

CSCI 4211: Introduction to Computer Networks

Spring 2024

PROGRAMMING PROJECT 3: SDN and Mininet

Phase 3: Self-Learning Ethernet Switch

Due: April 28th, 2024 at 11:59 p.m.

Note: The use of chatGPT and similar bots is strictly prohibited for this project.

Table of Contents

Table of Contents.....	1
1. Description.....	2
2. Design and Implement An Ethernet-Based Self-Learning Algorithm.....	2
2.1 High-Level Algorithm Logic.....	2
2.2 Code Execution.....	4
2.2.1 Running Your Controller with POX.....	4
2.2.2 Running Your Controller with Floodlight.....	4
2.3 Step-by-Step Implementation Instructions.....	5
2.4 Evaluate Your Ethernet-Based Self-Learning Algorithm.....	8
2.5 Extra Credit [15 Points].....	10
2.5.1 Description.....	10
2.5.2 Testing Your Router.....	11
2.5.3 Helpful Information.....	12
2.6 Provided Starter Files.....	13
2.6.1 Self-Learning Ethernet Switch.....	13
2.6.2 Topologies A and B.....	13
2.6.3 Topology C.....	13
2.6.4 Topology D [Extra Credit].....	13
2.6.5 Self-Learning IP Router [Extra Credit].....	13
3. Helpful Resources and General Advice.....	14
3.1 Handy Terminal Commands.....	14
3.2 References.....	14
4. Submission Information.....	15
4.1 Rubric.....	15
4.2 What To Submit.....	17
4.3 Common Error Checks.....	18

[Back To The Table of Contents](#)

1. Description

In this phase, you will learn how to use Mininet to implement an Ethernet-based self-learning algorithm using an SDN controller.

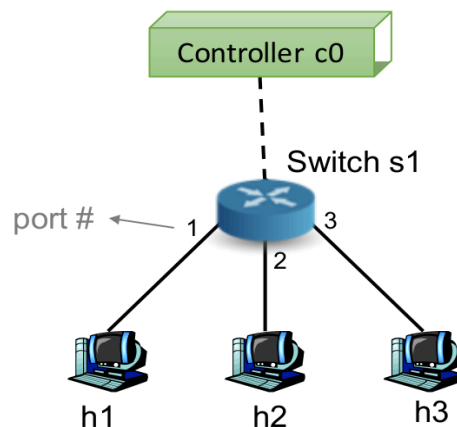
The controller can be implemented using Python's POX or Java's Floodlight:

- Both provide a framework for communicating with SDN switches using the well-defined application programming interface (API), OpenFlow protocol. The controller exercises direct control over the state in the SDN switches via OpenFlow protocol.
- POX uses Python, while Floodlight is intended to run with standard JDK tools and can optionally be run in Eclipse.

2. Design and Implement An Ethernet-Based Self-Learning Algorithm

2.1 High-Level Algorithm Logic

Consider Topology C:



Switch s1 maintains a flow table that contains match-action rules that are added/removed/updated by the controller. In the beginning, there are no rules installed in the switch's flow table. When a packet is sent from one host to another, and s1 doesn't know how to reach the packet's destination host, it will send the packet to c0, the controller. The controller will extract information from this packet and use it to build/update an internal global data structure (such as a hashmap, dictionary, etc.) to represent the network topology. In other words, the controller maintains information about where each host can be located using which port on a per-switch basis (i.e., a

[Back To The Table of Contents](#)

collection of switch tables that each map host MAC addresses to one of the switch's ports). The controller will use this data structure to decide what rules should be installed in the switch's flow table.

For example, consider host h1 wants to send a packet to h2. When switch s1 receives this packet from h1, it checks if there is any corresponding match-action rule (or flow entry) for this packet in its flow table. If there is no matching entry, then the switch will encapsulate the packet and send it inside an `OFPacketIn` message to the controller. On the other hand, if there already *is* a matching flow entry, then switch s1 will execute the action associated with the installed rule. However, as discussed earlier, the flow table is initially empty. Therefore, the switch will forward the packet to the controller.

