CS341 Network Lab 3: Building Your Own Network with SDN

Logistics

- The due date is 11:59pm on May 19th
- Submit a single zip file composed of `topology.py`, `route.py`, and `graph.py` via Gradescope.
- Discussion and Q&A: Lab 3 @ Piazza

Overview

In Lab 3, you will implement a fully-functioning network of multiple routers/switches on your personal computer or a single cloud. An open-source testbed tool, mininet (http://mininet.org/) will be used throughout this lab. Mininet lets network experimenters avoid many manual hassles for setting up routers and hosts at a large scale.

After implementing your own multi-router-and-host network and testing basic connectivity, you are asked to implement a couple of simple routing applications using software-defined networking (SDN) principles. The application will be implemented using the OpenFlow protocol and the POX controller. You will then show that the routing protocol you implement correctly forward packets between hosts.

The lab will be graded based on the correctness of your implementation.

Mininet Overview

What is Mininet?

Mininet is a complete network emulation of hosts, links, and switches. It is useful for testing OpenFlow and SDN, providing Python APIs. In this lab, we will use Mininet for emulating custom networks. For more information, visit Mininet official document.
Currently, we provide two ways to setup environments:

1. Automatic setup using Vagrant and Virtualbox
2. On Ubuntu 20.04 machine, setup

Environment Setup via Vagrant and Virtualbox

Virtualbox and Vagrant installation

To ensure Mininet works correctly, we provide a VM with Mininet and requirements installed. We use Virtualbox for a VM system, and Vagrant as a VM management software. Follow the official document for setup. We recommend you to conduct this lab on your own machine because mininet is lightweight and designed to run on a single laptop. But if you have trouble installing them on your machine, you may use the Kcloud VM you have received in the previous lab.

- Virtualbox: Installation document, Download page
- Vagrant: Installation document, Download page

Important: Please troubleshoot installation problems with Virtualbox or Vagrant on your machine or Kcloud on your own. TAs are not familiar with all the configuration issues with the tools, and thus it is your responsibility to install these widely used software projects. Search the Internet first before you ask any questions at piazza.
For your information, we have tested with following environments:

- Ubuntu 22.04 with Intel CPU
- Windows 11 with Intel CPU (hyper-V disabled)
- Not working: macOS 13 with M1
    - Please install Ubuntu 20.04 by yourself, and follow Environment Setup on Ubuntu 20.04

Environment Setup

Start from cloning the github repository.

```
$ git clone https://github.com/NetSP-KAIST/cs341-23s-lab3-handout.git
```

Setup VM environment by following command:

```
$ cd cs341-23s-lab3-handout
$ vagrant up
```

This will show following messages:

```
cs341-23s-lab3-handout$ vagrant up
Bringing machine 'mnvm' up with 'virtualbox' provider...
==> mnvm: Importing base box 'ubuntu/focal64'...
==> mnvm: Matching MAC address for NAT networking...
==> mnvm: Checking if box 'ubuntu/focal64' version '20230506.0.0' is up to date...
==> mnvm: Setting the name of the VM:
cs341-23s-lab3-handout_mnvm_xxxxxxxxxxxxx_xxxxx
==> mnvm: Clearing any previously set network interfaces...
==> mnvm: Preparing network interfaces based on configuration...

…

    mnvm: The following additional packages will be installed:
    mnvm:   binutils binutils-common binutils-x86-64-linux-gnu blt cpp cpp-9
    mnvm:   fontconfig-config fonts-dejavu-core gcc-9 gcc-9-base libasan5
libatomic1

…

    mnvm: ++ sudo cp /vagrant/controller.py /vagrant/dump.py /vagrant/graph.py
/vagrant/route.py /vagrant/task1.py /vagrant/task2.py /vagrant/topology.py ./
    mnvm: ++ sudo chown -R vagrant /home/vagrant/
    mnvm: ++ sudo ln -s /home/vagrant/controller.py /home/vagrant/pox/pox/misc/
```

Now, connect to vm by following command:

```
$ vagrant ssh
```

This will show the following message, as you connect to the remote server via ssh:

```
cs341-23s-lab3-handout$ vagrant ssh
Welcome to Ubuntu 20.04.6 LTS (GNU/Linux 5.4.0-148-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Tue May  9 18:51:53 UTC 2023

  System load:  0.03              Processes:               124
  Usage of /:   5.6% of 38.70GB   Users logged in:         0
  Memory usage: 13%               IPv4 address for enp0s3: 10.0.2.15
  Swap usage:   0%


Expanded Security Maintenance for Applications is not enabled.

0 updates can be applied immediately.

4 additional security updates can be applied with ESM Apps.
Learn more about enabling ESM Apps service at https://ubuntu.com/esm

New release '22.04.2 LTS' available.
Run 'do-release-upgrade' to upgrade to it.


vagrant@mnvm:~$
```

