Asynchronous I/O with Protocol Class Abstractions

Up to this point the examples have all avoided mingling concurrency and I/O operations to focus on one concept at a time. However, switching contexts when I/O blocks is one of the primary use cases for `asyncio`. Building on the concurrency concepts already introduced, this section examines two sample programs implementing a simple echo server and client, similar to the examples used in the `socket` and `socketserver` sections. A client can connect to the server, send some data, and then receive the same data as a response. Each time an I/O operation is initiated, the executing code gives up control to the event loop, allowing other tasks to run until the I/O is ready.

Echo Server

The server starts by importing the modules it needs to set up `asyncio` and `logging`, and then it creates an event loop object.

```
# asyncio_echo_server_protocol.py

import asyncio
import logging
import sys

SERVER_ADDRESS = ('localhost', 10000)

logging.basicConfig(
    level=logging.DEBUG,
    format='%(name)s: %(message)s',
    stream=sys.stderr,
)
log = logging.getLogger('main')

event_loop = asyncio.get_event_loop()
```

It then defines a subclass of `asyncio.Protocol` to handle client communication. The protocol object's methods are invoked based on events associated with the server socket.

```
class EchoServer(asyncio.Protocol):
```

Each new client connection triggers a call to `connection_made()`. The transport argument is an instance of `asyncio.Transport`, which provides an abstraction for doing asynchronous I/O using the socket. Different types of communication provide different transport implementations, all with the same API. For example, there are separate transport classes for working with sockets and for working with pipes to subprocesses. The address of the incoming client is available from the transport through `get_extra_info()`, an implementation-specific method.

```
    def connection_made(self, transport):
        self.transport = transport
        self.address = transport.get_extra_info('peername')
        self.log = logging.getLogger(
            'EchoServer_{}_{}'.format(*self.address)
        )
        self.log.debug('connection accepted')
```

After a connection is established, when data is sent from the client to the server the `data_received()` method of the protocol is invoked to pass the data in for processing. Data is passed as a byte string, and it is up to the application to decode it in an appropriate way. Here the results are logged, and then a response is sent back to the client immediately by calling `transport.write()`.

```
    def data_received(self, data):
        self.log.debug('received {!r}'.format(data))
        self.transport.write(data)
        self.log.debug('sent {!r}'.format(data))
```

Some transports support a special end-of-file indicator ("EOF"). When an EOF is encountered, the `eof_received()` method is

called. In this implementation, the EOF is sent back to the client to indicate that it was received. Because not all transports support an explicit EOF, this protocol asks the transport first whether it is safe to send EOF.

```
    def eof_received(self):
        self.log.debug('received EOF')
        if self.transport.can_write_eof():
            self.transport.write_eof()
```

When a connection is closed, either normally or as the result of an error, the protocol's `connection_lost()` method is called. If there was an error, the argument contains an appropriate exception object. Otherwise it is `None`.

```
    def connection_lost(self, error):
        if error:
            self.log.error('ERROR: {}'.format(error))
        else:
            self.log.debug('closing')
        super().connection_lost(error)
```

There are two steps to starting the server. First the application tells the event loop to create a new server object using the protocol class and the hostname and socket on which to listen. The `create_server()` method is a coroutine, so the results must be processed by the event loop in order to actually start the server. Completing the coroutine produces a `asyncio.Server` instance tied to the event loop.

```
# Create the server and let the loop finish the coroutine before
# starting the real event loop.
factory = event_loop.create_server(EchoServer, *SERVER_ADDRESS)
server = event_loop.run_until_complete(factory)
log.debug('starting up on {} port {}'.format(*SERVER_ADDRESS))
```

Then the event loop needs to be run in order to process events and handle client requests. For a long-running service, the `run_forever()` method is the simplest way to do this. When the event loop is stopped, either by the application code or by signaling the process, the server can be closed to clean up the socket properly, and then the event loop can be closed to finish handling any other coroutines before the program exits.

```
# Enter the event loop permanently to handle all connections.
try:
    event_loop.run_forever()
finally:
    log.debug('closing server')
    server.close()
    event_loop.run_until_complete(server.wait_closed())
    log.debug('closing event loop')
    event_loop.close()
```

Echo Client

Constructing a client using a protocol class is very similar to constructing a server. The code again starts by importing the modules it needs to set up `asyncio` and `logging`, and then creating an event loop object.

```
# asyncio_echo_client_protocol.py

import asyncio
import functools
import logging
import sys

MESSAGES = [
    b'This is the message. ',
    b'It will be sent ',
    b'in parts.',
]
SERVER_ADDRESS = ('localhost', 10000)

logging.basicConfig(
    level=logging.DEBUG,
    format='%(name)s: %(message)s',
    stream=sys.stderr,
)
log = logging.getLogger('main')
```

```
log = logging.getLogger('main')

event_loop = asyncio.get_event_loop()
```

The client protocol class defines the same methods as the server, with different implementations. The class constructor accepts two arguments, a list of the messages to send and a `Future` instance to use to signal that the client has completed a cycle of work by receiving a response from the server.

```
class EchoClient(asyncio.Protocol):

    def __init__(self, messages, future):
        super().__init__()
        self.messages = messages
        self.log = logging.getLogger('EchoClient')
        self.f = future
```

When the client successfully connects to the server, it starts communicating immediately. The sequence of messages is sent one at a time, although the underlying networking code may combine multiple messages into one transmission. When all of the messages are exhausted, an EOF is sent.

