

OMSCS/OMSCY GEORGIA TECH

# SDN Firewall with POX

---

**CS 6250**

**Summer 2021**

Copyright 2021

Georgia Institute of Technology

All rights reserved.

This is solely to be used for current CS6250 students. Any public posting of the material contained within is strictly forbidden by the Honor code.

# SDN Firewall with POX

## Table of Contents

PROJECT GOAL .....	3
Part 0: Project References.....	3
YouTube Videos .....	4
Part 1: Files Layout .....	4
Part 2: Before You Begin .....	6
Part 3: Review of Mininet.....	6
Part 4: Wireshark.....	8
Part 4a: Wireshark Example Packet Capture.....	8
Part 4b: Packet Capture Assignment (packetcapture.pcap) .....	11
Part 5: SDN Firewall Implementation Details.....	11
Part 5b: Implementing the Firewall in Code.....	14
Part 5c: How to Test .....	16
Part 6: Configuration Rules .....	19
Part 7: Error Conditions and Helpful Tips.....	20
What to Turn In .....	21
What you can and cannot share .....	21
Rubric .....	21
Appendix A: POX API Excerpt.....	23
ofp_flow_mod - Flow table modification.....	23
Example: Installing a table entry .....	24
Match Structure .....	24
Partial Matches and Wildcards.....	25
ofp_match Methods .....	26
Defining a match from an existing packet.....	27
Example: Matching Web Traffic .....	27
OpenFlow Actions .....	27

Output..... 28

Example: Sending a FlowMod ..... 28

## PROJECT GOAL

In this project, you will use Software Defined Networking (SDN) principles to create a configurable firewall using an OpenFlow enabled Switch. The Software Defined Networking function allows you to programmatically control the flow of traffic on the network

This project will start with a review of Mininet (this was first used in the optional Simulating Networks project). This review will explain the basic concepts of Mininet and the functionality you may need to complete this project.

The next phase will involve examining network traffic using Wireshark. This will allow you to see the header contents that will be important in building the code necessary to implement the firewall as well as the necessary ruleset you will create to test the firewall.

After this, you will need to perform two tasks that need to be conducted in parallel:

1. You will create a configuration file ruleset that describes certain types of traffic that should be blocked or allowed between individual hosts and networks. You will define this “ruleset” using header packet parameters such as Source IP Address, Destination Port Number, IP Protocol, and Destination MAC Address (there are more parameters, these are given as an example). Your ruleset will contain instruction on whether certain traffic should be blocked or should be allowed. By default, all traffic will be allowed. You will need to specify “routes” that need to be blocked and any specific exceptions to the block that you want to allow.
2. You will create python code that will take the parameters of the configuration from the first task above and create a flow policy object using the POX OpenFlow SDN frameworks.

Please start early on this project, especially if you are unfamiliar working with Python APIs.

## Part 0: Project References

You will find the following resources useful in completing this project. It is recommended that you review these resources before starting the project.

- IP Header - <https://erg.abdn.ac.uk/users/gorry/course/inet-pages/ip-packet.html>
- TCP Packet Header - [https://en.wikipedia.org/wiki/Transmission\\_Control\\_Protocol](https://en.wikipedia.org/wiki/Transmission_Control_Protocol)
- UDP Packet Header - [https://en.wikipedia.org/wiki/User\\_Datagram\\_Protocol](https://en.wikipedia.org/wiki/User_Datagram_Protocol)
- The ICMP Protocol - [https://en.wikipedia.org/wiki/Internet\\_Control\\_Message\\_Protocol](https://en.wikipedia.org/wiki/Internet_Control_Message_Protocol)
- POX Reference Manual - <https://noxrepo.github.io/pox-doc/html/>
- Flow Modification- <https://noxrepo.github.io/pox-doc/html/#openflow-messages>

- Packet Matching - <https://noxrepo.github.io/pox-doc/html/#match-structure>
- Output Actions - <https://noxrepo.github.io/pox-doc/html/#openflow-actions>
- IP Protocols - [https://en.wikipedia.org/wiki/List\\_of\\_IP\\_protocol\\_numbers](https://en.wikipedia.org/wiki/List_of_IP_protocol_numbers)
- TCP and UDP Service and Port References - [https://en.wikipedia.org/wiki/List\\_of\\_TCP\\_and\\_UDP\\_port\\_numbers](https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers)
- Wireshark - [https://www.wireshark.org/docs/wsug\\_html/](https://www.wireshark.org/docs/wsug_html/)
- CIDR Calculator - <https://account.arin.net/public/cidrCalculator>
- CIDR - [https://en.wikipedia.org/wiki/Classless\\_Inter-Domain\\_Routing](https://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing)

## YouTube Videos

Several YouTube videos have been posted concerning this project. These consist of the following:

- Broad Overview and Administration - <https://youtu.be/RngSesWqLTO>
- Wireshark Walkthrough - <https://youtu.be/CR4226dHE5Y>
- Implementation Walkthrough - <https://youtu.be/XEou4cFq-7c>
- Ruleset Walkthrough - <https://youtu.be/aytQysZ6-RA>

These videos are optional but can greatly assist you in completing this project.

