a human. For example, the password “password” may be hashed to a string that looks more like `'a12bc21df'`. While no one can read or recover the input data, we’re still able to check if a piece of data matches a hash. This is useful for scenarios such as validating a user’s password on login or checking if a piece of data has been tampered with.

There are many different hashing algorithms today, such as SHA512, BLAKE2, and scrypt, though SHA is not the best choice for storing passwords, as it is susceptible to brute-force attacks. Several of these algorithms are implemented in Python’s `hashlib` library. Many functions in this library release the GIL when hashing data greater than 2048 bytes, so multithreading is an option to improve this library’s performance. In addition, the `scrypt` function, used for hashing passwords, always releases the GIL.

Let’s introduce a (hopefully) hypothetical scenario to see when multithreading might be useful with `hashlib`. Imagine you’ve just started a new job as principal software architect at a successful organization. Your manager assigns you your first bug to get started learning the company’s development process—a small issue with the login system. To debug this issue, you start to look at a few database tables, and to your horror you notice that all your customers’ passwords are stored in plaintext! This means that if your database is compromised, attackers could get all your customers’ passwords and log in as them, potentially exposing sensitive data such as saved credit card numbers. You bring this to your manager’s attention, and they ask you to find a solution to the problem as soon as possible.

Using the `scrypt` algorithm to hash the plaintext passwords is a good solution for this kind of problem. It is secure and the original password is unrecoverable, as it introduces a *salt*. A salt is a random number that ensures that the hash we get for the

password is unique. To test out using scrypt, we can quickly write a synchronous script to create random passwords and hash them to get a sense of how long things will take. For this example, we'll test on 10,000 random passwords.

Listing 7.16 Hashing passwords with scrypt

```
import hashlib
import os
import string
import time
import random


def random_password(length: int) -> bytes:
    ascii_lowercase = string.ascii_lowercase.encode()
    return b''.join(bytes(random.choice(ascii_lowercase)) fo


passwords = [random_password(10) for _ in range(10000)]


def hash(password: bytes) -> str:
    salt = os.urandom(16)
    return str(hashlib.scrypt(password, salt=salt, n=2048, p


start = time.time()

for password in passwords:
    hash(password)
```

```
  end = time.time()
  print(end - start)
```

We first write a function to create random lowercase passwords and then use that to create 10,000 random passwords of 10 characters each. We then hash each password with the `scrypt` function. We'll gloss over the details (n, p, and r parameters of the `scrypt` function), but these are used to tune how secure we'd like our hash to be and memory/CPU usage.

Running this on the servers you have, which are 2.4 Ghz 8-core machines, this code completes in just over 40 seconds, which is not too bad. The issue is that you have a large user base, and you need to hash 1,000,000,000 passwords. Doing the calculation based on this test, it will take a bit over 40 days to hash the entire database! We could split up our data set and run this procedure on multiple machines, but we'd need a lot of machines to do that, given how slow this is. Can we use threading to improve the speed and therefore cut down on the time and machines we need to use? Let's apply what we know about multithreading to give this a shot. We'll create a thread pool and hash passwords in multiple threads.

Listing 7.17 Hashing with multithreading and asyncio

```
  import asyncio
  import functools
  import hashlib
  import os
  from concurrent.futures.thread import ThreadPoolExecutor
  import random
  import string
```

```
from util import async_timed


def random_password(length: int) -> bytes:
    ascii_lowercase = string.ascii_lowercase.encode()
    return b''.join(bytes(random.choice(ascii_lowercase)) fo


passwords = [random_password(10) for _ in range(10000)]


def hash(password: bytes) -> str:
    salt = os.urandom(16)
    return str(hashlib.scrypt(password, salt=salt, n=2048, p


@async_timed()
async def main():
    loop = asyncio.get_running_loop()
    tasks = []

    with ThreadPoolExecutor() as pool:
        for password in passwords:
            tasks.append(loop.run_in_executor(pool, functool

    await asyncio.gather(*tasks)


asyncio.run(main())
```

This approach involves us creating a thread pool executor and creating a task for each password we want to hash. Since `hashlib` releases the GIL we realize some decent performance gains. This code runs in about 5 seconds as opposed to the 40 we got earlier. We've just cut our runtime down from 47 days to a bit over 5! As a next step, we could take this application and run it concurrently on different machines to further cut runtime, or we could get a machine with more CPU cores.

Multithreading with NumPy

NumPy is an extremely popular Python library, widely used in data science and machine learning projects. It has a multitude of mathematical functions common to arrays and matrices that tend to outperform plain Python arrays. This increased performance is because much of the underlying library is implemented in C and Fortran that are low-level languages and tend to be more performant than Python.

Because many of this library's operations are in low-level code outside of Python, this opens the opportunity for NumPy to release the GIL and allow us to multithread some of our code. The caveat here is this functionality is not well-documented, but it is generally safe to assume matrix operations can potentially be multithreaded for a performance win. That said, depending on how the `numpy` function is implemented, the win could be large or small. If the code directly calls C functions and releases the GIL there is a potential bigger win; if there is a lot of supporting Python code around any low-level calls, the win will be smaller. Given that this is not well documented, you may have to try adding multithreading to specific bottlenecks in your application (you can determine where the bottlenecks are with profiling) and benchmarking what gains you get. You'll then need to decide if the extra complexity is worth any potential gains you get.

To see this in practice, we'll create a large matrix of 4,000,000,000 data points in 50 rows. Our task will be to obtain the mean for reach row. NumPy has an efficient function, `mean`, to compute this. This function has an axis parameter which lets us calculate all the means across an axis without having to write a loop. In our case, an axis of 1 will calculate the mean for every row.

Listing 7.18 Means of a large matrix with NumPy

```
import numpy as np
import time

data_points = 4000000000
rows = 50
columns = int(data_points / rows)

matrix = np.arange(data_points).reshape(rows, columns)

s = time.time()

res = np.mean(matrix, axis=1)

e = time.time()
print(e - s)
```

This script first creates an array with 4,000,000,000 integer data points, ranging from 1,000,000,000-4,000,000,000 (note that this takes quite a bit of memory; if your application crashes with an out-of-memory error, lower this number). We then "reshape" the array into a matrix with 50 rows. Finally, we call NumPy's `mean` function with an axis of 1, calculating the mean for each individual row. All told, this script runs in about 25–30 seconds on an 8-core 2.4 Ghz CPU. Let's adapt this code

slightly to work with threads. We'll run the median for each row in a separate thread and use `asyncio.gather` to wait for all the median of all rows.

