

# 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()`

# 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*/

#ifdef _WIN32
#include <winsock2.h>
#else
#include <unistd.h>
#endif
```

```
#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.

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/writable/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_ISSET(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

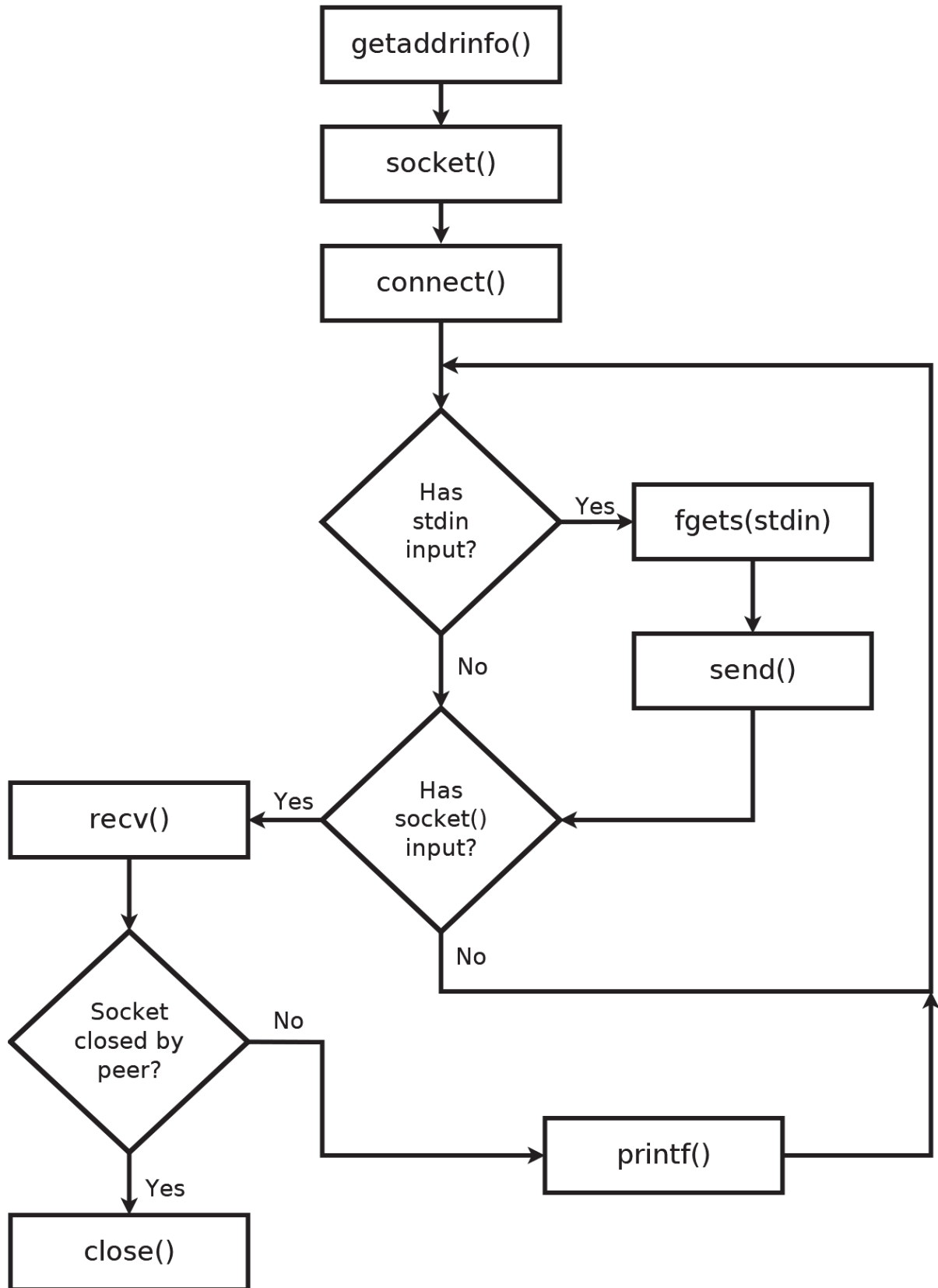
On Unix-based systems, `select()` can also be used on file and terminal I/O, which can be extremely useful. This doesn't work on Windows, though. Windows only supports `select()` for sockets.

