```
server_socket.listen()
server_socket.setblocking(False)

connections = []

try:
    while True:
        try:
            connection, client_address = server_socket.accep
            connection.setblocking(False)
            print(f'I got a connection from {client_address}
            connections.append(connection)
        except BlockingIOError:
            pass

        for connection in connections:
            try:
                buffer = b''

                while buffer[-2:] != b'\r\n':
                    data = connection.recv(2)
                    if not data:
                        break
                    else:
                        print(f'I got data: {data}!')
                        buffer = buffer + data

                print(f"All the data is: {buffer}")
                connection.send(buffer)
            except BlockingIOError:
                pass
```

```
  finally:
      server_socket.close()
```

Each time we go through an iteration of our infinite loop, none of our calls to `accept` or `recv` every block, and we either instantly throw an exception that we ignore, or we have data ready to process and we process it. Each iteration of this loop happens quickly, and we're never dependent on anyone sending us data to proceed to the next line of code. This addresses the issue of our blocking server and allows multiple clients to connect and send data concurrently.

This approach works, but it comes at a cost. The first is code quality. Catching exceptions any time we might not yet have data will quickly get verbose and is potentially error-prone. The second is a resource issue. If you run this on a laptop, you may notice your fan starts to sound louder after a few seconds. This application will always be using nearly 100% of our CPU's processing power (figure 3.3). This is because we are constantly looping and getting exceptions as fast as we can inside our application, leading to a workload that is CPU heavy.