## Part 1: Files Layout

Unzip the `SDNFirewall.zip` file into your Virtual Machine. Do this by running the following command:

### **unzip SDNFirewall.zip**

This will extract the files for this project into a directory named `SDNFirewall` at your current path. The following files will be extracted:

- `cleanup.sh` – this file called by using following command line: `./cleanup.sh`  
This file will clean up the Mininet Environment and kill all zombie Python and POX processes.
- `configure.pol` - this file is where you will supply the configuration to the firewall that specifies the traffic that should either be blocked or allowed (override blocks). The format of this file will be specified later in this document. This file is one of the deliverables that must be included in your ZIP submission to Canvas.

- `sdn-firewall.py` – this file is the other deliverable that must be included in your ZIP submission to Canvas. This file implements the firewall using POX and OpenFlow functions. It receives a copy of the contents of the `configure.pol` file as a python list containing a dictionary for each rule and you will need to implement the code necessary to process these items into POX policies to create the firewall.
- `sdn-topology.py` – this file creates the Mininet topology used in this assignment. This is like what you created in the Simulating Networks project. When evaluating your code against the ruleset specified in this project, do not change it. However, you are encouraged to make your own topologies (and rules) to test the firewall. Look at the `start-topology.sh` file to see how to start a different topology.
- `ws-topology.py` – this file is substantially similar to `sdn-topology.py`, but it does not call the POX Controller. You will use this during the wireshark exercise.
- `setup-firewall.py` – this file sets up the frameworks used in this project. **DO NOT MODIFY THIS FILE**. This file will create the appropriate POX framework and then integrates the rules implemented in `sdn-firewall.py` into the OpenFlow engine. It will also read in the values from the `configure.pol` file and validate that the entries are valid. If you make changes to this file, the autograder will likely have issues with your final code as the autograder uses the unaltered distribution version of this file.
- `start-firewall.sh` – this is the shell script that starts the firewall. This file must be started before the topology is started. It will copy files to the appropriate directory and then start the POX OpenFlow controller. This file called by using following command line:  
**`./start-firewall.sh`**
- `start-topology.sh` – this is the shell script that starts the Mininet topology used in the assignment. All it does is call the `sdn-topology.py` file with superuser permissions. This file called by using following command line: **`./start-topology.sh`**
- `test-client.py` – this is a python test client program used to test your firewall. This file is called using the following command line:  
**`python test-client.py PROTO SERVERIP PORT SOURCEPORT`** where PROTO is either T for TCP or U for UDP, SERVERIP is the IP address of the server (destination), PORT is the destination port, and optionally SOURCEPORT allows you to configure the source port that you are using (**IMPORTANT: ONLY USE SOURCE PORTS > 1024 OR YOU WILL DEAL WITH PERMISSIONS ISSUES**)  
Example: **`python test-client.py T 10.0.1.1 80`**
- `test-server.py` – this is a python test server program used to test your firewall. This file is called using the following command line:  
**`python test-server.py PROTO SERVERIP PORT`** where PROTO is either T for TCP or U for UDP, SERVERIP is the IP address of the server (the server you are running this script on), and PORT is the service port.  
Example: **`python test-server.py T 10.0.1.1 80`**

## Part 2: Before You Begin

This project assumes basic knowledge about IP and TCP/UDP Protocols. It is highly encouraged that you review the following items before starting. This will help you in understanding the contents of IP packet headers and what you may need to match.

- What is the IP (Internet Protocol)? What are the different types of Network Layer protocols?
- Review TCP and UDP? How does TCP or UDP differ from IP?
- Examine the packet header for a generic IP protocol entry. Contrast that with the packet header for a TCP packet, and for a UDP packet. What are the differences? What does each field mean?
- What constitutes a TCP Connection? How does this contrast with a UDP connection.
- A special IP protocol is ICMP. Why is ICMP important? What behavior happens when you do an ICMP Ping? If you block an ICMP response, what would you expect to see?
- If you block a host from ICMP, will you be able to send TCP/UDP traffic to it?
- Can you explain what happens if you get a ICMP Destination Unreachable response?
- What is CIDR notation? How do you subnet a network?

## Part 3: Review of Mininet

Mininet is a network simulator that allows you to explore SDN techniques by allowing you to create a network topology including virtual switches, links, hosts/nodes, and controllers. It will also allow you to set the parameters for each of these virtual devices and will allow you to simulate real-world applications on the different hosts/nodes.

