**Johns Hopkins University**

**EN.601.444/644 Network Security**

Fall 2018

Seth James Nielson

Lab #1

***Revision 1.0 (2018-09-18:0900)***

**ASSIGNED:** 2018-09-17

**DUE:** 23:59:59 on 10/15/2017

**NOTE:**

THIS IS A LONG DOCUMENT. There’s a lot of documentation here. Skim through it the first time to get a feel for what is covered. There’s a lot of sample code. Work through it. The entire first week is dedicated to just getting the environment setup.

After skimming through it a few times, and perhaps trying a code sample or two, read all the way through it carefully. Send the teaching staff questions. Get a handle on what the assignment is.

Once you start coding the assignment, refer back to the document for reference.

**Introduction**

Lab 1 is a very difficult assignment. You have four weeks, but should get started immediately. Not only is the engineering more complicated than it might appear but the software you create in this lab must also be compatible with the code from all the other students!

Another way Lab 1 is complicated is that you are creating what could be called “middleware.” In most classes, you create “applications”. On the other hand, Lab 1 is a network protocol layer intended to be used by application-layers.

In order to get started, you’re going to have to learn a few mechanics of the Playground system.

* You need to know how to create packets using the Playground “PacketType”
* You need to know how to create an application-layer protocol and connect it to the playground network
* You need to know how to connect “middleware” protocols to upper and lower protocols

Once you have those basics figured out, you will have the capacity to implement the protocol specified in the PRFC.

**The Limitations of the Playground Wire Protocol**

Why are we doing this lab?

Well, currently, all the data we’re sending over the Playground Network is not being sent reliably, nor even in sessions.

What is a session?

The Playground Wire Protocol is *packet oriented*. It is designed to get a packet from point A to point B. That is obviously very important and it works reasonably well for protocols that have very few communication exchanges. But as soon as two packets have to be related to each other, the Wire Protocol is ineffective. In other words, if the meaning of one packet is dependent upon a previous packet, the current Playground network just won’t work.

But we tested echo protocol in class. Why did that work? Hopefully you can see that the packets sent in echo weren’t dependent on previous packets even if *it appeared to be connected semantically*

Let’s review:

CLIENT -> SERVER: Hello

SERVER -> CLIENT: Hello

CLIENT -> SERVER: test two

SERVER -> CLIENT: test two

These packets *look* tied together, but they are not.

The playground wire protocol associates data together based on precisely four variables: source address, source port, destination address, destination port. On the server side, the hypothetical source code listens on a playground port number. Let’s suppose that it’s listening on playground address 20184.1.2.3 and port 5000.

On the client side of this imagined scenario, the client would create a connection to 20184.1.2.3, port 5000. Let’s supposed that it’s connecting from the playground address 20184.40.50.60. When creating an outbound connection, playground will automatically assign a source port. We’ll just arbitrarily pick source port 2222.

But, at this point, *no data has been sent*! When I say “create a connection”, this protocol has no handshake, so it is simply a reservation for future communication.

Once the client decides to actually send using echo protocol, it creates a Playground Wire Packet and wraps the application data. It includes four extra pieces of data: the source address (20174.40.50.60), the source port (2222), the destination address (20174.1.2.3), and the destination port (5000). This packet is sent out on the network. As it does so, it associates (20174.40.50.60, 2222, 20174.1.2.3, 5000) with the client protocol.

When the server receives this packet, it looks at these four pieces of data and determines that it has no current protocols associated therewith. But, it also determines that there is a listener associated with the destination port (5000). Accordingly, it produces a new playground wire protocol instance and associates it with the incoming data.

The playground wire protocol strips the data from the packet and calls the application layer’s data\_received with that data. The echo server, in response, sends the echoed packet back to the playground wire protocol via transport.write(). This data is put into a PWP packet with source and destination data and sent back to the original sender.

When the data comes back to the sender, remember that it associated the source, source port, destination, destination port with the client protocol. When it sees these four data, it looks up the protocol, pops out the data, and passes the data to the protocol using data\_received.

This continues back and forth with both sides sending and receiving their packets.

Both sides can choose to shut down after the final packets, closing the protocol and releasing the ports.