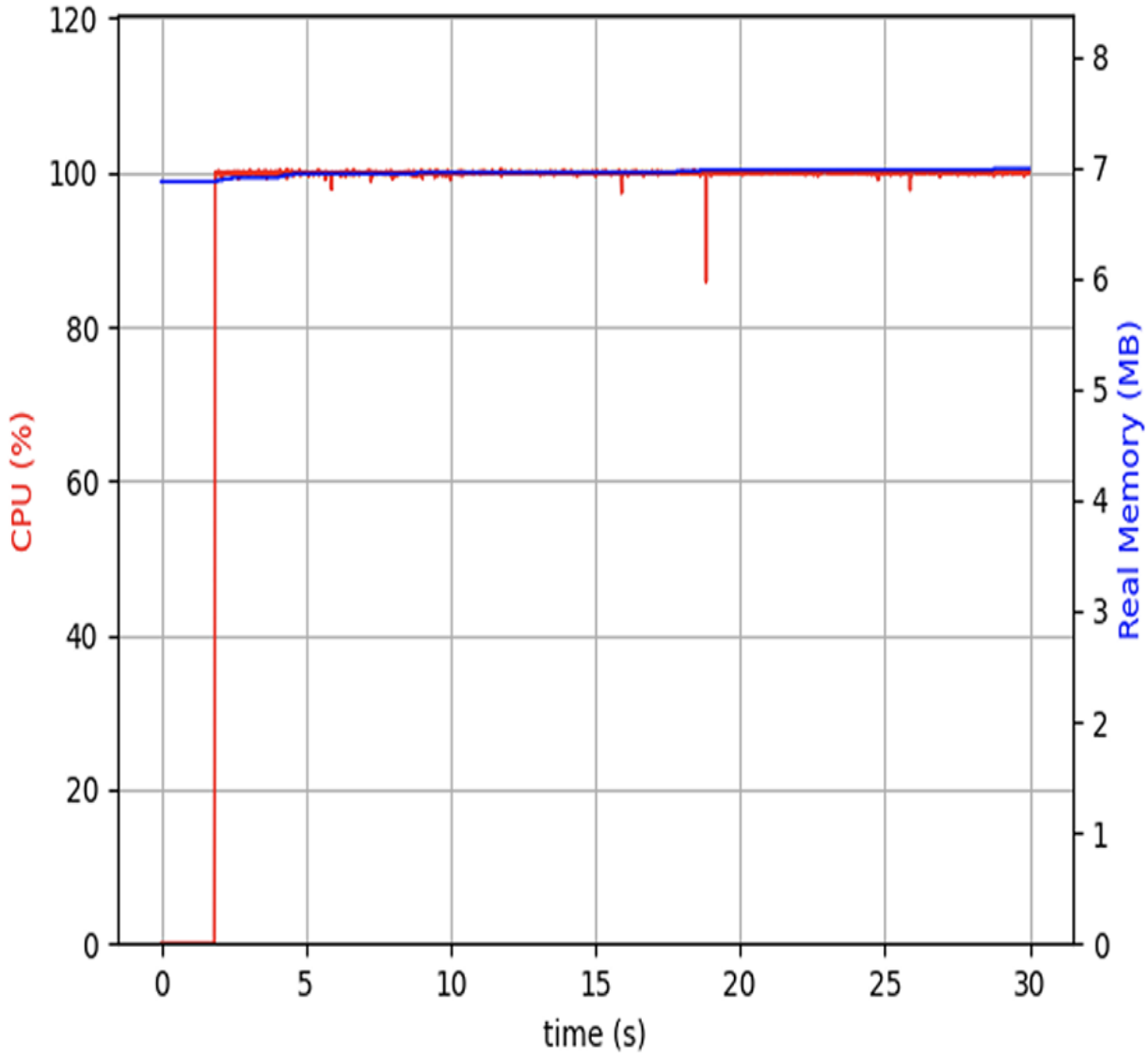

Figure 3.3 When looping and catching exceptions, CPU usage spikes to 100% and stays there.

Earlier, we mentioned operating system-specific event notification systems that can notify us when sockets have data that we can act on. These systems rely on hardware-level notifications and don't involve polling with a `while` loop, as we just did. Python has a library for using this event notification system built in. Next, we'll use this to resolve our CPU utilization issues and build a mini event loop for socket events.

ing the selectors module to build a socket event loop

Operating systems have efficient APIs that let us watch sockets for incoming data and other events built in. While the actual API is dependent on the operating system (kqueue, epoll, and IOCP are a few common ones), all of these I/O notification systems operate on a similar concept. We give them a list of sockets we want to monitor for events, and instead of constantly checking each socket to see if it has data, the operating system tells us explicitly when sockets have data.

Because this is implemented at the hardware level, very little CPU utilization is used during this monitoring, allowing for efficient resource usage. These notification systems are the core of how asyncio achieves concurrency. Understanding how this works gives us a view of how the underlying machinery of asyncio works.

The event notification systems are different depending on the operating system. Luckily, Python's `selectors` module is abstracted such that we can get the proper event for wherever we run our code. This makes our code portable across different operating systems.

This library exposes an abstract base class called `BaseSelector`, which has multiple implementations for each event notification system. It also contains a `DefaultSelector` class, which automatically chooses which implementation is most efficient for our system.

The `BaseSelector` class has important concepts. The first is *registration*. When we have a socket that we're interested in getting notifications about, we register it with the selector and tell it which events we're interested in. These are events such as read and write. Inversely, we can also deregister a socket we're no longer interested in.

The second major concept is *select*. `select` will block until an event has happened, and once it does, the call will return with a list of sockets that are ready for processing along with the event that triggered it. It also supports a timeout, which will return an empty set of events after a specified amount of time.

Given these building blocks, we can create a non-blocking echo server that does not stress our CPU. Once we create our server socket, we'll register it with the default selector, which will listen for any connections from clients. Then, any time someone connects to our server socket, we'll register the client's connection socket with the selector to watch for any data sent. If we get any data from a socket that isn't our server socket, we know it is from a client that has sent data. We then receive that data and write it back to the client. We will also add a timeout to demonstrate that we can have other code execute while we're waiting for things to happen.

Listing 3.7 Using selectors to build a non-blocking server

```
import selectors
import socket
from selectors import SelectorKey
from typing import List, Tuple

selector = selectors.DefaultSelector()

server_socket = socket.socket()
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEA

server_address = ('127.0.0.1', 8000)
server_socket.setblocking(False)
server_socket.bind(server_address)
server_socket.listen()
```

```
selector.register(server_socket, selectors.EVENT_READ)

while True:
    events: List[Tuple[SelectorKey, int]] = selector.select(

    if len(events) == 0:
        print('No events, waiting a bit more!')

    for event, _ in events:
        event_socket = event.fileobj

        if event_socket == server_socket:
            connection, address = server_socket.accept()
            connection.setblocking(False)
            print(f"I got a connection from {address}")
            selector.register(connection, selectors.EVENT_RE
        else:
            data = event_socket.recv(1024)
            print(f"I got some data: {data}")
            event_socket.send(data)
```

❶ Create a selector that will timeout after 1 second.

❷ If there are no events, print it out. This happens when a timeout occurs.

❸ Get the socket for the event, which is stored in the fileobj field.

❹ If the event socket is the same as the server socket, we know this is a connection attempt.

❺ Register the client that connected with our selector.

❻ If the event socket is not the server socket, receive data from the client, and echo it back.

When we run listing 3.7, we'll see "No events, waiting a bit more!" printed roughly every second unless we get a connection event. Once we get a connection, we register that connection to listen for read events. Then, if a client sends us data, our selector will return an event that we have data ready and we can read it with `socket.recv`.

This is fully functioning echo server that supports multiple clients. This server has no issues with blocking, as we only read or write data when we have data to act on. It also has very little CPU utilization as we're using the operating system's efficient event notification system (figure 3.4).

