Johns Hopkins University
EN.601.444/644 Network Security
Fall 2017
Seth James Nielson
Lab #1c
Revision 1.2 (9/10/2017:2003)

ASSIGNED: 9/6/2017
DUE: Midnight 9/11/2017

Introduction

It's time to start setting up some networking! The goal of this lab is to actually implement the protocol you described in lab 1[a] and using the packets you tested in 1[b].

As a quick reminder, here are the mini-labs we are working on in phase 1:

- ~~1[a] – Design a simple protocol with at least three packet types~~
- ~~1[b] – Install the Playground framework and implement the packets from 1[a]~~
- 1[c] (this lab) – Create a TCP/IP client server implementation of your protocol
- 1[d] – Convert your protocol from TCP/IP to Playground Network
- 1[e] – Install a simple "pass-through" network layer for the Playground Network.

And the due dates are

- ~~1[a] is due 9/4~~
- ~~1[b] is due 9/6~~
- 1[c] is due 9/11
- 1[d] is due 9/13
- 1[e] is due 9/18

This will be a programming assignment in python.

An Overview of Python 3's Asyncio

We've already done some examples in class of connecting via sockets. But in this class, we are going to use Python's asyncio library instead of (direct) socket access. Sockets are a form of "synchronous" communication. That means that the code moves in the typical flow from beginning to end.

Asynchronous communication works using an "event loop" and code is called when there's an event that triggers it. So, instead of calling "socket.recv" to wait for data, you will create a

"Protocol" class with a "data_received" method. You will register this protocol instance with the asyncio's event loop and wait for this loop to tell your protocol when data is ready.

The first thing you should do for this lab is read Python 3's documentation:

- https://docs.python.org/3/library/asyncio.html
- https://docs.python.org/3/library/asyncio-protocol.html
- https://docs.python.org/3/library/asyncio-eventloop.html

Read these pages a couple of times. Look at the examples. Try them yourself. Get comfortable with them. You're going to be using this framework for the rest of this class!

Although I want you to read this documentation as your primary source, here are a few key points ***as well as a few items specific to our class you should pay attention to***:

The Protocol Class:

The most basic element of most of your networking throughout this semester will be a "Protocol" class. A Protocol class has three critical methods:

- connection_made(self, transport)
- connection_lost(self, exc)
- data_received(self, data)

An instance of a protocol will be hooked up to a network connection. When the network connection is established, the system will call the protocol's connection_made. At any point afterwards when data arrives, the protocol's data_received will be called. When the connection is closed, the connection_lost method is called.

You should understand that data_received will only be called after connection_made so you can use connection_made to do setup you need for your data handling methods. Similarly, connection_lost is called once no more data will be sent to data_received.

So, here's a simple Echo Server Protocol:

```
class EchoServer(Protocol):
  def connection_made(self, transport):
    print("Echo Server Connected to Client")
    self.transport =transport

  def data_received(self, data):
    self.transport.write(data)

  def connection_lost(self, exc):
```

```
    print("Echo Server Connection Lost because {}".format(exc))
```

So, in this simple example, the protocol is given a transport in connection_made that it saves in self.transport. Then, whenever it receives data thereafter in data_received, it simply writes it back using self.transport.write.

The Transport:

You'll notice that your protocol class has an input method (i.e., data_received()), but no corresponding output method. Instead, the "transport" object passed to the protocol in connection_made provides a "write" method for sending data back out.

It should be obvious that you don't get a transport until connection_made. Before that, what would transport even mean?

Using Asyncio's Event Loop:

You cannot use a protocol directly, nor can you create your own Transport (well… you *could*… but don't try!). Instead, you will use asyncio's event loop to attach a protocol to either a server or an outbound TCP connection.

As described in the documentation, here is how you setup a listening protocol (e.g., a server):

```
import asyncio
…
loop = asyncio.get_event_loop()
loop.create_server(lambda: MyProtocolClass(), port=8000)
```

Note that the first argument to create_server is "lambda: MyProtocolClass." If you aren't familiar with lambda's, it is Python's anonymous function creator. A lambda function evaluates to the value of a single expression. For example:

```
f1 = lambda: 3
x = f1()    # x is now 3

f2 = lambda: "test"
y = f2()   # y is now "test"

f3 = lambda x,y: x+y
z = f3(10,5)    # z is now 15

f4 = lambda: MyProtocolClass()
a = f4()    # a is now an instance of the MyProtocolClass
```

The create_server operation calls the first parameter a “factory.” The purpose of the factory is to construct a new protocol instance for each incoming connection (remember that for a server, there will be multiple connections per port!). So the anonymous function above will be called each time a new connection arrives producing a separate protocol instance each time. Sometimes it makes sense to create a more complicated factory, but many times a simple lambda like this is sufficient.

To create an outbound connection, use the asyncio loop’s create_connection method.

```
loop.create_connection(lambda: MyClientProtocol(),
host=”127.0.0.1”, port=8000)
```

You’ll notice that this method also uses a factory even though, unlike the server, it will only ever spawn one protocol. Although there might be a few valid reasons for this, I imagine it is simply to keep the two operations create_server and create connection “symmetric.”

Once you’ve created your server and client files, each one will need to have the event loop started. Remember, your Protocol code stops executing. It’s the event loop that wakes it up when data is ready. So unless the event loop is running, your protocol will never run! Loop can execute a “run_forever” method that basically loops until ctrl-C, etc.

Coroutines: Prepare to have your Minds Blown!

If you’ve read the documentation for asyncio, you might have been completely confused by something called a “coroutine.”