Listing 7.19 Threading with NumPy

```
import functools
from concurrent.futures.thread import ThreadPoolExecutor
import numpy as np
import asyncio
from util import async_timed


def mean_for_row(arr, row):
    return np.mean(arr[row])


data_points = 4000000000
rows = 50
columns = int(data_points / rows)

matrix = np.arange(data_points).reshape(rows, columns)


@async_timed()
async def main():
    loop = asyncio.get_running_loop()
    with ThreadPoolExecutor() as pool:
        tasks = []
        for i in range(rows):
            mean = functools.partial(mean_for_row, matrix, i
            tasks.append(loop.run_in_executor(pool, mean))
```

```
        results = asyncio.gather(*tasks)


asyncio.run(main())
```

First, we create a `mean_for_row` function that calculates the mean for one row. Since our plan is to calculate the mean for every row in a separate thread, we can no longer use the `mean` function with an axis as we did before. We then create a main coroutine with a thread pool executor and create a task to calculate the mean for each row, waiting for all the calculations to finish with `gather`.

On the same machine, this code runs in roughly 9–10 seconds, nearly a 3× boost in performance! Multithreading can help us in certain cases with NumPy, although the documentation for what can benefit from threads is lacking at the time of writing. When in doubt, if threading will help a CPU-bound workload, the best way to see if it will help is to test it out and benchmark.

In addition, keep in mind that your NumPy code should be as vectorized as possible before trying threading or multiprocessing to improve performance. This means avoiding things like Python loops or functions like NumPy's `apply_along_axis`, which just hides a loop. With NumPy, you will often see much better performance by pushing as much computation as you can to the library's low-level implementations.

ary

- We've learned how to run I/O-bound work using the threading module.
- We've learned how to cleanly terminate threads on application shutdown.

- We've learned how to use thread pool executors to distribute work to a pool of threads. This allows us to use asyncio API methods like `gather` to wait for results from threads.
- We've learned how to take existing blocking I/O APIs, such as requests, and run them in threads with thread pools and asyncio for performance wins.
- We've learned how to avoid race conditions with locks from the threading module. We've also learned how to avoid deadlocks with reentrant locks.
- We've learned how to run the asyncio event loop in a separate thread and send coroutines to it in a thread-safe manner. This lets us build responsive user interfaces with frameworks such as Tkinter.
- We've learned how to use multithreading with `hashlib` and `numpy`. Low-level libraries will sometimes release the GIL, which lets us use threading for CPU-bound work.

8 Streams

This chapter covers

- Transports and protocols
- Using streams for network connections
- Processing command-line input asynchronously
- Creating client/server applications with streams

When writing network applications, such as our echo clients in prior chapters, we've employed the socket library to read from and write to our clients. While directly using sockets is useful when building low-level networking libraries, they are ultimately complex creatures with nuances outside the scope of this book. That said, many use cases of sockets rely on a few conceptually simple operations, such as starting a server, waiting for client connections, and sending data to clients. The designers of asyncio realized this and built network stream APIs to abstract away handling the nuances of sockets for us. These higher-level APIs are much easier to work with than sockets, making any client-server applications easier to build and more robust than using sockets ourselves. Using streams is the recommended way to build network-based applications in asyncio.

In this chapter, we'll first learn using the lower-level transport and protocol APIs by building a simple HTTP client. Learning about these APIs will give us the foundation for understanding how the higher-level stream APIs work in the background. We'll then use this knowledge to learn about stream readers and writers and use them to build a non-blocking command-line SQL client. This application will asynchronously process user input, allowing us to run multiple queries concurrently from the

command line. Finally, we'll learn how to use asyncio's server API to create client and server applications, building a functional chat server and chat client.

roducing streams

In asyncio, *streams* are a high-level set of classes and functions that create and manage network connections and generic streams of data. Using them, we can create client connections to read and write to servers, or even create servers and manage them ourselves. These APIs abstract a lot of knowledge around managing sockets, such as dealing with SSL or lost connections, making our lives as developers a little easier.

The stream APIs are built on top of a lower-level set of APIs known as *transports* and *protocols*. These APIs directly wrap the sockets we used in previous chapters (generally, any generic stream of data), providing us with a clean API for reading and writing data to sockets.

These APIs are structured a little differently from others in that they use a callback style design. Instead of actively waiting for data from a socket like we did previously, a method on a class we implement is called for us when data is available. We then process the data we receive in this method as needed. To get started learning how these callback-based APIs work, let's first see how to use the lower-level transport and protocol APIs by building a basic HTTP client.

ansports and protocols

At a high level, a transport is an abstraction for communication with an arbitrary stream of data. When we communicate with a socket or any data stream such as standard input, we work with a familiar set of operations. We read data from or write

data to a source, and when we're finished working with it, we close it. A socket cleanly fits how we've defined this transport abstraction; that is, we read and write data to it and once we've finished, we close it. In short, a transport provides definitions for sending and receiving data to and from a source. Transports have several implementations depending on which type of source we're using. We're mainly concerned with `ReadTransport`, `WriteTransport`, and `Transport`, though there are others for dealing with UDP connections and subprocess communication. Figure 8.1 illustrates the class hierarchy of transports.

