Establishing UDP Connections

In this chapter, we will look at how to send and receive **User Datagram Protocol** (**UDP**) packets. UDP socket programming is very similar to **Transmission Control Protocol** (**TCP**) socket programming, so it is recommended that you read and understand `Chapter 3`, *An In-Depth Overview of TCP Connections*, before beginning this one.

The following topics are covered in this chapter:

- The differences between TCP socket programming and UDP socket programming
- The `sendto()` and `recvfrom()` functions
- How `connect()` works on a UDP socket
- Implementing a UDP server using only one socket
- Using `select()` to tell when a UDP socket has data ready

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 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/chap04
```

Each example program in this chapter runs on Windows, Linux, and macOS. While compiling on Windows, each example program requires being linked 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 they are 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, `chap04.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 content of `chap04.h` is shown in the following code:

```
#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>
```

How UDP sockets differ

The socket API for UDP sockets is only very slightly different than what we've already learned for TCP. In fact, they are similar enough that we can take the TCP client from the last chapter and turn it into a fully functional UDP client by changing only one line of code:

1. Take `tcp_client.c` from `Chapter 3`, *An In-Depth Overview of TCP Connections*, and find the following line of code:

```
            hints.ai_socktype = SOCK_STREAM;
```

2. Change the preceding code to the following:

```
            hints.ai_socktype = SOCK_DGRAM;
```

This modification is included in this chapter's code as `udp_client.c`.

You can recompile the program using the same commands as before, and you'll get a fully functional UDP client.

Unfortunately, changing the TCP servers of the previous chapters to UDP won't be as easy. TCP and UDP server code are different enough that a slightly different approach is needed.

Also, don't assume that because we had to change only one line of the code that the client behaves exactly the same way – this won't happen. The two programs are using a different protocol, after all.

Remember from `Chapter 2`, *Getting to Grips with Socket APIs* that UDP does not try to be a reliable protocol. Lost packets are not automatically re-transmitted, and packets may be received in a different order than they were sent. It is even possible for one packet to erroneously arrive twice! TCP attempts to solve all these problems, but UDP leaves you to your own devices.

Do you know what the best thing about a UDP joke is? I don't care if you get it or not.

Despite UDP's (lack of) reliability, it is still appropriate for many applications. Let's look at the methods that are used by UDP clients and servers.

UDP client methods

Sending data with TCP requires calling `connect()` to set the remote address and establish the TCP connection. Thus, we use `send()` with TCP sockets, as shown in the following code:

```
connect(tcp_socket, peer_address, peer_address_length);
send(tcp_socket, data, data_length, 0);
```

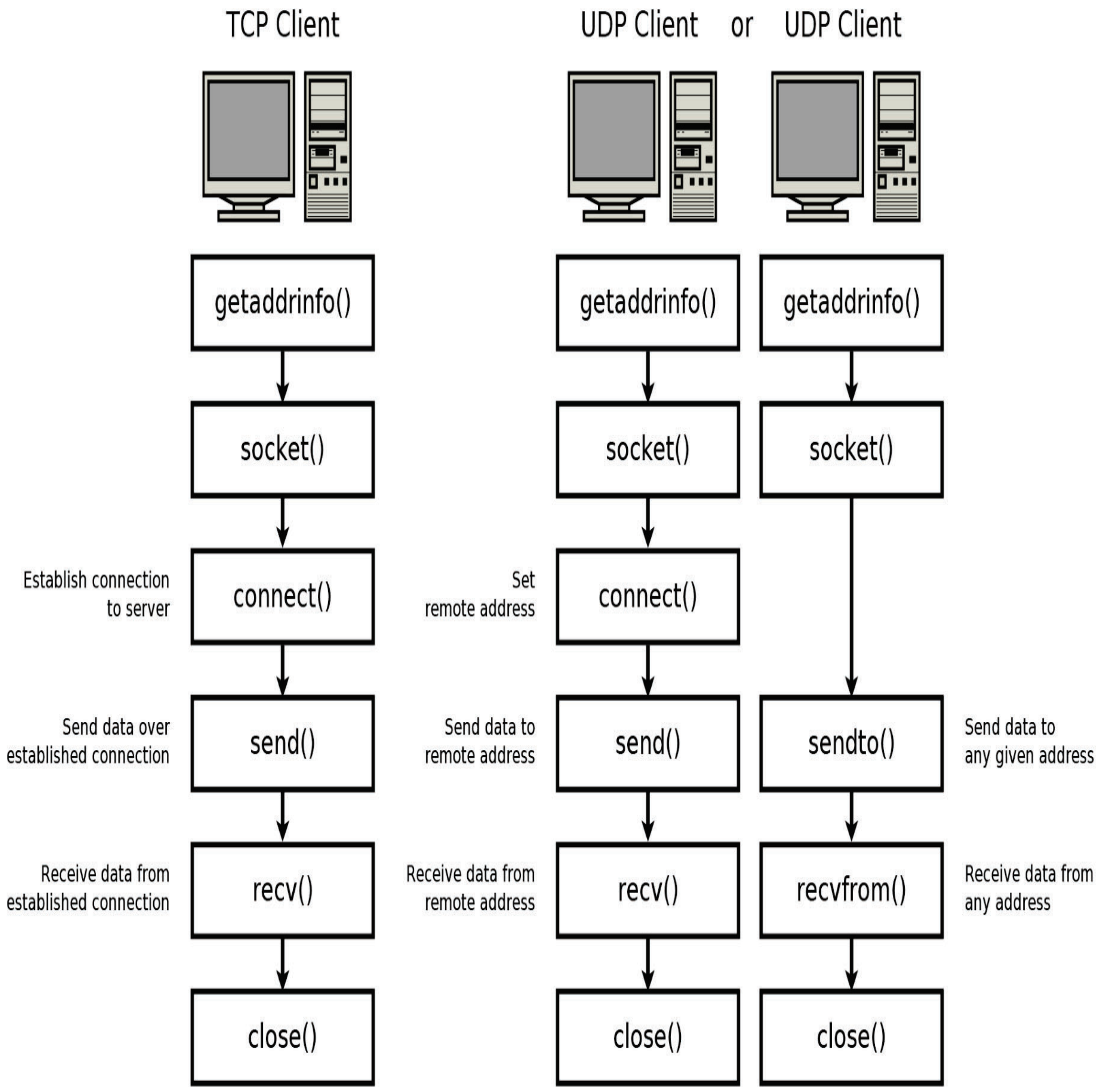
UDP is a connectionless protocol. Therefore, no connection is established before sending data. A UDP connection is never established. With UDP, data is simply sent and received. We can call `connect()` and then `send()`, as we mentioned previously, but the socket API provides an easier way for UDP sockets in the form of the `sendto()` function. It works like this:

```
sendto(udp_socket, data, data_length, 0,
        peer_address, peer_address_length);
```

`connect()` on a UDP socket works a bit differently. All `connect()` does with a UDP socket is associate a remote address. Thus, while `connect()` on a TCP socket involves a handshake for sending packets over the network, `connect()` on a UDP socket only stores an address locally.

So, a UDP client can be structured in two different ways, depending on whether you use `connect()`, `send()`, and `recv()`, or instead use `sendto()` and `recvfrom()`.

The following diagram compares the program flow of a **TCP Client** to a **UDP Client** using either method:


TCP Client
UDP Client
or
UDP Client
getaddrinfo()
getaddrinfo()
getaddrinfo()
socket()
socket()
socket()
Establish connection to server
connect()
Set remote address
connect()
Send data over established connection
send()
Send data to remote address
send()
sendto()
Send data to any given address
Receive data from established connection
recv()
Receive data from remote address
recv()
recvfrom()
Receive data from any address
close()
close()
close()

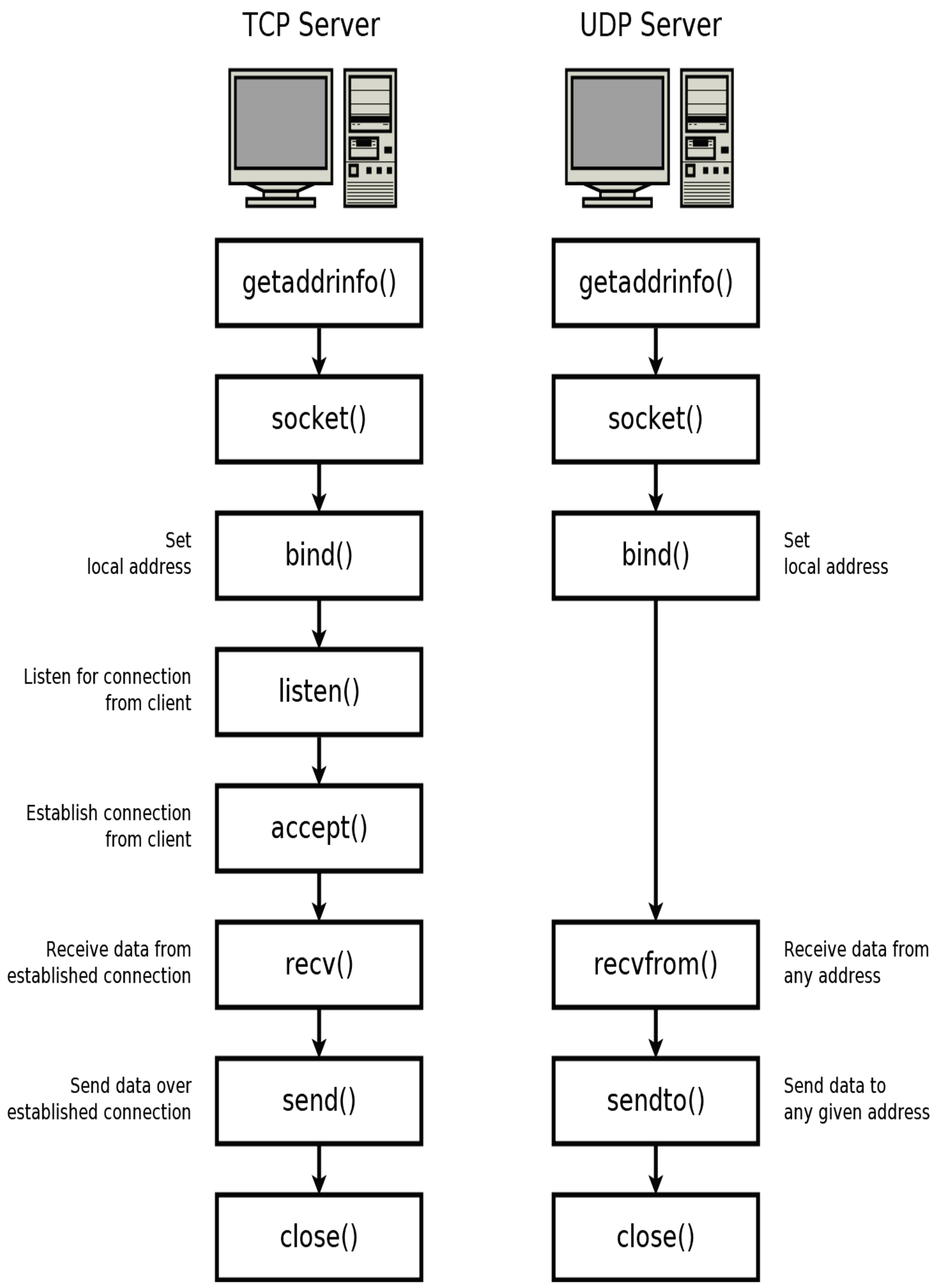
Note that, while using `connect()`, the **UDP Client** only receives data from the peer having the IP address and the port that is given to `connect()`. However, when not using `connect()`, the `recvfrom()` function returns data from any peer that addresses us! Of course, that peer would need to know our address and port. Unless we call `bind()`, our local address and port is assigned automatically by the operating system.

UDP server methods

Programming a UDP server is a bit different than TCP. TCP requires managing a socket for each peer connection. With UDP, our program only needs one socket. That one socket can communicate with any number of peers.

While the TCP program flow required us to use `listen()` and `accept()` to wait for and establish new connections, these functions are not used with UDP. Our UDP server simply binds to the local address, and then it can immediately start sending and receiving data.

The program flow of a **TCP Server** compared to a **UDP Server** is as follows:


TCP Server
UDP Server
getaddrinfo()
socket()
bind()
Set local address
listen()
Listen for connection from client
accept()
Establish connection from client
recv()
Receive data from established connection
send()
Send data over established connection
close()
getaddrinfo()
socket()
bind()
Set local address
recvfrom()
Receive data from any address
sendto()
Send data to any given address
close()

With either a TCP or UDP server, we use `select()` when we need to check/wait for incoming data. The difference is that a **TCP Server** using `select()` is likely monitoring many separate sockets, while a **UDP Server** often only needs to monitor one socket. If your program uses both TCP and UDP sockets, you can monitor them all with only one call to `select()`.

A first UDP client/server

To drive these points home, it will be useful to work through a full UDP client and UDP server program.

To keep things simple, we will create a UDP client program that simply sends the `Hello World` string to `127.0.0.1` on port `8080`. Our UDP server listens on `8080`. It prints any data it receives, along with the sender's address and port number.

We will begin by implementing the simple UDP server.

A simple UDP server

We will start with the server, since we already have a usable UDP client, that is, `udp_client.c`.

Like all of our networked programs, we will begin by including the necessary headers, starting with the `main()` function, and initializing Winsock as follows:

```
/*udp_recvfrom.c*/

#include "chap04.h"

int main() {

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

If you've been working through this book in order, this code should be very routine for you by now. If you haven't, then please refer to `Chapter 2`, *Getting to Grips with Socket APIs*.

Then, we must configure the local address that our server listens on. We use `getaddrinfo()` for this, as follows:

```
/*udp_recvfrom.c continued*/

    printf("Configuring local address...\n");
    struct addrinfo hints;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_DGRAM;
    hints.ai_flags = AI_PASSIVE;

    struct addrinfo *bind_address;
    getaddrinfo(0, "8080", &hints, &bind_address);
```

This differs only slightly from how we've done it before. Notably, we set `hints.ai_socktype = SOCK_DGRAM`. Recall that `SOCK_STREAM` was used there for TCP connections. We are still setting `hints.ai_family = AF_INET` here. This makes our server listen for IPv4 connections. We could change that to `AF_INET6` to make our server listen for IPv6 connections instead.

After we have our local address information, we can create the socket, as follows:

```
/*udp_recvfrom.c continued*/

    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;
    }
```

This code is exactly the same as in the TCP case. The call to `socket()` uses our address information from `getaddrinfo()` to create the proper type of socket.

We must then bind the new socket to the local address that we got from `getaddrinfo()`. This is as follows:

```
/*udp_recvfrom.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);
```

Again, that code is exactly the same as in the TCP case.

Here is where the UDP server diverges from the TCP server. Once the local address is bound, we can simply start to receive data. There is no need to call `listen()` or `accept()`. We listen for incoming data using `recvfrom()`, as shown here:

```
/*udp_recvfrom.c continued*/

    struct sockaddr_storage client_address;
    socklen_t client_len = sizeof(client_address);
    char read[1024];
    int bytes_received = recvfrom(socket_listen,
            read, 1024,
            0,
            (struct sockaddr*) &client_address, &client_len);
```

In the previous code, we created a `struct sockaddr_storage` to store the client's address. We also defined `socklen_t client_len` to hold the address size. This keeps our code robust in the case that we change it from IPv4 to IPv6. Finally, we created a buffer, `char read[1024]`, to store incoming data.

`recvfrom()` is used in a similar manner to `recv()`, except that it returns the sender's address, as well as the received data. You can think of `recvfrom()` as a combination of the TCP server `accept()` and `recv()`.

Once we've received data, we can print it out. Keep in mind that the data may not be null terminated. It can be safely printed with the `%.*s` `printf()` format specifier, as shown in the following code:

```
/*udp_recvfrom.c continued*/

    printf("Received (%d bytes): %.*s\n",
            bytes_received, bytes_received, read);
```

It may also be useful to print the sender's address and port number. We can use the `getnameinfo()` function to convert this data into a printable string, as shown in the following code:

```
/*udp_recvfrom.c continued*/

    printf("Remote address is: ");
    char address_buffer[100];
    char service_buffer[100];
    getnameinfo(((struct sockaddr*)&client_address),
            client_len,
            address_buffer, sizeof(address_buffer),
            service_buffer, sizeof(service_buffer),
            NI_NUMERICHOST | NI_NUMERICSERV);
    printf("%s %s\n", address_buffer, service_buffer);
```

The last argument to `getnameinfo()` (`NI_NUMERICHOST | NI_NUMERICSERV`) tells `getnameinfo()` that we want both the client address and port number to be in numeric form. Without this, it would attempt to return a hostname or protocol name if the port number matches a known protocol. If you do want a protocol name, pass in the `NI_DGRAM` flag to tell `getnameinfo()` that you're working on a UDP port. This is important for the few protocols that have different ports for TCP and UDP.

It's also worth noting that the client will rarely set its local port number explicitly. So, the port number returned here by `getnameinfo()` is likely to be a high number that's chosen randomly by the client's operating system. Even if the client did set its local port number, the port number we can see here might have been changed by **network address translation** (**NAT**).

In any case, if our server were to send data back, it would need to send it to the address and port stored in `client_address`. This would be done by passing `client_address` to `sendto()`.