When the controller receives the `OFPacketIn` message from switch s1, a `PacketIn` event is triggered. Two tasks should be performed at the controller:

- a. Using the packet's source MAC address and input port of the sending switch, the controller will update its internal network topology map if this information hasn't been encountered before.
- b. Using its current internal state and view of the network topology, the controller decides on what action(s) s1 should take to send this packet to h2 based on the following:
 1. If the controller already knows what interface s1 should forward the packet to reach h2, then:
 - a. Check if the input port matches the output port. If so, then drop the packet and return from the function.
 - b. Otherwise, inform s1 and install one or more flow table rules in it which could be used for future packet transmissions of a similar kind (see [Section 2.3](#)).
 2. Otherwise, if the controller does not know what interface s1 should use to forward the packet to reach h2, then:
 - a. Instruct the switch to flood the packet to all its interfaces except the one the packet was originally received from by s1. This flooding instruction is a one-time action without installing any rules in the switch. Also, increase a global counter to count the number of total flooded messages according to the number of ports on the switch.

Hint: Revisit the ethernet-learning algorithm from Lecture 16: Network Layer - Data Plane Part IV as a refresher.

2.2 Code Execution

When you start your topology (using Python) it will connect to this controller, so run your controller first in one terminal, and then run the target topology in another terminal. Also, in the `main()` function of `topology-a.py` and `topology-b.py`, you will need to comment out and comment in the code lines that have been indicated in the provided TODO comments so that your controller is used instead of the default controller. However, in `topology-c.py`, you don't need to change the code at all.

2.2.1 Running Your Controller with POX

Copy the `ethernet-learning.py` file into the `/home/mininet/pox/pox/samples` directory. You will then run the controller module using the following commands:

- `you@yourmachine:~$ cd pox`
- `you@yourmachine:~/pox$./pox.py samples.ethernet-learning`

Note: Do not include the `.py` file extension when running `ethernet-learning.py`.

2.2.2 Running Your Controller with Floodlight

Note: This is a more involved process. You may have to first follow the prerequisite instructions for [modifying the VM](#).

Follow the following steps:

- Put the `EthernetLearning.java` file in the `src/main/java/net/floodlightcontroller/ethernetlearning/` directory.
- You need Floodlight to load the module on startup. First, you must add the module to the loader. This is done by adding the fully qualified module name on its own line in `src/main/resources/META-INF/services/net.floodlightcontroller.core.module.IFloodlightModule`. Open that file and append this line to the end of it:
`net.floodlightcontroller.ethernetlearning.EthernetLearning`.
- To load the module, modify the Floodlight module configuration file to append the `EthernetLearning` module. The default one is `src/main/resources/floodlightdefault.properties`.
 - The key is `floodlight.modules` and the value is a comma-separated list of fully qualified modules
 - `names.floodlight.modules = <leave the default list of modules in place>`,
`net.floodlightcontroller.ethernetlearning.EthernetLearning`

[Back To The Table of Contents](#)

- Proceed with re-building the controller using the following commands:
 - `you@yourmachine:~$ cd floodlight/target`
 - `you@yourmachine:~/floodlight/target$ ant`
- Finally, run the controller by using the `floodlight.jar` file produced by the `ant` command from within the floodlight directory:
 - `you@yourmachine$ java -jar target/floodlight.jar`
- Floodlight will start running and print the log and debug output to your console.

2.3 Step-by-Step Implementation Instructions

Note: All screenshots taken and answers to the questions below must be included in your submitted Phase 3 report.

1. In a terminal, run your controller that hasn't been implemented yet (in `ethernet-learning.py`) using the appropriate steps in [Section 2.2](#).
2. In a terminal, run Topology C: `sudo python topology-c.py`
3. In the `mininet` terminal, do the following:
 - a. Attempt to have h1 Ping h2: `h1 ping h2 -c 1`
 - i. Was this attempt successful?
 - b. Dump the flow rules currently installed on the switch: `dpctl dump-flows`
 - i. Have any rules been installed yet?
 - c. Briefly explain what could be the cause of your answers to questions 3a.i and 3b.i.
 - d. Take a screenshot of the terminal output for the commands in parts 3a and 3b.
 - e. Shut down the controller by using `Ctrl+C` and Topology C.
4. Now, implement `ethernet-learning.py` according to [Section 2.1](#). When you get to the part of the algorithm where the controller installs flow table rules in a switch, install rules that match on all fields of a given packet.