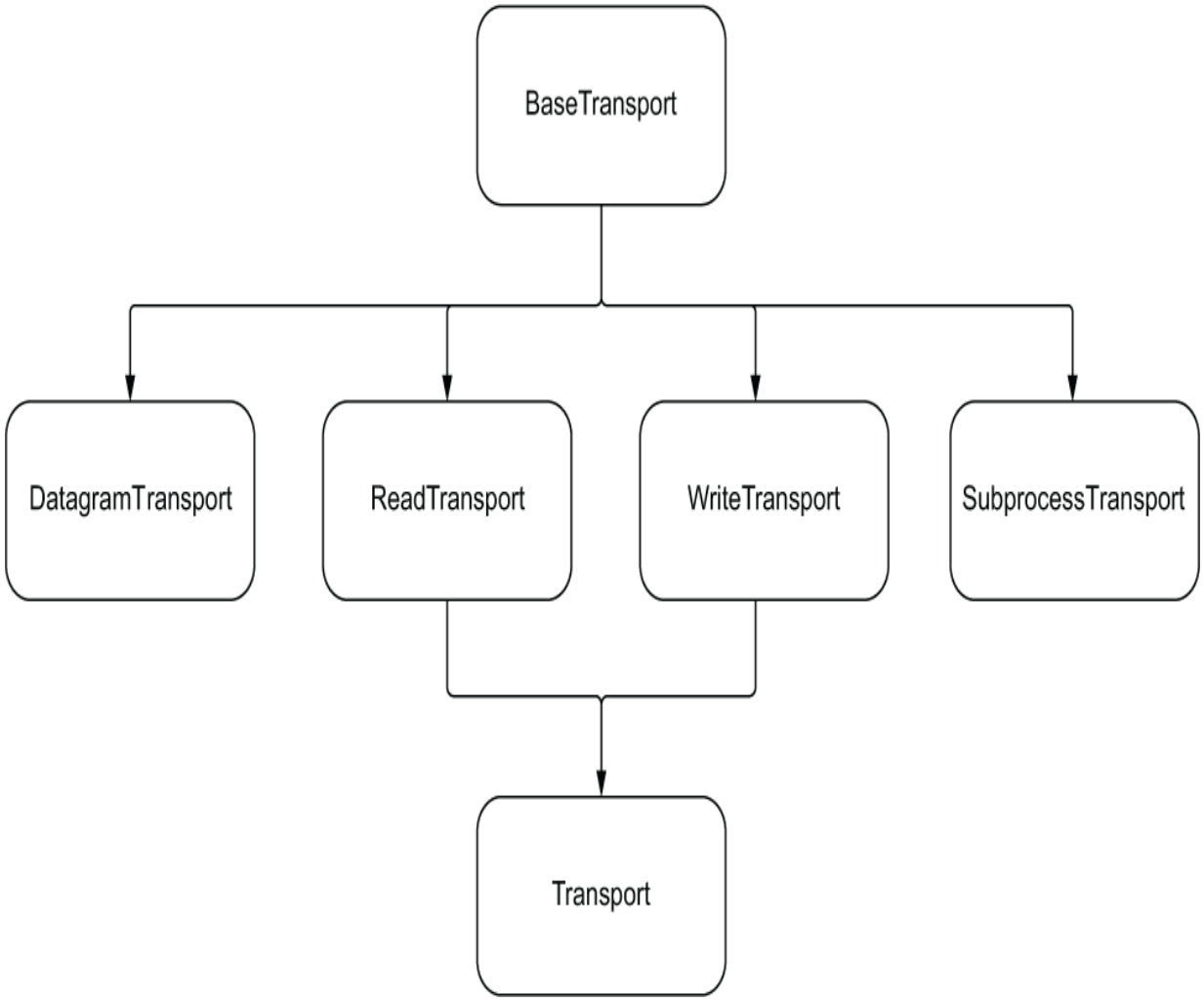

Figure 8.1 The class hierarchy of transports

Transmitting data to and from a socket is only part of the equation. What about the lifecycle of a socket? We establish a connection; we write data and then process any response we get. These are the set of operations a protocol owns. Note that a protocol simply refers to a Python class here and not a protocol like HTTP or FTP. A transport manages data transmission and calls methods on a protocol when events occur, such as a connection being established or data being ready to process, as shown in figure 8.2.

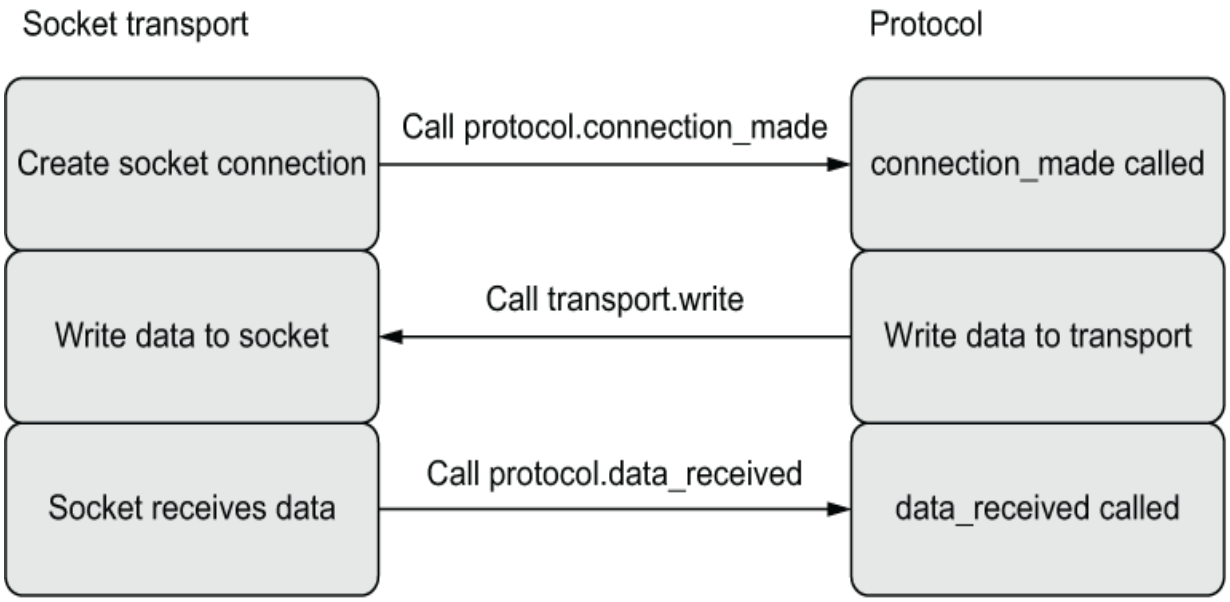

Figure 8.2 A transport calls methods on a protocol when events happen. A protocol can write data to a transport.

To understand how transports and protocols work together, we'll build a basic application to run a single HTTP GET request. The first thing we'll need to do is define a class that extends `asyncio.Protocol`. We'll implement a few methods from the base class to make the request, receive data from the request, and handle any errors with the connection.

The first protocol method we'll need to implement is `connection_made`. The transport calls this method when the underlying socket has successfully connected with the HTTP server. This method employs a `Transport` as an argument that we can use to communicate with the server. In this case, we'll use the transport to send the HTTP request immediately.

The second method we'll need to implement is `data_received`. The transport calls this method whenever it receives data, passing it to us as bytes. This method can be called multiple times, so we'll need to create an internal buffer to store the data.

The question now becomes, how do we tell when our response is finished? To answer this, we'll implement a method called `eof_received`. This method is called when we receive the *end of file*, which, in the case of a socket, happens when the server closes the connection. Once this method is called, we are guaranteed that `data_received` will never be called again. The `eof_received` method returns a `Boolean` value that determines how to shut down the transport (close the client socket in this example). Returning `False` ensures that the transport will shut itself down, whereas `True` means that the protocol implementation we wrote will shut things down. In this case, as we don't need to do any special logic on shutdown, our method should return `False,` so we don't need to handle closing the transport ourselves.

With what we've described, we have only a way to store things in an internal buffer. So, how do consumers of our protocol get the result once the request is finished? To do this, we can create a `Future` internally to hold the result when it is complete. Then, in the `eof_received` method we'll set the result of the `future` to the result of the HTTP response. We'll then define a coroutine we'll name `get_response` that will await the future.

Let's take what we've described above and implement it as our own protocol. We'll call it `HTTPGetClientProtocol`.

