Johns Hopkins University
EN.601.444/644 Network Security
Fall 2017
Seth James Nielson
Lab #1b
Revision 1.2 (9/6/2017:2148)

ASSIGNED: 9/4/2017
DUE: Midnight 9/6/2017

Introduction

By now, you should have completed your lab 1[a] to create a simple protocol with at least three packet types. In this lab, you are going to use python 3 + Playground to create representations of the packets and a simple unit test to illustrate that they work.

You are not actually going to implement the protocol in this phase! You are ONLY going to demonstrate that you can create the packets for the protocol.

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] (this lab) – 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] – 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 Playground's Packet Library

Playground provides a simple, but powerful, mechanism for creating new types of packets. Once a packet type is defined, it is equally simple to create instances of the packets, serialize them, and de-serialize them.

To get started, you will first need to install Playground. Currently, only Linux is fully supported. Mac OS/X will probably work with some tweaks, but I will not provide any technical support. You might also consider using the Windows Subsystem for Linux (I do), but again, no promised of technical support.

The easiest way to use Playground if you don't have a linux machine is to download a Virtual Box Virtual Machine for Linux.

Once you have Linux, make sure you have Python 3 (at least version 3.5) installed. Then, install Playground. You can install playground using pip, but I ***strongly recommend*** that you learn to use the python virtual environments for this.

Python has a built-in virtual directory system call "venv". Using venv, you can create a sub directory that has its own python binary, its own libraries, and so forth. That means you can install a bunch of modules and, if something breaks, you can erase everything with no impact on your system.

To setup a venv do the following:

```
mkdir ~/somedir
python3 -m venv ~/somedir
cd ~/somedir
source bin/activate
```

That's it! Now, everything you do in that terminal (bash session) will use the python from the virtual dir, and all of the packages therein. If you do a pip install, it will only install to this virtual environment.

To install playground, use pip (preferably from your venv environment)

```
pip install git+https://github.com/CrimsonVista/Playground3.git
```

Once installed, you also need to create your own repository. Please create a github repository that has the following code path:

```
/netsec_fall2017/lab_1b/submission.py
```

You can create other files, but your basic unit test (described later in this document) must be in this file.

Once you have your git repo ready, you should start experimenting with PacketType's. 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.

Assignment Description

Your assignment for lab 1[b] is to implement the packets (at least three) you described in your lab 1[a]. For example, in lab 1[a], I described the following protocol:

For example, consider this "Math Test Protocol":

1. The client begins the protocol by sending a request to the server for a math problem.
2. The server responds by sending a math problem (e.g., what is the sum of 3 and 7?)
3. The client sends back its answer (e.g., 10)
4. The server responds with a result (e.g., correct)

And I further identified the following packets.

1. RequestMathQuestion packet
    a. No parameters/fields
2. MathQuestion packet
    a. Operation Type (add, subtract, multiply, divide)
    b. Operand One
    c. Operand Two
    d. ID
3. MathSolution packet
    a. Solution
    b. ID
4. MathResult packet
    a. Result (correct/incorrect)
    b. ID

For this lab, if this was your protocol, you would need to use the Playground Framework to create four packet types with sufficient fields to transmit all of the necessary data.

Moreover, you also need to create a simple unit test that tests these various functions I suggest that you do something like this:

```
def basicUnitTest():
     packet1 = RequestMathOperation()
     packet1Bytes = packet1.__serialize__()
     packet1a = RequestMathOperation.Deserialize(packet1Bytes)
     assert packet1 == packet1a

     packet2 = MathQuestion()
     packet2.questionType = "add"
     packet2.op1 = 10
     packet2.op2 = -10
```

```
    packet2.id = 1
    packet2Bytes = packet2.__serialize__()
    …

    # You should do a **LOT** more testin!
    # Play around with it until you understand it!

if __name__=="__main__":
    basicUnitTest()
```

Grading

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

- 10: Program runs unit test thoroughly testing all packet types
- 7-9: Program runs unit test but tests are incomplete
- 4-6: Program does not run a unit test
- 1-3: Not all packets are defined, or not well defined.

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.

ALSO, PLEASE NOTE: On lab 1[a], I noticed a lot of submissions that looked suspiciously similar, and other submissions that took my example from the lab and just changed a word here or there. These submissions will not receive credit on Lab 1[b]. If you submit a lab that is questionable, even if it was accidental, you will be asked to resubmit (but there will be no penalty).

Due Date and Late Policy

The lab is due by midnight on Wednesday 9/6/2017. You need to submit to me and the TA the name of your repository so we can pull down your code.

I am getting this lab out later than I wanted. We will use all of class time on 9/6 to work on this together. It shouldn't take too long, but if we can't finish by the 6th, we'll extend the date a day or two.

Unless, however, I extend the due date, your lab will be marked down one point per day late. It will receive a 0 after 9/9/2017.