The following code sets up a basic Mininet topology similar to what is used for this project:

```
#!/usr/bin/python

from mininet.topo import Topo
from mininet.net import Mininet
from mininet.node import CPULimitedHost, RemoteController
from mininet.util import custom
from mininet.link import TCLink
from mininet.cli import CLI

class FirewallTopo(Topo):
    def __init__(self, cpu=.1, bw=10, delay=None, **params):
        super(FirewallTopo, self).__init__()

        # Host in link configuration
        hconfig = {'cpu': cpu}
        lconfig = {'bw': bw, 'delay': delay}

        # Create the firewall switch
        s1 = self.addSwitch('s1')
```

```

hq1 = self.addHost('hq1', ip='10.0.0.1', mac='00:00:00:00:00:1e', **hconfig)
self.addLink(s1, hq1)

us1 = self.addHost('us1', ip='10.0.1.1', mac='00:00:00:01:00:1e', **hconfig)
self.addLink(s1, us1)

```

This code defines the following virtual objects:

- Switch s1 – this is a single virtual switch with the label 's1'. In Mininet, you may have as many virtual ports as you need – for Mininet, “ports” are considered to be a virtual ethernet jack, not an application port that you would use in building your firewall.
- Hosts hq1 and us1 – these are individual virtual hosts that you can access via xterm and other means. You can define the IP Address, MAC/Hardware Addresses, and configuration parameters that can define cpu speed and other parameters using the hconfig dictionary.
- Links between s1 and hq1 and s1 and us1 – consider these like an ethernet cable that you would run between a computer and the switch port. You can define individual port numbers on each side (i.e., port on the host and port on the virtual switch), but it is advised to let Mininet automatically wire the network. Like hosts, you can define configuration parameters to set link speed, bandwidth, and latency. **REMINDER – PORTS ARE WIRING PORTS ON THE VIRTUAL SWITCH, NOT APPLICATION PORT NUMBERS.**

Useful Mininet Commands:

- For this project, you can start Mininet and load the firewall topology by running the **`./start-topology.sh`** from the project directory. You can quit Mininet by typing in the **`exit`** command.
- After you are done running Mininet, it is recommended that you cleanup Mininet. There are two ways of doing this. The first is to run the **`sudo mn -c`** command from the terminal and the second is to use the **`./cleanup.sh`** script provided in the project directory. Do this after every run to minimize any problems that might hang or crash Mininet.
- You can use the xterm command to start an xterm window for one of the virtual hosts. This command is run from the mininet> prompt. For example, you can type in **`us1 xterm &`** to open a xterm window for the virtual host us1. The & causes the window to open and run in the background. In this project, you will run the test-\*-client.py and test-\*-server.py in each host to test connectivity.
- The **`pingall`** command run from the mininet> prompt will cause all hosts to ping all other hosts. Note that this may take a long time. To run a ping between two hosts, you can



specify `host1 ping host2` (for example, `us1 ping hq1` which will show the result of host us1 pinging hq1).

- The `help` command will show all Mininet commands and `dump` will show information about all hosts in the topology.

## Part 4: Wireshark

Wireshark is a network packet capture program that will allow you to capture a stream of network packets and examine them. Wireshark is used extensively to troubleshoot computer networks and in the field of information security. We will be using Wireshark to examine the packet headers to learn how to use this information to match traffic that will be affected by the firewall we are constructing.

However, Wireshark is NOT currently installed on the VM, so please run the following commands on your VM:

```
sudo apt update
```

```
sudo apt install wireshark tshark -y
```

When you install, you may get a popup screen that asks whether non-superusers be able to capture packets. Select “<No>” for this.

tshark is a command line version of Wireshark that we will be using to capture the packets between mininet hosts and we will use Wireshark for the GUI to examine these packets. However, you will be allowed to use the Wireshark GUI if you would like in doing the packet capture.

### Part 4a: Wireshark Example Packet Capture

Please watch the video referenced in Part 2 if you would like to follow along in time for a live packet capture.

- Step 1: Open up a Terminal Window and change directory to the SDNFirewall directory that was extracted in Part 1.
- Step 2: The first action is to startup the Mininet topology used for the wireshark capture exercise. This topology matches the topology that you will be using when creating and testing your firewall. To start this topology, run the following command:

```
sudo python ws-topology.py
```

This will startup a Mininet session with all hosts created.

- Step 3: Start up two xterm windows for hosts us1 and us2. After you start each xterm window, it is recommended that you run the following command in each xterm window as you load them to avoid confusion about which xterm belongs to which host:

```
export PS1="hostname >"
```

replacing hostname with the actual hostname.

Type in the following commands at the Mininet prompt.

```
us1 xterm & (then run export PS1="us1 >" in the xterm window that pops up)  
us2 xterm & (likewise, run export PS1="us2 >" in the second xterm window)
```

- Step 4: In this step, we want to start capturing all the traffic that traverses through the ethernet port on host us1. We do this by running tshark (or alternatively, wireshark) as follows from the mininet prompt:

```
us1 sudo tshark -w /tmp/capture.pcap
```

This will start tshark and will output a pcap formatted file to /tmp/capture.pcap. Note that this file is created as root, so you will need to change ownership to mininet to use it in future steps – **chown mininet:mininet /tmp/capture.pcap**

If you wish to use the Wireshark GUI instead of tshark, you would call **us1 sudo wireshark &**. You may use this method, but the TA staff will not provide support for any issues.