Once the data has been received, we'll end our simple UDP server by closing the connection, cleaning up Winsock, and ending the program:

```
/*udp_recvfrom.c continued*/

    CLOSESOCKET(socket_listen);

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

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

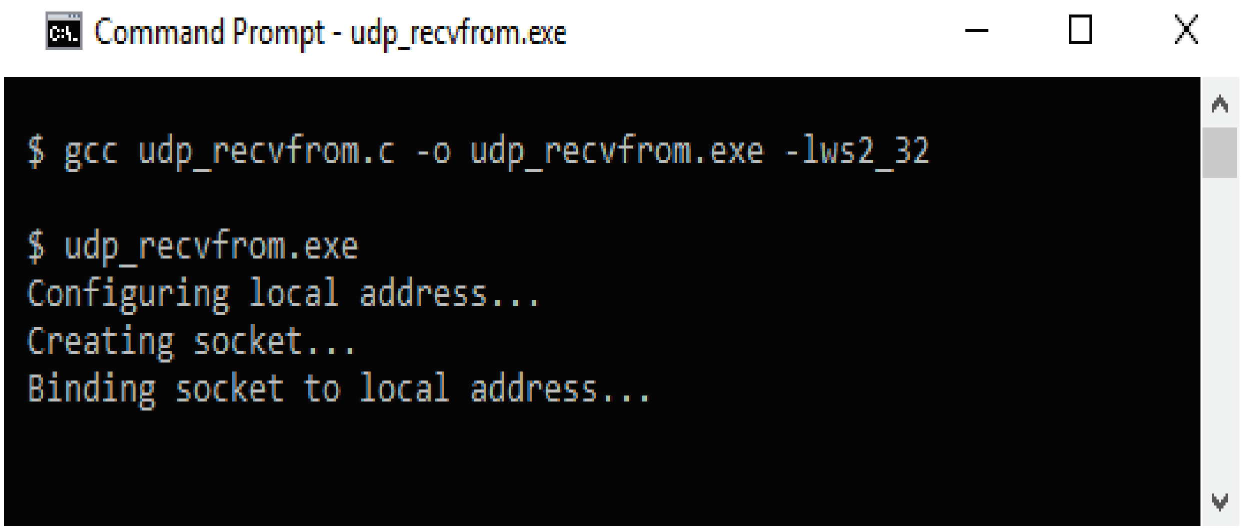
You can compile and run `udp_recvfrom.c` on Linux and macOS by using the following commands:

```
gcc udp_recvfrom.c -o udp_recvfrom
./udp_recvfrom
```

Compiling and running on Windows with MinGW is done as follows:

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

While running, it simply waits for an incoming connection:



```
$ gcc udp_recvfrom.c -o udp_recvfrom.exe -lws2_32

$ udp_recvfrom.exe
Configuring local address...
Creating socket...
Binding socket to local address...
```

You could use `udp_client` to connect to `udp_recvfrom` for testing, or you can implement `udp_sendto`, which we will do next.

A simple UDP client

Although we've already shown a fairly full-featured UDP client, `udp_client.c`, it is worthwhile building a very simple UDP client. This client shows only the minimal required steps to get a working UDP client, and it uses `sendto()` instead of `send()`.

Let's begin the same way we begin each program, by including the necessary headers, starting `main()`, and initializing Winsock, as follows:

```
/*udp_sendto.c*/

#include "chap04.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 configure the remote address using `getaddrinfo()`. In this minimal example, we use `127.0.0.1` as the remote address and `8080` as the remote port. This means that it connects to the UDP server only if it's running on the same computer.

Here is how the remote address is configured:

```
/*udp_sendto.c continued*/

    printf("Configuring remote address...\n");
    struct addrinfo hints;
    memset(&hints, 0, sizeof(hints));
    hints.ai_socktype = SOCK_DGRAM;
    struct addrinfo *peer_address;
    if (getaddrinfo("127.0.0.1", "8080", &hints, &peer_address)) {
        fprintf(stderr, "getaddrinfo() failed. (%d)\n", GETSOCKETERRNO());
        return 1;
    }
```

Notice that we hardcoded `127.0.0.1` and `8080` into the call to `getaddrinfo()`. Also, notice that we've set `hints.ai_socktype = SOCK_DGRAM`. This tells `getaddrinfo()` that we are connecting over UDP. Notice that we did not set `AF_INET` or `AF_INET6`. This allows `getaddrinfo()` to return the appropriate address for IPv4 or IPv6. In this case, it is IPv4 because the address, `127.0.0.1`, is an IPv4 address. We will cover `getaddrinfo()` in more detail in Chapter 5, *Hostname Resolution and DNS*.

We can print the configured address using `getnameinfo()`. The call to `getnameinfo()` is the same as in the previous UDP server, `udp_recvfrom.c`. It works as follows:

```
/*udp_sendto.c continued*/

    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  | NI_NUMERICSERV);
    printf("%s %s\n", address_buffer, service_buffer);
```

Now that we've stored the remote address, we are ready to create our socket with a call to `socket()`. We pass in fields from `peer_address` to create the appropriate socket type. The code for this is as follows:

```
/*udp_sendto.c continued*/

    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;
    }
```

Once the socket has been created, we can go straight to sending data with `sendto()`. There is no need to call `connect()`. Here is the code to send `Hello World` to our UDP server:

```
/*udp_sendto.c continued*/

    const char *message = "Hello World";
    printf("Sending: %s\n", message);
    int bytes_sent = sendto(socket_peer,
            message, strlen(message),
            0,
            peer_address->ai_addr, peer_address->ai_addrlen);
    printf("Sent %d bytes.\n", bytes_sent);
```

Notice that `sendto()` is much like `send()`, except that we need to pass in an address as the last parameter.

It is also worth noting that we do not get an error back if sending fails. `send()` simply tries to send a message, but if it gets lost or is misdelivered along the way, there is nothing we can do about it. If the message is important, it is up to the application protocol to implement the corrective action.