Doesn’t this mean the packets were associated with each other?

No, not at all. Imagine that when the user sends the original echo, the server already is receiving data from dozens of echo clients. For whatever reason, the server gets bogged down and can’t respond quickly. The user that sent the packet thinks that something is wrong and kills the program using CTRL-C. The user then sends a new request to the echo server.

In the mean time, the Echo Server finally gets around to responding to the original request. If, by chance, the restarted client uses the same outbound port (remember, the source port is chosen at random on the client side), when the packet arrives at the server, the server will assume that the data belongs to the old communication system.

Worse, the server, when it responds to the *original* request sends back the original echo. Thus, the client will receive a response that corresponds to a “cancelled” session. But because there is no session, really, it accepts the data.

In old versions of playground, ports were reused almost immediately. In these older versions, the outgoing source port would likely be the same. In the most recent version of Playground, it’s unlikely to *immediately* reuse the same port. Each version has a slightly different effect.

What is needed is something like TCP. TCP creates a “session” for every connection. Before sending any data, TCP sends a handshake that indicates the beginning of a session, and then numbers all the bytes being sent. Different sessions have different numberings so it is unlikely that data from two different sessions could get mixed together indistinguishably.

The other reason the Wire Protocol is insufficient is because it doesn’t deal with errors. If any data is corrupted, delivered out of order, or duplicated it will not be detected by PWP.

So far, the Playground Switches have not messed with the data. But at the conclusion of this lab, the switch will be configured to damage data to test the effectiveness of your protocol.

Hopefully this will explain why we need our new protocol.

**PacketType Operations**

Sending data over a network connection can be a pain. You have to convert various data types into bytes, concatenate everything, and include metadata necessary to get it all back out on the other side.

The Playground PacketType is designed to make this easier and you should start experimenting with PacketType’s *outside of your protocol*. You can start by importing the PacketType module:

from playground.network.packet import PacketType

Now, for a given packet, start by declaring a new class that inherits from PacketType:

class MyPacket(PacketType):

Whenever you create a new packet type, it requires two pieces of information. A name and a version number. The name can be any string, but you might consider using some kind of qualified path (e.g., “a.b.c”) for keeping namespaces unpolluted. Version numbers must be a string of the form “x.y.”

class MyPacket(PacketType):

DEFINITION\_IDENTIFIER = “lab2b.student\_x.MyPacket”

DEFINITION\_VERSION = “1.0”

At this point, you can begin to define the important part of your packet: the fields that define it. Fields require a name and a type. The types are not Python types. Rather, they are types that I have created to represent data that will be sent over a network. The currently defined types are all in playground.network.packet.fieldtypes and include:

* UINT (with UINT8, UINT16, UINT32, and UINT64 variants)
* INT (with INT8, INT16, INT32, and INT64 variants)
* BOOL
* LIST
* STRING
* BUFFER
* ComplexFieldType

We’ll save ComplexFieldType for another lab. For now, the other types should be sufficient. Let’s discuss each one briefly.

UINT and INTs are integers (no decimal) and UINT’s are unsigned (>= 0). The numbers that follow are how many bits. An INT8 is an 8-bit integer, and can hold any value between -126 and +127.

A BOOL is a true/false.

Strings and Buffers are for holding Python strings and bytes. You can search around on the Internet for an explanation of the difference (see, e.g., <https://stackoverflow.com/questions/6224052/what-is-the-difference-between-a-string-and-a-byte-string>). But for just a quick practical explanation:

s1 = “this is a string”

b1 = b”these are bytes” # note the ‘b’ in front of the quotes

And finally, let’s discuss LIST. LIST allows you to send multiple items *of the same type* in a packet. It is always declared with a second type (e.g., LIST(UINT8), LIST(STRING), etc).

In addition to the value that each type can hold, each type can also have a “null” value that is represented by the FIELD\_NOT\_SET value. This value needs to be imported from playground.network.packet as well.