# 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);
#ifdef _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 */

#ifdef _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);

#ifdef _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:



```
root@ubby16: /home/lv/chap03  
root@ubby16:/home/lv/chap03# ./tcp_client example.com http  
Configuring remote address...  
Remote address is: 93.184.216.34 http  
Creating socket...  
Connecting...  
Connected.  
To send data, enter text followed by enter.  
GET / HTTP/1.1  
Sending: GET / HTTP/1.1  
Sent 15 bytes.  
Host: example.com  
Sending: Host: example.com  
Sent 18 bytes.  
  
Sending:  
Sent 1 bytes.  
Received (1592 bytes): HTTP/1.1 200 OK  
Cache-Control: max-age=604800  
Content-Type: text/html; charset=UTF-8  
Date: Tue, 30 Oct 2018 19:59:46 GMT  
Etag: "1541025663+ident"  
Expires: Tue, 06 Nov 2018 19:59:46 GMT  
Last-Modified: Fri, 09 Aug 2013 23:54:35 GMT  
Server: ECS (ord/4CD5)  
Vary: Accept-Encoding  
X-Cache: HIT  
Content-Length: 1270  
  
<!doctype html>  
<html>  
<head>  
    <title>Example Domain</title>  
  
    <meta charset="utf-8" />  
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />  
    <meta name="viewport" content="width=device-width, initial-scale=1" />  
    <style type="text/css">  
body {  
    background-color: #f0f0f2;  
    margin: 0;  
    padding: 0;  
    font-family: "Open Sans", "Helvetica Neue", Helvetica, Arial, sans  
-serif;  
}
```

# A TCP server

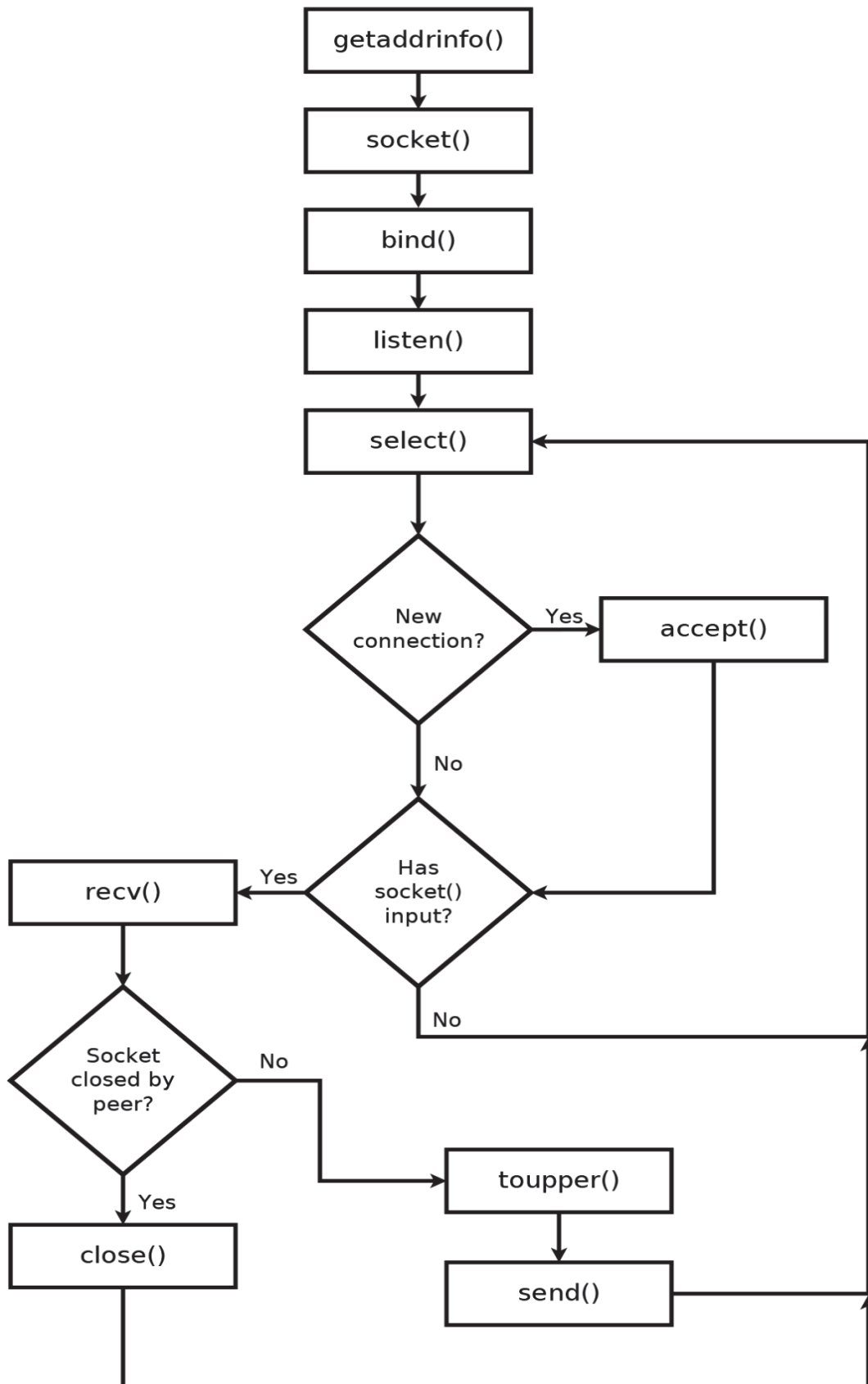
Microservices have become increasingly popular in recent years. The idea of microservices is that large programming problems can be split up into many small subsystems that communicate over a network. For example, if your program needs to format a string, you could add code to your program to do that, 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 (!ISVALID_SOCKET(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 (!ISVALIDDSOCKET(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);

