Johns Hopkins University
EN.601.444/644 Network Security
Fall 2017
Seth James Nielson
Lab #1e
Revision 1.0 (9/14/2017:1600)

ASSIGNED: 9/14/2017
DUE: Midnight 9/18/2017

Introduction

It's time to start working with Networking Stacks and prepare for Lab 2! This lab has a few non-programming assignments, so please read the instructions carefully.

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] – Create a TCP/IP client server implementation of your protocol~~
- ~~1[d] – Convert your protocol from TCP/IP to Playground Network~~
- 1[e] (this lab) – 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 9/13~~
- ~~1[d] is due 9/13~~
- 1[e] is due 9/18

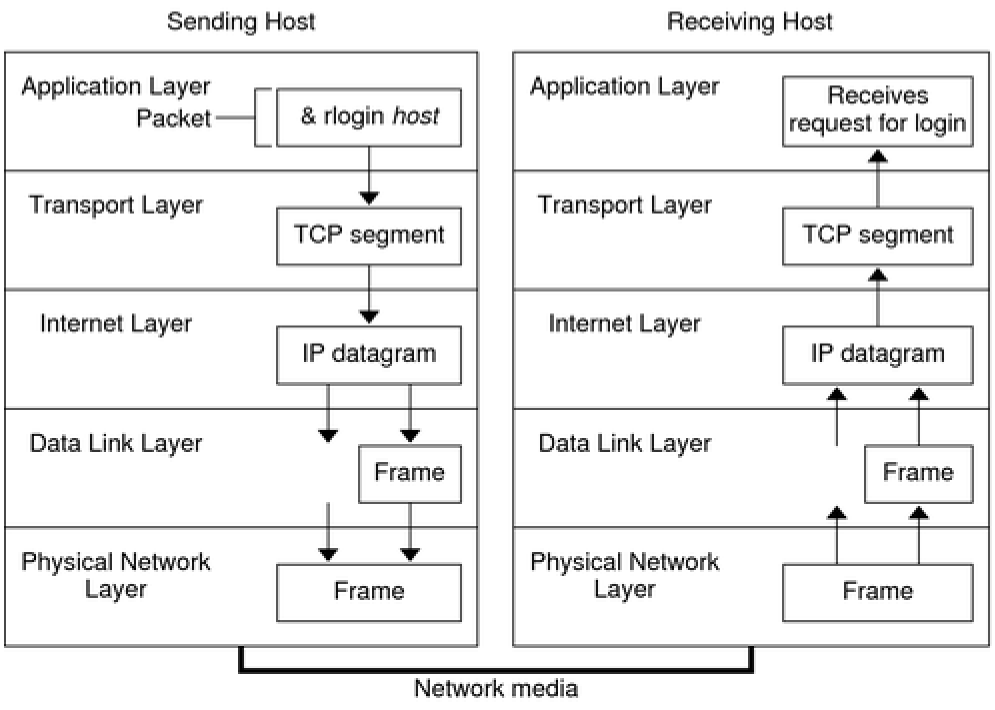
This will be a programming assignment in python using the Playground framework along with a few administrative assignments.

An Overview of Network Stacks

So far, you've worked with protocols that operate on their own. They really aren't designed to work with other protocols.

But in the networking world, different protocols work together to get things done. Protocols that work together are said to be "a stack." Why?

Here's a visualization of a typical networking protocol stack:



Each box is a separate protocol. There is an "Application Layer" protocol at the top, a "Transport Layer" protocol underneath, and so forth. The term "stack" came from this layered type of visualization.

Importantly, each protocol operates independently on the data and then either passes it up or down. Or, put another way, if a protocol receives data from a lower protocol, it processes it and then passes the processed data "up." If it receives data from a higher protocol, it processes it and passes the processed data down.

Let's walk through how this works in the real internet with an HTTP request. First, the protocols:

- Application Layer – HTTP Client (e.g., a browser requesting a page)
- Transport Layer – TCP
- Internet Layer – IP
- Data Link Layer – Ethernet
- Physical Network Layer – The Ethernet physical connection

When you type http://cnn.com into your browser, the browser creates an HTTP packet with that request and sends it over a socket. But what happens when that packet is sent on the socket?

The TCP layer receives that data and processes it. In the act of processing it, it creates a TCP packet that *contains* (encapsulates) the original http request. In fact, if it were a much larger packet (like an HTTP response), it will create many TCP packets that break the data into smaller chunks.

Once TCP gets done, it passes the newly created TCP packet or packets down to the Internet Protocol. The IP layer will create IP packets *that contain the TCP packets that contain the HTTP data!* IP may create just one packet, or it may need to fragment each TCP packet into several IP packets. Once it is finished, it passes it down to the MAC layer.

Once the Ethernet layer gets it, it also processes it, creates packets (e.g., Ethernet Frames) that contain the IP packets that contain the TCP packets, that contain the HTTP request. And it sends those over the physical layer to an ethernet layer on the other side.

When these Ethernet packets are received, they are processed and the original IP packets de-encapsulated. These IP packets are passed up to the IP layer.

At the IP layer, the IP protocol takes all of the IP packets and re-combines them until it can extract original TCP packets. Once those are extracted, it passes the TCP packets up to the TCP layer. Similarly, the TCP layer combines TCP packets until it has the HTTP request and passes that up to the HTTP server listening on the socket.

What I really haven't described in this process is all the processing that each layer does. For example, the TCP layer, before it ever sends a data packet, does a "handshake" by sending and receiving control messages from the other TCP layer. Once it starts sending data, it numbers each packet. When packets are received, they are acknowledged by the TCP receiver sending back a special acknowledgement packet. If a packet is received out of order, it corrects it. If a packet is not received (e.g., it receives #3 and #5, but not 4), it will either wait for a re-transmission or explicitly re-request.