Let’s get back to creating our packet. First, let’s make a packet that has some UINT32’s, a STRING, and a BUFFER. So, we need to import those types accordingly:

from playground.network.packet.fieldtypes import UINT32, STRING, BUFFER

Now let’s define a few fields:

from playground.network.packet import PacketType

from playground.network.packet.fieldtypes import UINT32, STRING, BUFFER

class MyPacket(PacketType):

DEFINITION\_IDENTIFIER = “lab2b.student\_x.MyPacket”

DEFINITION\_VERSION = “1.0”

FIELDS = [

(“counter1”, UINT32),

(“counter2”, UINT32),

(“name”, STRING),

(“data”, BUFFER)

]

That’s it! The packet is completely defined. Each field in the “FIELDS” list identifies a field by it’s name and its type. These will be automatically populated when creating an instance of the packet. Let’s do that next:

packet1 = MyPacket()

packet1.counter1 = 100

Where did counter1 come from? The PacketType class, upon instantiation, creates variables named after the field names. In this case, it created counter1, counter2, name, and data. And, when setting the data, it will do some basic type checking. For example:

packet1.counter2 = -100

This line above will throw an exception because it will note that counter2 is an unsigned int and cannot be negative.

Once the packet is created, it can be serialized into a stream of bytes. In the next lab, you will send the bytes over the network but, for now, we just want to test that this serialization and de-serialization back into an object works as expected. To serialize a packet, call the \_\_serialize\_\_() method.

packetBytes = packet1.serialize()

If you are trying the example so far, this line above should throw an exception. The problem is that a packet won’t serialize unless all required values are set. Remember the FIELD\_NOT\_SET value? If any non-optional field is FIELD\_NOT\_SET, serialization will fail. We’ll deal with optional values another time. For now, let’s set all the fields of MyPacket:

packet1.counter1 = 100

packet1.counter2 = 200

packet1.name = “Dr. Nielson”

packet1.data = b“This may look like a string but it’s actually a sequence of bytes.”

Now we can serialize:

packetBytes = packet1.\_\_serialize\_\_()

You may want to print these bytes out just to see what they look like. These bytes are appropriate for sending over a network. Once the bytes are received, they can be de-serialized back into an object. There are two ways of doing this.

The first way is to use the Deserialize class method of PacketType (or MyPacket). This method assumes you have enough bytes to completely de-serialize. Let’s try that out:

packet2 = PacketType.Deserialize(packetBytes)

if packet1 == packet2:

print(“These two packets are the same!”)

What happened here is we took packet1, turned it into a stream of bytes, and then used Deserialize to make an equivalent object. The two objects can be compared together and, so long as their fields match, they’ll be found equivalent as shown in the example above.

Deserialize works great but in network operations, you don’t always receive all the data at once. And sometimes, you might receive the data from two packets at the same time. How do you know if you have enough to deserialize? And how do you know if you need to deserialize more than one packet?

Fortunately, the PacketType class also provides a Deserializer object that deals with all of these problems. The Deserializer object takes network bytes in chunks and returns as many packets as it can unpack. Here is how it works:

deserializer = PacketType.Deserializer()

deserilaizer.update(data)

for packet in deserializer.nextPackets():

# now I have a packet!

Here’s an example using the MyPacket example:

packet1 = MyPacket()

# fill in packet1 fields

packet2 = MyPacket()

# fill in packet2 fields

packet3 = MyPacket()

# fill in packet3 fields

pktBytes = packet1.\_\_serialize\_\_() + packet2.\_\_serialize\_\_() + packet3.\_\_serialize\_\_()

Ok, so far so good. We have all three packets serialized into a single stream of bytes. How can we test the Deserializer object?

Let’s create a test where Deserializer only receives 10 bytes at a time.

deserializer = PacketType.Deserializer()

print(“Starting with {} bytes of data”.format(len(pktBytes)))

while len(pktBytes) > 0:

# let’s take of a 10 byte chunk

chunk, pktBytes = pktBytes[:10], pktBytes[10:]

deserializer.update(chunk)

print(“Another 10 bytes loaded into deserializer. Left={}”.format(len(pktBytes)))

for packet in deserializer.nextPackets():

print(“got a packet!”)

if packet == packet1: print(“It’s packet 1!”)

elif packet == packet2: print(It’s packet 2!”)

elif packet == packet3: print(“It’s packet 3!”)

Try playing with this until it make sense and you feel comfortable.