Listing 8.1 Running a HTTP request with transports and protocols

```
import asyncio
from asyncio import Transport, Future, AbstractEventLoop
from typing import Optional


class HTTPGetClientProtocol(asyncio.Protocol):

    def __init__(self, host: str, loop: AbstractEventLoop):
        self._host: str = host
        self._future: Future = loop.create_future()
        self._transport: Optional[Transport] = None
        self._response_buffer: bytes = b''

    async def get_response(self):
        return await self._future

    def _get_request_bytes(self) -> bytes:
        request = f"GET / HTTP/1.1\r\n" \
                  f"Connection: close\r\n" \
                  f"Host: {self._host}\r\n\r\n"
        return request.encode()

    def connection_made(self, transport: Transport):
        print(f'Connection made to {self._host}')
        self._transport = transport
        self._transport.write(self._get_request_bytes())
```

```
    def data_received(self, data):
        print(f'Data received!')
        self._response_buffer = self._response_buffer + data

    def eof_received(self) -> Optional[bool]:
        self._future.set_result(self._response_buffer.decode
        return False

    def connection_lost(self, exc: Optional[Exception]) -> N
        if exc is None:
            print('Connection closed without error.')
        else:
            self._future.set_exception(exc)
```

❶ Await the internal future until we get a response from the server.

❷ Create the HTTP request.

❸ Once we've established a connection, use the transport to send the request.

❹ Once we have data, save it to our internal buffer.

❺ Once the connection closes, complete the future with the buffer.

❻ If the connection closes without error, do nothing; otherwise, complete the future with an exception.

Now that we've implemented our protocol, let's use it to make a real request. To do this, we'll need to learn a new coroutine method on the asyncio event loop named `create_connection`. This method will create a socket connection to a given host

and wrap it in an appropriate transport. In addition to a host and port, it takes in a *protocol factory*. A protocol factory is a function that creates protocol instances; in our case, an instance of the `HTTPGetClientProtocol` class we just created. When we call this coroutine, we're returned both the transport that the coroutine created along with the protocol instance the factory created.

Listing 8.2 Using the protocol

```
import asyncio
from asyncio import AbstractEventLoop
from chapter_08.listing_8_1 import HTTPGetClientProtocol


async def make_request(host: str, port: int, loop: AbstractE
    def protocol_factory():
        return HTTPGetClientProtocol(host, loop)

    _, protocol = await loop.create_connection(protocol_fact

    return await protocol.get_response()


async def main():
    loop = asyncio.get_running_loop()
    result = await make_request('www .example .com', 80, loo
    print(result)


asyncio.run(main())
```

We first define a `make_request` method that takes in the host and port we'd like to make a request to, and the server's response. Inside this method, we create an inner method for our protocol factory that creates a new `HTTPGetClientProtocol`. We then call `create_connection` with the host and port that returns both a transport and the protocol our factory created. We won't need the transport, and we ignore it, but we will need the protocol because we'll want to use the `get_response` coroutine; therefore, we'll keep track of it in the `protocol` variable. Finally, we await the `get_response` coroutine of our protocol that will wait until the HTTP server has responded with a result. In our main coroutine, we await `make_request` and print the response. Executing this, you should see a HTTP response like the following (we've omitted the HTML body for brevity):

```
Connection made to www .example .com
Data received!
HTTP/1.1 200 OK
Age: 193241
Cache-Control: max-age=604800
Content-Type: text/html; charset=UTF-8
Connection closed without error.
```

We've learned to use transports and protocols. These APIs are lower-level and, as such, aren't the recommended way to work with streams in asyncio. Let's see how to use *streams*, a higher-level abstraction that expands on transports and protocols.

eam readers and stream writers

Transports and protocols are lower-level APIs that are best suited for when we need direct control over what is happening as we send and receive data. As an example, if

we're designing a networking library or web framework, we may consider transports and protocols. For most applications, we don't need this level of control, and using transports and protocols would involve us writing a bunch of repetitive code.

The designers of asyncio realized this and created the higher-level *streams* APIs. This API encapsulates the standard use cases of transports and protocols into two classes that are easy to understand and use: `StreamReader` and `StreamWriter`. As you can guess, they handle reading from and writing to streams, respectively. Using these classes is the recommended way to develop networking applications in asyncio.

To get an understanding of how to use these APIs, let's take our example of making a HTTP GET request and translate it into streams. Instead of directly instantiating `StreamReader` and `StreamWriter` instances, asyncio provides a library coroutine function named `open_connection` that will create them for us. This coroutine takes in a host and port that we'll connect to and returns a `StreamReader` and a `StreamWriter` as a tuple. Our plan will be to use the `StreamWriter` to send out the HTTP request and the `StreamReader` to read the response. `StreamReader` methods are easy to understand, and we have a convenient `readline` coroutine that waits until we have a line of data. Alternatively, we could also use `StreamReader`'s `read` coroutine that waits for a specified number of bytes to arrive.

`StreamWriter` is a little more complex. It has a `write` method as we'd expect, but it is a plain method and *not* a coroutine. Internally, stream writers try to write to a socket's output buffer right away, but this buffer can be full. If the socket's write buffer is full, the data is instead stored in an internal queue where it can later go into to the buffer. This poses a potential problem in that calling `write` does not necessarily send out data immediately. This can cause potential memory issues.

Imagine our network connection becomes slow, able to send out 1 KB per second, but our application is writing out 1 MB per second. In this case, our application's write buffer will fill up at a much faster rate than we can send the data out to the socket's buffer, and eventually we'll start to hit memory limits on the machine, inviting a crash.

How can we wait until all our data is properly sent out? To solve this issue, we have a coroutine method called `drain`. This coroutine will block until all queued data gets sent to the socket, ensuring we've written everything before moving on. The pattern we'll want to use functions after we call `write` we'll always `await` a call to `drain`. Technically, it's not necessary to call `drain` after every `write`, but it is a good idea to help prevent bugs.

Listing 8.3 A HTTP request with stream readers and writers

```
import asyncio
from asyncio import StreamReader
from typing import AsyncGenerator


async def read_until_empty(stream_reader: StreamReader) -> A
    while response := await stream_reader.readline():
        yield response.decode()


async def main():
    host: str = 'www .example .com'
    request: str = f"GET / HTTP/1.1\r\n" \
                   f"Connection: close\r\n" \
                   f"Host: {host}\r\n\r\n"
```

```
    stream_reader, stream_writer = await asyncio.open_connec

    try:
        stream_writer.write(request.encode())
        await stream_writer.drain()

        responses = [response async for response in read_unt

        print(''.join(responses))
    finally:
        stream_writer.close()
        await stream_writer.wait_closed()


asyncio.run(main())
```

❶ Read a line and decode it until we don't have any left.

