The Networks of Reinvention

DAVID BEAZLEY

David Beazley is an open source developer and author of the *Python Essential Reference* (4th Edition, Addison-Wesley, 2009). He is also known as the creator of Swig (http://www.swig.org) and Python Lex-Yacc (http://www.dabeaz.com/ply.html). Beazley is based in Chicago, where he also teaches a variety of Python courses. dave@dabeaz.com

Network programming has been a part of Python since its earliest days. Not only are there a wide variety of standard library modules ranging from low-level socket programming to various aspects of Web programming, there are a large number of third-party packages that simplify various tasks or provide different kinds of I/O models. As the network evolves and new standards emerge, though, one can't help but wonder whether the existing set of networking libraries are up to the task. For example, if you start to look at technologies such as websockets, HTTP/2, or coroutines, the whole picture starts to get rather fuzzy. Is there any room for innovation in this space? Or must one be resigned to legacy approaches from its past. In this article, I spend some time exploring some projects that are at the edge of Python networking. This includes my own Curio project as well as some protocol implementation libraries, including hyper-h2 and h11.

Introduction

For the past year, I've been working on a side project, Curio, that implements asynchronous I/O in Python using coroutines and the newfangled `async` and `await` syntax added in Python 3.5 [1]. Curio allows you to write low-level network servers almost exactly like you would with threads. Here is an example of a simple TCP Echo server:

```
from curio import run, spawn
from curio.socket import *

async def echo_server(address):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.setsockopt(SOL_SOCKET, SO_REUSEADDR, True)
    sock.bind(address)
    sock.listen(1)
    while True:
        client, addr = await sock.accept()
        print('Connection from', addr)
        await spawn(echo_handler(client))

async def echo_handler(client):
    async with client:
        while True:
            data = await client.recv(100000)
            if not data:
                break
            await client.sendall(data)
    print('Connection closed')

if __name__ == '__main__':
    run(echo_server(('',25000)))
```

If you haven't been looking at Python 3 recently, there is a certain risk that this example will shatter your head. Functions defined with `async` are coroutines. Coroutines can be called from other coroutines using the special `await` qualifier. Admittedly, it looks a little funny, but if you run the above code, you'll find that it has no problem serving up thousands of concurrent connections even though threads are nowhere to be found. You'll also find it to be rather fast. This article isn't about the details of my pet project though.

Here's the real issue—even though I've written an experimental networking library, where do I take it from here? Most developers don't want to program directly with sockets. They want to use higher-level protocols such as HTTP. However, how do I support that? Because of the use of coroutines, `async`, and `await`, none of Python's standard libraries are going to work (coroutines are "all-in"—to use them they must be used everywhere). I could look at code in third-party libraries such as Twisted, Tornado, or Gevent, but they each implement HTTP in their own way that can't be applied to my problem in isolation. I'm left with few choices except to reimplement HTTP from scratch—and that's probably the last thing the world needs. Another custom implementation of HTTP, that is.

It turns out that other developers in the Python world have been pondering such problems. For example, how are all of the different networking libraries and frameworks going to go about supporting the new HTTP/2 protocol? There is no support for this in the standard library, and the protocol itself is significantly more complicated than HTTP/1.1. Is every library going to reimplement the protocol from scratch? If so, how many thousands of hours are going to be wasted sorting out all of the bugs and weird corner cases? How many developers even understand HTTP/2 well enough to do it? I am not one of those developers.

At PyCon 2016, Cory Benfield gave a talk, "Building Protocol Libraries the Right Way," in which he outlined the central problem with I/O libraries [2]. In a nutshell, these libraries mix I/O and protocol parsing together in a way that makes it nearly impossible for anyone to reuse code. As a result, everyone ends up reinventing the wheel. It causes a lot of problems as everyone makes the same mistakes, and there is little opportunity to reap the rewards of having a shared effort. To break out of that cycle, an alternative approach is to decouple protocol parsing entirely from I/O. Cory has done just that with the hyper-h2 project for HTTP/2 [3]. Nathaniel Smith, inspired by the idea, has taken a similar approach for the h11 library, which provides a standalone HTTP/1.1 protocol implementation [4].