- Step 5: Now we need to capture some traffic. Do the following tasks in the appropriate xterm window:

```
in us1 xterm: ping 10.0.1.2 (hit control C after a few ping requests)  
In us2 xterm: ping 10.0.1.1 (likewise hit control C after a few ping requests)  
In us1 xterm: python test-server.py T 10.0.1.1 80  
In us2 xterm: python test-client.py T 10.0.1.1 80  
In us1 xterm: press Control C to kill the server  
In Mininet Terminal: press Control C to stop tshark
```

- Step 6: At the mininet prompt, type in exit and press enter. Next, do the chown command as described in step 4 above to your packet capture. DO NOT TURN IN THIS

PACKET CAPTURE. You may also close the two xterm windows as they are finished.

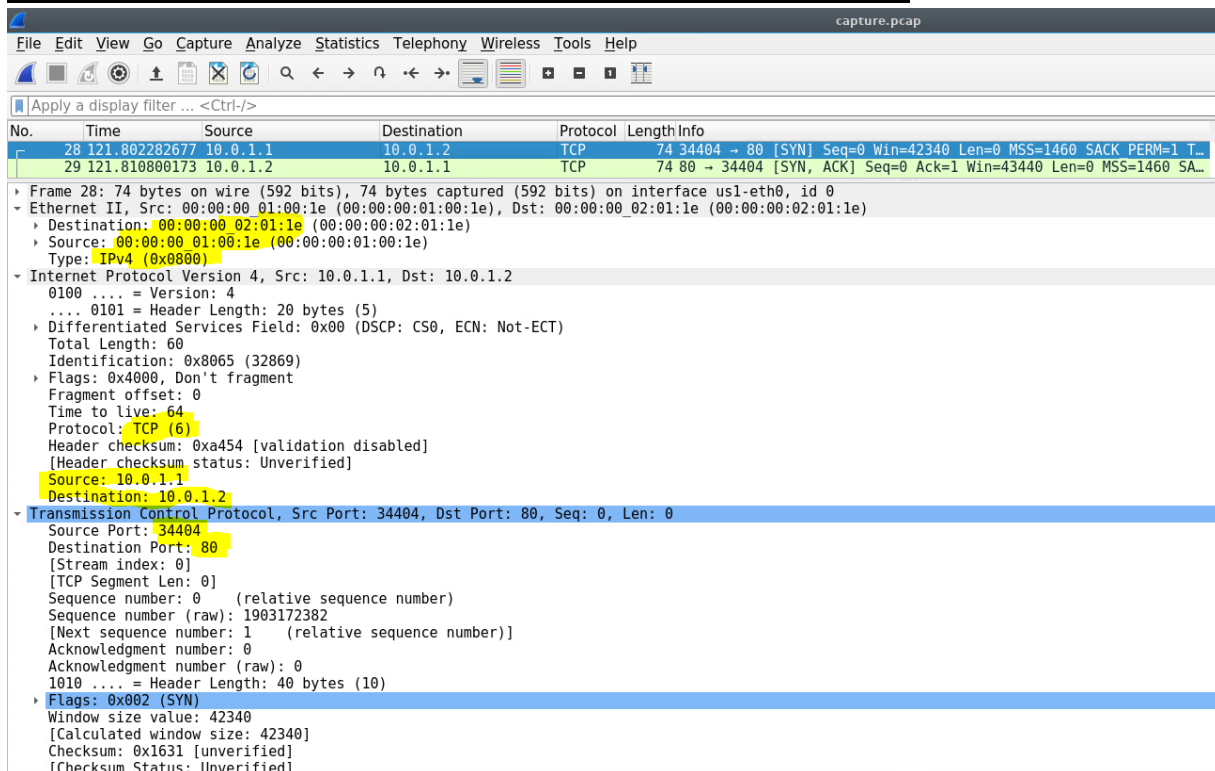
- Step 7: At the bash prompt on the main terminal, run:

**sudo wireshark**

Go to the File => Open menu item, browse to the /tmp directory and select the pcap file that you saved using tshark.

You will get a GUI that looks like the example packet capture. You will have a numbered list of all the captured packets with brief information consisting of source/destination, IP protocol, and a description of the packet. You can click on an individual packet and will get full details including the Layer 2 and Layer 3 packet headers, TCP/UDP/ICMP parameters for packets using those IP protocols, and the data contained in the packet.

### Example Packet Capture – Host us1 making web request to Host us2



Note the highlighted fields. You will be using the information from these fields to help build your firewall implementation and ruleset. Note the separate header information for TCP. This will also be the case for UDP packets.

Also, examine the three-way handshake that is used for TCP. What do you expect to find for UDP? ICMP?

### Example TCP Three-Way Handshake

28	121.802282677	10.0.1.1	10.0.1.2	TCP	74	34404 → 80	[SYN] Seq=0 Win=42340 Len=0 MSS=1460 SACK PERM=1 T...
29	121.810800173	10.0.1.2	10.0.1.1	TCP	74	80 → 34404	[SYN, ACK] Seq=0 Ack=1 Win=43440 Len=0 MSS=1460 SA...
30	121.810889156	10.0.1.1	10.0.1.2	TCP	66	34404 → 80	[ACK] Seq=1 Ack=1 Win=42496 Len=0 TSval=948059323 ...