After we've sent our data, we could reuse the same socket to send data to another address (as long as it's the same type of address, which is IPv4 in this case). We could also try to receive a reply from the UDP server by calling `recvfrom()`. Note that if we did call `recvfrom()` here, we could get data from anybody that sends to us – not necessarily the server we just transmitted to.

When we sent our data, our socket was assigned with a temporary local port number by the operating system. This local port number is called the **ephemeral port number**. From then on, our socket is essentially listening for a reply on this local port. If the local port is important, you can use `bind()` to associate a specific port before calling `send()`.

If multiple applications on the same system are connecting to a remote server at the same port, the operating system uses the local ephemeral port number to keep replies separate. Without this, it wouldn't be possible to know which application should receive which reply.

We'll end our example program by freeing the memory for `peer_address`, closing the socket, cleaning up Winsock, and finishing `main()`, as follows:

```
/*udp_sendto.c continued*/

    freeaddrinfo(peer_address);
    CLOSESOCKET(socket_peer);

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

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

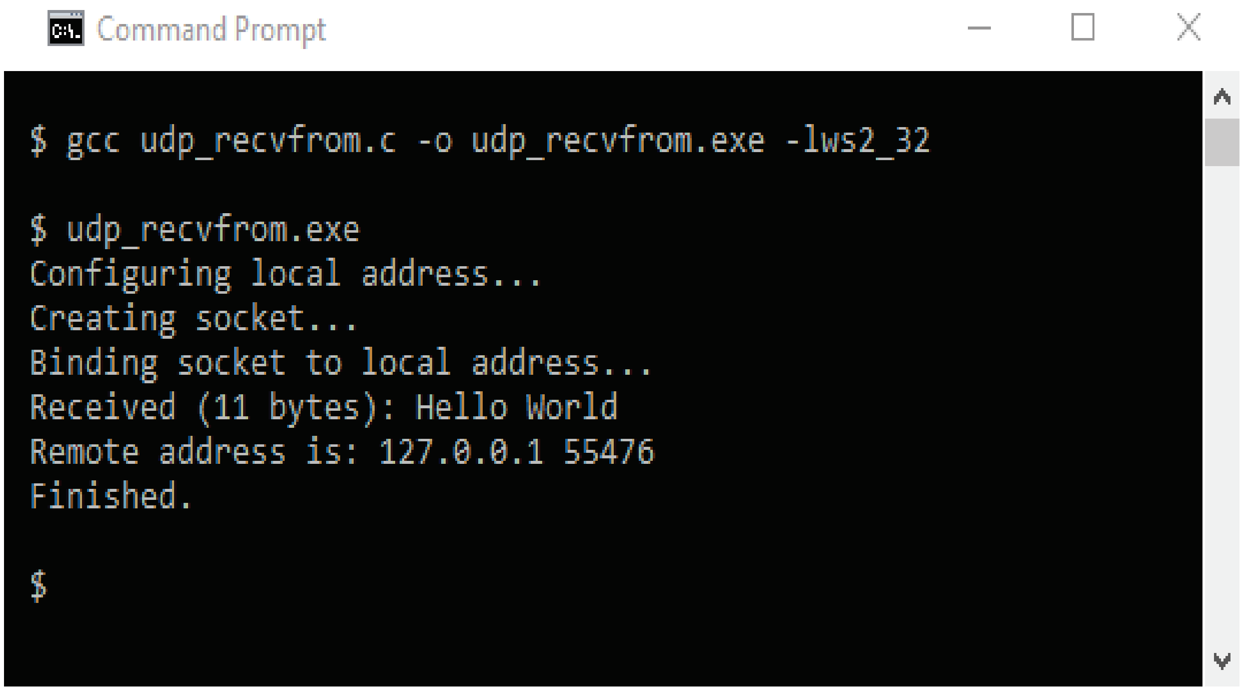
You can compile `udp_sendto.c` on Linux and macOS using the following command:

```
gcc udp_sendto.c -o udp_sendto
```

Compiling on Windows with MinGW is done in the following way:

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

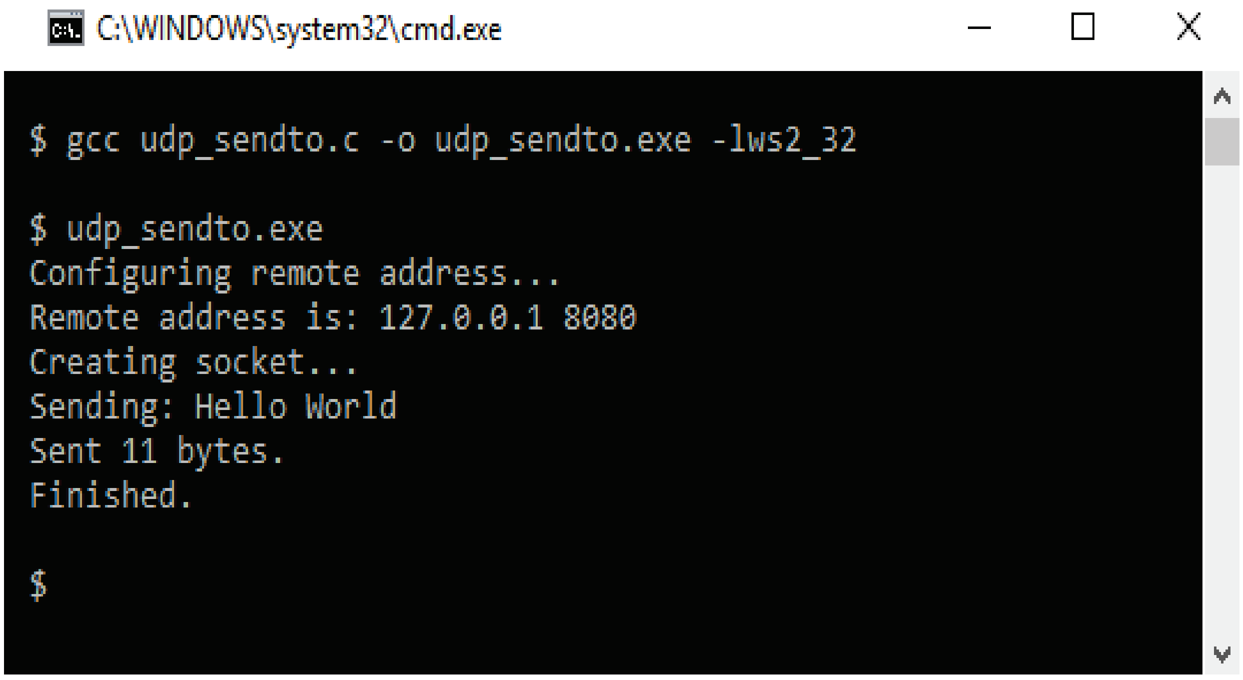
To test it out, first, start `udp_recvfrom` in a separate terminal. With `udp_recvfrom` already running, you can start `udp_sendto`. It should look as follows:



```
$ gcc udp_recvfrom.c -o udp_recvfrom.exe -lws2_32

$ udp_recvfrom.exe
Configuring local address...
Creating socket...
Binding socket to local address...
Received (11 bytes): Hello World
Remote address is: 127.0.0.1 55476
Finished.

$
```



```
$ gcc udp_sendto.c -o udp_sendto.exe -lws2_32

$ udp_sendto.exe
Configuring remote address...
Remote address is: 127.0.0.1 8080
Creating socket...
Sending: Hello World
Sent 11 bytes.
Finished.

$
```

If no server is running on port `8080`, `udp_sendto` still produces the same output. `udp_sendto` doesn't know that the packet was not delivered.

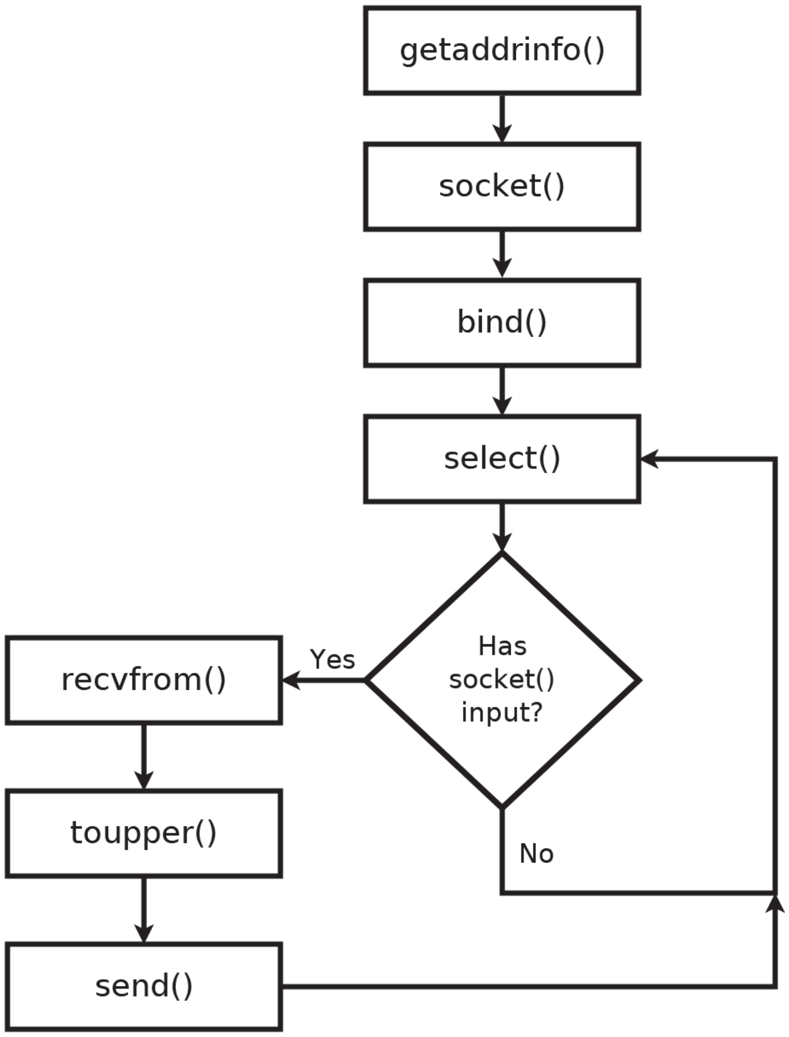
A UDP server

It will be useful to look at a UDP server that's been designed to service many connections. Fortunately for us, the UDP socket API makes this very easy.

We will take the motivating example from our last chapter, which was to provide a service that converts all text into uppercase. This is useful because you can directly compare the UDP code from here to the TCP server code from `Chapter 3`, *An In-Depth Overview of TCP Connections*.

Our server begins by setting up the socket and binding to our local address. It then waits to receive data. Once it has received a data string, it converts the string into all uppercase and sends it back.

The program flow looks as follows:



If you compare the flow of this program to the TCP server from the last chapter (Chapter 3, *An In-Depth Overview of TCP Connections*), you will find that it's much simpler. With TCP, we had to use `listen()` and `accept()`. With UDP, we skip those calls and go straight into receiving data with `recvfrom()`. With our TCP server, we had to monitor a listening socket for new connections while simultaneously monitoring an additional socket for each connected client. Our UDP server only uses one socket, so there is much less to keep track of.

Our UDP server program begins by including the necessary headers, starting the `main()` function, and initializing Winsock, as follows:

```
/*udp_serve_toupper.c*/

#include "chap04.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 find our local address that we should listen on, create the socket, and bind to it. This is all exactly the same as in our earlier server, `udp_recvfrom.c`. The only difference between this code and the TCP servers in Chapter 3, *An In-Depth Overview of TCP Connections*, is that we use `SOCK_DGRAM` instead of `SOCK_STREAM`. Recall that `SOCK_DGRAM` specifies that we want a UDP socket.