❷ Write the http request, and drain the writer.

❸ Read each line, and store it in a list.

❹ Close the writer, and wait for it to finish closing.

In the preceding listing, we first create a convenience async generator to read all lines from a `StreamReader`, decoding them into strings until we don't have any left to process. Then, in our main coroutine we open a connection to example.com, creating a `StreamReader` and `StreamWriter` instance in the process. We then write the request and drain the stream writer, using `write` and `drain`, respectively. Once

we've written our request, we use our async generator to get each line from the response back, storing them in the `responses` list. Finally, we close the `StreamWriter` instance by calling `close` and then awaiting the `wait_closed` coroutine. Why do we need to call a method *and* a coroutine here? The reason is that when we call `close` a few things happen, such as deregistering the socket and calling the underlying transport's `connection_lost` method. These all happen asynchronously on a later iteration of the event loop, meaning that immediately after we call `close` our connection isn't closed until sometime later. If you need to wait for the connection to close before proceeding or are concerned about any exceptions that may happen while you're closing, calling `wait_ closed` is best practice.

We've now learned the basics around the stream APIs by making web requests. The usefulness of these classes extends beyond web- and network-based applications. Next, we'll see how to utilize stream readers to create non-blocking command-line applications.

n-blocking command-line input

Traditionally in Python, when we need to get user input, we use the `input` function. This function will stop execution flow until the user has provided input and presses Enter. What if we want to run code in the background while remaining responsive to input? For example, we may want to let the user launch multiple long-running tasks concurrently, such as long-running SQL queries. In the case of a command-line chat application, we likely want the user to be able to type a message while receiving messages from other users.

Since asyncio is single-threaded, using `input` in an asyncio application means we stop the event loop from running until the user provides input, halting our entire

application. Even using tasks to kick off an operation in the background won't work. To demonstrate this, let's attempt to create an application where the user enters a time for the application to sleep. We'd like to be able to run multiple of these sleep operations concurrently while still accepting user input, so we'll ask for the number of seconds to sleep and create a `delay` task in a loop.

Listing 8.4 Attempting background tasks

```
import asyncio
from util import delay


async def main():
    while True:
        delay_time = input('Enter a time to sleep:')
        asyncio.create_task(delay(int(delay_time)))


asyncio.run(main())
```

If this code worked the way we intended, after we input a number we'd expect to see `sleeping for n second(s)` printed out followed by `finished sleeping for n second(s)` *n* seconds later. However, this isn't the case, and we see nothing except our prompt to enter a time to sleep. This is because there is no `await` inside our code and, therefore, the task never gets a chance to run on the event loop. We can hack around this by putting `await asyncio.sleep(0)` after the `create_task` line that will schedule the task (this is known as "yielding to the event loop" and will be covered in chapter 14). Even with this trick, as it stops the

entire thread the `input` call still blocks any background task we create from running to completion.

What we really want is for the `input` function to be a coroutine instead, so we could write something like `delay_time = await input('Enter a time to sleep:').` If we could do this, our task would schedule properly and continue to run while we waited for user input. Unfortunately, there is no coroutine variant of `input`, so we'll need to do something else.

This is where protocols and stream readers can help us out. Recall that a stream reader has the `readline` coroutine that is the type of coroutine we're looking for. If we had a way to hook a stream reader to standard input, we could then use this coroutine for user input.

asyncio has a coroutine method on the event loop called `connect_read_pipe` that connects a protocol to a file-like object, which is almost what we want. This coroutine method accepts a *protocol factory* and a *pipe*. A protocol factory is just a function that creates a protocol instance. A pipe is a file-like object, which is defined as an object with methods such as `read` and `write` on it. The `connect_read_pipe` coroutine will then connect the pipe to the protocol the factory creates, taking data from the pipe and sending it to the protocol.

In terms of standard console input, `sys.stdin` fits the bill of a file-like object that we can pass in to `connect_read_pipe`. Once we call this coroutine, we'll get a tuple of the protocol our factory function created and a `ReadTransport`. The question now becomes what protocol should we create in our factory, and how do we connect this with a `StreamReader` that has the `readline` coroutine we'd like to use?

asyncio provides a utility class called `StreamReaderProtocol` for connecting instances of stream readers to protocols. When we instantiate this class, we pass in an instance of a stream reader. The protocol class then delegates to the stream reader we created, allowing us to use the stream reader to read data from standard input. Putting all these pieces together, we can create a command-line application that does not block the event loop when waiting for user input.

For Windows users

Unfortunately, on Windows `connect_read_pipe` will not work with `sys.stdin`. This is due to an unfixed bug caused by the way Windows implements file descriptors. For this to work on Windows, you'll need to call `sys.stdin.readline()` in a separate thread using techniques we explored in chapter 7. You can read more about this issue at https://bugs.python.org/issue26832.

Since we'll be reusing the asynchronous standard in reader throughout the rest of the chapter, let's create it in its own file, `listing_8_5.py`. We'll then import it in the rest of the chapter.

Listing 8.5 An asynchronous standard input reader

```
import asyncio
from asyncio import StreamReader
import sys


async def create_stdin_reader() -> StreamReader:
    stream_reader = asyncio.StreamReader()
    protocol = asyncio.StreamReaderProtocol(stream_reader)
    loop = asyncio.get_running_loop()
```

```
    await loop.connect_read_pipe(lambda: protocol, sys.stdin
    return stream_reader
```

In the preceding listing, we create a reusable coroutine named `create_stdin_reader` that creates a `StreamReader` that we'll use to asynchronously read standard input. We first create a stream reader instance and pass it to a stream reader protocol. We then call `connect_read_pipe`, passing in a protocol factory as a lambda function. This lambda returns the stream reader protocol we created earlier. We also pass `sys.stdin` to connect standard input to our stream reader protocol. Since we won't need them, we ignore the transport and protocol that `connect_read_pipe` returns. We can now use this function to asynchronously read from standard input and build our application.