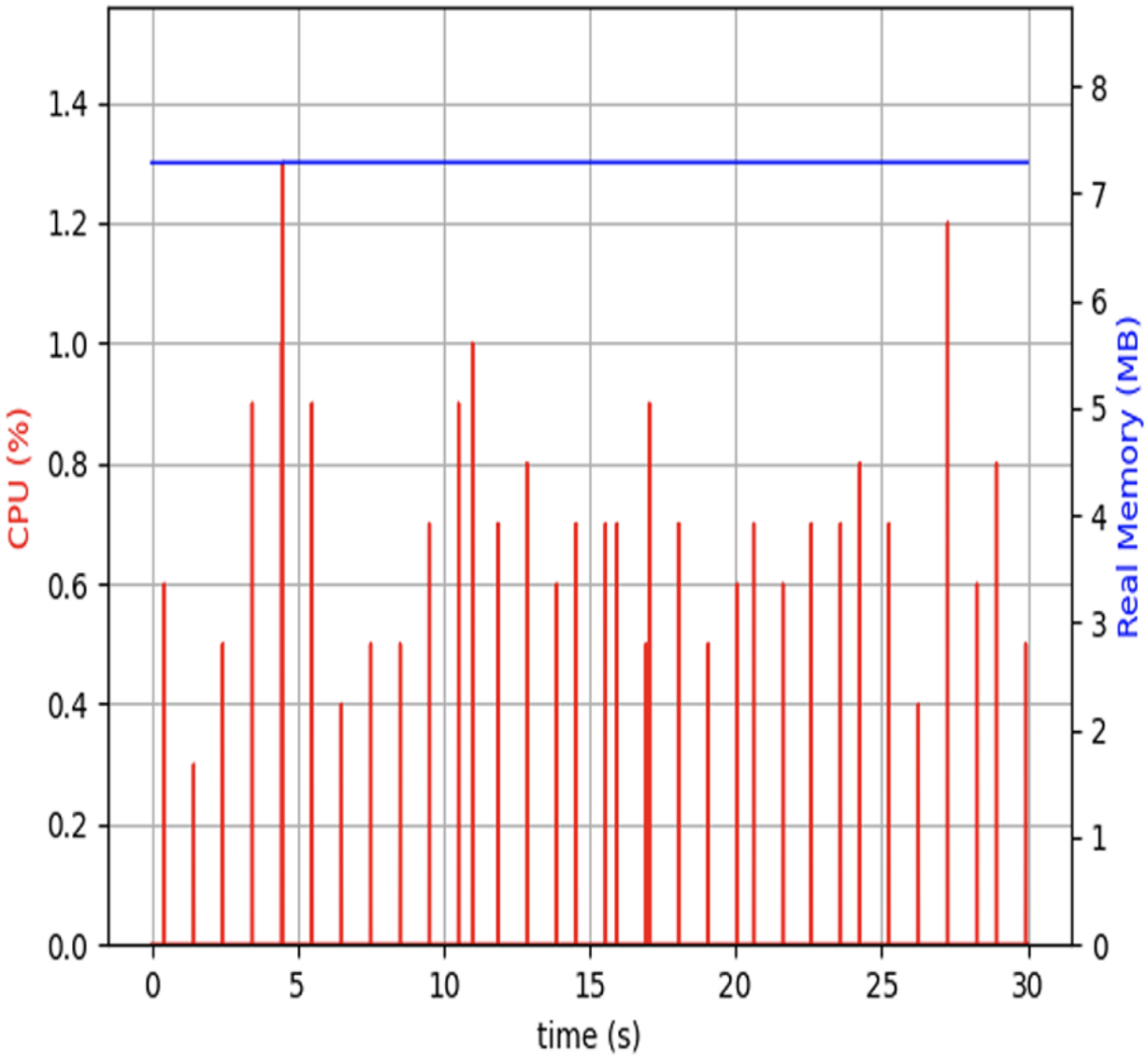

Figure 3.4 CPU graph of the echo server with selectors. Utilization hovers around 0 and 1 percent with this method.

What we've built is akin to a big part of what asyncio's event loop does under the hood. In this case, the events that matter are sockets receiving data. Each iteration of our event loop and the asyncio event loop is triggered by either a socket event happening, or a timeout triggering an iteration of the loop. In the asyncio event loop, when any of these two things happen, coroutines that are waiting to run will do so

until they either complete or they hit the next `await` statement. When we hit an `await` in a coroutine that utilizes a non-blocking socket, it will register that socket with the system's selector and keep track that the coroutine is paused waiting for a result. We can translate this into pseudocode that demonstrates the concept:

```
paused = []
ready = []

while True:
    paused, new_sockets = run_ready_tasks(ready)
selector.register(new_sockets)
    timeout = calculate_timeout()
    events = selector.select(timeout)
    ready = process_events(events)
```

We run any coroutines that are ready to run until they are paused on an `await` statement and store those in the `paused` array. We also keep track of any new sockets we need to watch from running those coroutines and register them with the selector. We then calculate the desired timeout for when we call `select`. While this timeout calculation is somewhat complicated, it is typically looking at things we have scheduled to run at a specific time or for a specific duration. An example of this is `asyncio.sleep`. We then call select and wait for any socket events or a timeout. Once either of those happen, we process those events and turn that into a list of coroutines that are ready to run.

