**Class Notes for Network Systems Foundations:**

Your task in this lab is to iterate over all packets we just read in (into the variable pkts), and if it is of type Ether, change it's source address to 11:11:11:11:11:11, and its destination address to 22:22:22:22:22:22

**for** pkt **in** pkts:

**if** isinstance(pkt, Ether):

pkt.src = "11:11:11:11:11:11"

pkt.dst = "22:22:22:22:22:22"

Code that passed 100% Successful on Programming Assignment : Modifying Ethernet Frames

*# your code here*

**from** **scapy.all** **import** \*

pkts = rdpcap("intro-wireshark-trace1.pcap")

pkt = pkts[1]

*# Do checks on what types are in the packet*

eth = Ether **in** pkt

print(eth)

*# You can get the name of the layers in the packet (requesting a specific one)*

layer = pkt.getlayer(0)

print(layer)

*# or, there's a variable within the pkt structure indicating the name of the type*

exp = pkt.name

print (exp)

*# You can list all the available protocols - won't, since it's large*

*#ls()*

*# Or list the fields in a particular layer*

ls(Ether)

*# Dump the packet contents in hex*

hexdump(pkt)

*# Dump the packet in a nice format*

pkt.show()

*# You can manipulate the packet through fields*

pkt.dst = "11:22:33:44:55:66"

pkt.show()

**for** pkt **in** pkts:

**if** isinstance(pkt, Ether):

pkt.src = "11:11:11:11:11:11"

pkt.dst = "22:22:22:22:22:22"

WEEK\_TWO Programming Assignment:

# Task: Create some functions for a simplified BGP router

# Specifically, the withdraw, update, and next\_hop functions of the Router

# The class Route will be used.

#

# withdraw(rt) - rt is type Route. If a simplified BGP router gets this message, it will

#

class Route:

def \_\_init\_\_(self, neighbor, prefix, mask, path):

"""

:param neighbor: IP address of the next-hop neighbor (string)

:param prefix: Network prefix (string, e.g., "192.168.0.0")

:param mask: Subnet mask length (integer, e.g., 24 for /24)

:param path: AS path (list of integers)

"""

self.neighbor = neighbor

self.prefix = prefix

self.mask = mask

self.path = path

def pfx\_str(self):

"""

Returns the prefix in 'a.b.c.d/x' format.

"""

return f"{self.prefix}/{self.mask}"

def \_\_repr\_\_(self):

"""

String representation for debugging.

"""

return f"Route(neighbor={self.neighbor}, prefix={self.prefix}, mask={self.mask}, path={self.path})"

class Router:

def \_\_init\_\_(self):

self.rib = {} # Routing Information Base (RIB)

def print\_rib(self):

"""

Prints the RIB for debugging.

"""

for prefix, routes in self.rib.items():

print(f"{prefix}: {routes}")

def update(self, rt):

"""

Update the RIB with a given route.

:param rt: Route object

"""

prefix\_str = rt.pfx\_str() # Get prefix in "a.b.c.d/x" format

if prefix\_str not in self.rib:

self.rib[prefix\_str] = []

# Check if a route from the same neighbor exists

for i, existing\_route in enumerate(self.rib[prefix\_str]):

if existing\_route.neighbor == rt.neighbor:

self.rib[prefix\_str][i] = rt # Replace the route

return

# Add the new route

self.rib[prefix\_str].append(rt)

def withdraw(self, rt):

"""

Withdraw a route from the RIB.

:param rt: Route object

"""

prefix\_str = rt.pfx\_str()

if prefix\_str in self.rib:

# Remove routes matching the neighbor

self.rib[prefix\_str] = [

route for route in self.rib[prefix\_str]

if route.neighbor != rt.neighbor

]

# Remove prefix if no routes remain

if not self.rib[prefix\_str]:

del self.rib[prefix\_str]

def convert\_to\_binary(self, ip):

"""

Convert an IP address to a binary string.

"""

return ''.join(f"{int(octet):08b}" for octet in ip.split('.'))

def next\_hop(self, ipaddr):

"""

Find the next hop for a given IP address.