Listing 8.6 Using stream readers for input

```
import asyncio
from chapter_08.listing_8_5 import create_stdin_reader
from util import delay


async def main():
    stdin_reader = await create_stdin_reader()
    while True:
        delay_time = await stdin_reader.readline()
        asyncio.create_task(delay(int(delay_time)))


asyncio.run(main())
```

In our main coroutine, we call `create_stdin_reader` and loop forever, waiting for input from the user with the `readline` coroutine. Once user presses Enter on the keyboard, this coroutine will deliver the input text entered. Once we have input from the user, we convert it into an integer (note here that for a real application, we should add code to handle bad input, as we'll crash if we pass in a string right now) and create a `delay` task. Running this, you'll be able to run multiple `delay` tasks concurrently while still entering command-line input. For instance, entering delays of 5, 4, and 3 seconds, respectively, you should see the following output:

```
5
sleeping for 5 second(s)
4
sleeping for 4 second(s)
3
sleeping for 3 second(s)
finished sleeping for 5 second(s)
finished sleeping for 4 second(s)
finished sleeping for 3 second(s)
```

This works, but this approach has a critical flaw. What happens if a message appears on the console while we're typing an input delay time? To test this out, we'll enter a delay time of 3 seconds and then start rapidly pressing 1. Doing this, we'll see something like the following:

```
3
sleeping for 3 second(s)
111111finished sleeping for 3 second(s)
11
```

While we were typing, the message from our delay task prints out, disrupting our input line and forcing it to continue on the next line. In addition, the input buffer is now only `11`, meaning if we press Enter, we'll create a `delay` task for that amount of time, losing the first few pieces of input. This is because, by default, the terminal runs in *cooked* mode. In this mode, the terminal echoes user input to standard output, and also processes special keys, such as Enter and CTRL-C. This issue arises because the `delay` coroutine writes to standard out at the same time the terminal is echoing output, causing a race condition.

There is also a single position on the screen where standard out writes to. This is known as a *cursor* and is much like a cursor you'd see in a word processor. As we enter input, the cursor rests on the line where our keyboard input prints out. This means that any output messages from other coroutines will print on the same line as our input, since this is where the cursor is, causing odd behavior.

To solve these issues, we need a combination of two solutions. The first is to bring the echoing of input from the terminal into our Python application. This will ensure that, while echoing input from the user, we don't write any output messages from other coroutines as we're single-threaded. The second is to move the cursor around the screen when we write output messages, ensuring that we don't write output messages on the same line as our input. We can do these by manipulating the settings of our terminal and using escape sequences.

Terminal raw mode and the read coroutine

Because our terminal is running in cooked mode, it handles echoing user input on `readline` for us outside of our application. How can we bring this processing into our application, so we can avoid the race conditions we saw previously?

The answer is switching the terminal to *raw* mode. In raw mode, instead of the terminal doing buffering, preprocessing, and echoing for us, every keystroke is sent to the application. It is then up to us to echo and preprocess as we'd like. While this means we must do extra work, it also means we have fine-grained control around writing to standard out, giving us the needed power to avoid race conditions.

Python allows us to change the terminal to raw mode but also allows for `cbreak` mode. This mode behaves like raw mode with the difference being that keystrokes like CTRL-C will still be interpreted for us, saving us some work. We can enter raw mode by using the `tty` module and the `setcbreak` function like so:

```
import tty
import sys
tty.setcbreak(sys.stdin)
```

Once we're in `cbreak` mode, we'll need to rethink how we designed our application. The `readline` coroutine will no longer work, as it won't echo any input for us in raw mode. Instead, we'll want to read one character at a time and store it in our own internal buffer, echoing each character typed in. The standard input stream reader we created has a method called `read` that takes in a number of bytes to read from the stream. Calling `read(1)` will read one character at a time, which we can then store in a buffer and echo to standard out.

We now have two pieces of the puzzle to solve this, entering `cbreak` mode and reading one input character at a time, echoing it to standard out. We need to think through how to display the output of the `delay` coroutines, so it won't interfere with our input.

Let’s define a few requirements to make our application more user-friendly and solve the issue with output writing on the same line as input. We’ll then let these requirements inform how we implement things:

1. The user input field should always remain at the bottom of the screen.
2. Coroutine output should start from the top of the screen and move down.
3. When there are more messages than available lines on the screen, existing messages should scroll up.

Given these requirements, how can we display the output from the `delay` coroutine? Given that we want to scroll messages up when there are more messages than available lines, writing directly to standard out with print will prove tricky. Instead of doing this, the approach we’ll take is keeping a *deque* (double-ended queue) of the messages we want to write to standard out. We’ll set the maximum number of elements in the deque to the number of rows on the terminal screen. This will give us the scrolling behavior we want when the deque is full, as items in the back of the deque will be discarded. When a new message is appended to the deque, we’ll move to the top of the screen and redraw each message. This will get us the scrolling behavior we desire without having to keep much information about the state of standard out. This makes our application flow look like the illustration in figure 8.3.

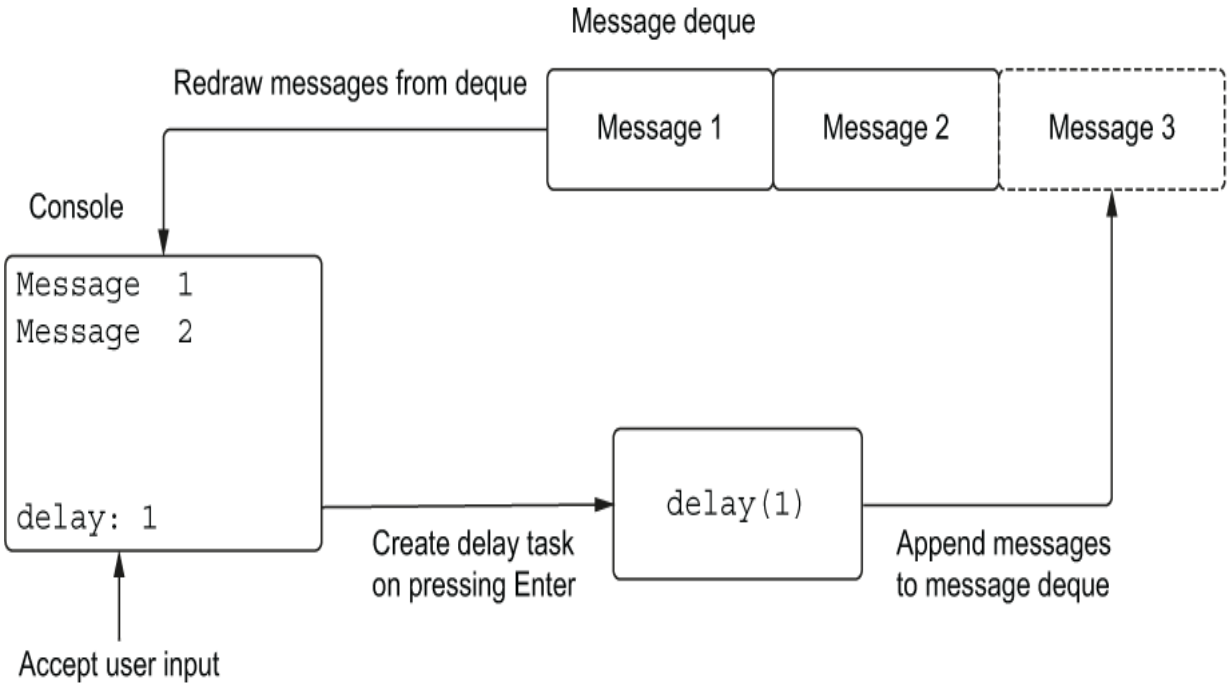

Figure 8.3 The delay console application

Our game plan for the application will then be as follows:

1. Move the cursor to the bottom of the screen, and when a key is pressed, append it to our internal buffer, and echo the keypress to standard out.
2. When the user presses Enter, create a `delay` task. Instead of writing output messages to standard out, we'll append them to a deque with a maximum number of elements equal to the number of rows on the console.
3. Once a message goes into the deque, we'll redraw the output on the screen. We first move the cursor to the top left of the screen. We then print out all messages in the deque. Once we're done, we return the cursor to the input row and column where it was before.

