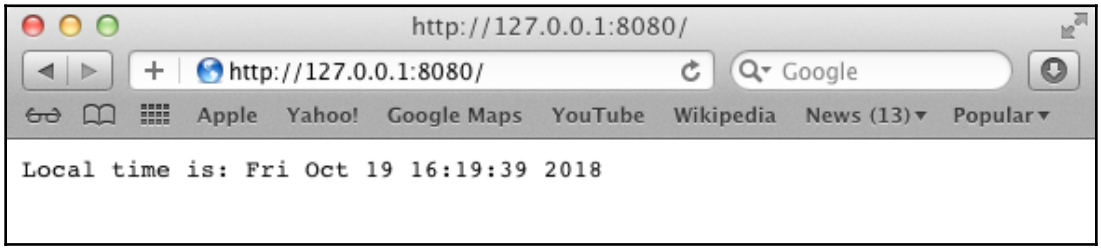
Keep in mind that the socket needs to be initially created as an IPv6 socket. This is accomplished with the `hints.ai_family = AF_INET6` line in our code.

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`:



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*.

Technical requirements

The example programs for 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 in this book's GitHub repository: `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/chap03
```

Each example program in this chapter runs on Windows, Linux, and macOS. While compiling on Windows, each example program requires that you link with the Winsock library. This can be accomplished by passing the `-lws2_32` option to `gcc`.

We provide the exact commands that are needed to compile each example as it is introduced.

All of the example programs in this chapter require the same header files and C macros that we developed in `Chapter 2`, *Getting to Grips with Socket APIs*. For brevity, we put these statements in a separate header file, `chap03.h`, which we can include in each program. For an explanation of these statements, please refer to `Chapter 2`, *Getting to Grips with Socket APIs*.

The contents of `chap03.h` is as follows:

```
/*chap03.h*/

#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


#if defined(_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


#include <stdio.h>
#include <string.h>
```

Multiplexing TCP connections

The socket APIs are blocking by default. When you use `accept()` to wait for an incoming connection, your program's execution is blocked until a new incoming connection is actually available. When you use `recv()` to read incoming data, your program's execution blocks until new data is actually available.

In the last chapter, we built a simple TCP server. This server only accepted one connection, and it only read data from that connection once. Blocking wasn't a problem then, because our server had no other purpose than to serve its one and only client.

In the general case, though, blocking I/O can be a significant problem. Imagine that our server from `Chapter 2`, *Getting to Grips with Socket APIs*, needed to serve multiple clients. Then, imagine that one slow client connected to it. Maybe this slow client takes a minute before sending its first data. During this minute, our server would simply be waiting on the `recv()` call to return. If other clients were trying to connect, they would have to wait it out.

3
An In-Depth Overview of TCP Connections

In `Chapter 2`, *Getting to Grips with Socket APIs*, we implemented a simple TCP server that served a web page with HTTP. In this chapter, we will begin by implementing a TCP client. This client is able to establish an IPv4 or IPv6 TCP connection with any listening TCP server. It will be a useful debugging tool that we can reuse in the rest of this book.

Our TCP server from the last chapter was limited to accepting only one connection. In this chapter, we will look at multiplexing techniques to allow our programs to handle many separate connections simultaneously.

The following topics are covered in this chapter:

- Configuring a remote address with `getaddrinfo()`
- Initiating a TCP connection with `connect()`
- Detecting terminal input in a non-blocking manner
- Multiplexing with `fork()`
- Multiplexing with `select()`
- Detecting peer disconnects
- Implementing a very basic microservice
- The stream-like nature of TCP
- The blocking behavior of `send()`

Blocking on `recv()` like this isn't really acceptable. A real application usually needs to be able to manage several connections simultaneously. This is obviously true on the server side, as most servers are built to manage many connected clients. Imagine running a website where hundreds of clients are connected at once. Serving these clients one at a time would be a non-starter.

Blocking also isn't usually acceptable on the client side either. If you imagine building a fast web browser, it needs to be able to download many images, scripts, and other resources in parallel. Modern web browsers also have a **tab** feature where many whole web pages can be loaded in parallel.

What we need is a technique for handling many separate connections simultaneously.

Polling non-blocking sockets

It is possible to configure sockets to use a non-blocking operation. One way to do this is by calling `fcntl()` with the `O_NONBLOCK` flag (`ioctlsocket()` with the `FIONBIO` flag on Windows), although other ways also exist. Once in non-blocking mode, a call to `recv()` with no data will return immediately. See `Chapter 13`, *Socket Programming Tips and Pitfalls*, for more information.

A program structured with this in mind could simply check each of its active sockets in turn, continuously. It would handle any socket that returned data and ignore any socket that didn't. This is called **polling**. Polling can be a waste of computer resources since most of the time, there will be no data to read. It also complicates the program somewhat, as the programmer is required to manually track which sockets are active and which state, they are in. Return values from `recv()` must also be handled differently than with blocking sockets.

For these reasons, we won't use polling in this book.

Forking and multithreading

Another possible solution to multiplexing socket connections is to start a new thread or process for each connection. In this case, blocking sockets are fine, as they block only their servicing thread/process, and they do not block other threads/processes. This can be a useful technique, but it also has some downsides. First of all, threading is tricky to get right. This is especially true if the connections must share any state between them. It is also less portable as each operating system provides a different API for these features.

On Unix-based systems, such as Linux and macOS, starting a new process is very easy. We simply use the `fork()` function. The `fork()` function splits the executing program into two separate processes. A multi-process TCP server may accept connections like this:

```
while(1) {
    socket_client = accept(socket_listen, &new_client, &new_client_length);
    int pid = fork();
    if (pid == 0) { //child process
        close(socket_listen);
        recv(socket_client, ...);
        send(socket_client, ...);
        close(socket_client);
        exit(0);
    }
    //parent process
    close(socket_client);
}
```

In this example, the program blocks on `accept()`. When a new connection is established, the program calls `fork()` to split into two processes. The child process, where `pid == 0`, only services this one connection. Therefore, the child process can use `recv()` freely without worrying about blocking. The parent process simply calls `close()` on the new connection and returns to listening for more connections with `accept()`.

Using multiple processes/threads is much more complicated on Windows. Windows provides `CreateProcess()`, `CreateThread()`, and many other functions for these features. However—and I can say this objectively—they are all much harder to use than Unix's `fork()`.

Debugging these multi-process/thread programs can be much more difficult compared to the single process case. Communicating between sockets and managing shared state is also much more burdensome. For these reasons, we will avoid `fork()` and other multi-process/thread techniques for the rest of this book.

That being said, an example TCP server using fork is included in this chapter's code. It's named `tcp_serve_toupper_fork.c`. It does not run on Windows, but it should compile and run cleanly on Linux and macOS. I would suggest finishing the rest of this chapter before looking at it.

The select() function

Our preferred technique for multiplexing is to use the `select()` function. We can give `select()` a set of sockets, and it tells us which ones are ready to be read. It can also tell us which sockets are ready to write to and which sockets have exceptions. Furthermore, it is supported by both Berkeley sockets and Winsock. Using `select()` keeps our programs portable.

Synchronous multiplexing with select()

The `select()` function has several useful features. Given a set of sockets, it can be used to block until any of the sockets in that set is ready to be read from. It can also be configured to return if a socket is ready to be written to or if a socket has an error. Additionally, we can configure `select()` to return after a specified time if none of these events take place.

The C function prototype for `select()` is as follows:

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
    fd_set *exceptfds, struct timeval *timeout);
```