Please examine the other packets that were captured to help you familiarize yourself with Wireshark.

### Part 4b: Packet Capture Assignment (packetcapture.pcap)

The deliverable for the Wireshark portion of the assignment is a new packet capture using different hosts and using protocols and ports similar to what you will be using to create your firewall configuration file as described in Part 6. This file will need to be titled packetcapture.pcap and submitted in the ZIP as described in the “What to Turn In” section.

Your packet capture may contain additional data in addition to the steps described below. Your submission will be evaluated to ensure that there is at least one packet captured that describes most of the events below. You will use techniques similar to what was described in Part 4a above – except using different hosts and different ports. For questions regarding the test-client.py and test-server.py file, you can either start the file without parameters to get instructions or refer to Part 5c below that describes in detail how these utilities are used.

You will need to use create xterm windows for hosts hq1, hq2, and cn1. **You will capture packets that traverse through host cn1.** You can find the IP address for these hosts by examining the ws-topology.py file.

Create a series of traffic as such:

- Have host hq1 ping host cn1.
- Have host cn1 ping host hq2.
- Make cn1 be a server for UDP Port 53 (DNS) and have host hq1 send a request (client) to cn1. Kill the server when finished capturing the connection.
- Make hq2 be a server for TCP Port 22 (SSH) and have cn1 send a request to hq2. Kill the server when finished capturing the connection.
- Make hq2 be a server for UDP Port 22 (SSH) and have cn1 send a request to hq2. Kill the server when finished capturing the connection.

### Part 5: SDN Firewall Implementation Details

Using the information that you learned above in running Wireshark, you will be creating two files – one is a firewall configuration file that will specify different header parameters to match in order to allow or block certain traffic and the second is the implementation code to create Openflow Flow Modification objects that will create the firewall using the parameters given in the firewall configuration file.

## Part 5a: Specifications of configure.pol

This file consists of a series of entries that describe each desired firewall rule. A particular final rule may contain any number of implementation rules (or lines) in this file. Each line has the following format:

**Rule Number, Action, Source MAC, Destination MAC, Source IP, Destination IP, Protocol, Source Port, Destination Port, Comment/Note**

- **Rule Number** = this is a rule number to help you track a particular rule - it is not used in the firewall implementation. It can be any value and is NOT validated in `setup-firewall.py`. **DO NOT USE this field to match traffic and DO NOT USE this field to assign priority to rules.**
- **Action = Block or Allow** **Block** rules will block traffic that matches the remaining parameters of this rule. **Allow** rules will override Block rules to allow specific traffic to pass through the firewall (see below for an example). The entry is a string in (Block,Allow).
- **Source / Destination MAC** address in form of xx:xx:xx:xx:xx:xx. You may use MAC Addresses to match an individual host. In the real world, you would use to match a particular piece of hardware. The MAC address of a particular host is defined inside the `sdn-topology.py` file.
- **Source / Destination IP** Address in form of xxx.xxx.xxx.xxx/xx in CIDR notation. You can use this to match either a single IP Address, a particular Subnet, or a particular subnetwork (based on CIDR). An entry here would look like: 10.0.0.1/32. NOTE: If you are using a CIDR mask, make sure that the IP Address shown is the Network Address. For instance, if you want to match the 192.168.1.x/24 subnet, the IP Address must be 192.168.1.0/24. This distinction becomes important if you use a sub-network CIDR mask – like a /30 network – the first address of that CIDR block is the network address you need to use. You can refer to <https://github.com/att/pox/blob/master/pox/lib/addresses.py> for more information on the format of CIDR addresses. The address must be in CIDR notation. The IP address of a particular host is defined inside the `sdn-topology.py` file.
- **Protocol** = integer IP protocol number per IANA (0-254). An example is ICMP is IP Protocol 1, TCP is IP Protocol 6, etc. This must be an integer.

- **Source / Destination Port** = if Protocol is TCP or UDP, this is the Application Port Number per IANA. For example, web traffic is generally TCP Port 80. **FOR THIS PROJECT, the following link is what will be used for rules checking (Yes and Unofficial only, not Assigned):**  
[https://en.wikipedia.org/wiki/List\\_of\\_TCP\\_and\\_UDP\\_port\\_numbers](https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers)
- **Comment/Note** = this is for your use in tracking rules.