To implement the application in this way, we'll first need to learn how to move the cursor around the screen. We can use ANSI *escape codes* to do this. These are special

codes we can write to standard out performing actions like changing the color of text, moving the cursor up or down, and deleting lines. Escape sequences are first introduced with an escape code; in Python, we can do this by printing `\033` to the console. Many of the escape sequences we'll need to use are introduced by *control sequence introducers*, which are started by printing `\033[.` To better understand this, let's see how to move the cursor to five lines below where it currently is.

```
        sys.stdout.write('\033[5E')
```

This escape sequence starts with the control sequence introducer followed by `5E`. 5 represents the number of rows from the current cursor row we'd like to move down, and E is the code for "move the cursor down this number of lines." Escape sequences are terse and a little hard to follow. In the next listing, we'll create several functions with clear names to explain what each escape code does, and we'll import them in future listings. If you'd like more explanation on ANSI escape sequences and how they work, the Wikipedia article on the subject has great information at https://en.wikipedia.org/wiki/ANSI_escape_code.

Let's think through how we'll need to move the cursor around the screen to figure out which functions we'll need to implement. First, we'll need to move the cursor to the bottom of the screen to accept user input. Then, once the user presses Enter, we'll need to clear any text they have entered. To print coroutine output messages from the top of the screen, we'll need to be able to move to the first line of the screen. We'll also need to save and restore the current position of the cursor, since while we're typing a message from a coroutine it may print a message, meaning we'll need to move it back to the proper spot. We can do these with the following escape code functions:

Listing 8.7 Escape sequence convenience functions

```
import sys
import shutil

def save_cursor_position():
    sys.stdout.write('\0337')


def restore_cursor_position():
    sys.stdout.write('\0338')


def move_to_top_of_screen():
    sys.stdout.write('\033[H')


def delete_line():
    sys.stdout.write('\033[2K')


def clear_line():
    sys.stdout.write('\033[2K\033[0G')


def move_back_one_char():
    sys.stdout.write('\033[1D')

def move_to_bottom_of_screen() -> int:
    _, total_rows = shutil.get_terminal_size()
    input_row = total_rows - 1
```

```
    sys.stdout.write(f'\033[{input_row}E')
    return total_rows
```

Now that we have a set of reusable functions to move the cursor around the screen, let's implement a reusable coroutine for reading standard input one character at a time. We'll use the `read` coroutine to do this. Once we have read a character, we'll write it to standard output, storing the character in an internal buffer. Since we also want to handle a user pressing Delete, we'll watch for the Delete key. When a user presses it, we'll delete the character from the buffer and standard output.

Listing 8.8 Reading input one character at a time

```
import sys
from asyncio import StreamReader
from collections import deque
from chapter_08.listing_8_7 import move_back_one_char, clear


async def read_line(stdin_reader: StreamReader) -> str:
    def erase_last_char():
        move_back_one_char()
        sys.stdout.write(' ')
        move_back_one_char()

    delete_char = b'\x7f'
    input_buffer = deque()
    while (input_char := await stdin_reader.read(1)) != b'\n
        if input_char == delete_char:
            if len(input_buffer) > 0:
                input_buffer.pop()
```

```
                erase_last_char()
                sys.stdout.flush()
        else:
            input_buffer.append(input_char)
            sys.stdout.write(input_char.decode())
            sys.stdout.flush()
    clear_line()
    return b''.join(input_buffer).decode()
```

❶ Convenience function to delete the previous character from standard output

❷ If the input character is backspace, remove the last character.

❸ If the input character is not backspace, append it to the buffer and echo.

Our coroutine takes in a stream reader that we've attached to standard input. We then define a convenience function to erase the previous character from standard output, as we'll need this when a user presses Delete. We then enter a `while` loop reading character by character until the user hits Enter. If the user presses Delete, we remove the last character from the buffer and from standard out. Otherwise, we append it to the buffer and echo it. Once the user presses Enter, we clear the input line and return the contents of the buffer.

Next, we'll need to define the queue where we'll store the messages we want to print to standard out. Since we want to redraw output whenever we append a message, we'll define a class that wraps a deque and takes in a callback awaitable. The callback we pass in will be responsible for redrawing output. We'll also add an `append` coroutine method to our class that will append items to the deque and call the callback with the current set of items in the deque.

Listing 8.9 A message store

```
from collections import deque
from typing import Callable, Deque, Awaitable


class MessageStore:
    def __init__(self, callback: Callable[[Deque], Awaitable
        self._deque = deque(maxlen=max_size)
        self._callback = callback

    async def append(self, item):
        self._deque.append(item)
        await self._callback(self._deque)
```

Now, we have all the pieces to create the application. We'll rewrite our `delay` coroutine to add messages to the message store. Then, in our main coroutine, we'll create a helper coroutine to redraw messages in our deque to standard out. This is the callback we'll pass to our `MessageStore`. Then, we'll use the `read_line` coroutine we implemented earlier to accept user input, creating a delay task when the user hits Enter.

Listing 8.10 The asynchronous delay application

```
import asyncio
import os
import tty
from collections import deque
from chapter_08.listing_8_5 import create_stdin_reader
```

```
from chapter_08.listing_8_7 import *
from chapter_08.listing_8_8 import read_line
from chapter_08.listing_8_9 import MessageStore


async def sleep(delay: int, message_store: MessageStore):
    await message_store.append(f'Starting delay {delay}')
    await asyncio.sleep(delay)
    await message_store.append(f'Finished delay {delay}')


async def main():
    tty.setcbreak(sys.stdin)
    os.system('clear')
    rows = move_to_bottom_of_screen()

    async def redraw_output(items: deque):
        save_cursor_position()
        move_to_top_of_screen()
        for item in items:
            delete_line()
            print(item)
        restore_cursor_position()

    messages = MessageStore(redraw_output, rows - 1)

    stdin_reader = await create_stdin_reader()

    while True:
        line = await read_line(stdin_reader)
        delay_time = int(line)
        asyncio.create_task(sleep(delay_time, messages))
```

```
asyncio.run(main())
```

❶ Append the output messages to the message store.

