**Johns Hopkins University**

**EN.601.444/644 Network Security**

Fall 2018

Seth James Nielson

Lab #1 PRFC

***Revision 1.0 (2018/09/10:1830)***

**ASSIGNED:** 2018/09/10

**LAB #2 PRFC DUE:** 23:59:59 on 2018/09/12

For this assignment, you need to create a PRFC (Playground RFC) to specify a protocol that provides sessions and reliable delivery.

Requirements:

1. Within your github, the file MUST be named doc/prfc/drafts/reliable.xml
2. The XML file must be compatible with the xml2prfc.tcl script in the class GitHub
3. Your “Reliable” PRFC document must:
   1. Specify a ***name*** for the protocol. In past years, we’ve had:
      1. PCP – Playground Control Protocol
      2. KISS – Keep It Simple Stupid
   2. Provide for ***session*** establishment. A session is a group of packets associated together. In TCP, for example, when you open a socket it creates a session. All data sent over that socket is part of the same session.
   3. Provide for ***reliable delivery*** of the data within the session. That means:
      1. In-order
      2. Error-free (up to some “reasonable” error rate)
      3. Guaranteed delivery until session “tear down”
   4. Provide for reliable session “tear down.” Either side can tear down the session. ***Unlike TCP, there are NO guarantees in EITHER DIRECTION once tear down has been acknowledged.*** (This simplifies closing the connection considerably).
   5. Provide for ***byte-oriented*** delivery. TCP is NOT packet oriented, it is BYTE oriented. That means that as soon as TCP has any amount of data in order, even just one byte, it passes it to the application layer. It may break up data into TCP packets, but it does not do any reassembly. In fact, it acts like a ***stream*** protocol, not a packet protocol. Please ask me or the CA’s if this doesn’t make sense.

**Introduction to Playground**

This section is ***optional*** but probably very helpful. If you have the time, you ***should*** go through this. If nothing else, make sure you know how to specify a “packet” in Playground as you will need it for the PRFC.

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. (Serialization converts an object to bytes, and de-serialization converts the bytes back to an object).

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 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.

**RFC 793: TCP**

Take a look at RFC 793 for TCP (<https://tools.ietf.org/html/rfc793>). This is your template for your own protocol. Please pay careful attention to how the protocol is specified, including *state machines*.

You will notice that TCP’s RFC specifies many functionalities of TCP. You do NOT need to deal with multiplexing/demultiplexing (e.g., no ports). That is already handled. Your only job is to make sure that you specify a ***reliable, session-based*** protocol.

If you would like, you can also add in security features. The fact that TCP was written without security in mind haunts us to this day. Would you like to make your network more secure? Feel free.

But first, 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.

Because Playground’s wire protocol has ports, you can actually create a lot of protocols without an official transport layer, similar to what is possible with UDP.

Suppose we had one of our programming examples with the mathematics server. Although it was a little different, it could be described like this:

CLIENT -> SERVER: What is (3+4)\*2?

SERVER -> CLIENT: 14

Each packet in this protocol is meaningless (or at least, cannot be considered “correct”) without the previous one. The final packet obviously, is only meaningful if the first packet is “what is (3+4)\*2”

This would probably work somewhat in Playground even without a session, but only really by luck. Any “errors” of any kind would mess up the system. For example, suppose that the client crashed immediately after sending the question, and then immediately restarted. If it received the “14” answer, it would be very confused, but it would have no way of knowing that this packet was from a previous “session.”

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.

What is reliable delivery? It means that every byte is delivered in-order and error free. If there are too many errors to deliver the data error-free, the connection should get torn down.

**Authoring your PRFC**

There is a utility in the class GitHub called prfc/authoring/xml2prfc.tcl. This utility can convert a specially formatted xml document into an RFC-like document. An example is included at prfc/authoring/p\_rfc\_1.xml. Two sym links to xml2prfc are provided that are used to create html and txt versions of the PRFC.

To test this, check out the code in a POSIX-like environment (that supports sym links). To create an HTML version, run:

xml2html p\_rfc\_1.xml

To create a txt version, run:

xml2txt p\_rfc\_1.xml

You will need TCL installed.

Please note that the XML is very fragile. It’s easy to get it screwed up. Please write small-pieces and make sure it still converts before moving on.

The sample file will create PRFC 1. Please pay special attention to PRFC 1, which describes how you should write a PRFC.

And please remember, do NOT create a bit-oriented header for your protocol. Specify a playground packet as described above.

**Grading**

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

* 50 Points for addressing all necessary issues. You MUST address the following:
  + Purpose/technical background
  + Session establishment
  + Session termination
  + Data transmission, including in environments with errors
  + Error handling
  + Packet descriptions
  + Computational Algorithms
  + State machines
* 25 Points for thoroughness. We (the grading staff) will make a judgement as to how well someone could use your PRFC to implement a compliant protocol.
* 25 Points for grammar, correct punctuation, etc.

Please note that a full ***25% of your grade*** for the writing assignment is based on correct use of the English language. If you are not a native English speaker, ***I HIGHLY RECOMMEND GETTING EDITING HELP FROM AN ENGLISH SPEAKER OR JHU’S WRITING LAB.***

There will also be 25 points of extra credit awarded to the team with the best PRFC. Said PRFC will be selected as the official PRFC for the class.

***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, do not collaborate on any written portion. You may discuss ideas with class mates, but do not share any written work.