A coroutine gets its name from the fact that the function runs in parallel with others. As I described in class today, the “yield” statement causes a function to “freeze” and then continue later. So, because it won’t continue until the calling function triggers it, the two are co-routines together.

If you look closely, you’ll notice that loop.create_server and loop.create_connection return co-routines. Python is basically saying these two functions aren’t really functions at all. They don’t execute and then return. Instead, they pause in the middle of execution (i.e., while waiting to connect or receive connections) and the loop will advance them when connections or data is available.

You shouldn’t have to do much with coroutines at this point in the class. But please start reading up on these concepts and, perhaps, experimenting with asynchronous functions. You’ll want to understand this very soon.

Two Playground Specific Issues:

Beyond the basic Python 3 setup described in the online documentation, you need to be aware of the two following issues:

1. I want you to specify a null transport in the constructor and set it back to null in connection_lost.
2. You will want to use a deserializer in data_received.

Let's talk about each one of these in details.

First, back in the days when I was using twisted, a protocol always had a .transport attribute, even before connection_made. It was just that before connection_made it was set to None. But Python 3 makes no assumptions about transport. It passes it in to connection_made and then its up to the receiving protocol to decide what to do with it.

But the problem is, there are a lot of times when other code needs to determine if the protocol is connected or still connected. Playground does for reasons that will become clear in lab 1[e] and lab 2. But there's no standard way to make said determination. Accordingly, I'm adopting the Twisted approach of determining if a protocol is connected by the status of its transport.

So, when you write your Protocol class, you need to do something like this:

```
class MyProtocol(Protocol):
  def __init__(self):
    …
    self.transport = None

  def connection_made(self, transport):
    self.transport = transport
    …

  def dataReceived(self, data):
    …

  def connection_lost(self, exc):
    self.transport = None
    …
```

I hate requiring this because there's nothing in the code to enforce it and it's likely to introduce bugs. But I looked at a number of alternatives and came up with nothing better.

The second thing you need to do is use Deserializer to handle your incoming messages. You should instantiate the Deserializer in your constructor or connection_made method:

```
  def connection_made(self, transport):
    …
   self._deserializer = PacketType.Deserializer()

  def data_received(self, data):
    self._deserializer.update(data)
    for pkt in self._deserializer.nextPackets():
      # process pkt
```

Testing Protocols

When it comes time to **unit** test your protocols, you should generally *NOT* use the actual network. In fact, you shouldn't even use the real event loop.

I've provided a few classes in Playground to help. I suggest that you look at basicUnitTest() in playground\network\devices\vnic\VNIC.py for some ideas. Here are the useful classes:

- `from playground.asyncio_lib.testing import TestLoopEx`
- `playground.network.testing import MockTransportToStorageStream`
- `playground.network.testing import MockTransportToProtocol`

Let's say that you wanted to test an EchoServerProtocol and an EchoClientProtocol. The first thing you want to do is replace the real event loop with our test loop:

```
def basicUnitTest():
  asyncio.set_event_loop(TestLoopEx())
```

The next thing we'll do is manually construct our two classes.

```
  client = EchoClientProtocol()
  server = EchoServerProtocol()
```

Now, we need to construct some fake transports. Normally, a transport is provided by Asyncio and represents a connection to the network. But we'll use the Mock transports I provide to simulate sending data over the network. I wrote up two simple type of Mock transports. The MockTransportToStorageStream stores any data written to it to a storage stream of bytes that can be read later. This is useful for representing a connection to a protocol not under test. You can simply read the bytes written out and see if they're correct.

But if you want to communicate between two symmetric protocols, use MockTransportToProtocol, which writes its data directly to a connected protocol. That's what we'll use here:

```
  transportToServer = MockTransportToProtocol(server)
```

```
transportToClient = MockTransportToProtocol(client)
```

Basically, calling transportToServer's write method will route the data directly to server's data_received method.

Now that our transports are created, call the connection_made method. There are protocols for which order might matter (i.e., if a protocol writes data in connection_made). But for our simple tests, we can just connect them:

```
server.connection_made(transportToClient)
client.connection_made(transportToServer)
```

Now, we can test stuff by having the client send data to the server and check its responses. Note that in this example, we didn't really need the test loop. That piece is more useful for code that does more with coroutines, timed callbacks, etc. I'm not going into that for now, but you can look at the vnic.py code for example usage.

Assignment Description

Your assignment for lab 1[c] is to implement the protocol you described in your lab 1[a] using the packets you implemented in 1[b]. You must create two protocol classes, one for your client and one for your server, and be able to connect them over the TCP network. You must also write up at least one unit test that demonstrates that your protocol is implemented correctly.

Your Server protocol needs to show some kind of "brain." That is, it needs to decide on the exact response based on the input from the client. For example, in the lab 1[a] writeup, I described a math problem protocol. A minimum "brain" would at least check the result to decide whether to send back a "pass" or "fail" data.

Grading

Your assignment will be graded out of 10 points according to the following rubric:

- 10: Program can run in client/server mode, unit test completes and is thorough
- 7-9: Program runs unit test but tests are incomplete, or doesn't run in client/server mode.
- 4-6: Program does not run a unit test
- 1-3: Incomplete submission.

Collaboration Policy

For this assignment, you ***MAY NOT*** discuss with anyone other than the professor or the TA, and my not use the Internet (e.g., Google) for any ideas. I am testing your ability to program and I do not want you using other students.

Due Date and Late Policy

The lab is due by midnight on Monday 9/11/2017. We will expect to pull the data out of your repository at that time. Your primary file with the unit test should be in the repository as

/netsec_fall2017/lab_1c/submission.py