❷ Callback to move the cursor to the top of the screen; redraw output and move the cursor back.

Running this, you'll be able to create delays and watch input write to the console even as you type. While it is more complicated than our first attempt, we've built an application that avoids the problems writing to standard out that we faced earlier.

What we've built works for the `delay` coroutine, but what about something more real-world? The pieces we've just defined are robust enough we can make more useful applications by reusing them. For example, let's think through how to create a command-line SQL client. Certain queries may take a long time to execute, but we may want to run other queries in the meantime or cancel a running query. Using what we've just built, we can create this type of client. Let's build one using our previous e-commerce product database from chapter 5, where we created a schema with a set of clothing brands, products, and SKUs. We'll create a connection pool to connect to our database, and we'll reuse our code from previous examples to accept and run queries. We'll output basic information about the queries to the console—for now, just the number of rows returned.

Listing 8.11 An asynchronous command-line sql client

```
import asyncio
import asyncpg
```

```
import os
import tty
from collections import deque
from asyncpg.pool import Pool
from chapter_08.listing_8_5 import create_stdin_reader
from chapter_08.listing_8_7 import *
from chapter_08.listing_8_8 import read_line
from chapter_08.listing_8_9 import MessageStore


async def run_query(query: str, pool: Pool, message_store: M
    async with pool.acquire() as connection:
        try:
            result = await connection.fetchrow(query)
            await message_store.append(f'Fetched {len(result
        except Exception as e:
            await message_store.append(f'Got exception {e} f


async def main():
    tty.setcbreak(0)
    os.system('clear')
    rows = move_to_bottom_of_screen()

    async def redraw_output(items: deque):
        save_cursor_position()
        move_to_top_of_screen()
        for item in items:
            delete_line()
            print(item)
        restore_cursor_position()
```

```
    messages = MessageStore(redraw_output, rows - 1)

    stdin_reader = await create_stdin_reader()

    async with asyncpg.create_pool(host='127.0.0.1',
                                   port=5432,
                                   user='postgres',
                                   password='password',
                                   database='products',
                                   min_size=6,
                                   max_size=6) as pool:

        while True:
            query = await read_line(stdin_reader)
            asyncio.create_task(run_query(query, pool, messa


asyncio.run(main())
```

Our code is almost the same as before, with the difference that instead of a `delay` coroutine, we create a `run_query` coroutine. Instead of just sleeping for an arbitrary amount of time, this runs a query the user entered that can take an arbitrary amount of time. This lets us issue new queries from the command line while others are still running; it and also lets us see output from completed ones even as we are typing in new queries.

We now know how to create command-line clients that can handle input while other code executes and writes to the console. Next, we'll learn how to create servers using higher-level asyncio APIs.

eating servers

When we have built servers, such as our echo server, we've created a server socket, bound it to a port and waited for incoming connections. While this works, asyncio lets us create servers at a higher level of abstraction, meaning we can create them without ever worrying about managing sockets. Creating servers this way simplifies the code we need to write with sockets, and as such, using these higher-level APIs is the recommended way to create and manage servers using asyncio.

We can create a server with the `asyncio.start_server` coroutine. This coroutine takes in several optional parameters to configure things such as SSL, but the main parameters we'll be interested in are the `host`, `port`, and `client_connected_cb`. The host and port are like what we've seen before: the address that the server socket will listen for connections. The more interesting piece is `client_connected_cb`, which is either a callback function or a coroutine that will run whenever a client connects to the server. This callback takes in a `StreamReader` and `StreamWriter` as parameters that will let us read from and write to the client that connected to the server.

When we await `start_server`, it will return an `AbstractServer` object. This class lacks many interesting methods that we'll need to use, other than `serve_forever`, which runs the server forever until we terminate it. This class is also an asynchronous context manager. This means we can use an instance of it with `async with` syntax to have the server properly shut down on exit.

To get a handle on creating servers, let's create an echo server again but make it a little more advanced. Instead of just echoing back output, we'll display information about how many other clients are connected. We'll also display information when a

client disconnects from the server. To manage this, we'll create a class we'll call `ServerState` to manage how many users are connected. Once a user connects, we'll add them to the server state and notify other clients that they connected.

Listing 8.12 Creating an echo server with server objects

```
import asyncio
import logging
from asyncio import StreamReader, StreamWriter


class ServerState:

    def __init__(self):
        self._writers = []

    async def add_client(self, reader: StreamReader, writer:
        self._writers.append(writer)
        await self._on_connect(writer)
        asyncio.create_task(self._echo(reader, writer))

    async def _on_connect(self, writer: StreamWriter):
        writer.write(f'Welcome! {len(self._writers)} user(s)
        await writer.drain()
        await self._notify_all('New user connected!\n')

    async def _echo(self, reader: StreamReader, writer: Stre
        try:
            while (data := await reader.readline()) != b'':
                writer.write(data)
                await writer.drain()
```

```
                self._writers.remove(writer)
                await self._notify_all(f'Client disconnected. {l
            except Exception as e:
                logging.exception('Error reading from client.',
                self._writers.remove(writer)

    async def _notify_all(self, message: str):
        for writer in self._writers:
            try:
                writer.write(message.encode())
                await writer.drain()
            except ConnectionError as e:
                logging.exception('Could not write to client
                self._writers.remove(writer)


async def main():
    server_state = ServerState()

    async def client_connected(reader: StreamReader, writer:
     StreamWriter) -> None:
        await server_state.add_client(reader, writer)

    server = await asyncio.start_server(client_connected, '1

    async with server:
        await server.serve_forever()


asyncio.run(main())
```

❶ Add a client to the server state, and create an echo task.

❷ On a new connection, tell the client how many users are online, and notify others of a new user.

❸ Handle echoing user input when a client disconnects, and notify other users of a disconnect.

❹ Helper method to send a message to all other users. If a message fails to send, remove that user.

❺ When a client connects, add that client to the server state.

❻ Start the server, and start serving forever.

When a user connects to our server, our `client_connected` callback responds with a reader and writer for that user, which in turn calls the server state's `add_client` coroutine. In the `add_client` coroutine, we store the `StreamWriter`, so we can send messages to all connected clients and remove it when a client disconnects. We then call `_on_connect`, which sends a message to the client informing them how many other users are connected. In `_on_connect`, we also notify any other connected clients that a new user has connected.

The `_echo` coroutine is similar to what we've done in the past with the twist being that when a user disconnects, we notify any other connected clients that someone disconnected. When running this, you should have a functioning echo server that lets each individual client know when a new user connects and disconnects from the server.