Now, test if the VM is correctly configured by testing mininet with following command:

```
$ sudo mn --test pingall
```

This will show following messages:

```
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Waiting for switches to connect
s1
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
*** Stopping 1 controllers
c0
*** Stopping 2 links
..
*** Stopping 1 switches
s1
*** Stopping 2 hosts
h1 h2
*** Done
completed in 6.060 seconds
```

Congratulations! You now have installed a mininet on your local environment.

Environment Setup on Ubuntu 20.04

If your machine does not support Virtualbox, try installing Ubuntu 20.04 with other virtualization software, or natively. For M1/M2 macbooks, UTM is a possible candidate. Note that choosing an option to update while installing Ubuntu 20.04 might update Ubuntu to 22.04, which makes Mininet not work. Currently, Mininet 2.3.0 does not support Ubuntu 22.04 and some other latest OSes.
If you have prepared Ubuntu 20.04 machine, start from cloning github repository:

```
$ git clone https://github.com/NetSP-KAIST/cs341-23s-lab3-handout.git
```

Setup VM environment by following command:

```
$ cd cs341-23s-lab3-handout
$ ./setup.sh
```

Now, test if the VM is correctly configured by testing mininet with following command:

```
$ sudo mn --test pingall
```

This will show following messages:

```
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Waiting for switches to connect
s1
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
*** Stopping 1 controllers
c0
*** Stopping 2 links
..
*** Stopping 1 switches
s1
*** Stopping 2 hosts
h1 h2
*** Done
completed in 6.060 seconds
```

Congratulations! You now have installed a mininet on your local environment.

Task 1: Building Your Own Network

Here, you will implement a "network topology builder", which generates a network topology from a given network graph.

open topology.py, you will see following code:

```
#!/usr/bin/python3

from typing import List, Tuple

from mininet.topo import Topo

class Topology(Topo):
    def build(self, switches: List[str], hosts: List[str], links: List[Tuple[str,
int, str, int, int]]) -> None:
        # KAIST CS341 SDN Lab Task 1

        # input:
        # - switches: List[str]
        #     -> List of switch names
        # - hosts: List[str]
        #     -> List of host names
        # - links: List[Tuple[str,int,str,int,int]]
        #     -> List of links, which is represented by a tuple
        #        The first and the second components represents name of the
components
        #        The third component represents cost of the link; not used in this
task

        ###
        # YOUR CODE HERE
        ###
```

You can test your code by following:

```
$ sudo python3 test.py --task 1
```

This will open Mininet CLI. Type `help` for available commands and other information. You can check your link state with `net`, and check connectivity with `pingall`.
If properly configured, the result of the net should match with the given graph, and pingall should make a 0% drop (because we will use connected graphs only for testing in this lab).

```
mininet> net
h1 h1-eth0:s2-eth2
h2 h2-eth0:s4-eth4
h3 h3-eth0:s3-eth2
s1 lo:  s1-eth1:s4-eth2
s2 lo:  s2-eth1:s4-eth3 s2-eth2:h1-eth0
s3 lo:  s3-eth1:s5-eth2 s3-eth2:h3-eth0
s4 lo:  s4-eth1:s5-eth1 s4-eth2:s1-eth1 s4-eth3:s2-eth1 s4-eth4:h2-eth0
s5 lo:  s5-eth1:s4-eth1 s5-eth2:s3-eth1
c0
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3
h2 -> h1 h3
h3 -> h1 h2
*** Results: 0% dropped (6/6 received)
```

Task 2: Getting familiarized with POX

So far, all the networks in mininet you tested are composed of switches and hosts. From now on, there will be one more type of node in the network, a “controller”. In our experiments, there exists one controller in a network, and it manages forwarding rules of all the switches. In a given

Mininet network, a controller connects to all the switches, and pushes forwarding rules at the network startup phase.

Here, you will learn how to build a logic for a POX controller, using OpenFlow. You are required to implement a very basic forwarding rule, by writing a couple of algorithms for the controller.