Before calling `select()`, we must first add our sockets into an `fd_set`. If we have three sockets, `socket_listen`, `socket_a`, and `socket_b`, we add them to an `fd_set`, like this:

```
fd_set our_sockets;
FD_ZERO(&our_sockets);
FD_SET(socket_listen, &our_sockets);
FD_SET(socket_a, &our_sockets);
FD_SET(socket_b, &our_sockets);
```

It is important to zero-out the `fd_set` using `FD_ZERO()` before use.

Socket descriptors are then added to the `fd_set` one at a time using `FD_SET()`. A socket can be removed from an `fd_set` using `FD_CLR()`, and we can check for the presence of a socket in the set using `FD_ISSET()`.

You may see some programs manipulating an `fd_set` directly. I recommend that you use only `FD_ZERO()`, `FD_SET()`, `FD_CLR()`, and `FD_ISSET()` to maintain portability between Berkeley sockets and Winsock.

`select()` also requires that we pass a number that's larger than the largest socket descriptor we are going to monitor. (This parameter is ignored on Windows, but we will always do it anyway for portability.) We store the largest socket descriptor in a variable, like this:

```
SOCKET max_socket;
max_socket = socket_listen;
if (socket_a > max_socket) max_socket = socket_a;
if (socket_b > max_socket) max_socket = socket_b;
```

When we call `select()`, it modifies our `fd_set` of sockets to indicate which sockets are ready. For that reason, we want to copy our socket set before calling it. We can copy an `fd_set` with a simple assignment like this, and then call `select()` like this:

```
fd_set copy;
copy = our_sockets;

select(max_socket+1, &copy, 0, 0, 0);
```

This call blocks until at least one of the sockets is ready to be read from. When `select()` returns, `copy` is modified so that it only contains the sockets that are ready to be read from. We can check which sockets are still in `copy` using `FD_ISSET()`, like this:

```
if (FD_ISSET(socket_listen, &copy)) {
    //socket_listen has a new connection
    accept(socket_listen...
}

if (FD_ISSET(socket_a, &copy)) {
    //socket_a is ready to be read from
    recv(socket_a...
}

if (FD_ISSET(socket_b, &copy)) {
    //socket_b is ready to be read from
    recv(socket_b...
}
```

In the previous example, we passed our `fd_set` as the second argument to `select()`. If we wanted to monitor an `fd_set` for writability instead of readability, we would pass our `fd_set` as the third argument to `select()`. Likewise, we can monitor a set of sockets for exceptions by passing it as the fourth argument to `select()`.

select() timeout

The last argument taken by `select()` allows us to specify a timeout. It expects a pointer to `struct timeval`. The `timeval` structure is declared as follows:

```
struct timeval {
    long tv_sec;
    long tv_usec;
}
```

`tv_sec` holds the number of seconds, and `tv_usec` holds the number of microseconds (1,000,000th second). If we want `select()` to wait a maximum of 1.5 seconds, we can call it like this:

```
struct timeval timeout;
timeout.tv_sec = 1;
timeout.tv_usec = 500000;
select(max_socket+1, &copy, 0, 0, &timeout);
```

In this case, `select()` returns after a socket in `fd_set copy` is ready to read or after 1.5 seconds has elapsed, whichever is sooner.

If `timeout.tv_sec = 0` and `timeout.tv_usec = 0`, then `select()` returns immediately (after changing the `fd_set` as appropriate). As we saw previously, if we pass in a null pointer for the timeout parameter, then `select()` does not return until at least one socket is ready to be read.

`select()` can also be used to monitor for writeable sockets (sockets where we could call `send()` without blocking), and sockets with exceptions. We can check for all three conditions with one call:

```
select(max_sockets+1, &ready_to_read, &ready_to_write, &excepted,
&timeout);
```

On success, `select()` itself returns the number of socket descriptors contained in the (up to) three descriptor sets it monitored. The return value is zero if it timed out before any sockets were readable/writeable/excepted. `select()` returns `-1` to indicate an error.

Iterating through an fd_set

We can iterate through an `fd_set` using a simple `for` loop. Essentially, we start at `1`, since all socket descriptors are positive numbers, and we continue through to the largest known socket descriptor in the set. For each possible socket descriptor, we simply use `FD_ISSET()` to check if it is in the set. If we wanted to call `CLOSESOCKET()` for every socket in the `fd_set master`, we could do it like this:

```
SOCKET i;
for (i = 1; i <= max_socket; ++i) {
    if (FD_ISSSET(i, &master)) {
        CLOSESOCKET(i);
    }
}
```

This may seem like a brute-force approach, and it actually kind of is. However, these are the tools that we have to work with. `FD_ISSET()` runs very fast, and it's likely that processor time spent on other socket operations will dwarf what time was spent iterating through them in this manner. Nevertheless, you may be able to optimize this operation by additionally storing your sockets in an array or linked list. I don't recommend that you make this optimization unless you profile your code and find the simple `for` loop iteration to be a significant bottleneck.