The idea of decoupling network protocols from I/O is interesting. It's something that could have huge benefits for Python development down the road. However, it's also pretty experimental. Given the experimental nature of my own Curio project, I thought it might be interesting to put some of these ideas about protocols to the test to see whether they can work in practice. Admittedly, this is a bit self-serving, but Curio has the distinct advantage of being incompatible with everything. There is no other option than going it alone—so maybe these protocol libraries can make life a whole heck of a lot easier. Let's find out.

Reinventing Requests

As a test, my end goal is to reinvent a tiny portion of the popular requests library so that it works with coroutines [5]. Requests is a great way to fetch data from the Web, but it only works in a purely synchronous manner. For example, if I wanted to download a Web page, I'd do this:

```
>>> import requests
>>> r = requests.get('https://httpbin.org')
>>> r.status_code
200
>>> r.headers
{'connection': 'keep-alive', 'content-type': 'text/html;
charset=utf-8', 'access-control-allow-origin': '*',
'access-control-allow-credentials': 'true',
 'server': 'nginx', 'content-length': '12150',
  'date': 'Tue, 28 Jun 2016 14:58:04 GMT'}
>>> r.content
b'<!DOCTYPE html>\n<html>\n<head>\n  ...'
>>>
```

Under the covers, requests uses the `urllib3` library for HTTP handling [6]. To adapt requests to coroutines, one approach might be to port it and `urllib3` to coroutines—a task that would involve identifying and changing every line of code related to I/O. It's probably not an impossible task, but it would require a lot of work. Let's not do that.

Understanding the Problem

The core of the problem is that existing I/O libraries mix protocol parsing and I/O together. For example, if I wanted to establish an HTTP client connection, I could certainly use the built-in `http.client` library like this:

```
>>> import http.client
>>> c = http.client.HTTPConnection('httpbin.org')
>>> c.request('GET', '/')
>>> r = c.getresponse()
>>> r.status
200
>>> data = r.read()
>>>
```

However, this code performs both the task of managing the protocol and the underlying socket I/O. There is no way to replace the I/O with something different or to extract just the code related to the protocol itself. I'm stuck.

Using a Protocol Parser

Let's look at the problem in a different way using the h11 module. First, you'll want to install h11 directly from its GitHub page [4]. h11 is an HTTP/1.1 protocol parser. It performs no I/O itself. Let's play with it to see what this means:

```
>>> import h11
>>> conn = h11.Connection(our_role=h11.CLIENT)
>>>
```

At first glance, this is going to look odd. We created a "Connection," but there is none of the usual network-related flavor to it. No host name. No port number. What exactly is this connected to? As it turns out, it's not "connected" to anything. Instead, it is an abstract representation of an HTTP/1.1 connection. Let's make a request on the connection:

```
>>> request = h11.Request(method='GET', target='/',
headers=[('Host', 'httpbin.org')])
>>> bytes_to_send = conn.send(request)
>>> bytes_to_send
b'GET / HTTP/1.1\r\nhost: httpbin.org\r\n\r\n'
>>>
```

Notice that the `send()` method of the connection returned a byte string? These are the bytes that need to be sent someplace. To do that, you are on your own. For example, you could create a socket and do it manually:

```
>>> from socket import *
>>> sock = socket(AF_INET, SOCK_STREAM)
>>> sock.connect(('httpbin.org', 80))
>>> sock.sendall(bytes_to_send)
>>>  # Send end of message to mark the end of the request
>>> sock.sendall(conn.send(h11.EndOfMessage()))
>>>
```

Upon sending the request over a socket, the server will respond. Let's read a tiny fragment of the response:

```
>>> data = sock.recv(10)    # Read 10 bytes
>>> data
b'HTTP/1.1 2'
>>>
```

This is just a fragment of what's being sent to us, but let's feed it into the HTTP/1.1 connection object and see how it responds:

```
>>> conn.receive_data(data)
>>> conn.next_event()
NEED_DATA
>>>
```

The `receive_data()` call feeds incoming data to the connection. After you do that, the `next_event()` call will tell you more about what has been received. In this case, the `NEED_DATA` response means that incomplete data has been received (the connection has not received a full response). You have to read more data. Let's do that.

```
>>> data = sock.recv(1000)
>>> conn.receive_data(data)
>>> conn.next_event()
Response(status_code=200, headers=[(b'server', b'nginx'),
(b'date', b'Tue, 28 Jun 2016 15:31:50 GMT'),
(b'content-type', b'text/html; charset=utf-8'),
(b'content-length', b'12150'), (b'connection', b'keep-alive'),
(b'access-control-allow-origin', b'*'),
(b'access-control-allow-credentials', b'true')], http
_version=b'1.1')
>>>
```

Aha! Now we see the response and some headers. There is a very important but subtle aspect to all of this. When using the protocol library, you just feed it byte fragments without worrying about low-level details (e.g., splitting into lines, worrying about the header size, etc.). It just works. If more data is needed, the library lets you know with the `NEED_DATA` event.