Although it appears that the data is all being sent immediately, in fact the transport object buffers the outgoing data and sets up a callback to actually transmit when the socket's buffer is ready to receive data. This is all handled transparently, so the application code can be written as though the I/O operation is happening right away.

```
    def connection_made(self, transport):
        self.transport = transport
        self.address = transport.get_extra_info('peername')
        self.log.debug(
            'connecting to {} port {}'.format(*self.address)
        )
        # This could be transport.writelines() except that
        # would make it harder to show each part of the message
        # being sent.
        for msg in self.messages:
            transport.write(msg)
            self.log.debug('sending {!r}'.format(msg))
        if transport.can_write_eof():
            transport.write_eof()
```

When the response from the server is received, it is logged.

```
    def data_received(self, data):
        self.log.debug('received {!r}'.format(data))
```

And when either an end-of-file marker is received or the connection is closed from the server's side, the local transport object is closed and the future object is marked as done by setting a result.

```
    def eof_received(self):
        self.log.debug('received EOF')
        self.transport.close()
        if not self.f.done():
            self.f.set_result(True)

    def connection_lost(self, exc):
        self.log.debug('server closed connection')
        self.transport.close()
        if not self.f.done():
            self.f.set_result(True)
        super().connection_lost(exc)
```

Normally the protocol class is passed to the event loop to create the connection. In this case, because the event loop has no facility for passing extra arguments to the protocol constructor, it is necessary to create a `partial` to wrap the client class and pass the list of messages to send and the `Future` instance. That new callable is then used in place of the class when calling `create_connection()` to establish the client connection.

```
client_completed = asyncio.Future()

client_factory = functools.partial(
    EchoClient,
    messages=MESSAGES,
```

```
    messages=MESSAGES,
    future=client_completed,
)
factory_coroutine = event_loop.create_connection(
    client_factory,
    *SERVER_ADDRESS,
)
```

To trigger the client to run, the event loop is called once with the coroutine for creating the client and then again with the `Future` instance given to the client to communicate when it is finished. Using two calls like this avoids having an infinite loop in the client program, which likely wants to exit after it has finished communicating with the server. If only the first call was used to wait for the coroutine to create the client, it might not process all of the response data and clean up the connection to the server properly.

```
log.debug('waiting for client to complete')
try:
    event_loop.run_until_complete(factory_coroutine)
    event_loop.run_until_complete(client_completed)
finally:
    log.debug('closing event loop')
    event_loop.close()
```

Output

Running the server in one window and the client in another produces the following output.

```
$ python3 asyncio_echo_client_protocol.py
asyncio: Using selector: KqueueSelector
main: waiting for client to complete
EchoClient: connecting to ::1 port 10000
EchoClient: sending b'This is the message. '
EchoClient: sending b'It will be sent '
EchoClient: sending b'in parts.'
EchoClient: received b'This is the message. It will be sent in parts.'
EchoClient: received EOF
EchoClient: server closed connection
main: closing event loop

$ python3 asyncio_echo_client_protocol.py
asyncio: Using selector: KqueueSelector
main: waiting for client to complete
EchoClient: connecting to ::1 port 10000
EchoClient: sending b'This is the message. '
EchoClient: sending b'It will be sent '
EchoClient: sending b'in parts.'
EchoClient: received b'This is the message. It will be sent in parts.'
EchoClient: received EOF
EchoClient: server closed connection
main: closing event loop

$ python3 asyncio_echo_client_protocol.py
asyncio: Using selector: KqueueSelector
main: waiting for client to complete
EchoClient: connecting to ::1 port 10000
EchoClient: sending b'This is the message. '
EchoClient: sending b'It will be sent '
EchoClient: sending b'in parts.'
EchoClient: received b'This is the message. It will be sent in parts.'
EchoClient: received EOF
EchoClient: server closed connection
main: closing event loop
```

Although the client always sends the messages separately, the first time the client runs the server receives one large message and echoes that back to the client. These results vary in subsequent runs, based on how busy the network is and whether the network buffers are flushed before all of the data is prepared.

```
$ python3 asyncio_echo_server_protocol.py
asyncio: Using selector: KqueueSelector
main: starting up on localhost port 10000
EchoServer ::1 63347: connection accepted
```

```
EchoServer_::1_63347: received b'This is the message. It will be sent in parts.'
EchoServer_::1_63347: sent b'This is the message. It will be sent in parts.'
EchoServer_::1_63347: received EOF
EchoServer_::1_63347: closing

EchoServer_::1_63387: connection accepted
EchoServer_::1_63387: received b'This is the message. '
EchoServer_::1_63387: sent b'This is the message. '
EchoServer_::1_63387: received b'It will be sent in parts.'
EchoServer_::1_63387: sent b'It will be sent in parts.'
EchoServer_::1_63387: received EOF
EchoServer_::1_63387: closing

EchoServer_::1_63389: connection accepted
EchoServer_::1_63389: received b'This is the message. It will be sent '
EchoServer_::1_63389: sent b'This is the message. It will be sent '
EchoServer_::1_63389: received b'in parts.'
EchoServer_::1_63389: sent b'in parts.'
EchoServer_::1_63389: received EOF
EchoServer_::1_63389: closing
```

Synchronization Primitives

Asynchronous I/O Using Coroutines and Streams

Quick Links

Echo Server
Echo Client
Output

This page was last updated 2016-12-26.

Navigation

Synchronization Primitives
Asynchronous I/O Using Coroutines and Streams

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?

This Site

Module Index
Index

Other Writing

- Blog
- The Python Standard Library By Example