### Special Notes:

- Any field not being used for a match should have a '-' character as its entry. A '-' means that the item is not being used for matching traffic. It is valid for any rule element except for **Action** to have a '-'. (i.e., a rule like: **1,Block,-,-,-,-,-,Block the world** is valid)
- When a rule states to block the world from accessing a particular host, this means that you are matching against all possible hosts which may include hosts that are not in your topology. An example host world has been included to help you test this condition.
- Note that a rule does not necessarily need a MAC or IP Address. Also, it is possible to have a rule that only has network addresses and no ports/protocols. What won't ever be tested is using a src/dst port WITHOUT an IP Protocol. Read Appendix A for the reasons for this.
- What is the difference between source and destination? Source makes a request of the destination. For ports, you will most often use destination ports, but make sure that your firewall implements both source and destination ports. For IP and MAC addresses, you will use both.
- When should I use MAC vs IP Addresses? You will want to interchange them in this file to test the robustness of your implementation. It is valid to specify a Source MAC address and a Destination IP Address. **NOTE THAT THE AUTOGRADER WILL TEST SCENARIOS THAT INCLUDE COMBINATIONS OF MAC AND IP ADDRESSES.**

### Example Rules:

```
1,Block,-,-,10.0.0.1/32,10.0.1.0/24,6,-,80,Block 10.0.0.1 from accessing a web server on the 10.0.1.0/32 network
```

```
2,Allow,-,-,10.0.0.1/32,10.0.1.125/32,6,-,80,Allow 10.0.0.1 to access a web server on 10.0.1.125 overriding previous rule
```

What do these rules do?

The first rule basically blocks host **hq1** (IP Address 10.0.0.1) from accessing a web server on any host on the **us** network (the subnet 10.0.1.0/24 network). The web server is running on the TCP IP Protocol (6) and uses TCP Port 80.

The second rule overrides the initial rule to allow **hq1** (IP Address 10.0.0.1) to access a web server running on **us5** (IP Address 10.0.1.125)

By definition – from the sdn-topology.py file:

```
This class defines the Mininet Topology for the network used in this project.  
This network consists of the following hosts/networks:
```

```
    Headquarters Network (hq1-hq5). Subnet 10.0.0.1/24
```

```
    US Network (us1-us5). Subnet 10.0.1.0/24
```

```
    India Network (in1-in5). Subnet 10.0.2.0/24
```

```
    China Network (cn1-cn5). Subnet 10.0.3.0/24
```

```
    UK Network (uk1-uk5). Subnet 10.0.4.0/24
```

There is also a single host named **wo1** that is at 10.0.200.1/32. This is to be used for testing a “world” scenario.

In Part 6, you will be given a set of firewall conditions that you will need to create the configure.pol needed for your submission.

You may create temporary rulesets in order to help you complete Part 5b below.

### Part 5b: Implementing the Firewall in Code

For this step, you will be implementing the coding that will convert the entries from the configure.pol file into a rule that POX will understand. Other than basic python syntax, all you need to know to implement the firewall can be found in these three sections of the POX manual (and detailed in Appendix A attached to this documentation):

- Flow Modification- <https://noxrepo.github.io/pox-doc/html/#openflow-messages>
- Packet Matching - <https://noxrepo.github.io/pox-doc/html/#match-structure>
- Output Actions - <https://noxrepo.github.io/pox-doc/html/#openflow-actions>

Your code will need to do the following:

- Create a OpenFlow Flow Modification object
- Create a POX Packet Matching object that will integrate the elements from the firewall configuration policy that are passed to it using the policy dictionary.

- Create a POX Output Action, if needed, to specify what to do with the traffic.

Your code will go into a section that will repeat itself for every line in the firewall configuration file that is passed to it. The “rule” item that is added to the “rules” list is an OpenFlow Modification object. The process of injecting this rule into the POX controller is handled automatically for you in the start-firewall.py file.

### Key Information:

- policies is a python list that contains one entry for each rule line contained in your configure.pol file. policy is a dictionary that consists of one line from the policies list. This dictionary has the following keys:
  - policy['mac-src'] = Source MAC Address
  - policy['mac-dst'] = Destination MAC Address
  - policy['ip-src'] = Source IP Address (10.0.10.1/32)
  - policy['ip-src-address'] = Source IP Address part (10.0.10.1)
  - policy['ip-src-subnet'] = Source IP Address subnet (32)
  - policy['ip-dst'] = Destination IP Address (10.0.10.1/32)
  - policy['ip-dst-address'] = Destination IP Address part (10.0.10.1)
  - policy['ip-dst-subnet'] = Destination IP Address subnet (32)
  - policy['ipprotocol'] = IP Protocol
  - policy['port-src'] = Source Port for TCP/UDP
  - policy['port-dst'] = Destination Port for TCP/UDP
  - policy['rulenum'] = Rule Number
  - policy['comment'] = Comment
  - policy['action'] = Allow or Block

Use these to help match traffic. Please note that all fields are strings and may contain a '-' character.

- You will need to assume that all traffic is IPV4.
- **Do not hard code your solution. By hardcoding, it is meant that you should not be using actual IP/MAC Addresses and Ports in your coding. Use the policy[] dictionary items above to make the implementation generic. Failure to heed this instruction may result in the loss of 1/3 of your points when the project is graded with an alternate topology and/or ruleset.**
- Hint 1: Priority is your friend. Do NOT use the rulenum as your priority. See Appendix A for a discussion on priority level. This is how you will make an Allow rule override a Block rule.