select() on non-sockets

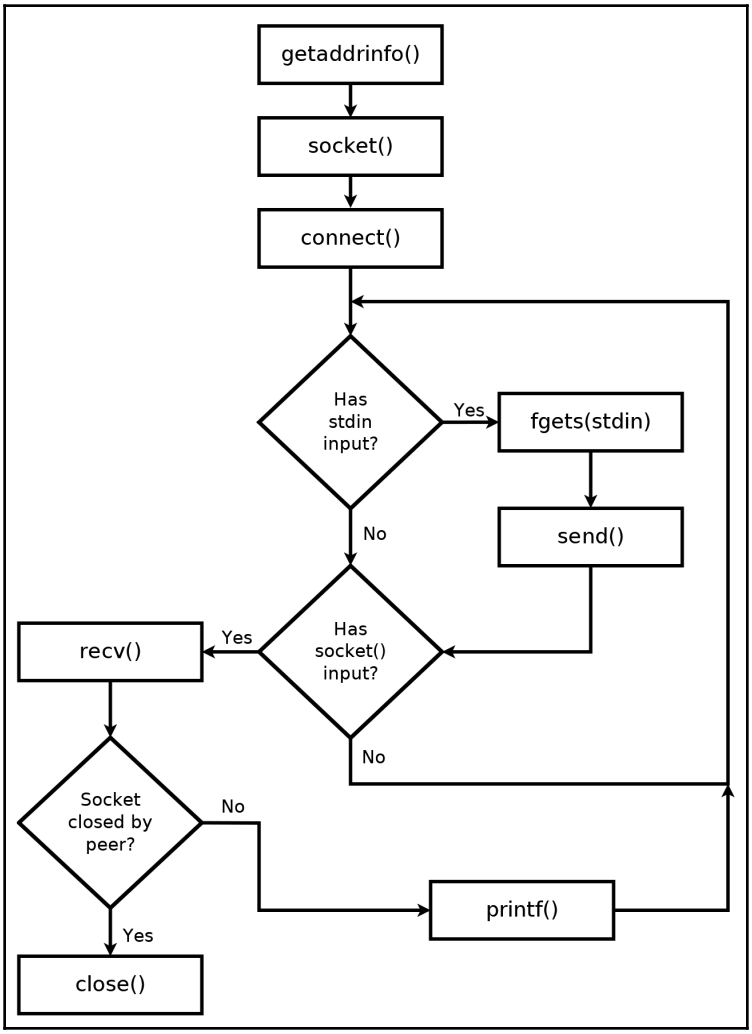
On Unix-based systems, `select()` can also be used on file and terminal I/O, which can be extremely useful. This doesn't work on Windows, though. Windows only supports `select()` for sockets.

A TCP client

It will be useful for us to have a TCP client that can connect to any TCP server. This TCP client will take in a hostname (or IP address) and port number from the command line. It will attempt a connection to the TCP server at that address. If successful, it will relay data that's received from that server to the terminal and data inputted into the terminal to the server. It will continue until either it is terminated (with *Ctrl* + *C*) or the server closes the connection.

This is useful as a learning opportunity to see how to program a TCP client, but it is also useful for testing the TCP server programs we develop throughout this book.

Our basic program flow looks like this:



Our program first uses `getaddrinfo()` to resolve the server address from the command-line arguments. Then, the socket is created with a call to `socket()`. The fresh socket has `connect()` called on it to connect to the server. We use `select()` to monitor for socket input. `select()` also monitors for terminal/keyboard input on non-Windows systems. On Windows, we use the `_kbhit()` function to detect terminal input. If terminal input is available, we send it over the socket using `send()`. If `select()` indicated that socket data is available, we read it with `recv()` and display it to the terminal. This `select()` loop is repeated until the socket is closed.

TCP client code

We begin our TCP client by including the header file, `chap03.h`, which was printed at the beginning of this chapter. This header file includes the various other headers and macros we need for cross-platform networking:

```
/*tcp_client.c*/

#include "chap03.h"
```

On Windows, we also need the `conio.h` header. This is required for the `_kbhit()` function, which helps us by indicating whether terminal input is waiting. We conditionally include this header, like so:

```
/*tcp_client.c*/

#if defined(_WIN32)
#include <conio.h>
#endif
```

We can then begin the `main()` function and initialize Winsock:

```
/*tcp_client.c*/

int main(int argc, char *argv[]) {

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

We would like our program to take the hostname and port number of the server it should connect to as command-line arguments. This makes our program flexible. We have our program check that these command-line arguments are given. If they aren't, it displays usage information:

```
/*tcp_client.c*/

    if (argc < 3) {
        fprintf(stderr, "usage: tcp_client hostname port\n");
        return 1;
    }
```

`argc` contains the number of argument values available to us. Because the first argument is always our program's name, we check that there is a total of at least three arguments. The actual values themselves are stored in `argv[]`.

We then use these values to configure a remote address for connection:

```
/*tcp_client.c*/

    printf("Configuring remote address...\n");
    struct addrinfo hints;
    memset(&hints, 0, sizeof(hints));
    hints.ai_socktype = SOCK_STREAM;
    struct addrinfo *peer_address;
    if (getaddrinfo(argv[1], argv[2], &hints, &peer_address)) {
        fprintf(stderr, "getaddrinfo() failed. (%d)\n", GETSOCKETERRNO());
        return 1;
    }
```

This is similar to how we called `getaddrinfo()` in `Chapter 2`, *Getting to Grips with Socket APIs*. However, in `Chapter 2`, *Getting to Grips with Socket APIs*, we wanted it to configure a local address, whereas this time, we want it to configure a remote address.

We set `hints.ai_socktype = SOCK_STREAM` to tell `getaddrinfo()` that we want a TCP connection. Remember that we could set `SOCK_DGRAM` to indicate a UDP connection.

In `Chapter 2`, *Getting to Grips with Socket APIs*, we also set the family. We don't need to set the family here, as we can let `getaddrinfo()` decide if IPv4 or IPv6 is the proper protocol to use.

For the call to `getaddrinfo()` itself, we pass in the hostname and port as the first two arguments. These are passed directly in from the command line. If they aren't suitable, then `getaddrinfo()` returns non-zero and we print an error message. If everything goes well, then our remote address is in the `peer_address` variable.

`getaddrinfo()` is very flexible about how it takes inputs. The hostname could be a domain name like `example.com` or an IP address such as `192.168.17.23` or `::1`. The port can be a number, such as `80`, or a protocol, such as `http`.

After `getaddrinfo()` configures the remote address, we print it out. This isn't really necessary, but it is a good debugging measure. We use `getnameinfo()` to convert the address back into a string, like this:

```
/*tcp_client.c*/

    printf("Remote address is: ");
    char address_buffer[100];
    char service_buffer[100];
    getnameinfo(peer_address->ai_addr, peer_address->ai_addrlen,
            address_buffer, sizeof(address_buffer),
            service_buffer, sizeof(service_buffer),
            NI_NUMERICHOST);
    printf("%s %s\n", address_buffer, service_buffer);
```

We can then create our socket:

```
/*tcp_client.c*/

    printf("Creating socket...\n");
    SOCKET socket_peer;
    socket_peer = socket(peer_address->ai_family,
            peer_address->ai_socktype, peer_address->ai_protocol);
    if (!ISVALIDSOCKET(socket_peer)) {
        fprintf(stderr, "socket() failed. (%d)\n", GETSOCKETERRNO());
        return 1;
    }
```

This call to `socket()` is done in exactly the same way as it was in `Chapter 2`, *Getting to Grips with Socket APIs*. We use `peer_address` to set the proper socket family and protocols. This keeps our program very flexible, as the `socket()` call creates an IPv4 or IPv6 socket as needed.

After the socket has been created, we call `connect()` to establish a connection to the remote server:

```
/*tcp_client.c */

    printf("Connecting...\n");
    if (connect(socket_peer,
                peer_address->ai_addr, peer_address->ai_addrlen)) {
        fprintf(stderr, "connect() failed. (%d)\n", GETSOCKETERRNO());
```

```
            return 1;
        }
        freeaddrinfo(peer_address);
```

`connect()` takes three arguments—the socket, the remote address, and the remote address length. It returns `0` on success, so we print an error message if it returns non-zero. This call to `connect()` is extremely similar to how we called `bind()` in `Chapter 2`, *Getting to Grips with Socket APIs*. Where `bind()` associates a socket with a local address, `connect()` associates a socket with a remote address and initiates the TCP connection.

After we've called `connect()` with `peer_address`, we use the `freeaddrinfo()` function to free the memory for `peer_address`.

If we've made it this far, then a TCP connection has been established to the remote server. We let the user know by printing a message and instructions on how to send data:

```
/*tcp_client.c */

    printf("Connected.\n");
    printf("To send data, enter text followed by enter.\n");
```

Our program should now loop while checking both the terminal and socket for new data. If new data comes from the terminal, we send it over the socket. If new data is read from the socket, we print it out to the terminal.

It is clear we cannot call `recv()` directly here. If we did, it would block until data comes from the socket. In the meantime, if our user enters data on the terminal, that input is ignored. Instead, we use `select()`. We begin our loop and set up the call to `select()`, like this:

```
/*tcp_client.c */

    while(1) {

        fd_set reads;
        FD_ZERO(&reads);
        FD_SET(socket_peer, &reads);
#if !defined(_WIN32)
        FD_SET(0, &reads);
#endif

        struct timeval timeout;
        timeout.tv_sec = 0;
        timeout.tv_usec = 100000;

        if (select(socket_peer+1, &reads, 0, 0, &timeout) < 0) {
```

```
            fprintf(stderr, "select() failed. (%d)\n", GETSOCKETERRNO());
            return 1;
        }
```

First, we declare a variable, `fd_set reads`, to store our socket set. We then zero it with `FD_ZERO()` and add our only socket, `socket_peer`.

On non-Windows systems, we also use `select()` to monitor for terminal input. We add `stdin` to the `reads` set with `FD_SET(0, &reads)`. This works because `0` is the file descriptor for `stdin`. Alternatively, we could have used `FD_SET(fileno(stdin), &reads)` to the same effect.

The Windows `select()` function only works on sockets. Therefore, we cannot use `select()` to monitor for console input. For this reason, we set up a timeout to the `select()` call for 100 milliseconds (100,000 microseconds). If there is no socket activity after 100 milliseconds, `select()` returns, and we can check for terminal input manually.

After `select()` returns, we check to see whether our socket is set in `reads`. If it is, then we know to call `recv()` to read the new data. The new data is printed to the console with `printf()`:

```
/*tcp_client.c*/

        if (FD_ISSET(socket_peer, &reads)) {
            char read[4096];
            int bytes_received = recv(socket_peer, read, 4096, 0);
            if (bytes_received < 1) {
                printf("Connection closed by peer.\n");
                break;
            }
            printf("Received (%d bytes): %.*s",
                    bytes_received, bytes_received, read);
        }
```

Remember, the data from `recv()` is not null terminated. For this reason, we use the `%.*s` `printf()` format specifier, which prints a string of a specified length.

`recv()` normally returns the number of bytes read. If it returns less than `1`, then the connection has ended, and we break out of the loop to shut it down.

After checking for new TCP data, we also need to check for terminal input:

```
/*tcp_client.c */

#if defined(_WIN32)
        if(_kbhit()) {
```

```
#else
        if(FD_ISSET(0, &reads)) {
#endif
            char read[4096];
            if (!fgets(read, 4096, stdin)) break;
            printf("Sending: %s", read);
            int bytes_sent = send(socket_peer, read, strlen(read), 0);
            printf("Sent %d bytes.\n", bytes_sent);
        }
```

On Windows, we use the `_kbhit()` function to indicate whether any console input is waiting. `_kbhit()` returns non-zero if an unhandled key press event is queued up. For Unix-based systems, we simply check if `select()` sets the `stdin` file descriptor, `0`. If input is ready, we call `fgets()` to read the next line of input. This input is then sent over our connected socket with `send()`.

Note that `fgets()` includes the newline character from the input. Therefore, our sent input always ends with a newline.

If the socket has closed, `send()` returns `-1`. We ignore this case here. This is because a closed socket causes `select()` to return immediately, and we notice the closed socket on the next call to `recv()`. This is a common paradigm in TCP socket programming to ignore errors on `send()` while detecting and handling them on `recv()`. It allows us to simplify our program by keeping our connection closing logic all in one place. Later in this chapter, we will discuss other concerns regarding `send()`.

This `select()` based terminal monitoring works very well on Unix-based systems. It also works equally well if input is piped in. For example, you could use our TCP client program to send a text file with a command such as `cat my_file.txt | tcp_client 192.168.54.122 8080`.

The Windows terminal handling leaves a bit to be desired. Windows does not provide an easy way to tell whether `stdin` has input available without blocking, so we use `_kbhit()` as a poor proxy. However, if the user presses a non-printable key, such as an arrow key, it still triggers `_kbhit()`, even though there is no character to read. Also, after the first key press, our program will block on `fgets()` until the user presses the *Enter* key. (This doesn't happen on shells that buffer entire lines, which is common outside of Windows.) This blocking behavior is acceptable, but you should know that any received TCP data will not display until after that point. `_kbhit()` does not work for piped input. Doing proper piped and console input on Windows is possible, of course, but it's very complicated.

We would need to use separate functions for each (`PeekNamedPipe()` and `PeekConsoleInput()`), and the logic for handling it would be as long as this entire program! Since handling terminal input isn't the purpose of this book, we're going to accept `_kbhit()` function's limitations and move on.

At this point, our program is essentially done. We can end the `while` loop, close our socket, and clean up Winsock:

```
/*tcp_client.c */

    }

    printf("Closing socket...\n");
    CLOSESOCKET(socket_peer);

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

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

That's the complete program. You can compile it on Linux and macOS like this:

```
gcc tcp_client.c -o tcp_client
```

Compiling on Windows with MinGW is done like this:

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

To run the program, remember to pass in the remote hostname/address and port number, for example:

```
tcp_client example.com 80
```

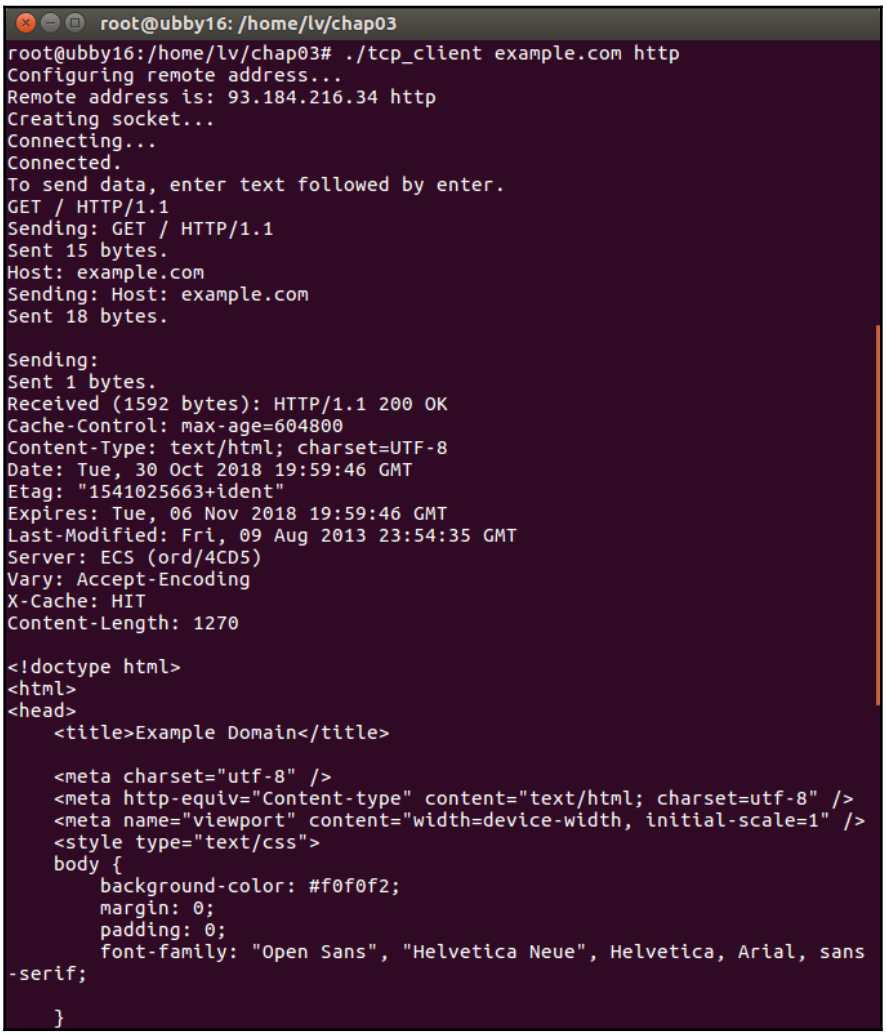
Alternatively, you can use the following command:

```
tcp_client 127.0.0.1 8080
```

A fun way to test out the TCP client would be to connect to a live web server and send an HTTP request. For example, you could connect to `example.com` on port `80` and send the following HTTP request:

```
GET / HTTP/1.1
Host: example.com
```

You must then send a blank line to indicate the end of the request. You'll receive an HTTP response back. It might look something like this:



A TCP server

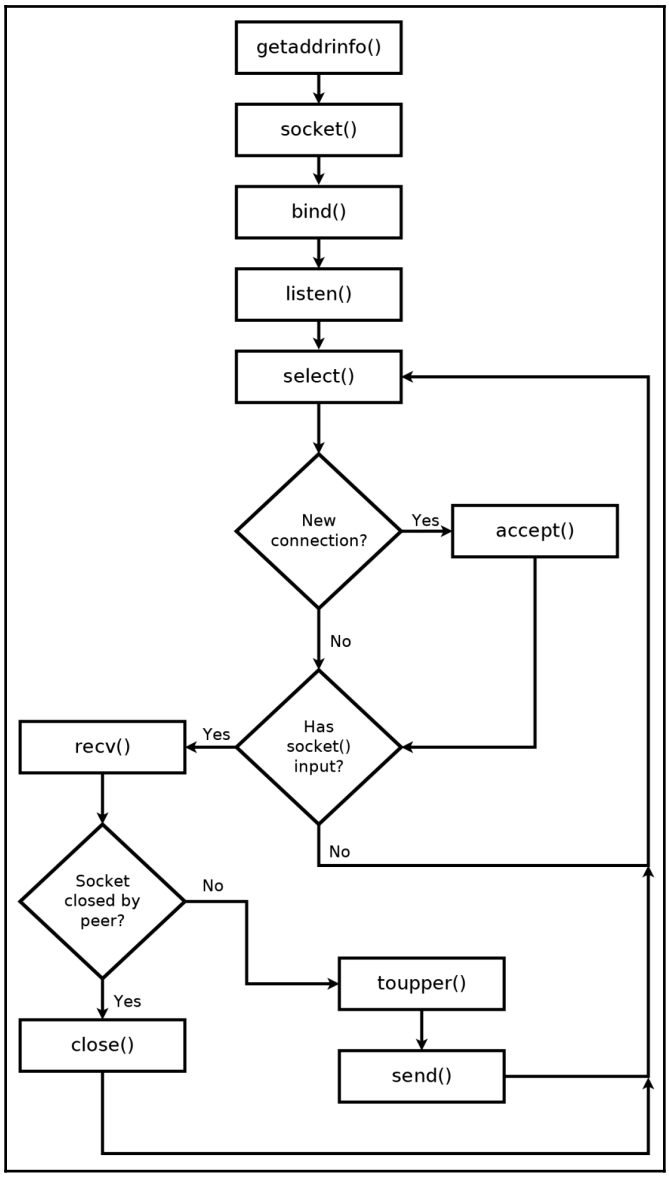
Microservices have become increasingly popular in recent years. The idea of microservices is that large programming problems can be split up into many small subsystems that communicate over a network. For example, if your program needs to format a string, you could add code to your program to do that, but writing code is hard. Alternatively, you could keep your program simple and instead connect to a service that provides string formatting for you. This has the added advantage that many programs can use this same service without reinventing the wheel.