The `message.match` object has quite a few potential matching fields:

Attribute	Meaning
in_port	Switch port number the packet arrived on
dl_src	Ethernet source address
dl_dst	Ethernet destination address
dl_type	Ethertype / length (e.g. 0x0800 = IPv4)
nw_tos	IPv4 TOS/DS bits
nw_proto	IPv4 protocol (e.g., 6 = TCP), or lower 8 bits of ARP opcode
nw_src	IPv4 source address
nw_dst	IP destination address
tp_src	TCP/UDP source port
tp_dst	TCP/UDP destination port

It is also possible to create a `message.match` object that matches all fields of a given packet. Use the following code line in your implementation:

```
message.match = of.ofp_match.from_packet(packet, event.port)
```

- a. Run your controller again using the steps in [Section 2.2](#).
- b. In a terminal, run Topology C: `sudo python topology-c.py`
- c. In the `mininet` terminal, open three XTerm terminals, one for each host in Topology C: `xterm h1 h2 h3`
- d. In the terminals for h2 and h3, run `tcpdump` to monitor the packets received by each:
 - i. For h2: `tcpdump -n -i h2-eth0`
 - ii. For h3: `tcpdump -n -i h3-eth0`
- e. In the terminal for h1, have h1 Ping h2 three times: `ping 10.0.0.2 -c 3`
 - i. Comment on what you see in each of the three host terminals and provide screenshots of them.
 - ii. After the first Ping packet is sent and the flow rules are installed in s1, is there a difference in latency observed by h1 for the other two Ping packets? Why or why not?

Note: Your screenshots should support your answer.

[Back To The Table of Contents](#)

- f. Close the h3 terminal and have h2 exit `tcpdump` by using `Ctrl+C`.
- g. In the h1 and h2 terminals, run iPerf:
 - i. For h2: `iperf -s -p 4000`
 - ii. For h1: `iperf -c 10.0.0.2 -p 4000`

Note: The iPerf server must run before the iPerf client.

- iii. Did iPerf work?
 - iv. Were any of the iPerf packets sent to the controller?
 - v. How many rules are now installed in s1?
 - vi. Briefly explain what could be the cause of your answers.
 - vii. Take a screenshot of the terminal output h1 and h2.
 - h. Shut down the controller by using `Ctrl+C` and Topology C.
5. In your controller code, now install flow table rules that match only on the source and destination MAC addresses, instead of all fields, of a given packet.

Note: Don't delete your old code from Part 4. Just comment it out and write a comment above it saying that it was from Part 4 of Section 2.3.

Use the following code line in your implementation (you will need to replace `x` and `y` with the correct MAC addresses:

```
message.match = of.ofp_match(d1_src = x, d1_dst = y)
```

Note: You will need to install two flow table rules in the switch that match on the following:

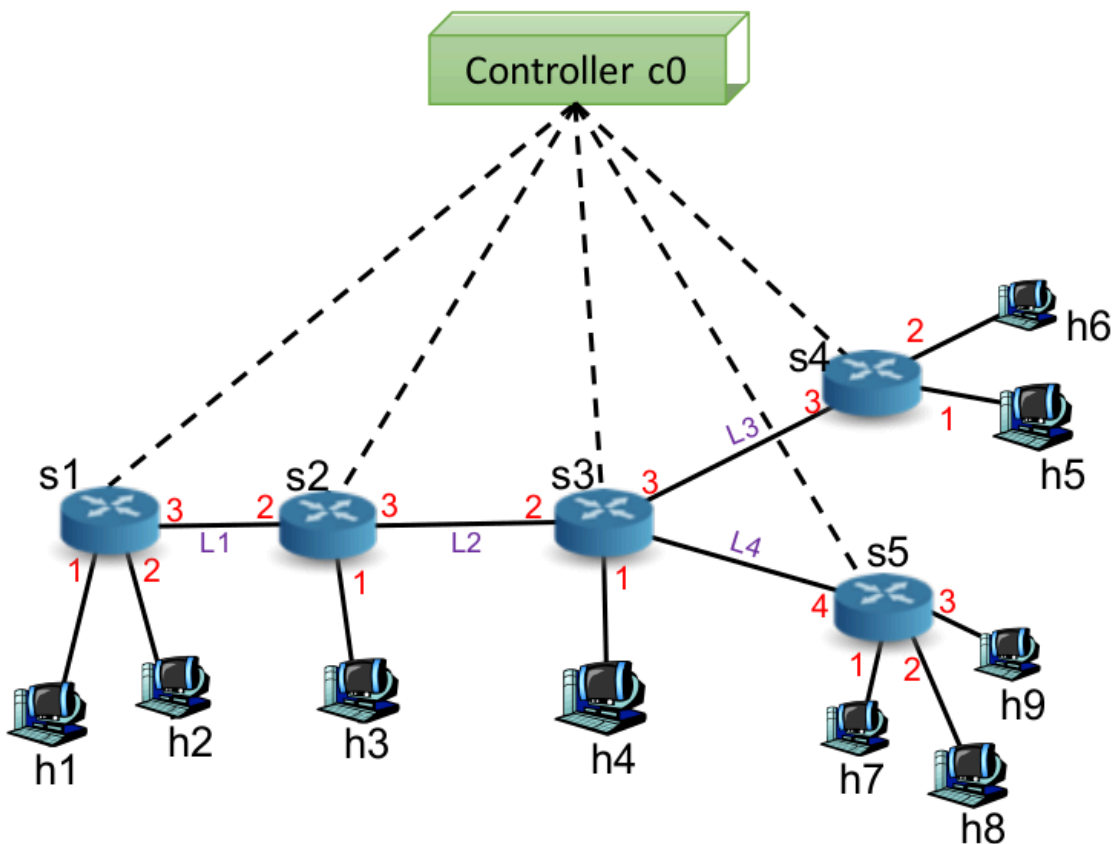
1. `d1_src` = Packet's source MAC address, `d1_dst` = Packet's destination MAC address. Set the action's output port to the one that reaches the destination host.
 2. `d1_src` = Packet's destination MAC address, `d1_dst` = Packet's source MAC address. Set the action's output port to the one that reaches the source host.
6. Run Ping and iPerf again using the same steps in Part 4 of this section.
- a. Did iPerf work?
 - b. Were any of the iPerf packets sent to the controller?
 - c. How many rules are installed in s1?
 - d. Briefly justify your answer.
 - e. Take a screenshot of the terminal output h1 and h2.

7. Exit the controller and Topology C and restart them both. Do the following actions and include answers to the following questions and screenshots:
 - a. The Ping results that each host can reach all the other hosts. Do this by running the `pingall` command after running a topology.
 - i. How many packets were sent to the controller?
 - ii. Show the rules that are installed in all switches by the controller:
`dpctl dump-flows`
 - iii. Show the number of messages flooded by the controller.
 - b. Run `pingall` again. How many packets were sent to the controller this time? Briefly justify your answer.