#ifdef _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:

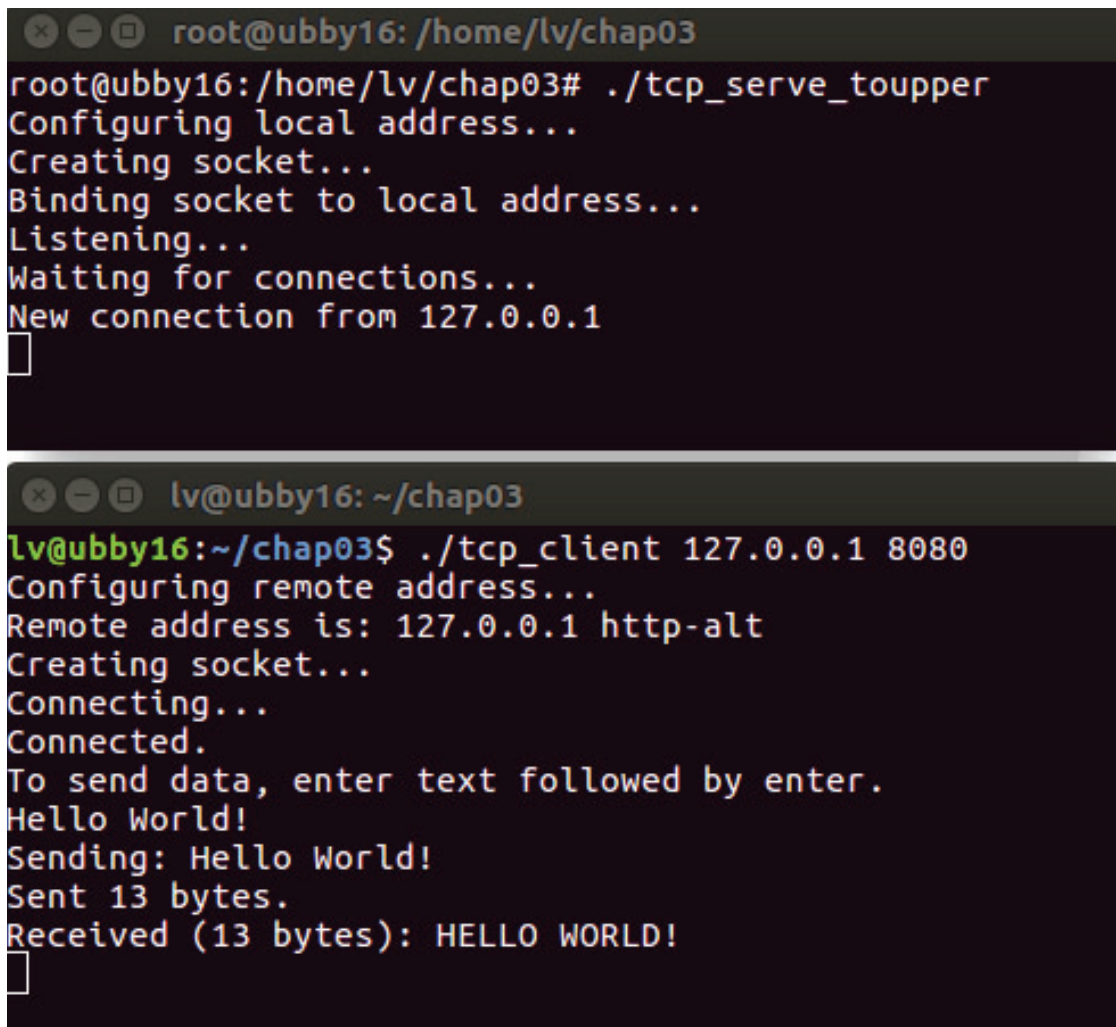
```
gcc tcp_serve_toupper.c -o tcp_serve_toupper.exe -lws2_32
tcp_serve_toupper.exe
```