While the event loop we've built is only for socket events, it demonstrates the main concept of using selectors to register sockets we care about, only being woken up when something we want to process happens. We'll get more in-depth with how to construct a custom event loop at the end of this book.

Now, we understand a large part of the machinery that makes asyncio tick. However, if we just use selectors to build our applications, we would resort to implementing our own event loop to achieve the same functionality, as provided by asyncio. To see how to implement this with asyncio, let's take what we have learned and translate it into `async` / `await` code and use an event loop already implemented for us.

echo server on the asyncio event loop

Working with `select` is a bit too low-level for most applications. We may want to have code run in the background while we're waiting for socket data to come in, or we may want to have background tasks run on a schedule. If we were to do this with only selectors, we'd likely build our own event loop, while asyncio has a nicely implemented one ready to use. In addition, coroutines and tasks provide abstractions on top of selectors, which make our code easier to implement and maintain, as we don't need to think about selectors at all.

Now that we have a deeper understanding on how the asyncio event loop works, let's take the echo server that we built in the last section and build it again using coroutines and tasks. We'll still use lower-level sockets to accomplish this, but we'll use asyncio-based APIs that return coroutines to manage them. We'll also add some more functionality to our echo server to demonstrate a few key concepts to illustrate how asyncio works.

Event loop coroutines for sockets

Given that sockets are a relatively low-level concept, the methods for dealing with them are on asyncio's event loop itself. There are three main coroutines we'll want to work with: `sock_accept`, `sock_recv` and `sock_sendall`. These are analogous to the socket methods that we used earlier, except that they take in a socket

as an argument and return coroutines that we can `await` until we have data to act on.

Let's start with `sock_accept`. This coroutine is analogous to the `socket.accept` method that we saw in our first implementation. This method will return a tuple (a data structure that stores an ordered sequence of values) of a socket connection and a client address. We pass it in the socket we're interested in, and we can then `await` the coroutine it returns. Once that coroutine completes, we'll have our connection and address. This socket must be non-blocking and should already be bound to a port:

```
connection, address = await loop.sock_accept(socket)
```

`sock_recv` and `sock_sendall` are called similarly to `sock_accept`. They take in a socket, and we can then `await` for a result. `sock_recv` will `await` until a socket has bytes we can process. `sock_sendall` takes in both a socket and data we want to send and will wait until all data we want to send to a socket has been sent and will return `None` on success:

```
data = await loop.sock_recv(socket)
success = await loop.sock_sendall(socket, data)
```

With these building blocks, we'll be able to translate our previous approaches into one using coroutines and tasks.

Designing an asyncio echo server

In chapter 2, we introduced coroutines and tasks. So when should we use just a coroutine, and when should we wrap a coroutine in a task for our echo server? Let's examine how we want our application to behave to make this determination.

We'll start with how we want to listen for connections in our application. When we are listening for connections, we will only be able to process one connection at a time as `socket.accept` will only give us one client connection. Behind the scenes, incoming connections will be stored in a queue known as the *backlog* if we get multiple connections at the same time, but here, we won't get into how this works.

Since we don't need to process multiple connections concurrently, a single coroutine that loops forever makes sense. This will allow other code to run concurrently while we're paused waiting for a connection. We'll define a coroutine called `listen_for_connections` that will loop forever and listen for any incoming connections:

```
async def listen_for_connections(server_socket: socket,
                                 loop: AbstractEventLoop):
    while True:
        connection, address = await loop.sock_accept(server_
        connection.setblocking(False)
        print(f"Got a connection from {address}")
```

Now that we have a coroutine for listening to connections, how about reading and writing data to the clients who have connected? Should that be a coroutine, or a coroutine we wrap in a task? In this case, we will have multiple connections, each of which could send data to us at any time. We don't want to wait for data from one

connection to block another, so we need to read and write data from multiple clients concurrently. Because we need to handle multiple connections at the same time, creating a task for each connection to read and write data makes sense. On every connection we get, we'll create a task to both read data from and write data to that connection.

We'll create a coroutine named `echo` that is responsible for handling data from a connection. This coroutine will loop forever listening for data from our client. Once it receives data it will then send it back to the client.

Then, in `listen_for_connections` we'll create a new task that wraps our `echo` coroutine for each connection that we get. With these two coroutines defined, we now have all we need to build an asyncio echo server.