- Hint 2: Blocking a packet is a null action.
- Outputting extra print debug lines will not adversely impact the autograder.
- You will need to rewrite the rule = None to reference your Flow Modification object.

## Part 5c: How to Test

This is the process to manually test your code while implementing your code and firewall configuration file. An unofficial testing script that will find most errors will be posted to EdStem that can be used after implementation of the code and the configure.pol file is complete. However, in implementation, the manual process is the way to test. To start out, consider testing the first example rule, which is:

**1,Block,-,-,10.0.0.1/32,10.0.1.0/24,6,-,80,Block 10.0.0.1 from accessing a web server on the 10.0.1.0/32 network**

Procedure:

- Step 1: Open two terminal windows or tabs on the VM and change to the SDNFirewall directory.
- Step 2: In the first terminal window, type in: **./start-firewall.sh configure.pol**  
This should start up POX, read in your rules, and start up an OpenFlow Controller. You will see something like this in your terminal window:

```
mininet@mininet:~/SDNFirewall$ ./start-firewall.sh configure.pol
~/pox ~/SDNFirewall
POX 0.7.0 (gar) / Copyright 2011-2020 James McCauley, et al.
Starting POX Instance
Starting date and time : 2021-02-08 01:09:59

WARNING:version:Support for Python 3 is experimental.
INFO:core:POX 0.7.0 (gar) is up.
INFO:openflow.of_01:[00-00-00-00-00-01 1] connected
List of Policy Objects imported from configure.pol:
-----
[{'rulenum': '1', 'action': 'Block', 'mac-src': '-', 'mac-dst': '-', 'ip-src': '10.0.0.1/32', 'ip-dst': '10.0.1.0/24', 'ipprotocol': '6', 'port-src': '-', 'port-dst': '80', 'comment': 'Block 10.0.0.1 from accessing a web server on the 10.0.1.0/32 network'}, {'rulenum': '2', 'action': 'Allow', 'mac-src': '-', 'mac-dst': '-', 'ip-src': '10.0.0.1/32', 'ip-dst': '10.0.1.125/32', 'ipprotocol': '6', 'port-src': '-', 'port-dst': '80', 'comment': 'Allow 10.0.0.1 to access a web server on 10.0.1.125 overriding previous rule'}]
Added Rule 1 : Block 10.0.0.1 from accessing a web server on the 10.0.1.0/32 network
Added Rule 2 : Allow 10.0.0.1 to access a web server on 10.0.1.125 overriding previous rule
```

- Step 3: In the second terminal window, type in: **./start-topology.sh**

This should start up mininet and load the topology. You should see the following:

```
mininet@mininet:~/SDNFirewall$ ./start-topology.sh
Starting Mininet Topology...
If you see a Unable to Contact Remote Controller, you have an error in your code...
Remember that you always use the Server IP Address when calling test scripts...
mininet> █
```

This is where most of the work will occur. At the mininet prompt, you can use all of the standard mininet commands as described in Part 3.

- Step 4: The rule we are testing involves the **hq1** host attempting to connect to the web server port (TCP Port 80) on host **us1**. At the mininet prompt, type in the following two commands on two different lines:

```
hq1 xterm &
us1 xterm &
```

Two windows should have popped up. You can always identify which xterm is which by running the command: **ip address** from the bash shell. This will give you the IP address for the xterm, which will then let you discover which xterm belongs to which host.

- Step 5: In the xterm window for **us1** (which is the destination host of the rule – remember that the destination is always the server), type in the following command:

```
python test-server.py T 10.0.1.1 80
```

This sets up the test server for us1 that will be listening on TCP port 80. The IP Address specified is always the IP address of the machine you are running it on. If you attempt to start the test-server on a machine that does not have the IP address that is specified in the command, you will get the following error: `OSError: [Errno 99] Cannot assign requested address`.

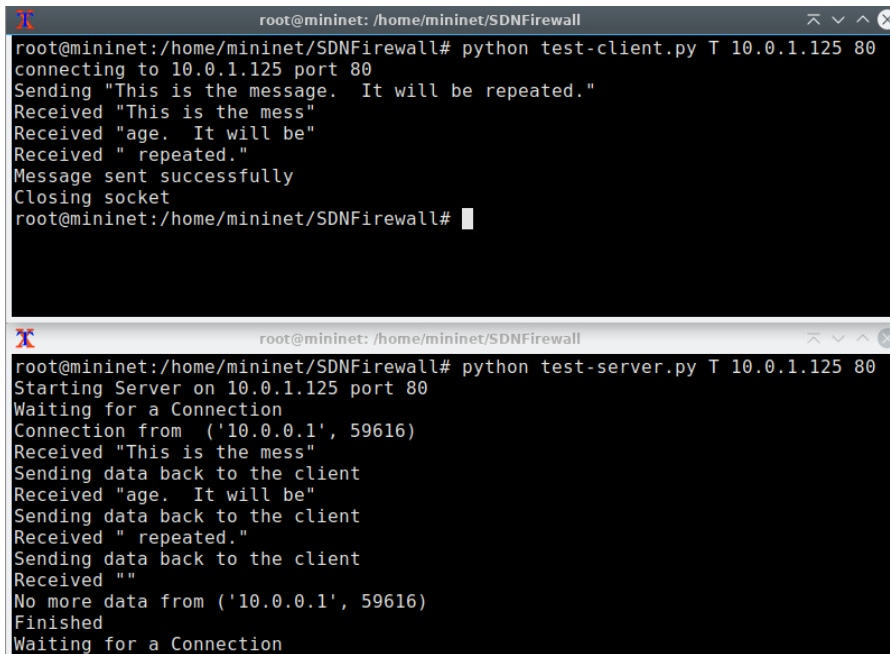
- Step 6: In the xterm window for **hq1** (which is the destination host of the rule – remember that the destination is always the server), type in the following command:

```
python test-client.py T 10.0.1.1 80
```