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:



```
root@ubby16: /home/lv/chap03
root@ubby16:/home/lv/chap03# ./tcp_serve_toupper
Configuring local address...
Creating socket...
Binding socket to local address...
Listening...
Waiting for connections...
New connection from 127.0.0.1
█

lv@ubby16: ~/chap03
lv@ubby16:~/chap03$ ./tcp_client 127.0.0.1 8080
Configuring remote address...
Remote address is: 127.0.0.1 http-alt
Creating socket...
Connecting...
Connected.
To send data, enter text followed by enter.
Hello World!
Sending: Hello World!
Sent 13 bytes.
Received (13 bytes): HELLO WORLD!
█
```

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:

```
root@ubby16: /home/lv/chap03
root@ubby16:/home/lv/chap03# ./tcp_serve_chat
Configuring local address...
Creating socket...
Binding socket to local address...
Listening...
Waiting for connections...
New connection from 127.0.0.1
New connection from 127.0.0.1
New connection from 127.0.0.1
lv@ubby16: ~/chap03
lv@ubby16:~/chap03$ ./tcp_client 127.0.0.1 8080
Configuring remote address...
Remote address is: 127.0.0.1 http-alt
Creating socket...
Connecting...
Connected.
To send data, enter text followed by enter.
Received (23 bytes): Hello from terminal 1.
Hello from a different terminal!
Sending: Hello from a different terminal!
Sent 33 bytes.
lv@ubby16: ~/chap03
lv@ubby16:~/chap03$ ./tcp_client 127.0.0.1 8080
Configuring remote address...
Remote address is: 127.0.0.1 http-alt
Creating socket...
Connecting...
Connected.
To send data, enter text followed by enter.
Hello from terminal 1.
Sending: Hello from terminal 1.
Sent 23 bytes.
Received (33 bytes): Hello from a different terminal!
lv@ubby16: ~/chap03
lv@ubby16:~/chap03$ ./tcp_client 127.0.0.1 8080
Configuring remote address...
Remote address is: 127.0.0.1 http-alt
Creating socket...
Connecting...
Connected.
To send data, enter text followed by enter.
Received (23 bytes): Hello from terminal 1.
Received (33 bytes): Hello from a different terminal!
```

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*.