Listing 3.8 Building an asyncio echo server

```
import asyncio
import socket
from asyncio import AbstractEventLoop
async def echo(connection: socket,
               loop: AbstractEventLoop) -> None:
    while data := await loop.sock_recv(connection, 1024):
        await loop.sock_sendall(connection, data)


async def listen_for_connection(server_socket: socket,
                                loop: AbstractEventLoop):
    while True:
        connection, address = await loop.sock_accept(server_
        connection.setblocking(False)
```

```
            print(f"Got a connection from {address}")
            asyncio.create_task(echo(connection, loop))


    async def main():
        server_socket = socket.socket(socket.AF_INET, socket.SOC
        server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_RE

        server_address = ('127.0.0.1', 8000)
        server_socket.setblocking(False)
        server_socket.bind(server_address)
        server_socket.listen()

        await listen_for_connection(server_socket, asyncio.get_e

    asyncio.run(main())
```

❶ Loop forever waiting for data from a client connection

❷ Once we have data, send it back to that client.

❸ Whenever we get a connection, create an echo task to listen for client data.

❹ Start the coroutine to listen for connections.

The architecture for the preceding listing looks like figure 3.5. We have one coroutine, `listen_for_connection` , listening for connections. Once a client connects, our coroutine spawns an `echo` task for each client which then listens for data and writes it back out to the client.

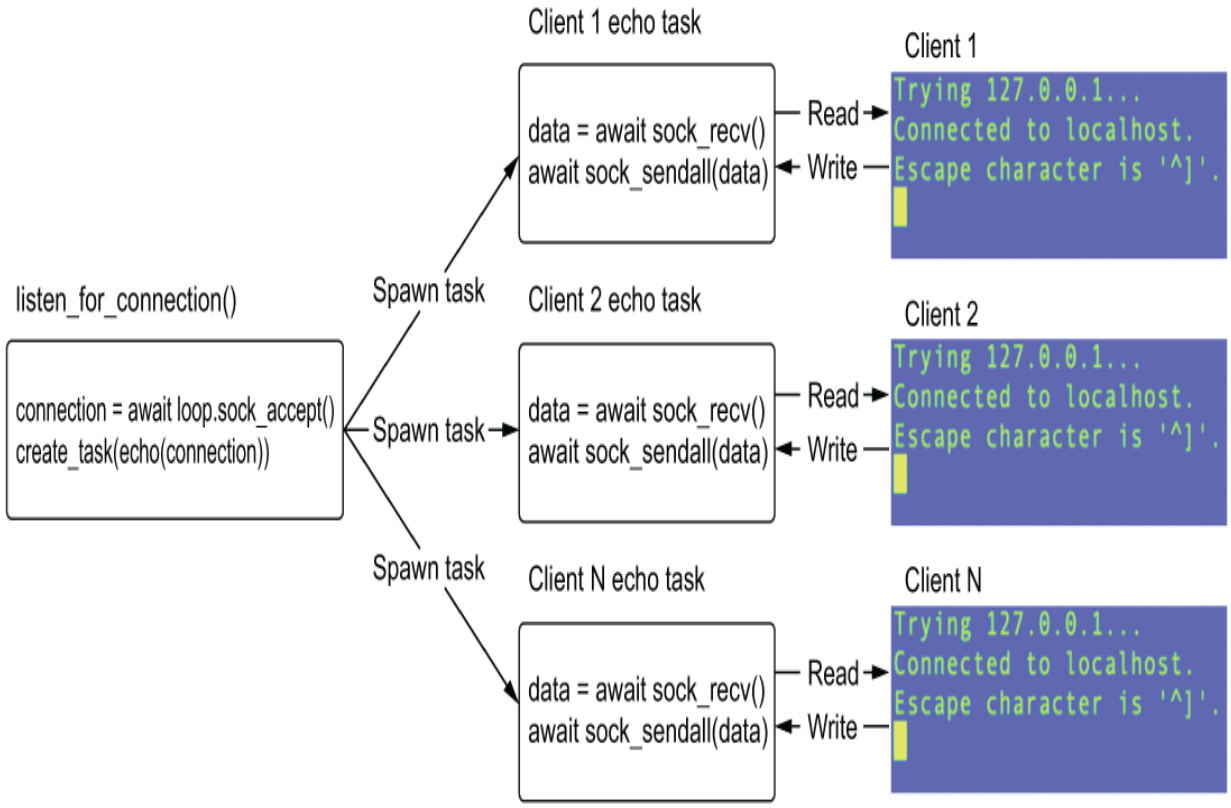

Figure 3.5 The coroutine listening for connections spawns one task per each connection it gets.