Now open `route.py`, you will see following code:

```
import pox.openflow.libopenflow_01 as of

# KAIST CS341 SDN Lab Task 2, 3, 4
#
# Task 2: Getting familiarized with POX
#   - Let switches "flood" packets
#
# Task 3: Implementing a Simple Routing Protocol
#   - Let switches route via Dijkstra
#   - Match ARP and IPv4 packets
#
# Task 4: Implementing a Custom Routing Protocol
#   - Let switches route via Fog routing
#   - Keep routing ARP packets with Dijkstra, and use Fog routing on IPv4 only

###
# If you want, define global variables
###

def init(net):
    #
    # net argument has following structure:
    #
    # net = {
    #    'hosts': {
    #         'h1': {
    #             'name': 'h1',
    #             'IP': '10.0.0.1',
    #             'links': [
    #                 # (node1, port1, node2, port2, link cost)
    #                 ('h1', 1, 's1', 2, 3)
    #             ],
    #         },
    #         ...
    #     },
    #     'switches': {
    #         's1': {
    #             'name': 's1',
    #             'links': [
    #                 # (node1, port1, node2, port2, link cost)
    #                 ('s1', 2, 'h1', 1, 3)
    #             ]
    #         },
    #         ...
    #     }
    # }
    #
    ###
    # YOUR CODE HERE
    ###
    pass

def addrule(switch, connection):
    #
    # This function is invoked when a new switch is connected to controller
    # Install table entry to the switch's routing table
    #
    # For more information about POX openflow API,
    # Refer to [POX official document](https://noxrepo.github.io/pox-doc/html/),
    # Especially [ofp_flow_mod - Flow table
modification](https://noxrepo.github.io/pox-doc/html/#ofp-flow-mod-flow-table-modif
ication)
    # and [Match
Structure](https://noxrepo.github.io/pox-doc/html/#match-structure)
    #
    # your code will be look like:
    # msg = ....
    # connection.send(msg)
    ###
    # YOUR CODE HERE
    ###
    pass
```

There are two functions: init and addrule

- The init is called only once at network initialisation, before the addrule function. The function learns how the network consists via the first argument. The argument includes information about hosts, switches, and links.
- The addrule is called when a controller connects to a switch. Because the controller connects to all the switches, the function will be called N times, where N is the number of switches in the network. The first argument is the name of the connected switch (ex: s1), and the second argument is a connection object described in the official document. You do not need to know too many details about the object, the only thing you will need is that you can push new rules to switch by `connection.send`. You can find several examples in the official document.

In this task, your controller should let switches to “flood” incoming packets, by forwarding packets to all ports except the ingress port. In this Lab, we will only test two types of packets, ARP and ICMP via IPv4, and you can ignore other packets. You may refer to EtherType to check if packets are those types.

To test your code, you should open two sessions with two terminals. The first session will execute the POX controller, and the second session will initiate the Mininet composed of switches and hosts. Launch POX controller first, and Mininet later, so that switches in the Minnet can find the controller to connect. Execute following commands in that order, at each terminal session:

`$ sudo pox/pox.py misc.controller`	`$ sudo python3 test.py --task 2`

This command will initialize the Mininet network with POX controller with your implemented logic. The startup takes some time, waiting for all switches to connect a controller and pushing rules. This command will finally open Mininet CLI as you have already used in Task 1. You can test the network with `pingall` command, which should yield 0% drop rate.

- This task will *not* be graded. However, we recommend you to do this task, as we think this helps you understand how to use OpenFlow before following tasks.
- You may not need to use the init function here.

Task 3: Implementing a Simple Routing Protocol

In this task, you will implement a simple, straightforward routing protocol: Dijkstra. You should let the switches route packets with minimum cost. Now, you should compute how to forward packets beforehand, make rules for all packets, and push rules to switches.
Your task is composed of three sub tasks:

1. Parse network structure at init function
2. Compute forwarding rules for all switches
3. Push forwarding rules at addrule function

You can test your code by following command in each terminal session:

`$ sudo pox/pox.py misc.controller`	`$ sudo python3 test.py --task 3`

As described in Task 2, this command will initialize the Mininet network with POX controller with your implemented logic. You can test your logic with the `pingall` command, which should make 0% packet drop rate. You may use shell commands in Mininet CLI, with pre-defined variables: host and switch names. For example, `h1 ping h2` and `h1 tcpdump -w h1.pcap &`. You may get more information from the official document.