:param ipaddr: IP address (string)

:return: Next-hop IP address (string) or None if no match found

"""

binary\_ip = self.convert\_to\_binary(ipaddr)

best\_match = None

best\_prefix\_length = -1

for prefix\_str, routes in self.rib.items():

prefix, length = prefix\_str.split("/")

length = int(length)

binary\_prefix = self.convert\_to\_binary(prefix)

# Check if the IP matches the prefix

if binary\_ip[:length] == binary\_prefix[:length]:

# Prefer longer prefixes

if length > best\_prefix\_length:

best\_match = routes

best\_prefix\_length = length

if not best\_match:

return None # No matching route

# Find the best route (shortest AS path)

best\_route = min(best\_match, key=lambda route: len(route.path))

return best\_route.neighbor

def test\_router():

router = Router()

# Add routes

router.update(Route("1.1.1.1", "10.0.0.0", 24, [1, 2, 3]))

router.update(Route("2.2.2.2", "10.0.0.0", 24, [4, 5]))

router.update(Route("3.3.3.3", "10.0.0.0", 22, [6]))

router.update(Route("4.4.4.4", "12.0.0.0", 16, [7, 8, 9]))

# Print RIB

print("RIB after updates:")

router.print\_rib()

# Test next hop

print("Next hop for 10.0.0.1:", router.next\_hop("10.0.0.1"))

print("Next hop for 12.0.0.1:", router.next\_hop("12.0.0.1"))

print("Next hop for 192.168.1.1:", router.next\_hop("192.168.1.1"))

# Withdraw a route

router.withdraw(Route("1.1.1.1", "10.0.0.0", 24, [1, 2, 3]))

print("RIB after withdrawal:")

router.print\_rib()

if \_\_name\_\_ == "\_\_main\_\_":

test\_router()

**How to Complete Transport Layer Lab!**

MMarshall

Learner

2 days ago

So... I finally passed both tests. Here's how.

First, make sure you understood the first lab in this course (link layer lab) and how you could print out individual packets and get their data. This is helpful for debugging and how to access different parts of each packet.

Ignore *create-simple-pcap.py*.

All you're doing in *FindMaxBytesInFlight.py* is building the function *findMaxBytesInFlight(pcapfile)*. You need to extract the packets like you did in the link layer lab. Then, pass those parsed packets into the readHandShake.

The function you're building is already set up to track the maxBytesInFlight. You must iterate through every packet and keep track of 2 more variables: the highest server sequence number, and the highest client acknowledgement number.

Notice the helper function to tell you which direction the packet is flowing. Everything going from server to client will help you calculate the highest server sequence number. Everything going from client to server will help you calculate the highest client acknowledgement number.

HERE'S THE BIG MYSTERY:

Remember what you learned on *Reliable Transfer*. **A** sends to **B** seq=500 with **PAYLOAD** length of 10. The acknowledgement from the client back to the server will be 510. The next successful packet will be sent with the sequence number of 510. **SO**, to calculate the highest server sequence number, it will be the sequence number of the packet **PLUS** the length of the payload -> len(packet[TCP].payload) to be exact for this lab (if you don't do it like that it won't work).

As you iterate through every packet, you can keep track of the highest server sequence number and highest client acknowledgement number. Make sure you access the values like this -> packet[TCP].child

And remember, you can know what kind of children there are by finding the packet structure learned from the link layer lab.

I will admit, it was slightly frustrating to be thrown into a lab where you don't necessarily understand the python package you're working with and the instructions weren't very clear, yet with a willing mind to problem solve you *can* figure it out.

In the end, the logic is quite simple for how daunting it seems to be. You can do it!