When we run this application, we'll be able to connect multiple clients concurrently and send data to them concurrently. Under the hood, this is all using selectors as we saw before, so our CPU utilization remains low.

We've now built a fully functioning echo server entirely using asyncio! So is our implementation error free? It turns out that the way we have designed this echo server does have an issue when our echo task fails that we'll need to handle.

Handling errors in tasks

Network connections are often unreliable, and we may get exceptions we don't expect in our application code. How would our application behave if reading or

writing to a client failed and threw an exception? To test this out, let's change our implementation of `echo` to throw an exception when a client passes us a specific keyword:

```
async def echo(connection: socket,
               loop: AbstractEventLoop) -> None:
    while data := await loop.sock_recv(connection, 1024):
        if data == b'boom\r\n':
            raise Exception("Unexpected network error")
        await loop.sock_sendall(connection, data)
```

Now, whenever a client sends "boom" to us, we will raise an exception and our task will crash. So, what happens when we connect a client to our server and send this message? We will see a traceback with a warning like the following:

```
Task exception was never retrieved
future: <Task finished name='Task-2' coro=<echo() done, defi
Traceback (most recent call last):
  File "asyncio_echo.py", line 9, in echo
    raise Exception("Unexpected network error")
Exception: Unexpected network error
```

The important part here is `Task exception was never retrieved`. What does this mean? When an exception is thrown inside a task, the task is considered done with its result as an exception. This means that no exception is thrown up the call stack. Furthermore, we have no cleanup here. If this exception is thrown, we can't react to the task failing because we never retrieved the exception.

To have the exception reach us, we must use the task in an `await` expression. When we await a task that failed, the exception will get thrown where we perform the await, and the traceback will reflect that. If we don't `await` a task at some point in our application, we run the risk of never seeing an exception that a task raised. While we did see the exception output in the example, which may lead us to think it isn't that big an issue, there are subtle ways we could change our application so that we would never see this message.

As a demonstration of this, let's say that, instead of ignoring the echo tasks we create in `listen_for_connections`, we kept track of them in a list like so:

```
tasks = []

async def listen_for_connection(server_socket: socket,
                                loop: AbstractEventLoop):
    while True:
        connection, address = await loop.sock_accept(server_
        connection.setblocking(False)
        print(f"Got a connection from {address}")
        tasks.append(asyncio.create_task(echo(connection, lo
```

One would expect this to behave in the same way as before. If we send the "boom" message, we'll see the exception printed along with the warning that we never retrieved the task exception. However, this isn't the case, since we'll actually see nothing printed until we forcefully terminate our application!

This is because we've kept a reference around to the task. asyncio can only print this message and the traceback for a failed task when that task is garbage collected. This is because it has no way to tell if that task will be awaited at some other point in the

application and would therefore raise an exception then. Due to these complexities, we'll either need to `await` our tasks or handle all exceptions that our tasks could throw. So how do we do this in our echo server?

The first thing we can do to fix this is wrap the code in our echo coroutine in a `try/catch` statement, log the exception, and close the connection:

```
import logging

async def echo(connection: socket,
               loop: AbstractEventLoop) -> None:
    try:
        while data := await loop.sock_recv(connection, 1024)
            print('got data!')
            if data == b'boom\r\n':
                raise Exception("Unexpected network error")
            await loop.sock_sendall(connection, data)
    except Exception as ex:
        logging.exception(ex)
    finally:
        connection.close()
```

This will resolve the immediate issue of an exception causing our server to complain that a task exception was never retrieved because we handle it in the coroutine itself. It will also properly shut down the socket within the `finally` block, so we won't be left with a dangling unclosed exception in the event of a failure.

It's important to note that this implementation will properly close any connections to clients we have open on application shutdown. Why is this? In chapter 2, we noted

that `asyncio.run` will cancel any tasks we have remaining when our application shuts down. We also learned when we cancel a task, a `CancelledError` is raised whenever we try to `await` it.

The important thing here is noting where that exception is raised. If our task is waiting on a statement such as `await loop.sock_recv`, and we cancel that task, a `CancelledError` is thrown from the `await loop.sock_recv` line. This means that in the above case our `finally` block will be executed, since we threw an exception on an `await` expression when we canceled the task. If we change the exception block to catch and log these exceptions, you will see one `CancelledError` per each task that was created.

We've now handled the immediate issue of handling errors when our echo tasks fail. What if we want to provide some cleanup of any errors or leftover tasks when our application shuts down? We can do this with asyncio's signal handlers.

utting down gracefully

Now, we've created an echo server that handles multiple concurrent connections and also properly logs errors and cleans up when we get an exception. What happens if we need to shut down our application? Wouldn't it be nice if we could allow any in-flight messages to complete before we shut down? We can do this by adding custom shutdown logic to our application that allows any in-progress tasks a few seconds to finish sending any messages they might want to send. While this won't be a production-worthy implementation, we'll learn the concepts around shutting down as well as canceling all running tasks in our asyncio applications.