After getting the basic response, you can move on to reading data. To do that, call `conn.next_event()` again.

```
>>> conn.next_event()
Data(data=bytearray(b"<!DOCTYPE html>\n<html>\n<head>\n..."))
>>>
```

This is a block of data found in the last read of the underlying sock. You can continue to call `conn.receive_data()` and `conn.next_event()` to process the entire stream.

```
>>> conn.next_event()
NEED_DATA
>>> data = sock.recv(100000)    # Get more data
>>> conn.receive_data(data)
>>> conn.next_event()
Data(data=bytearray(b'p h3+pre {margin-top:5px}\n .mp img ...')
>>> conn.next_event()
EndOfMessage(headers=[])
>>>
```

The `EndOfMessage` event means that the end of the received data has been reached. At this point, you're done. You've made a request and completely read the response.

The really critical part of this example is that the HTTP/1.1 protocol handling is completely decoupled from the underlying socket. Yes, bytes are sent and received on the socket, but none of that is mixed up with protocol handling. This means that you can make the I/O as crazy as you want to make it.

A More Advanced Example: HTTP/2

The hyper-h2 library implements a similar idea as the previous example, but for the HTTP/2 protocol [3]. HTTP/2 is a much more complicated beast, but here is an example of what it looks like to drive it with h2. This program makes a request to `https://http2bin.org` and prints out the events that are returned.

```
from socket import *
import ssl

# Establish a socket connection with TLS
sock = socket(AF_INET, SOCK_STREAM)
sock.connect(('http2bin.org', 443))
ssl_context = ssl.create_default_context()
sock = ssl_context.wrap_socket(sock, server_hostname=
      'http2bin.org')

# Create HTTP/2 connection
import h2.connection
import h2.events

conn = h2.connection.H2Connection(client_side=True)
conn.initiate_connection()
sock.sendall(conn.data_to_send())

# Get the response to the initial connection request
data = sock.recv(10000)
events = conn.receive_data(data)
for evt in events:
   print(evt)

# Send out a request
request_headers = [
   (':method', 'GET'),
   (':authority', 'http2bin.org'),
   (':scheme', 'https'),
   (':path', '/')
]
conn.send_headers(1, request_headers, end_stream=True)
sock.sendall(conn.data_to_send())

# Read all responses until stream is ended
done = False
while not done:
   data = sock.recv(100000)
   events = conn.receive_data(data)
   for evt in events:
      print(evt)
      if isinstance(evt, h2.events.StreamEnded):
         done = True
```

If you run this, you should get output like this:

```
<RemoteSettingsChanged changed_settings:{3:
ChangedSetting(setting=3,
original_value=None, new_value=100),
4: ChangedSetting(setting=4, original_value=65535,
new_value=16777216)}>
<SettingsAcknowledged changed_settings:{}>
<ResponseReceived stream_id:1, headers:[(':status', '200'),
('server', 'h2o/1.7.0'),
('date', 'Tue, 28 Jun 2016 16:12:48 GMT'), ('content-type',
'text/html; charset=utf-8'),
('access-control-allow-origin', '*'),
('access-control-allow-credentials', 'true'),
('x-clacks-overhead', 'GNU Terry Pratchett'),
('content-length', '11729')]>
<DataReceived stream_id:1, flow_controlled_length:11729,
data:3c21444f43545950452068746d6c3e0a3c68746d>
<StreamEnded stream_id:1>
```

This probably looks fairly low-level, but the key part of it is that all of the protocol handling and the underlying I/O are decoupled. There is a TLS socket on which low-level `sendall()` and `recv()` operations are performed. However, all interpretation of the data is performed by the `H2Connection` instance. I don't have to worry about the precise nature of those details.