I hope this helped! Have a blessed day :)

LikeReply

WEEK THREE CODE:

#!/usr/bin/python3

from scapy.all import \*

# This class captures some information about a unidirectional flow

class FlowTracking:

def \_\_init\_\_(self, startSeqNum, ackNumReceived, srcIP, dstIP):

self.startSeqNum = startSeqNum

self.ackNumReceived = ackNumReceived

self.highestSeqNum = startSeqNum

self.pktLenOfHighestSeqNumPacket = 0

self.srcIP = srcIP

self.dstIP = dstIP

# Returns FlowTracking object for the server side

def readHandShake(pkts):

# Read SYN (client -> server)

syn\_pkt = pkts[0]

if not (TCP in syn\_pkt and syn\_pkt[TCP].flags == "S"):

raise ValueError("First packet is not a SYN packet.")

seqInit = syn\_pkt[TCP].seq

srcInit = syn\_pkt[IP].src

dstInit = syn\_pkt[IP].dst

# Read SYN-ACK (server -> client)

syn\_ack\_pkt = pkts[1]

if not (TCP in syn\_ack\_pkt and syn\_ack\_pkt[TCP].flags == "SA"):

raise ValueError("Second packet is not a SYN-ACK packet.")

if syn\_ack\_pkt[TCP].ack != seqInit + 1:

raise ValueError(f"SYN-ACK ack number mismatch: expected {seqInit + 1}, got {syn\_ack\_pkt[TCP].ack}")

if syn\_ack\_pkt[IP].src != dstInit or syn\_ack\_pkt[IP].dst != srcInit:

raise ValueError("SYN-ACK IP addresses do not match SYN packet.")

seqOther = syn\_ack\_pkt[TCP].seq

# Read ACK (client -> server)

ack\_pkt = pkts[2]

if not (TCP in ack\_pkt and ack\_pkt[TCP].flags == "A"):

raise ValueError("Third packet is not an ACK packet.")

if ack\_pkt[TCP].ack != seqOther + 1:

raise ValueError(f"ACK ack number mismatch: expected {seqOther + 1}, got {ack\_pkt[TCP].ack}")

if ack\_pkt[IP].src != srcInit or ack\_pkt[IP].dst != dstInit:

raise ValueError("ACK IP addresses do not match SYN packet.")

return FlowTracking(seqOther, seqOther + 1, dstInit, srcInit)

# Returns true if the packet is in the direction of the unidirectional flow represented by f

def isFlowEgress(pkt, flow):

return pkt[IP].src == flow.srcIP and pkt[IP].dst == flow.dstIP

# Given a pcap file name as a string, this function will return the max number of bytes

# that were in flight (unacknowledged) for this stream.

def findMaxBytesInFlight(pcap\_file):

try:

pkts = rdpcap(pcap\_file)

except FileNotFoundError:

print(f"Error: File {pcap\_file} not found.")

return -1

# Read the handshake to initialize flow tracking

try:

flow = readHandShake(pkts)

except ValueError as e:

print(f"Error reading handshake: {e}")

return -1

max\_bytes\_in\_flight = 0

current\_bytes\_in\_flight = 0

# Process packets after the handshake

for pkt in pkts[3:]:

if IP in pkt and TCP in pkt:

if isFlowEgress(pkt, flow):

# Update highest sequence number and packet length

if pkt[TCP].seq > flow.highestSeqNum:

flow.highestSeqNum = pkt[TCP].seq

flow.pktLenOfHighestSeqNumPacket = len(pkt[TCP].payload)

# Calculate bytes in flight

current\_bytes\_in\_flight = flow.highestSeqNum - flow.ackNumReceived + flow.pktLenOfHighestSeqNumPacket

max\_bytes\_in\_flight = max(max\_bytes\_in\_flight, current\_bytes\_in\_flight)

elif pkt[IP].src == flow.dstIP and pkt[IP].dst == flow.srcIP:

# Update acknowledgment number

flow.ackNumReceived = max(flow.ackNumReceived, pkt[TCP].ack)

return max\_bytes\_in\_flight

if \_\_name\_\_ == "\_\_main\_\_":

pcap\_files = [

"simple-tcp-session.pcap",

"out\_10m\_0p.pcap",

"out\_10m\_2p.pcap"

]

for pcap\_file in pcap\_files:

print(f"Processing {pcap\_file}:")

max\_bytes = findMaxBytesInFlight(pcap\_file)

if max\_bytes != -1:

print(f"Max bytes in flight: {max\_bytes}")