Unfortunately, the microservice paradigm has largely avoided the C ecosystem; until now!

As a motivating example, we are going to build a TCP server that converts strings into uppercase. If a client connects and sends `Hello`, then our program will send `HELLO` back. This will serve as a very basic microservice. Of course, a real-world microservice might do something a bit more advanced (such as left-pad a string), but this to-uppercase service works well for our pedagogical purposes.

For our microservice to be useful, it does need to handle many simultaneous incoming connections. We again use `select()` to see which connections need to be serviced.

Our basic program flow looks like this:



Like in `Chapter 2`, *Getting to Grips with Socket APIs*, our TCP server uses `getaddrinfo()` to obtain the local address to listen on. It creates a socket with `socket()`, uses `bind()` to associate the local address to the socket, and uses `listen()` to begin listening for new connections. Up until that point, it is essentially identical to our TCP server from `Chapter 2`, *Getting to Grips with Socket APIs*.

However, our next step is not to call `accept()` to wait for new connections. Instead, we call `select()`, which alerts us if a new connection is available or if any of our established connections have new data ready. Only when we know that a new connection is waiting do we call `accept()`. All established connections are put into an `fd_set`, which is passed to every subsequent `select()` call. In this same way, we know which connections would block on `recv()`, and we only service those connections that we know will not block.

When data is received by `recv()`, we run it through `toupper()` and return it to the client using `send()`.

This is a complicated program with several new concepts. Don't worry about understanding all the details right now. The flow is only intended to give you an overview of what to expect before we dive into the actual code.

TCP server code

Our TCP server code begins by including the needed headers, starting `main()`, and initializing Winsock. Refer to `Chapter 2`, *Getting to Grips with Socket APIs*, if this doesn't seem familiar:

```
/*tcp_serve_toupper.c*/

#include "chap03.h"
#include <ctype.h>

int main() {

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

We then get our local address, create our socket, and `bind()`. This is all done exactly as explained in `Chapter 2`, *Getting to Grips with Socket APIs*:

```
/*tcp_serve_toupper.c */

    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);


    printf("Creating socket...\n");
    SOCKET socket_listen;
    socket_listen = socket(bind_address->ai_family,
            bind_address->ai_socktype, bind_address->ai_protocol);
    if (!ISVALIDSOCKET(socket_listen)) {
        fprintf(stderr, "socket() failed. (%d)\n", GETSOCKETERRNO());
        return 1;
    }
```

Note that we are going to listen on port `8080`. You can, of course, change that. We're also doing an IPv4 server here. If you want to listen for connections on IPv6, then just change `AF_INET` to `AF_INET6`.

We then `bind()` our socket to the local address and have it enter a listening state. Again, this is done exactly as in `Chapter 2`, *Getting to Grips with Socket APIs*:

```
/*tcp_serve_toupper.c*/

    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);

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

This is the point where we diverge from our earlier methods. We now define an `fd_set` structure that stores all of the active sockets. We also maintain a `max_socket` variable, which holds the largest socket descriptor. For now, we add only our listening socket to the set. Because it's the only socket, it must also be the largest, so we set `max_socket = socket_listen` too:

```
/*tcp_serve_toupper.c */

    fd_set master;
    FD_ZERO(&master);
    FD_SET(socket_listen, &master);
    SOCKET max_socket = socket_listen;
```

Later in the program, we will add new connections to `master` as they are established.

We then print a status message, enter the main loop, and set up our call to `select()`:

```
/*tcp_serve_toupper.c */

    printf("Waiting for connections...\n");

    while(1) {
        fd_set reads;
        reads = master;
        if (select(max_socket+1, &reads, 0, 0, 0) < 0) {
            fprintf(stderr, "select() failed. (%d)\n", GETSOCKETERRNO());
            return 1;
        }
```

This works by first copying our `fd_set master` into `reads`. Recall that `select()` modifies the set given to it. If we didn't copy `master`, we would lose its data.

We pass a timeout value of `0` (NULL) to `select()` so that it doesn't return until a socket in the `master` set is ready to be read from. At the beginning of our program, `master` only contains `socket_listen`, but as our program runs, we add each new connection to `master`.

We now loop through each possible socket and see whether it was flagged by `select()` as being ready. If a socket, `X`, was flagged by `select()`, then `FD_ISSET(X, &reads)` is true. Socket descriptors are positive integers, so we can try every possible socket descriptor up to `max_socket`. The basic structure of our loop is as follows:

```
/*tcp_serve_toupper.c */

        SOCKET i;
        for(i = 1; i <= max_socket; ++i) {
```

```
            if (FD_ISSET(i, &reads)) {
                //Handle socket
            }
        }
```

Remember, `FD_ISSET()` is only true for sockets that are ready to be read. In the case of `socket_listen`, this means that a new connection is ready to be established with `accept()`. For all other sockets, it means that data is ready to be read with `recv()`. We should first determine whether the current socket is the listening one or not. If it is, we call `accept()`. This code snippet and the one that follows replace the `//Handle socket` comment in the preceding code:

```
/*tcp_serve_toupper.c */

                if (i == socket_listen) {
                    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;
                    }

                    FD_SET(socket_client, &master);
                    if (socket_client > max_socket)
                        max_socket = socket_client;

                    char address_buffer[100];
                    getnameinfo((struct sockaddr*)&client_address,
                            client_len,
                            address_buffer, sizeof(address_buffer), 0, 0,
                            NI_NUMERICHOST);
                    printf("New connection from %s\n", address_buffer);
```

If the socket is `socket_listen`, then we `accept()` the connection much as we did in `Chapter 2`, *Getting to Grips with Socket APIs*. We use `FD_SET()` to add the new connection's socket to the `master` socket set. This allows us to monitor it with subsequent calls to `select()`. We also maintain `max_socket`. As a final step, this code prints out the client's address using `getnameinfo()`.

If the socket `i` is not `socket_listen`, then it is instead a request for an established connection. In this case, we need to read it with `recv()`, convert it into uppercase using the built-in `toupper()` function, and send the data back:

```
/*tcp_serve_toupper.c */

                } else {
                    char read[1024];
                    int bytes_received = recv(i, read, 1024, 0);
                    if (bytes_received < 1) {
                        FD_CLR(i, &master);
                        CLOSESOCKET(i);
                        continue;
                    }

                    int j;
                    for (j = 0; j < bytes_received; ++j)
                        read[j] = toupper(read[j]);
                    send(i, read, bytes_received, 0);
                }
```

If the client has disconnected, then `recv()` returns a non-positive number. In this case, we remove that socket from the `master` socket set, and we also call `CLOSESOCKET()` on it to clean up.