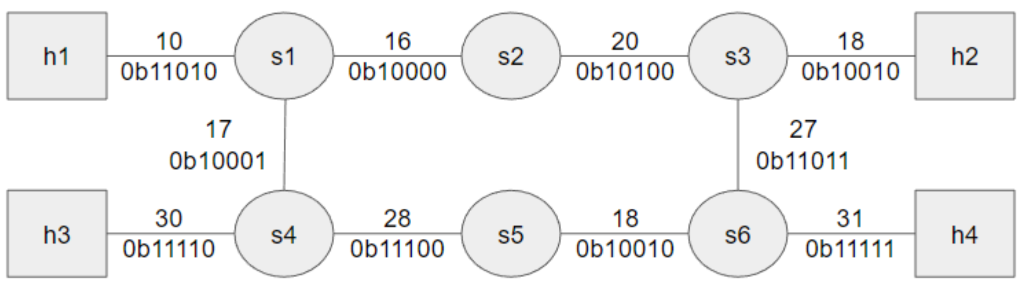
- Your switch should forward one packet to one output port.
- You may want to edit `graph.py`, managing construction of network structure
- You cannot use flood rules here. If you simply flood messages to all ports (i.e., no meaningful routing in practice) you will surely get 0 points even if pingall yield a 0% drop rate.
- Remember that you need to apply Dijkstra to both ARP and IPv4 packets. We will test both when grading.

Task 4: Implementing a Variant Protocol of Dijkstra

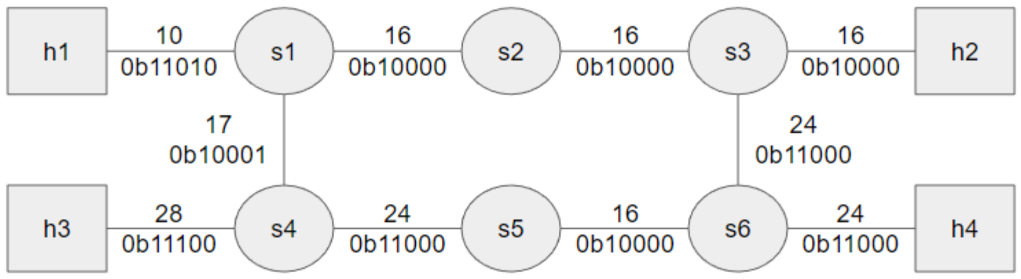
Here, you will implement a variant routing protocol of Dijkstra, which we call a fog routing, to your network. This is a routing protocol we invent for this Lab (so, waste no time for googling it). This routing protocol has the key characteristics of "blurring" out link weight information that is farther away from the observer; hence, named a "fog" routing. The fog routing works as following:

- To hide link cost information to switches, a link cost gets "blurred" when the distance between a link and a switch is far.
- The "blurring" is done by removing the least significant positive bit; i.e., changing the rightmost 1 to 0 in a binary representation of link weights while not making link weights zero.
    - Ex) Assume a four-bit link weight. A link weight of 13 becomes 12, and 8 as it propagates through each hop of switches; and it stays at 8 after that. This is because as follows: 13 (0b1101) → 12 (0b1100) → 8 (0b1000) → 8 (0b1000).
- If the link is N hops away from the switch, the "blurring" is done N times in a row.
    - Assume a network with following network:

Ground-truth link cost value



link cost value in s1's point-of-view



    - In normal Dijkstra, s1 might route packets to h4 by forwarding it to either s2 or s4, since total cost is same (16+20+27+31 = 17+28+18+31)
    - In Fog routing, s1 should route packets to h4 by forwarding to s2 (16+16+24+24 < 17+24+16+24)

Your main task is to implement the fog routing to your IPv4 packets, and keep Dijkstra routing for your ~~ICMP~~ARP packets.

Testing command is similar with Task 3:

`$ sudo pox/pox.py misc.controller`	`$ sudo python3 test.py --task 4`

Task 5: Finding bug in the Custom Routing Protocol

In this task, you should provide an example network that explains why the fog routing is rather problematic. The fog routing is an interesting tweak to Dijkstra but it contains a serious problem. Find a network structure where the routing protocol demonstrates the problem, and show it.

Note that you do not need to fix the problem. Demonstrating the problem would be sufficient for this lab.