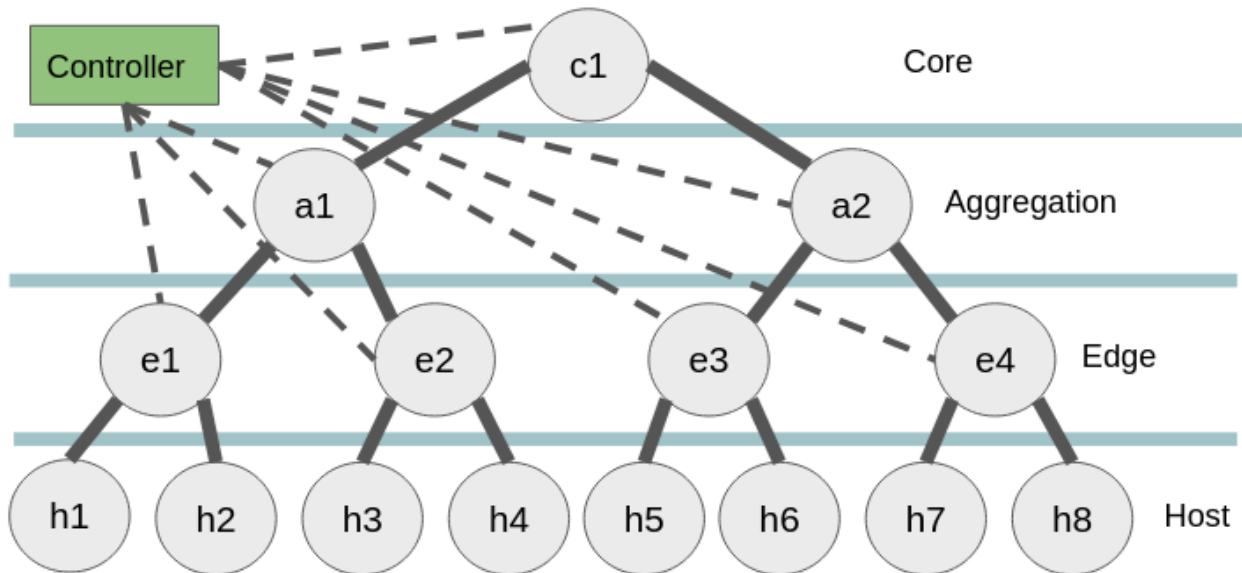
2.4 Evaluate Your Ethernet-Based Self-Learning Algorithm

Test your algorithm using POX (Python) or Floodlight (Java) when flow table rules are installed matching on just the source and destination MAC addresses of a given packet:

- Against Topology A:



- Against Topology B:



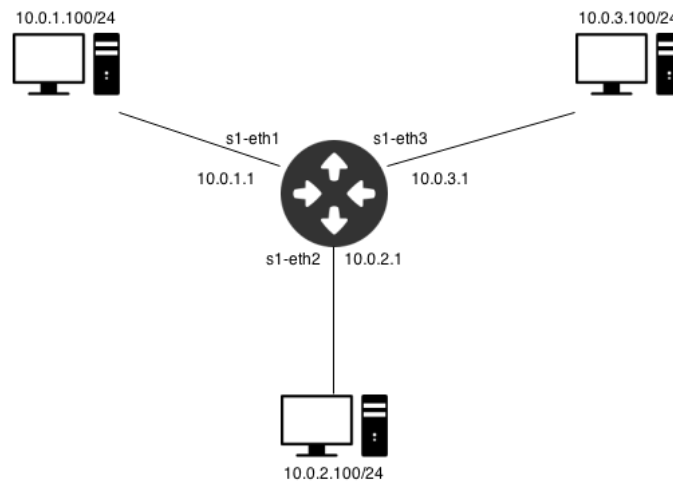
In your report for both topologies A and B, include answers to the following proposed questions and the required screenshots:

- Show the Ping results that each host can reach every other host. Do this by running the `pingall` command after running a topology.
 - How many packets were sent to the controller?
 - Show the rules that are installed in all switches by the controller: `dpctl dump-flows`
 - Show the number of messages flooded by the controller.
- Measure the estimated latency and throughput for each link between the switches using the exact same process you used to complete Phase 2. Make sure to include your results in your report.
- Are the performance results similar to what is defined for each link in the topology definition (i.e., in `topology-a.py` and `topology-b.py`)? Are your results similar to what you estimated in Phase 2? Why or why not?

2.5 Extra Credit [15 Points]

2.5.1 Description

Consider Topology D:



In this part, your task is to make a static layer-3 forwarder/switch that will emulate a network router. It will examine the IP header of a received packet, remove and process its Data Link layer header, modify the IP header if required, and replace the Data Link layer header for forwarding. Since this is a static router, you do not need to send or receive routing table updates.