Signals on Windows

Windows does not support signals. Therefore, this section only applies to Unix-based systems. Windows uses a different system to handle this, that, at the time of writing this book, does not perform with Python. To learn more about how to make this code work in a cross-platform way, see the following answer on Stack Overflow: https://stackoverflow.com/questions/35772001.

Listening for signals

Signals are a concept in Unix-based operating systems for asynchronously notifying a process of an event that occurred at the operating system level. While this sounds very low-level, you're probably familiar with some signals. For instance, a common signal is SIGINT, short for *signal interrupt*. This is triggered when you press CTRL-C to kill a command-line application. In Python, we can often handle this by catching the `KeyboardInterrupt` exception. Another common signal is SIGTERM, short for *signal terminate*. This is triggered when we run the `kill` command on a particular process to stop its execution.

To implement custom shutdown logic, we'll implement listeners in our application for both the SIGINT and SIGTERM signals. Then, in these listeners we'll implement logic to allow any echo tasks we have a few seconds to finish.

How do we listen for signals in our application? The asyncio event loop lets us directly listen for any event we specify with the `add_signal_handler` method. This differs from the signal handlers that you can set in the signal module with the `signal .signal` function in that `add_signal_handler` can safely interact with the event loop. This function takes in a signal we want to listen for and a function that we'll call when our application receives that signal. To demonstrate this, let's look at adding a signal handler that cancels all currently running tasks. asyncio

has a convenience function that returns a set of all running tasks named `asyncio.all_tasks`.

Listing 3.9 Adding a signal handler to cancel all tasks

```
import asyncio, signal
from asyncio import AbstractEventLoop
from typing import Set

from util.delay_functions import delay


def cancel_tasks():
    print('Got a SIGINT!')
    tasks: Set[asyncio.Task] = asyncio.all_tasks()
    print(f'Cancelling {len(tasks)} task(s).')
    [task.cancel() for task in tasks]


async def main():
    loop: AbstractEventLoop = asyncio.get_running_loop()

    loop.add_signal_handler(signal.SIGINT, cancel_tasks)

    await delay(10)


asyncio.run(main())
```

When we run this application, we'll see that our delay coroutine starts right away and waits for 10 seconds. If we press CTRL-C within these 10 seconds we should see

`got a SIGINT!` printed out, followed by a message that we're canceling our tasks. We should also see a `CancelledError` thrown from `asyncio.run(main())`, since we've canceled that task.

Vaiting for pending tasks to finish

In the original problem statement, we wanted to give our echo server's echo tasks a few seconds to keep running before shutting down. One way for us to do this is to wrap all our echo tasks in a `wait_for` and then `await` those wrapped tasks. Those tasks will then throw a `TimeoutError` once the timeout has passed and we can terminate our application.

One thing you'll notice about our shutdown handler is that this is a normal Python function, so we can't run any `await` statements inside of it. This poses a problem for us, since our proposed solution involves `await`. One possible solution is to just create a coroutine that does our shutdown logic, and in our shutdown handler, wrap it in a task:

```
async def await_all_tasks():
    tasks = asyncio.all_tasks()
    [await task for task in tasks]


async def main():
    loop = asyncio.get_event_loop()
    loop.add_signal_handler(signal.SIGINT,
                            lambda: asyncio.create_task(awai
```

An approach like this will work, but the drawback is that if something in `await_all_tasks` throws an exception, we'll be left with an orphaned task that failed and a "exception was never retrieved" warning. So, is there a better way to do this?

We can deal with this by raising a custom exception to stop our main coroutine from running. Then, we can catch this exception when we run the main coroutine and run any shutdown logic. To do this, we'll need to create an event loop ourselves instead of using `asyncio.run`. This is because on an exception `asyncio.run` will cancel all running tasks, which means we aren't able to wrap our echo tasks in a `wait_for`:

```
class GracefulExit(SystemExit):
    pass

def shutdown():
    raise GracefulExit()

loop = asyncio.get_event_loop()

loop.add_signal_handler(signal.SIGINT, shutdown)

try:
    loop.run_until_complete(main())
except GracefulExit:
    loop.run_until_complete(close_echo_tasks(echo_tasks))
finally:
    loop.close()
```

With this approach in mind, let's write our shutdown logic:

```
async def close_echo_tasks(echo_tasks: List[asyncio.Task]):
    waiters = [asyncio.wait_for(task, 2) for task in echo_ta
    for task in waiters:
        try:
            await task
        except asyncio.exceptions.TimeoutError:
            # We expect a timeout error here
            pass
```

In `close_echo_tasks`, we take a list of echo tasks and wrap them all in a `wait_for` task with a 2-second timeout. This means that any echo tasks will have 2 seconds to finish before we cancel them. Once we've done this, we loop over all these wrapped tasks and `await` them. We catch any `TimeoutErrors`, as we expect this to be thrown from our tasks after 2 seconds. Taking all these parts together, our echo server with shutdown logic looks like the following listing.