**An Overview of Python 3’s Asyncio**

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.

*The Transport:*

You’ll notice that the 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. Instead, you will use asyncio’s event loop to attach a protocol to either a server or an outbound TCP connection. I will show you the code for setting up a TCP client/server first, and then we will show how to switch it over to the playground network.

As described in the Asyncio documentation, here is how you setup a listening (TCP server) protocol:

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. If you’re unfamiliar with Python’s “yield” statement, you should probably look it up. But briefly, 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.

Co-routines can take a while to get used to. We’ll talk about them more in class. But for now, start reading up on these concepts and, perhaps, experimenting with asynchronous functions. You’ll want to understand this very soon.

*Using Playground instead of TCP:*

First, let’s talk about two basic issues for your protocol classes.

1. Specify a null transport in the constructor and set it back to null in connection\_lost.
2. Use playground’s PacketType and 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. Accordingly, I’m adopting the Twisted approach of determining if a protocol is connected by the status of its transport. I need you to set it to “None” in the constructor so that Playground can tell it isn’t connected until connection\_made().

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

The only other change is how you setup the client and server in the asyncio loop. It’s actually very easy. In fact, a one-line change for each. Once you have playground networking set up, all you need to do is convert any references to “loop.create\_server” and “loop.create\_connection” with the playground equivalents.

For create\_server, use the following command instead:

playground.getConnector().create\_playground\_server(factory, port)

For create\_connection, use the following command instead:

playground.getConnector().create\_playground\_connection (factory, playgroundAddress, port)

Please note that it is the playground address you’re connecting to, not the IP address. If you used 20184.1.1.1 for your VNIC, that is the address you would put in here.

The playground code includes an echo client/server. You should run this and get comfortable with playground packets, addresses, protocols, and client/server functions.

The Playground Network Stack

So far, we’ve talked about protocols that are application-layer protocols. But we need to figure out how to “stack” protocols together. 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 want to subclass StackingTransport for creating your reliable layer such that the transport does the processing before calling the lower transport’s write().

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 A “Passthrough” Layer

To get comfortable with creating the network stack, we’re going to create a pass-through layer. A pass through layer does exactly what it says. It simply takes data and passes it through. The point of this exercise is to figure out when the passthrough layer should communicate with the upper or lower layers and how.

Start by asking yourself these questions.

1. When should my pass-through layer call the higher layer’s connection\_made and with what arguments?
2. When should my pass-through layer call the higher layer’s data\_received and with what arguments?
3. When should my pass-through layer call the higher layer’s connection\_lost and with what arguments?

The actual code should be very short. Perhaps about 15 lines of code total.

Remember that you can have different protocols for client-side and server-side. There’s no good reason to write two different classes for “pass through,” but I want you to anyway to get used to thinking in terms of client protocols and server protocols. 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. Although there is only one protocol in this “stack,” you still have to use a StackingProtocolFactory:

f = StackingProtocolFactory(lambda: PassThroughClientProtocol())

You may be wondering why your stack is just one protocol. Where is the application layer, and where is the playground wire protocol? Well, because those are “fixed” at the top and bottom, the purpose of the stack is to define all the layers in between. Right now, there’s just one. By the end of the semester, there will be two.

To use your newly created stack, we need to adjust playground’s connectors. Playground can associate different stacks with different string identifiers. When creating a playground connection with GetConnector(), passing a string label indicates which stack to use in between the application layer and playground wire. If you pass no argument, no stack is used, and the application layer connects directly to the playground wire protocol.

You will basically define a new connector that uses your stack. A Playground Connector takes a pair of a StackingProtocolFactorys as a parameters for the client and server:

f\_client = StackingProtocolFactory(lambda: PassThroughClientProtocol())

f\_server = StackingProtocolFactory(lambda: PassThroughServerProtocol())

ptConnector = playground.Connector(protocolStack=(f\_client, f\_server))

playground.setConnector("passthrough", ptConnector)

Now, to use the stack, the application layer connection needs to change from this:

playground.getConnector().create\_playground\_server

with

playground.getConnector(‘passthrough’).create\_playground\_server

Playground also has a system for auto-loading stacks. Place a module within .playground/connectors and have the \_\_init\_\_.py file within the module set the connector. These modules will be auto loaded by playground.