Each network node will have a configured subnet. If a packet is destined for a host within that subnet, then the node acts as a switch and forwards the packet with no changes to a known port or broadcasts it, just like how you did with the self-learning Ethernet switch. If a packet is destined for some IP address for which the router knows the next hop, then it should modify the source and destination MAC addresses and forward the packet to the correct port.

A router generally has to respond to ARP requests. You will see Ethernet broadcasts which will (initially at least) be forwarded to the controller. Your controller should construct ARP replies and forward them to the appropriate ports.

Static Routing:

- Once ARP has been handled, you will need to handle routing for the static network configuration. Since we know what is connected to each port, we can match on the destination IP address (or prefix, in the case of the remote subnet) and forward the packet to the appropriate port.

- You need to handle all IPv4 traffic that comes through the router by forwarding it to the correct subnet. The only change in the packet should be the source and destination MAC addresses.
- If the router does not know the MAC address of the destination host, then it will have to send an ARP request for that host.

Additionally, your controller may receive ICMP echo (Ping) requests for the router, which it should respond to. Lastly, packets for unreachable subnets should be responded to with ICMP network unreachable messages.

Sample Static Routing Table:

```
[subnet prefix, IP address of a host, interface name, interface address, switch port]
['10.0.1.100/24', '10.0.1.100', 's1-eth1', '10.0.1.1', 1],
['10.0.2.100/24', '10.0.2.100', 's1-eth2', '10.0.2.1', 3],
['10.0.3.100/24', '10.0.3.100', 's1-eth3', '10.0.3.1', 2]
```

2.5.2 Testing Your Router

Your code is considered completely correct if it can handle all of the following test cases:

- Attempts to send a Ping from 10.0.1.100 to an unknown address range like 10.99.0.1 should yield an ICMP network unreachable message being received by the original sender and the Ping should ultimately fail. Show the Ping messages sent and received by taking screenshots of your terminal and Wireshark output.
- Packets sent to hosts on a known address range should have their destination MAC address changed accordingly. Should the source MAC address also be changed in this case, why or why not? If yes, then include this in your implementation.
- All of the router's interfaces (IP addresses) should be Pingable, and the router should generate an ICMP echo reply in response to an ICMP echo request.
- First, use iPerf and measure the throughput. Note the results of this test. In your code's current state, the router sends all of the traffic to the controller via **OFPacketIn** messages, which makes a routing decision and sends an **OFPacketOut** message to the router. The next step is to install flow modification rules, which should yield better performance with iPerf.

2.5.3 Helpful Information

Host setup:

- You'll need to configure each host with a subnet, IP, gateway, and netmask.
- Example:

```
host1 = self.addHost('h1', ip = "10.0.1.100/24", defaultRoute =  
    "via 10.0.1.1")
```

General packet structure/header fields and parsing code:

- `packet = event.parsed` : The event contains the Ethernet packet.
- `protocol = packet.payload` : Strips the Ethernet header and contains the IPv4 or ARP packet. Your code should check this information and behave accordingly.
- `packet.payload.payload` : Strips the protocol header and contains the underlying packet, which may be an ICMP packet.
- `packet.payload.payload.payload` : Strips the ICMP header and contains the echo request or reply or network unreachable message, assuming this is an ICMP packet.

ARP packet structure/header fields:

- `hwsrc` : Source hardware address.
- `hwdst` : Destination hardware address which is what we are asking for in the ARP reply.
- `protosrc` : Source IP address.
- `protodst` : Destination IP address.
- `opcode` : The type of ARP packet (e.g., REQUEST, REPLY, REV_REQUEST, REV_REPLY).

2.6 Provided Starter Files

Note: These files include useful TODO comments for what you need to do and where. Remember that the TODO comments are purposefully not a complete step-by-step guide that you should blindly follow and that you will need to think independently about what else you may need to implement. If a piece of code doesn't have a TODO comment, then it should not be modified at all.

2.6.1 Self-Learning Ethernet Switch

Implement your algorithm/controller logic according to the design and implementation details in Sections [2.1](#) and [2.3](#) using the provided skeleton code in `ethernet-learning.py` or `EthernetLearning.java`. Also, don't forget to use Lecture 16: Network Layer - Data Plane Part IV and Lecture 17: Network Layer - Data Plane Part V as a guide.