And that's one of the reasons why we break everything up into different protocols instead of just one big protocol. It's easier to have each protocol be responsible for its own processing.

It's also important to note that, for the most part, each protocol is only aware of its "peer" protocol. That is, the TCP layer, for all intents and purposes, is only communicating with the other TCP layer on the other computer. The IP layer is only communicating with the IP layer, and the Ethernet layer is only communicating with the Ethernet layer.

The Playground Network Stack

For a number of reasons, Playground just smooshes the bottom layers of the diagram above into a single layer. Basically, the Playground Wire protocol handles data link and internet over

TCP/IP (which is the physical layer). Your job in lab #2, is to design and implement a transport layer.
But for this lab, we're just going to do some preparatory setup
.
First, let's talk about our Python programming. I've created a special set of classes for "stacking" protocols, transports, and factories. These are:

- StackingProtocol
- StackingTransport
- StackingProtocolFactory

All of these classes are imported from playground.network.common. You will also need to import playground.

The StackingProtocol is a class that extends Protocol. It includes a "higherProtocol()" method that returns the higher protocol if there is one. To make a protocol a stacking protocol, all you need to do is inherit from StackingProtocol instead of Protocol. Nothing else changes but you can access the higher protocol from within your "connection_lost", "data_received," and "connection_made" methods using the "self.higherProtocol()" method mentioned above. So, for example, if you're ready to pass data up, you'd call, "self.higherProtocol().data_received(data)".

PLEASE NOTE: If you write a constructor (__init__) method for your StackingProtocol class, please make sure to call "super().__init__" or make sure that "self.transport=None." If you don't, you will get weird failures.

StackingTransport is a Transport subclass that writes data to a lower transport. Why would you want such a thing? Well, remember that a higher protocol has to write to a lower one. This is how it does that. The general idea is that a lower protocol should create a stacking protocol and pass it to the higher protocol.

You will not need to subclass StackingTransport here, but you will later in lab #2. In that lab, you will modify the transport so that it processes data before sending it to the lower transport. For this lab, you will only need to create an instance of StackingTransport with the lower transport as the first argument. So, for example "higherTransport = StackingTransport(self.transport)"

Finally, a StackingProtocolFactory is a special class that takes protocol factories that produce StackingProtocol's and chains them together so that all of the protocols have their higher protocols set correctly. You should generally not need to subclass this. To use it, you pass the component factories into the constructor: "Stack = StackingProtocolFactory(factory1, factory2)"

The component factories can be normal asyncio factories (e.g., "lambda: Protocol1()") so long as the protocols they build are StackingProtocols and not just regular Protocols.

Creating Your Passthrough Layer

Your assignment for 1[e] is to create two "pass through" protocol layers and insert them into your playground stack. First, let's talk about a "pass through" layer and then how to insert it.

A pass through layer does exactly what it says. It simply takes data and passes it through. No modification, no processing. If it receives data from data_received and passes it to the higher layer. It provides a transport to the higher layer that, when write is called, simply passes data to the lower layer.

What you have to figure out is ***when to do these various operations.*** For this lab, start by asking yourself these questions.

1. When should my pass-through layer call the higher layer's connection_made?
2. What should my pass-through layer give to the higher layer's connection_made?
3. When should my pass-through layer call the higher layer's data_received?
4. When should my pass-through layer call the higher layer's connection_lost?

Once you've figured out the answers to these questions, writing the pass-through layer should be super easy. My pass-through layer is about 15 lines of code total.

Your two pass-through layers can be identical, but I want you to have two. As you're working with these, put some logging messages into your connection_made, data_received, and connection_lost methods of these two classes so you can see when one thing is built, and then the next.

Once you have your two classes, create two factories and chain them together using a StackingProtocolFactory. Assuming protocol 1 is PassThrough1 and protocol2 is PassThrough2, you could write the following:

```
f = StackingProtocolFactory(lambda: PassThrough1(), lambda: PassThrough2())
```

Congratulations! You've made a pass-through networking stack!

To use your newly created stack, we need to adjust playground's connectors. You will basically define a new connector that uses your stack. Like so:

```
f = StackingProtocolFactory(lambda: PassThrough1(), lambda: PassThrough2())

ptConnector = playground.Connector(protocolStack=f)

playground.setConnector("passthrough", ptConnector)
```

Now, to use the stack, go back to your lab 1[d] code and replace:

```
playground.getConnector().create_playground_server
```

with

playground.getConnector('passthrough').create_playground_server

Now when you run your l[d] client and server, you should see the logging messages showing the intermediary layers getting created, passing data, etc.

Assignment Description

You have multiple assignments for lab 1[e]:

1. Implement the pass-through layers as described above. There is no unit test; just check-in the code in github and we will review it manually.
2. Send me and the TA's an email with your Lab2 team. If you don't have a lab2 team by the time this assignment is due, let me know that you're still looking.
3. ***Skim*** over RFC 793 online. It is the 1981 specification of TCP. If it absolutely makes no sense, try reading a bit about TCP in Wikipedia. Try to get a sense for how it works and then start to think about how you might do your own transmission protocol if you got to start from scratch. In the same email, simply indicate that you have reviewed this information for credit.

Grading

Your assignment will be graded out of 10 points as pass fail (10/0).

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 9/18/2017. We will expect to pull the data out of your repository at that time. Your files should be in the following directory:

/netsec_fall2017/lab_1e/