So, recall that pnetworking looks to a .playground directory for networking configuration. If you followed my instructions, this directory will be in a playground environment directory that I’ll call playground\_env for the example. The directory structure should look like this:

playground\_env/

playground\_env/.playground/

playground\_env/.playground/connectors

If you’re unfamiliar with a Python module, it is just a directory with a \_\_init\_\_.py file. So you could create a module within your playground\_env like this:

playground\_env/.playground/connectors/passthrough/\_\_init\_\_.py

playground\_env/.playground/connectors/passthrough/passthrough.py

Assuming your protocol is defined within passthrough.py, the \_\_init\_\_.py file would import the protocol, create the stack and connector, and call setConnector(“passthrough”, passThroughConnector). This would be auto-imported by playground when getConnector() is called.

You can use this to test out your pass-through layer with echotest. Echotest takes a -stack=stack\_name as an optional argument. So, to start a server and client using your pass through layer:

echotest.py server 9999 -stack=passthrough

echotest.py <server\_address> -stack=passthrough

**Assignment Description**

For Lab 1, you need to create a new middleware protocol that will provide sessions and reliable delivery according to the class-chosen PRFC.

The PETF will have to “fix” the protocol in stages as people uncover the “bugs” in the standard. That’s probably not something you’re happy about, because it means that you will have to change you protocol as the standards are updated and extended. Make the most of it. Express your views on Slack in the lab1-prfc channel to identify bugs, etc.

But it is necessary for your teams to implement these *reference protocols* partially to help the PETF clarify and improve the specifications. And it is better to be ahead and have to change some relatively minor details. Even if something major has to change, you will have better understanding of why the major changes are needed.

It is recommended to do the lab in the following stages:

* Week 1: Figure out pass-through layers and demonstrate using them with echotest
* Week 2: Get a handshake working
* Week 3: Error-free data transmissions
* Week 4: Error-correcting data transmissions

Once you have your protocol working (or even parts of it working), you can plug it in for testing on your own. You should already be familiar with manually setting a connector as you did in lab 1[e]. For that lab, however, you could put just one protocol stack in for both sides of the connection (client and server). For this lab, however, you’ll need a different protocol stack for each side.

You can test on your own computer, of course, but this will not help you determine if you are compatible with your classmates. We will either provide a switch that you and other students can connect to for testing transmissions, or we will give you instructions for how to setup your own switches.

**Due Date and Submission Process**

Lab1 is due by midnight (technically, 23:59:59) on 10/15/2017. We will expect to pull the data out of your repository at that time. Your submission must use the following format:

* You must create a python module named “lab1\_protocol”
  + It must be located at: /labs/lab2/src/lab1\_protocol/
  + It must be importable (i.e., /labs/lab2/src/lab1\_protocol/\_\_init\_\_.py)
* Upon import, it must configure a connector named “lab1protocol.” The easiest way to do this is simply to have your \_\_init\_\_.py do the following:

import playground

lab2Connector = playground.Connector(protocolStack=(

lab1ClientFactory,

lab1ServerFactory))

playground.setConnector("lab1protocol", lab1Connector)

Obviously, you would replace “lab1ClientFactory” and “lab1ServerFactory” with whatever names you have used in your code.

**Grading**

Your Lab #1 assignment will be graded out of 100 points according to the following schedule:

* 25 Points for session establishment and session termination
* 25 Points for error-free data transmissions
* 25 Points for low-error rate data transmissions
* 25 Points for medium-error rate data transmissions

There will also be extra credit for the following:

* 10-30 Points for high-error rate data transmissions
* 25 Points for winning a throughput test (more about this later)

***LATE POLICY: You will be marked off 5 pts for day late. After 10 days late, you will receive NO CREDIT for this lab. That’s a significant portion of your final grade. DON’T BE LATE.***

**Collaboration Policy**

For this assignment, you may discuss and test with ***ANYONE***. This includes other students. However, you ***MUST NOT SHARE CODE*.** Another student is welcome to help you debug (figure out where bugs are), but must not write any of your code for you, nor share with you their own code. If you get stuck, ask a CA.