2.6.2 Topologies A and B

To test your controller, you will need to modify `topology-a.py` and `topology-b.py` so that when they run, they will use your controller and not the default controller. This can be done in the `main()` function of these files by commenting out and commenting in the code lines that have been indicated in the provided TODO comments. No other modifications are required to be made to these files for this phase.

Note: If you couldn't get Topology B working for Phase 2, then you can request a copy of our implementation through email to use for Phase 3 only after you've submitted Phase 2 or the Phase 2 deadline has passed.

2.6.3 Topology C

The Topology C script (`topology-c.py`) is already fully implemented. It shouldn't be modified at all. This file should only be used when completing parts of [Section 2.3](#) of Phase 3.

2.6.4 Topology D [Extra Credit]

The Topology D script (`topology-d.py`) is a skeleton that you will need to complete according to the network topology figure in [Section 2.5.1](#) and the hint in [Section 2.5.3](#).

2.6.5 Self-Learning IP Router [Extra Credit]

Implement your algorithm/controller logic according to the design and implementation details in [Section 2.5.1](#) using the provided skeleton code in `ip-learning.py` or `IPLearning.java`.

[Back To The Table of Contents](#)

3. Helpful Resources and General Advice

3.1 Handy Terminal Commands


Analyze network traffic:

- `xterm`, `wireshark`, `tcpdump`
- Send a Ping from h1 to h2, run `tcpdump` for h2 and h3. The Ping packets are now going up to the controller, which then floods them out of all interfaces except the sending one.
 - a. `tcpdump -XX -n -i h2-eth0`

Print the rules currently installed on all switches: `dpctl dump-flows`

3.2 References

While almost *any* reference can be helpful to you, the following should be particularly useful when completing this project:

- [OpenFlow Tutorial](#)
-  [Mininet and Remote SDN Controllers \(Floodlight + Pox\)](#)
- [Installing POX — POX Manual Current documentation](#)
- [Floodlight Documentation](#)
- [POX Code Example](#)
- [Create a Learning Switch](#)
- [Mininet Python API Reference Manual](#)
- [Router Exercise \(Extra Credit\)](#)

Get Started Early! Don't wait until the last minute. Starting early will allow you the opportunity to have more time to promptly receive any help that you may require from the instructors and to calmly debug your code if any unexpected and/or difficult problems arise.

4. Submission Information

4.1 Rubric

Before submitting the project, be sure to verify all of your work by using the following rubric:

Required (60 points)					
Criteria	Approximate % Grade	Excellent (100%)	Adequate (80%)	Poor (60%)	Not Met (0%)
Source Code	50%				
Ethernet Algorithm Code	50%	The remote controller runs with no errors. The source code logic is correct and includes useful documentation and comments.	The remote controller runs with no errors. The source code logic is mostly correct (a couple of minor errors) and includes useful documentation and comments.	The remote controller runs with no errors. The source code logic is partially correct (several errors) and may or may not include useful documentation and comments.	1.) The remote controller cannot be run because it has errors. OR 2.) The remote controller runs with no errors but the logic is completely incorrect.
Report	50%				
Implementation Process	30%	Provided correct answers to all of the questions in Section 2.3 and all required screenshots.	Provided correct answers to most of the questions in Section 2.3 and at least most of the required screenshots.	Provided correct answers to some of the questions in Section 2.3 and at least some of the required screenshots.	1.) Provided incorrect answers to all of the questions in Section 2.3 and none or some of the required screenshots. OR 2.) The report is completely missing answers to the questions asked in Section 2.3 .
Ethernet Algorithm Evaluation	20%	Included all of the required parts from Section 2.4 for both topologies A and B.	Included most of the required parts from Section 2.4 for both topologies A and B.	Included some of the required parts from Section 2.4 for both topologies A and B.	Included none of the required parts from Section 2.4 for both topologies A and B.