Listing 3.10 A graceful shutdown

```
import asyncio
from asyncio import AbstractEventLoop
import socket
import logging
import signal
from typing import List


async def echo(connection: socket,
               loop: AbstractEventLoop) -> None:
    try:
```

```
        while data := await loop.sock_recv(connection, 1024)
            print('got data!')
            if data == b'boom\r\n':
                raise Exception("Unexpected network error")
            await loop.sock_sendall(connection, data)
    except Exception as ex:
        logging.exception(ex)
    finally:
        connection.close()

echo_tasks = []


async def connection_listener(server_socket, loop):
    while True:
        connection, address = await loop.sock_accept(server_
        connection.setblocking(False)
        print(f"Got a connection from {address}")
        echo_task = asyncio.create_task(echo(connection, loo
        echo_tasks.append(echo_task)


class GracefulExit(SystemExit):
    pass


def shutdown():
    raise GracefulExit()


async def close_echo_tasks(echo_tasks: List[asyncio.Task]):
    waiters = [asyncio.wait_for(task, 2) for task in echo_ta
```

```
    for task in waiters:
        try:
            await task
        except asyncio.exceptions.TimeoutError:
            # We expect a timeout error here
            pass


async def main():
    server_socket = socket.socket()
    server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_RE

    server_address = ('127.0.0.1', 8000)
    server_socket.setblocking(False)
    server_socket.bind(server_address)
    server_socket.listen()

    for signame in {'SIGINT', 'SIGTERM'}:
        loop.add_signal_handler(getattr(signal, signame), sh
    await connection_listener(server_socket, loop)


loop = asyncio.new_event_loop()

try:
    loop.run_until_complete(main())
except GracefulExit:
    loop.run_until_complete(close_echo_tasks(echo_tasks))
finally:
    loop.close()
```

Assuming we have at least one client connected, if we stop this application with either CTRL-C, or we issue a `kill` command to our process, our shutdown logic will execute. We will see the application wait for 2 seconds, while it allows our echo tasks some time to finish before it stops running.

There are a couple reasons why this is not a production-worthy shutdown. The first is we don't shut down our connection listener while we're waiting for our echo tasks to complete. This means that, as we're shutting down, a new connection could come in and then we won't be able to add a 2-second shutdown. The other problem is that in our shutdown logic we await every echo task we're shutting down and only catch `TimeoutExceptions`. This means that if one of our tasks threw something other than that, we would capture that exception and any other subsequent tasks that may have had an exception will be ignored. In chapter 4, we'll see some asyncio methods for more gracefully handling failures from a group of awaitables.

While our application isn't perfect and is a toy example, we've built a fully functioning server using asyncio. This server can handle many users concurrently—all within one single thread. With a blocking approach we saw earlier, we would need to turn to threading to be able to handle multiple clients, adding complexity and increased resource utilization to our application.

ary

In this chapter, we've learned about blocking and non-blocking sockets and have explored more in depth how the asyncio event loop functions. We've also made our first application with asyncio, a highly concurrent echo server. We have examined how to handle errors in tasks and add custom shutdown logic in our application.

- We've learned how to create simple applications with blocking sockets. Blocking sockets will stop the entire thread when they are waiting for data. This prevents us from achieving concurrency because we can get data from only one client at a time.
- We've learned how to build applications with non-blocking sockets. These sockets will always return right away, either with data because we have it ready, or with an exception stating we have no data. These sockets let us achieve concurrency because their methods never block and return instantly.
- We've learned how to use the selectors module to listen for events on sockets in an efficient manner. This library lets us register sockets we want to track and will tell us when a non-blocking socket is ready with data.
- If we put select in an infinite loop, we've replicated the core of what the asyncio event loop does. We register sockets we are interested in, and we loop forever, running any code we want once a socket has data available to act on.
- We learned how to use asyncio's event loop methods to build applications with non-blocking sockets. These methods take in a socket and return a coroutine which we can then use this in an `await` expression. This will suspend our parent coroutine until the socket has data. Under the hood, this is using the selectors library.
- We've seen how to use tasks to achieve concurrency for an asyncio-based echo server with multiple clients sending and receiving data at the same time. We've also examined how to handle errors within those tasks.
- We've learned how to add custom shutdown logic to an asyncio application. In our case, we decided that when our server shuts down, we'd give it a few seconds for any remaining clients to finish sending data. Using this knowledge, we can add any logic our application needs when it is shutting down.