Here is the code for setting the address and creating a new socket:

```
/*udp_serve_toupper.c continued*/

    printf("Configuring local address...\n");
    struct addrinfo hints;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_DGRAM;
    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;
    }
```

Binding the new socket to the local address is done as follows:

```
/*udp_serve_toupper.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);
```

Because our server uses `select()`, we need to create a new `fd_set` to store our listening socket. We zero the set using `FD_ZERO()`, and then add our socket to this set using `FD_SET()`. We also maintain the maximum socket in the set using `max_socket`:

```
/*udp_serve_toupper.c continued*/

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

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

Note that we don't really have to use `select()` for this program, and omitting it would make the program simpler (see `udp_server_toupper_simple.c`). However, we are going to use `select()` because it makes our code more flexible. We could easily add in an additional socket (if we needed to listen on multiple ports, for example), and we could add in a `select()` timeout if our program needs to perform other functions. Of course, our program doesn't do those things, so we don't need `select()`, but I think that most programs do, so we will show it that way.

Now, we are ready for the main loop. It copies the socket set into a new variable, `reads`, and then uses `select()` to wait until our socket is ready to read from. Recall that we could pass in a timeout value as the last parameter to `select()` if we want to set a maximum waiting time for the next read. Refer to `Chapter 3`, *An In-Depth Overview of TCP Connections*, the *Synchronous multiplexing with select()* section, for more information on `select()`.

Once `select()` returns, we use `FD_ISSET()` to tell if our particular socket, `socket_listen`, is ready to be read from. If we had additional sockets, we would need to use `FD_ISSET()` for each socket.

If `FD_ISSET()` returns true, we read from the socket using `recvfrom()`. `recvfrom()` gives us the sender's address, so we must first allocate a variable to hold the address, that is, `client_address`. Once we've read a string from the socket using `recvfrom()`, we convert the string into uppercase using the C `toupper()` function. We then send the modified text back to the sender using `sendto()`. Note that the last two parameters to `sendto()` are the client's addresses that we got from `recvfrom()`.

The main program loop can be seen in the following code:

```
/*udp_serve_toupper.c continued*/

    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;
        }

        if (FD_ISSET(socket_listen, &reads)) {
            struct sockaddr_storage client_address;
            socklen_t client_len = sizeof(client_address);

            char read[1024];
            int bytes_received = recvfrom(socket_listen, read, 1024, 0,
                    (struct sockaddr *)&client_address, &client_len);
            if (bytes_received < 1) {
                fprintf(stderr, "connection closed. (%d)\n",
                        GETSOCKETERRNO());
                return 1;
            }

            int j;
            for (j = 0; j < bytes_received; ++j)
                read[j] = toupper(read[j]);
            sendto(socket_listen, read, bytes_received, 0,
                    (struct sockaddr*)&client_address, client_len);

        } //if FD_ISSET
    } //while(1)
```

We can then close the socket, clean up Winsock, and terminate the program. Note that this code never runs, because the main loop never terminates. We include this code anyway as good practice; in case the program is adapted in the future to have an exit function.

The cleanup code is as follows:

```
/*udp_serve_toupper.c continued*/
```

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

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

    printf("Finished.\n");

    return 0;
}
```

That's our complete UDP server program. You can compile and run it on Linux and macOS as follows:

```
gcc udp_serve_toupper.c -o udp_serve_toupper
./udp_serve_toupper
```

Compiling and running on Windows with MinGW is done in the following manner:

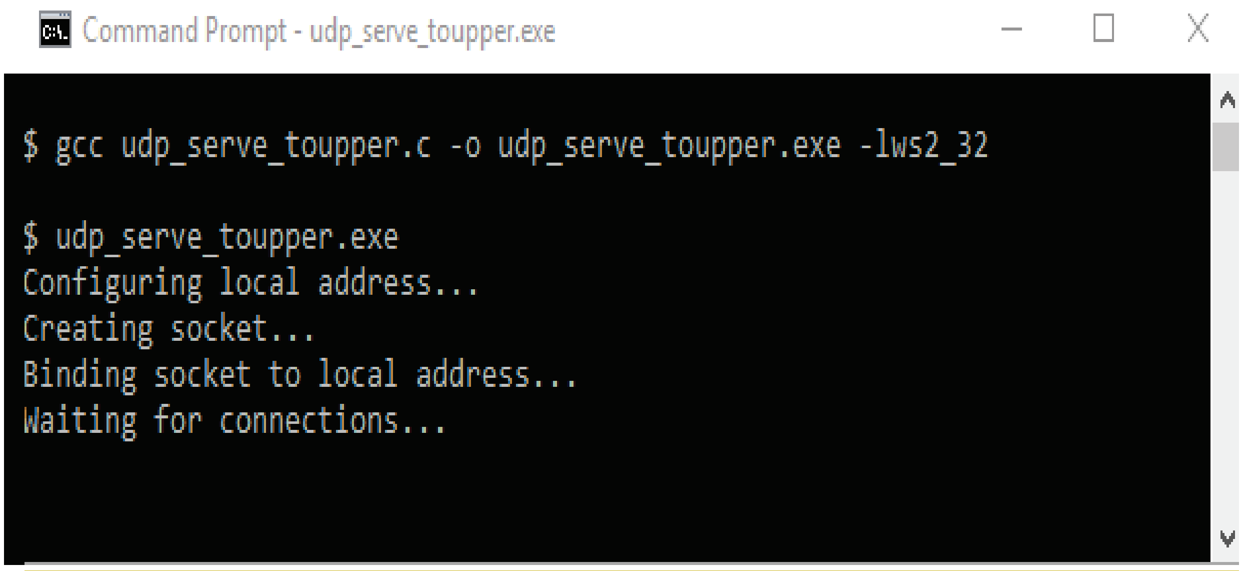
```
gcc udp_serve_toupper.c -o udp_serve_toupper.exe -lws2_32
udp_serve_toupper.exe
```