Extra Credit (15 Points)					
Criteria	Approximate % Grade	Excellent (100%)	Adequate (80%)	Poor (60%)	Not Met (0%)
Source Code	50%				
Topology D Creation	10%	Topology D was created with 0 errors.	Topology D was created with 1-2 minor errors.	Topology D was created with a substantial number of errors.	1.) Topology D was created completely incorrect. OR 2.) topology-d.py was not submitted.
Router Algorithm Code	40%	The remote controller runs with no errors. The source code logic is correct and includes useful documentation and comments.	The remote controller runs with no errors. The source code logic is mostly correct (a couple of minor errors) and includes useful documentation and comments.	The remote controller runs with no errors. The source code logic is partially correct (several errors) and may or may not include useful documentation and comments.	1.) The remote controller cannot be run because it has errors. OR 2.) The remote controller runs with no errors but the logic is completely incorrect.
Report	50%				
Description	20%	The provided high-level description of the algorithm makes complete sense and there are detailed explanations of the data structures that were used.	The provided high-level description of the algorithm mostly makes sense (only a few things weren't clear) and there are detailed explanations of the data structures that were used.	The provided high-level description of the algorithm doesn't make a lot of sense (many things weren't clear) and there may or may not be detailed explanations of the data structures that were used.	The provided high-level description of the algorithm makes absolutely no sense.
"Pseudocode"	10%	The provided "pseudocode" of the implemented algorithm makes complete sense and matches the submitted source code.	The provided "pseudocode" of the implemented algorithm mostly makes sense (only a few things weren't clear) and matches the submitted source code.	The provided "pseudocode" of the implemented algorithm doesn't make a lot of sense (many things weren't clear) and may or may not match the submitted source code.	The provided "pseudocode" of the implemented algorithm makes absolutely no sense.
Router Algorithm Testing	20%	All of the test cases listed in Section 2.5.2 pass.	Most of the test cases listed in Section 2.5.2 pass.	A few of the test cases listed in Section 2.5.2 pass.	None of the test cases listed in Section 2.5.2 pass.

[Back To The Table of Contents](#)

4.2 What To Submit

You must submit to the class Gradescope website a ZIP file (.zip formats), named g<GroupNumber>-project3-phase3.zip where <GroupNumber> is the number of the Mininet Group you joined on Canvas (example: g1-project3-phase3.zip), that contains the following files:

- Your modified Ethernet-based self-learning switch (as `ethernet-learning.py` or `EthernetLearning.java`). Ensure that your code is sufficiently documented and has useful comments for code that isn't immediately understandable to the reader.
- A PDF, named g<GroupNumber>-project3-phase3-report.pdf (example: g1-project3-phase3-report.pdf).
 - Refer to Sections [2.3](#) and [2.4](#) to see what this report should contain. Make sure to clearly label each section of the report.
- If you attempted the extra credit, then submit the following as well in the ZIP:
 - Your modified Topology D (as `topology-d.py`).
 - Your modified IP-based self-learning router (as `ip-learning.py` or `IPLearning.java`). Ensure that your code is sufficiently documented and has useful comments for code that isn't immediately understandable to the reader.
 - In your PDF report, also include the following sections:
 - Description section:
 1. Describe at a high level your self-learning algorithm.
 2. Including detailed explanations of your data structures and their purposes.
 - A "Pseudocode" section:
 1. Describe your algorithm following [Pseudocode](#) guidelines.
 - (Optional) Include any assumptions that were made or comments that you wish for the grader to know beforehand.
 - Screenshots of your results of the different testing scenarios for testing your router (see [Section 2.5.2](#)).

Note: Make sure that you have submitted the required files to Gradescope with the correct filenames.

4.3 Common Error Checks

Please read this bit – incorrect submissions will lose points, and “details” like file naming can easily be done wrong. Your submission process should be as follows:

1. Check your work against the provided rubric one last time.
2. Create a ZIP file of the folder containing your work.
3. Upload the ZIP file to Gradescope.
4. Download your submission from Gradescope.
5. Unzip it (where it is, usually in your Downloads folder).
6. Confirm that your submission still works as intended.