Open graph.py, and you will see following code:

```
#!/usr/bin/python3

import random

def gen_graph(task=1):

    # increase number of switches and hosts to make more complex and bigger network
    switch=5
    host=3
    switches = ['s{}'.format(i) for i in range(1, 1+switch)]
    hosts = ['h{}'.format(i) for i in range(1, 1+host)]
    links = []

    switchneighbors = {s:set() for s in switches}

    if task in range(1,5): # For task 1, 2, 3, 4
        # generate random tree topology
        # First, connect switches in random order
        switchorder = list(switches)
        random.shuffle(switchorder)
        for i in range(1, switch):
            # attach swith to one among existing switches
            node1 = random.choice(switchorder[:i])
            node2 = switchorder[i]
            links.append((
                node1,
                node2,
                random.randint(1 << 15, 1 << 30)
            ))
            switchneighbors[node1].add(node2)
            switchneighbors[node2].add(node1)
        # Next, connect hosts to switches
        for host in hosts:
            node1 = host
            node2 = random.choice(switches)
            links.append((
                node1,
                node2,
                random.randint(1 << 15, 1 << 30)
            ))

    if task in range(3,5): # For task 3, 4
        # For testing Dijkstra and Fog routing,
        # Add more links between switches, possibly making loop
        for _ in range(switch * 2):
            s1 = random.choice(switches)
            s2 = random.choice(switches)
            if (s1 != s2) and (s1 not in switchneighbors[s2]) and (s2 not in
switchneighbors[s1]):
                switchneighbors[s1].add(s2)
                switchneighbors[s2].add(s1)
                links.append((
                    s1,
                    s2,
                    random.randint(1 << 15, 1 << 30)
                ))
    if task == 5:
        ## Task 5: Finding bug in the Custom Routing Protocol
        #
        # Add links to connect switches and hosts
        # hosts should have only one link that is connected to a switch
        pass
    return (switches, hosts, links)
```

Currently, the gen_graph function generates a random network structure. It generates simple tree structure for task 1 and 2, and adds more links on the tree for Task 3 and 4. You should write your own example network on `task == 5` section, which makes the fog routing fails reliable routing. Again, testing code is similar with Task 2 to 4:

`$ sudo pox/pox.py misc.controller`	`$ sudo python3 test.py --task 5`

- your code should not make language-level problems: Exceptions, Errors, etc. Mininet CLI should be opened without any problem

Troubleshooting

On `vagrant` up, timed out while waiting for machine to boot error

When following messages show up:

```
cs341-23s-lab3-handout>vagrant up
Bringing machine 'mnvm' up with 'virtualbox' provider...
==> mnvm: Importing base box 'ubuntu/focal64'...
==> mnvm: Matching MAC address for NAT networking...
==> mnvm: Checking if box 'ubuntu/focal64' version '20230506.0.0' is up to date...
==> mnvm: Setting the name of the VM:
cs341-23s-lab3-handout_mnvm_1683656478422_26689
```

```
==> mnvm: Clearing any previously set network interfaces...
==> mnvm: Preparing network interfaces based on configuration...
    mnvm: Adapter 1: nat
==> mnvm: Forwarding ports...
    mnvm: 22 (guest) => 2222 (host) (adapter 1)
==> mnvm: Running 'pre-boot' VM customizations...
==> mnvm: Booting VM...
==> mnvm: Waiting for machine to boot. This may take a few minutes...
    mnvm: SSH address: 127.0.0.1:2222
    mnvm: SSH username: vagrant
    mnvm: SSH auth method: private key
Timed out while waiting for the machine to boot. This means that
Vagrant was unable to communicate with the guest machine within
the configured ("config.vm.boot_timeout" value) time period.

If you look above, you should be able to see the error(s) that
Vagrant had when attempting to connect to the machine. These errors
are usually good hints as to what may be wrong.

If you're using a custom box, make sure that networking is properly
working and you're able to connect to the machine. It is a common
problem that networking isn't setup properly in these boxes.
Verify that authentication configurations are also setup properly,
as well.

If the box appears to be booting properly, you may want to increase
the timeout ("config.vm.boot_timeout") value.
```

Open Virtualbox manually, and then open a newly created machine (named cs341-23s-lab3-handout-mnvm_xxxxxxxxxxxxx_xxxxx), and wait for its boot. If the VM screen shows login page (showing `ubuntu-focal login:` text), try `vagrant up` again

Address already in use error while executing POX controller

Error log similar to below shows up, failing to execute POX controller

```
INFO:core:POX 0.7.0 (gar) is up.
ERROR:openflow.of_01:Error 98 while binding 0.0.0.0:6633: Address already in use
ERROR:openflow.of_01: You may have another controller running.
ERROR:openflow.of_01: Use openflow.of_01 --port=<port> to run POX on another port.
```

Usually, killing already running POX controller solves problem:

```
vagrant@mnvm:~$ sudo pkill python3
```

If you want some python scripts not to be killed, you may check all listening ports, get process ID, and kill it manually.