Putting It All Together

Looking at the previous two examples, you might wonder who these libraries are for? Yes, they handle low-level protocol details, but they provide none of the convenience of higher-level libraries. In the big picture, the main beneficiaries are the authors of I/O libraries, such as myself. To see how this might work, here is bare-bones implementation of a requests-like clone using coroutines and Curio:

```
# example.py

from urllib.parse import urlparse
import socket
import curio
import h11

class Response(object):
   def __init__(self, url, status_code, headers, conn, sock):
      self.url = url
      self.status_code = status_code
      self.headers = headers
      self._conn = conn
      self._sock = sock
```

```
    # Asynchronous iteration for a streaming response
    async def __aiter__(self):
        return self

    async def __anext__(self):
        while True:
            evt = self._conn.next_event()
            if evt == h11.NEED_DATA:
                data = await self._sock.recv(100000)
                self._conn.receive_data(data)
            else:
                break

        if isinstance(evt, h11.Data):
            return evt.data
        elif isinstance(evt, h11.EndOfMessage):
            raise StopAsyncIteration
        else:
            raise RuntimeError('Bad response %r' % evt)

    # Asynchronous property for getting all content
    @property
    async def content(self):
        if not hasattr(self, '_content'):
            chunks = []
            async for data in self:
                chunks.append(data)
            self._content = b''.join(chunks)
        return self._content

async def get(url, params=None):
    url_parts = urlparse(url)
    if ':' in url_parts.netloc:
        host, _, port = url_parts.netloc.partition(':')
        port = int(port)
    else:
        host = url_parts.netloc
        port = socket.getservbyname(url_parts.scheme)

    # Establish a socket connection
    use_ssl = (url_parts.scheme == 'https')
    sock = await curio.open_connection(host,
        port,
        ssl=use_ssl,
        server_hostname=host if use_ssl else None)

    # Make a HTTP/1.1 protocol connection
    conn = h11.Connection(our_role=h11.CLIENT)
    request = h11.Request(method='GET',
                          target=url_parts.path,
                          headers=[('Host', host)])
    bytes_to_send = conn.send(request) + conn.send(h11.
EndOfMessage())
    await sock.sendall(bytes_to_send)

    # Read the response
    while True:
        evt = conn.next_event()
        if evt == h11.NEED_DATA:
            data = await sock.recv(100000)
            conn.receive_data(data)
        elif isinstance(evt, h11.Response):
            break
        else:
            raise RuntimeError('Unexpected %r' % evt)

    resp = Response(url, evt.status_code, dict(evt.headers),
conn, sock)
    return resp
```

Using this code, here is an example that makes an HTTP request and retrieves the result using coroutines:

```
import example
import curio

async def main():
    r = await example.get('http://httpbin.org/')
    print(r.status)
    print(r.headers)
    print(await r.content)

curio.run(main())
```

Or, if you want to stream the response back in chunks, you can do this:

```
async def main():
    r = await example.get('http://httpbin.org/bytes:100000')
    with open('out.bin', 'wb') as f:
        async for chunk in r:
            print('Writing', len(chunk))
            f.write(chunk)

curio.run(main())
```

Fleshing out this code to something more worthy of production would obviously require more work. However, it didn't take a lot of code to make a bare-bones implementation. Nor was it necessary to worry too much about underlying mechanics of the HTTP/1.1 protocol. That's pretty neat.

The Future

If it catches on, the whole idea of breaking out network protocols into libraries decoupled from I/O would be an interesting direction for Python. There are certain obvious benefits such as eliminating the need for every I/O library to implement its own protocol handling. It also makes experimental work in I/O handling much more feasible as it is no longer necessary to implement all of that code from scratch.

Although Cory Benfield's work on HTTP/2 handling may be the most visible, one can't help wonder whether a similar approach to other protocols might be useful. For example, isolating the protocols used for database engines (MySQL, Postgres), Redis, ZeroMQ, DNS, FTP, and other network-related technologies might produce interesting results if it enabled those protocols to be used in different kinds of I/O libraries. It seems that there might be opportunities here.

References

[1] Curio project: https://curio.readthedocs.io.

[2] C. Benfield, "Building Protocol Libraries the Right Way," PyCon 2016: https://www.youtube.com/watch?v=7cC3_jGwl_U.

[3] Hyper-h2 Project: http://python-hyper.org/projects/h2/.

[4] H11 Project: https://github.com/njsmith/h11.

[5] Requests Project: https://docs.python-requests.org.

[6] Urllib3 Project: https://urllib3.readthedocs.io.