Our program is now almost finished. We can end the `if FD_ISSET()` statement, end the `for` loop, end the `while` loop, close the listening socket, and clean up Winsock:

```
/*tcp_serve_toupper.c */

            } //if FD_ISSET
        } //for i to max_socket
    } //while(1)

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

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

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

Our program is set up to continuously listen for connections, so the code after the end of the `while` loop will never run. Nevertheless, I believe it is still good practice to include it in case we program in functionality later to abort the `while` loop.

That's the complete to-uppercase microservice TCP server program. You can compile and run it on Linux and macOS like this:

```
gcc tcp_serve_toupper.c -o tcp_serve_toupper
./tcp_serve_toupper
```

Compiling and running on Windows with MinGW is done like this:

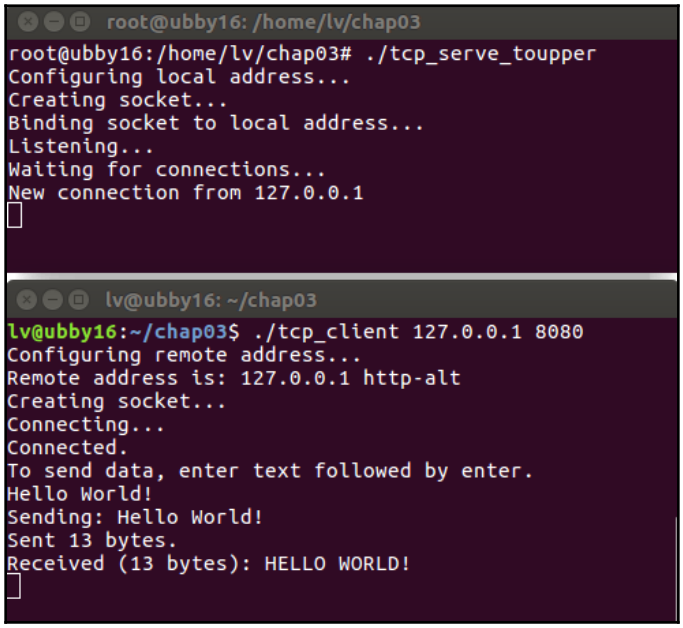
```
gcc tcp_serve_toupper.c -o tcp_serve_toupper.exe -lws2_32
tcp_serve_toupper.exe
```

You can abort the program's execution with *Ctrl* + *C*.

Once the program is running, I would suggest opening another terminal and running the `tcp_client` program from earlier to connect to it:

```
tcp_client 127.0.0.1 8080
```

Anything you type in `tcp_client` should be sent back as uppercase. Here's what this might look like:



As a test of the server program's functionality, try opening several additional terminals and connecting with `tcp_client`. Our server should be able to handle many simultaneous connections.

Also included with this chapter's code is `tcp_serve_toupper_fork.c`. This program only runs on Unix-based operating systems, but it performs the same functions as `tcp_serve_toupper.c` by using `fork()` instead of `select()`. The `fork()` function is commonly used by TCP servers, so I think it's helpful to be familiar with it.

Building a chat room

It is also possible, and common, to need to send data between connected clients. We can modify our `tcp_serve_toupper.c` program and make it a chat room pretty easily.

First, locate the following code in `tcp_serve_toupper.c`:

```
/*tcp_serve_toupper.c excerpt*/

                    int j;
                    for (j = 0; j < bytes_received; ++j)
                        read[j] = toupper(read[j]);
                    send(i, read, bytes_received, 0);
```

Replace the preceding code with the following:

```
/*tcp_serve_chat.c excerpt*/

                    SOCKET j;
                    for (j = 1; j <= max_socket; ++j) {
                        if (FD_ISSET(j, &master)) {
                            if (j == socket_listen || j == i)
                                continue;
                            else
                                send(j, read, bytes_received, 0);
                        }
                    }
```

This works by looping through all the sockets in the `master` set. For each socket, `j`, we check that it's not the listening socket and we check that it's not the same socket that sent the data in the first place. If it's not, we call `send()` to echo the received data to it.

You can compile and run this program in the same way as the previous one.

On Linux and macOS, this is done as follows:

```
gcc tcp_serve_chat.c -o tcp_serve_chat
./tcp_serve_chat
```

On Windows, this is done as follows:

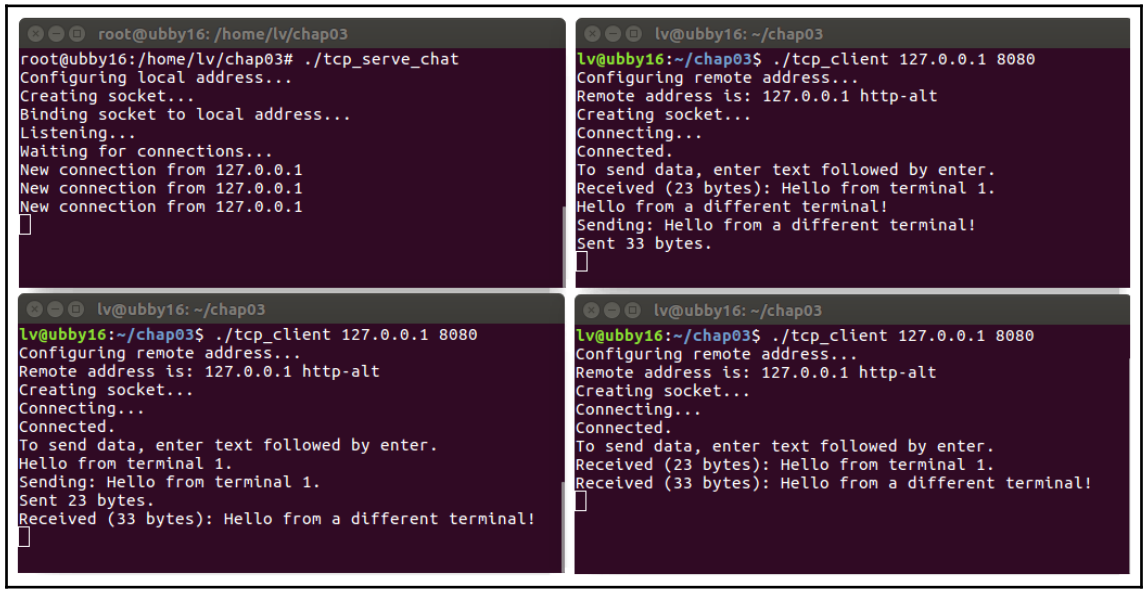
```
gcc tcp_serve_chat.c -o tcp_serve_chat.exe -lws2_32
tcp_serve_chat.exe
```

You should open two or more additional windows and connect to it with the following code:

```
tcp_client 127.0.0.1 8080
```

Whatever you type in one of the `tcp_client` terminals get sent to all of the other connected terminals.

Here is an example of what this may look like:



In the preceding screenshot, I am running `tcp_serve_chat` in the upper-left terminal windows. The other three terminal windows are running `tcp_client`. As you can see, any text entered in one of the `tcp_client` windows is sent to the server, which relays it to the other two connected clients.

Blocking on send()

When we call `send()` with an amount of data, `send()` first copies this data into an outgoing buffer provided by the operating system. If we call `send()` when its outgoing buffer is already full, it blocks until its buffer has emptied enough to accept more of our data.

In some cases where `send()` would block, it instead returns without copying all of the data as requested. In this case, the return value of `send()` indicates how many bytes were actually copied. One example of this is if your program is blocking on `send()` and then receives a signal from the operating system. In these cases, it is up to the caller to try again with any remaining data.

In this chapter's *TCP server code* section, we ignored the possibility that `send()` could block or be interrupted. In a fully robust application, what we need to do is compare the return value from `send()` with the number of bytes that we tried to send. If the number of bytes actually sent is less than requested, we should use `select()` to determine when the socket is ready to accept new data, and then call `send()` with the remaining data. As you can imagine, this can become a bit complicated when keeping track of multiple sockets.

As the operating system usually provides a large enough outgoing buffer, we were able to avoid this possibility with our earlier server code. If we know that our server may try to send large amounts of data, we should certainly check for the return value from `send()`.

The following code example assumes that `buffer` contains `buffer_len` bytes of data to send over a socket called `peer_socket`. This code blocks until we've sent all of `buffer` or an error (such as the peer disconnecting) occurs:

```
int begin = 0;
while (begin < buffer_len) {
    int sent = send(peer_socket, buffer + begin, buffer_len - begin, 0);
    if (sent == -1) {
        //Handle error
    }
    begin += sent;
}
```

If we are managing multiple sockets and don't want to block, then we should put all sockets with pending `send()` into an `fd_set` and pass it as the third parameter to `select()`. When `select()` signals on these sockets, then we know that they are ready to send more data.

`Chapter 13`, *Socket Programming Tips and Pitfalls*, addresses concerns regarding the `send()` function's blocking behavior in more detail.

TCP is a stream protocol

A common mistake beginners make is assuming that any data passed into `send()` can be read by `recv()` on the other end in the same amount. In reality, sending data is similar to writing and reading from a file. If we write 10 bytes to a file, followed by another 10 bytes, then the file has 20 bytes of data. If the file is to be read later, we could read 5 bytes and 15 bytes, or we could read all 20 bytes at once, and so on. In any case, we have no way of knowing that the file was written in two 10 byte chunks.

Using `send()` and `recv()` works the same way. If you `send()` 20 bytes, it's not possible to tell how many `recv()` calls these bytes are partitioned into. It is possible that one call to `recv()` could return all 20 bytes, but it is also possible that a first call to `recv()` returns 16 bytes and that a second call to `recv()` is needed to get the last 4 bytes.

This can make communication difficult. In many protocols, as we will see later in this book, it is important that received data be buffered up until enough of it has accumulated to warrant processing. We avoided this issue in this chapter with our to-uppercase sever by defining a protocol that operates just as well on 1 byte as it does on 100. This isn't true for most application protocols.

For a concrete example, imagine we wanted to make our `tcp_serve_toupper` server terminate if it received the `quit` command through a TCP socket. You could call `send(socket, "quit", 4, 0)` on the client and you may think that a call to `recv()` on the server would return `quit`. Indeed, in your testing, it is very likely to work that way. However, this behavior is not guaranteed. A call to `recv()` could just as likely return `qui`, and a second call to `recv()` may be required to receive the last `t`. If that is the case, consider how you would interpret whether a `quit` command has been received. The straightforward way to do it would be to buffer up data that's received from multiple `recv()` calls.

We will cover techniques for dealing with `recv()` buffering in `Section 2`, *An Overview of Application Layer Protocols*, of this book.

In contrast to TCP, UDP is not a stream protocol. With UDP, a packet is received with exactly the same contents as it was sent with. This can sometimes make handling UDP somewhat easier, as we will see in `Chapter 4`, *Establishing UDP Connections*.

Summary

TCP really serves as the backbone of the modern internet experience. TCP is used by HTTP, the protocol that powers websites, and by **Simple Mail Transfer Protocol** (**SMTP**), the protocol that powers email.

In this chapter, we saw that building a TCP client was fairly straightforward. The only really tricky part was having the client monitor for local terminal input while simultaneously monitoring for socket data. We were able to accomplish this with `select()` on Unix-based systems, but it was slightly trickier on Windows. Many real-world applications don't need to monitor terminal input, and so this step isn't always needed.

Building a TCP server that's suitable for many parallel connections wasn't much harder. Here, `select()` was extremely useful, as it allowed a straightforward way of monitoring the listening socket for new connections while also monitoring existing connections for new data.

We also touched briefly on some common pain points. TCP doesn't provide a native way to partition data. For more complicated protocols where this is needed, we have to buffer data from `recv()` until a suitable amount is available to interpret. For TCP peers that are handling large amounts of data, buffering to `send()` is also necessary.

The next chapter, `Chapter 4`, *Establishing UDP Connections,* is all about UDP, the counterpart to TCP. In some ways, UDP programming is simpler than TCP programming, but it is also very different.

Questions

Try answering these questions to test your knowledge on this chapter:

1. How can we tell whether the next call to `recv()` will block?
2. How can you ensure that `select()` doesn't block for longer than a specified time?
3. When we used our `tcp_client` program to connect to a web server, why did we need to send a blank line before the web server responded?
4. Does `send()` ever block?
5. How can we tell whether the socket has been disconnected by our peer?
6. Is data received by `recv()` always the same size as data sent with `send()`?
7. Consider the following code:

```
recv(socket_peer, buffer, 4096, 0);
printf(buffer);
```

   What is wrong with it?
   Also see what is wrong with this code:

```
recv(socket_peer, buffer, 4096, 0);
printf("%s", buffer);
```

The answers can be found in `Appendix A`, *Answers to Questions*.