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

Once the program is running, you should open another terminal window and run the `udp_client` program from earlier to connect to it, as follows:

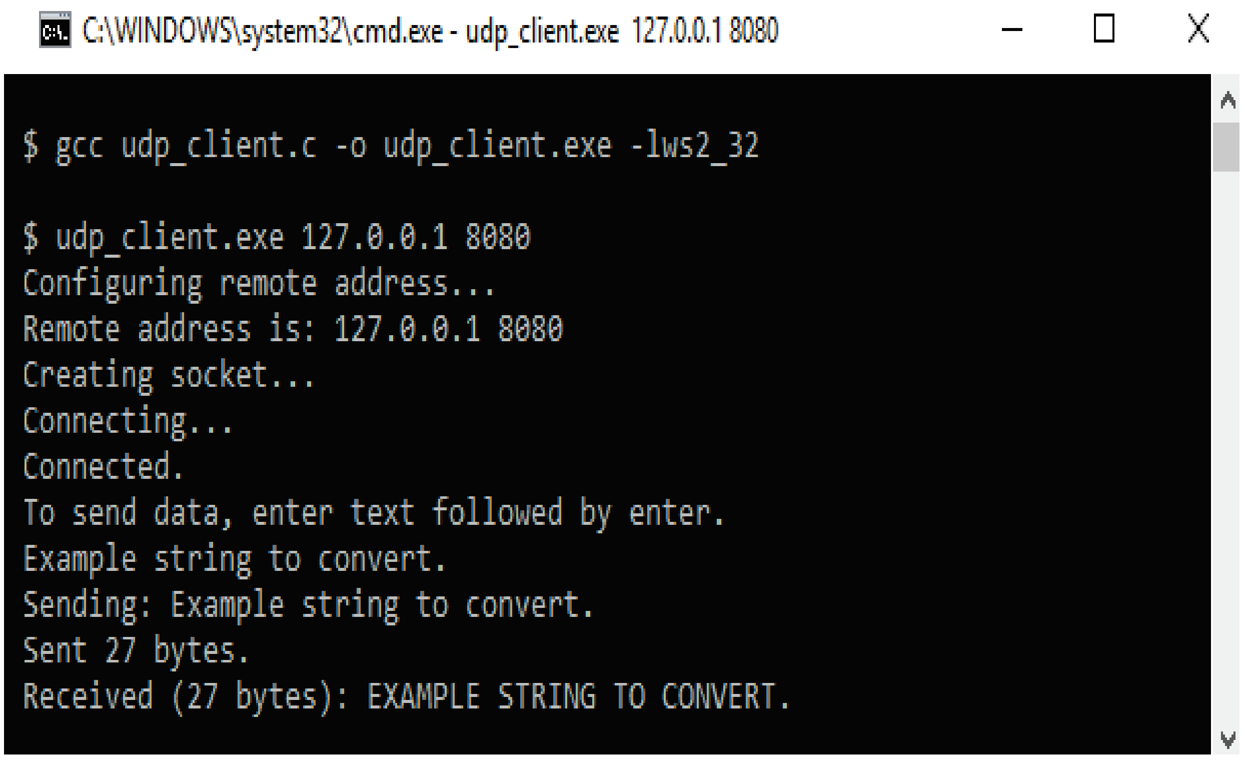
```
udp_client 127.0.0.1 8080
```

Anything you type in `udp_client` should be sent back to it in uppercase. Here's what that might look like:



```
$ gcc udp_serve_toupper.c -o udp_serve_toupper.exe -lws2_32

$ udp_serve_toupper.exe
Configuring local address...
Creating socket...
Binding socket to local address...
Waiting for connections...
```



```
$ gcc udp_client.c -o udp_client.exe -lws2_32

$ udp_client.exe 127.0.0.1 8080
Configuring remote address...
Remote address is: 127.0.0.1 8080
Creating socket...
Connecting...
Connected.
To send data, enter text followed by enter.
Example string to convert.
Sending: Example string to convert.
Sent 27 bytes.
Received (27 bytes): EXAMPLE STRING TO CONVERT.
```

You may also want to try opening additional terminal windows and connecting with `udp_client`.

See `udp_serve_toupper_simple.c` for an implementation that doesn't use `select()`, but manages to work just as well anyway.

Summary

In this chapter, we saw that programming with UDP sockets is somewhat easier than with TCP sockets. We learned that UDP sockets don't need the `listen()`, `accept()`, or `connect()` function calls. This is mostly because `sendto()` and `recvfrom()` deal with the addresses directly. For more complicated programs, we can still use the `select()` function to see which sockets are ready for I/O.

We also saw that UDP sockets are connectionless. This is in contrast to connection-oriented TCP sockets. With TCP, we had to establish a connection before sending data, but with UDP, we simply send individual packets directly to a destination address. This keeps UDP socket programming simple, but it can complicate application protocol design, and UDP does not automatically retry communication failures or ensure that packets arrive in order.

The next chapter, `Chapter 5`, *Hostname Resolution and DNS*, is all about hostnames. Hostnames are resolved using the DNS protocol, which works over UDP. Move on to `Chapter 5`, *Hostname Resolution and DNS*, to learn about implementing a real-world UDP protocol.

Questions

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

1. How do `sendto()` and `recvfrom()` differ from `send()` and `recv()`?
2. Can `send()` and `recv()` be used on UDP sockets?
3. What does `connect()` do in the case of a UDP socket?
4. What makes multiplexing with UDP easier than with TCP?
5. What are the downsides to UDP when compared to TCP?
6. Can the same program use UDP and TCP?

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