Asynchronous I/O Using Coroutines and Streams

This section examines alternate versions of the two sample programs implementing a simple echo server and client, using coroutines and the `asyncio` streams API instead of the protocol and transport class abstractions. The examples operate at a lower abstraction level than the `Protocol` API discussed previously, but the events being processed are similar.

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_coroutine.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 coroutine to handle communication. Each time a client connects, a new instance of the coroutine will be invoked so that within the function the code is only communicating with one client at a time. Python's language runtime manages the state for each coroutine instance, so the application code does not need to manage any extra data structures to track separate clients.

The arguments to the coroutine are `StreamReader` and `StreamWriter` instances associated with the new connection. As with the `Transport`, the client address can be accessed through the writer's method `get_extra_info()`.

```
async def echo(reader, writer):
    address = writer.get_extra_info('peername')
    log = logging.getLogger('echo_{}_{}'.format(*address))
    log.debug('connection accepted')
```

Although the coroutine is called when the connection is established, there may not be any data to read, yet. To avoid blocking while reading, the coroutine uses `await` with the `read()` call to allow the event loop to carry on processing other tasks until there is data to read.

```
    while True:
        data = await reader.read(128)
```

If the client sends data, it is returned from `await` and can be sent back to the client by passing it to the writer. Multiple calls to `write()` can be used to buffer outgoing data, and then `drain()` is used to flush the results. Since flushing network I/O can block, again `await` is used to restore control to the event loop, which monitors the write socket and invokes the writer when it is possible to send more data.

```
        if data:
            log.debug('received {!r}'.format(data))
            writer.write(data)
            await writer.drain()
            log.debug('sent {!r}'.format(data))
```

If the client has sent no data, `read()` returns an empty byte string to indicate that the connection is closed. The server needs to close the socket for writing to the client, and then the coroutine can return to indicate that it is finished.

```
        else:
            log.debug('closing')
            writer.close()
            return
```

There are two steps to starting the server. First the application tells the event loop to create a new server object using the coroutine and the hostname and socket on which to listen. The `start_server()` method is itself 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 = asyncio.start_server(echo, *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()
except KeyboardInterrupt:
    pass
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 coroutine 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_coroutine.py

import asyncio
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')

event_loop = asyncio.get_event_loop()
```

The `echo_client` coroutine takes arguments telling it where the server is and what messages to send.

```
async def echo_client(address, messages):
```

The coroutine is called when the task starts, but it has no active connection to work with. The first step, therefore, is to have the client establish its own connection. It uses `await` to avoid blocking other activity while the `open_connection()` coroutine runs.

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

    log.debug('connecting to {} port {}'.format(*address))
    reader, writer = await asyncio.open_connection(*address)
```

The `open_connection()` coroutine returns `StreamReader` and `StreamWriter` instances associated with the new socket. The next step is to use the writer to send data to the server. As in the server, the writer will buffer outgoing data until the socket is ready or `drain()` is used to flush the results. Since flushing network I/O can block, again `await` is used to restore control to the event loop, which monitors the write socket and invokes the writer when it is possible to send more data.

```
    # This could be writer.writelines() except that
    # would make it harder to show each part of the message
    # being sent.
    for msg in messages:
        writer.write(msg)
        log.debug('sending {!r}'.format(msg))
    if writer.can_write_eof():
        writer.write_eof()
    await writer.drain()
```

Next the client looks for a response from the server by trying to read data until there is nothing left to read. To avoid blocking on an individual `read()` call, `await` yields control back to the event loop. If the server has sent data, it is logged. If the server has sent no data, `read()` returns an empty byte string to indicate that the connection is closed. The client needs to close the socket for sending to the server and then return to indicate that it is finished.

```
    log.debug('waiting for response')
    while True:
        data = await reader.read(128)
        if data:
            log.debug('received {!r}'.format(data))
        else:
            log.debug('closing')
            writer.close()
            return
```

To start the client, the event loop is called with the coroutine for creating the client. Using `run_until_complete()` avoids having an infinite loop in the client program. Unlike in the protocol example, no separate future is needed to signal when the coroutine is finished, because `echo_client()` contains all of the client logic itself and it does not return until it has received a response and closed the server connection.

```
try:
    event_loop.run_until_complete(
        echo_client(SERVER_ADDRESS, MESSAGES)
    )
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_coroutine.py
asyncio: Using selector: KqueueSelector
echo_client: connecting to localhost port 10000
echo_client: sending b'This is the message. '
echo_client: sending b'It will be sent '
echo_client: sending b'in parts.'
echo_client: waiting for response
echo_client: received b'This is the message. It will be sent in parts.'
echo_client: closing
main: closing event loop

$ python3 asyncio_echo_client_coroutine.py
asyncio: Using selector: KqueueSelector
```

```
echo_client: connecting to localhost port 10000
echo_client: sending b'This is the message. '
echo_client: sending b'It will be sent '
echo_client: sending b'in parts.'
echo_client: waiting for response
echo_client: received b'This is the message. It will be sent in parts.'
echo_client: closing
main: closing event loop

$ python3 asyncio_echo_client_coroutine.py
asyncio: Using selector: KqueueSelector
echo_client: connecting to localhost port 10000
echo_client: sending b'This is the message. '
echo_client: sending b'It will be sent '
echo_client: sending b'in parts.'
echo_client: waiting for response
echo_client: received b'This is the message. It will be sent '
echo_client: received b'in parts.'
echo_client: closing
main: closing event loop
```

Although the client always sends the messages separately, the first two times 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_coroutine.py
asyncio: Using selector: KqueueSelector
main: starting up on localhost port 10000
echo_::1_64624: connection accepted
echo_::1_64624: received b'This is the message. It will be sent in parts.'
echo_::1_64624: sent b'This is the message. It will be sent in parts.'
echo_::1_64624: closing

echo_::1_64626: connection accepted
echo_::1_64626: received b'This is the message. It will be sent in parts.'
echo_::1_64626: sent b'This is the message. It will be sent in parts.'
echo_::1_64626: closing

echo_::1_64627: connection accepted
echo_::1_64627: received b'This is the message. It will be sent '
echo_::1_64627: sent b'This is the message. It will be sent '
echo_::1_64627: received b'in parts.'
echo_::1_64627: sent b'in parts.'
echo_::1_64627: closing
```

Asynchronous I/O with Protocol Class Abstractions

Using SSL

Quick Links
Echo Server
Echo Client
Output

This page was last updated 2016-12-26.

Navigation
Asynchronous I/O with Protocol Class Abstractions
Using SSL

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