If the problem still exists, try killing pox controller and ovs virtual switches:

```
vagrant@mnvm:~$ sudo pkill python3; sudo pkill ovs-vswit
```

and reload your machine:

```
yourname@yourmachine: ~/yourpath$ vagrant reload
```

Error occurs while executing Mininet: `File exists` exception

Error log similar to below shows up, failing to execute mininet operation

```
RTNETLINK answers: File exists
Traceback (most recent call last):
  File "test_task2.py", line 28, in <module>
    net = Mininet(topo=t, controller=RemoteController)
  File "/usr/local/lib/python3.8/dist-packages/mininet/net.py", line 178, in
__init__
    self.build()
  File "/usr/local/lib/python3.8/dist-packages/mininet/net.py", line 508, in build
    self.buildFromTopo( self.topo )
  File "/usr/local/lib/python3.8/dist-packages/mininet/net.py", line 495, in
buildFromTopo
    self.addLink( **params )
  File "/usr/local/lib/python3.8/dist-packages/mininet/net.py", line 406, in
addLink
    link = cls( node1, node2, **options )
  File "/usr/local/lib/python3.8/dist-packages/mininet/link.py", line 456, in
__init__
    self.makeIntfPair( intfName1, intfName2, addr1, addr2,
  File "/usr/local/lib/python3.8/dist-packages/mininet/link.py", line 501, in
makeIntfPair
    return makeIntfPair( intfname1, intfname2, addr1, addr2, node1, node2,
  File "/usr/local/lib/python3.8/dist-packages/mininet/util.py", line 270, in
makeIntfPair
    raise Exception( "Error creating interface pair (%s,%s): %s " %
Exception: Error creating interface pair (h2-eth0,s4-eth2): RTNETLINK answers: File
exists
```

This happens because the previous mininet execution did not end correctly, leaving leftovers. Remove leftovers by following command:

```
$ sudo mn -c
```

`struct.error` while sending openflow message to switch

Error log similar to below shows up, failing to send openflow message to switches

```
ERROR:core:Exception while handling OpenFlowNexus!ConnectionUp...
Traceback (most recent call last):
  File "/home/vagrant/pox/pox/lib/revent/revent.py", line 242, in
raiseEventNoErrors
    return self.raiseEvent(event, *args, **kw)
  File "/home/vagrant/pox/pox/lib/revent/revent.py", line 295, in raiseEvent
    rv = event._invoke(handler, *args, **kw)
  File "/home/vagrant/pox/pox/lib/revent/revent.py", line 168, in _invoke
    return handler(self, *args, **kw)
  File "/home/vagrant/pox/pox/misc/controller.py", line 41, in connectionUp
    route.addrule(switchname, event.connection)
  File "/home/vagrant/route.py", line 87, in addrule
    connection.send(msg)
  File "/home/vagrant/pox/pox/openflow/of_01.py", line 875, in send
    data = data.pack()
  File "/home/vagrant/pox/pox/openflow/libopenflow_01.py", line 2349, in pack
    packed += i.pack()
  File "/home/vagrant/pox/pox/openflow/libopenflow_01.py", line 1586, in pack
    packed += struct.pack("!HHHH", self.type, len(self), self.port,
struct.error: required argument is not an integer
```

In this case, you are passing arguments with the wrong type while you are creating open flow messages. In the example above, the port number should be integer. If you pass port numbers with types other than int, the error above occurs. Refer to POX document for correct argument types.

Useful Information

- If you have installed through vagrant, the files you should edit (/home/vagrant/*.py in the VM machine) are actually symbolic links of scripts in the synced folder. Modification of original file (/cs341-23s-lab3-handout/*.py in the host machine) is applied in the files in the VM, and vice versa.
- For Task 2 to Task 5, you can see forwarding of packets in the switches by tcpdump-ing network interfaces. While your Mininet is running, type ifconfig to see the interfaces of your switches; it will be displayed as `s1-eth1`, `s1-eth2`, `s2-eth1`, …
- If you want to filter unhandled packets in the controller session, try looking at controller.py
- For Lab1 to 4, random network structure is made at every test. You can find the network structure at `/tmp/net.json`

Submissions

Zip `topology.py`, `route.py`, and `graph.py` into single zip file
Upload the zip file via gradescope.
Gradescope sometimes returns turnitin error. In this case, try submitting a few minutes later.