This will start up a client that will connect to the TCP Port 80 on the server 10.0.1.1 (destination IP address) and will send a message string to the server. However, if the firewall is set to block this connection, you will never see the message pass on either of

the client or the server.

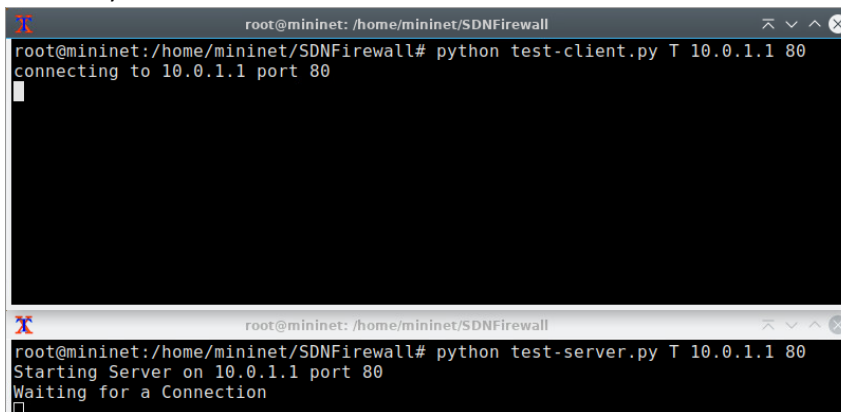
If traffic should have passed between the two systems, you would see the following (the top xterm is hq1 and in this case, the bottom xterm is us5, which shows the implementation of example rule 2):



```
root@mininet: /home/mininet/SDNFirewall
root@mininet:/home/mininet/SDNFirewall# python test-client.py T 10.0.1.125 80
connecting to 10.0.1.125 port 80
Sending "This is the message. It will be repeated."
Received "This is the mess"
Received "age. It will be"
Received " repeated."
Message sent successfully
Closing socket
root@mininet:/home/mininet/SDNFirewall#

root@mininet: /home/mininet/SDNFirewall
root@mininet:/home/mininet/SDNFirewall# python test-server.py T 10.0.1.125 80
Starting Server on 10.0.1.125 port 80
Waiting for a Connection
Connection from ('10.0.0.1', 59616)
Received "This is the mess"
Sending data back to the client
Received "age. It will be"
Sending data back to the client
Received " repeated."
Sending data back to the client
Received ""
No more data from ('10.0.0.1', 59616)
Finished
Waiting for a Connection
```

A blocked connection will look like this (note that the client may take a while to timeout):



```
root@mininet: /home/mininet/SDNFirewall
root@mininet:/home/mininet/SDNFirewall# python test-client.py T 10.0.1.1 80
connecting to 10.0.1.1 port 80
█

root@mininet: /home/mininet/SDNFirewall
root@mininet:/home/mininet/SDNFirewall# python test-server.py T 10.0.1.1 80
Starting Server on 10.0.1.1 port 80
Waiting for a Connection
█
```

You may hit Control C to kill both the server and the client.

A timeout looks like:

```

root@mininet:/home/mininet/SDNFirewall# python test-client.py T 10.0.1.1 80
connecting to 10.0.1.1 port 80
Traceback (most recent call last):
  File "test-client.py", line 29, in <module>
    sock.connect(server_address)
TimeoutError: [Errno 110] Connection timed out
root@mininet:/home/mininet/SDNFirewall# █

```

- Repeat this process to test other combinations. Make sure to test for conditions that should and should not be blocked.

When you are done testing, go to the second terminal and type in **quit** at the mininet window and then press Control-C on the first terminal window that was used to start the firewall.

## Part 6: Configuration Rules

You will need to submit a configure.pol file to create policies that implement the following scenarios. You may implement your rules in any manner that you want, but it is recommended using this step as an opportunity to check your firewall code implementation.

You work for GT-SDN Corporation that has offices in the US, China, India, and the UK, with a US headquarters that acts as the datacenter for the company. Your task is to implement a firewall that accomplishes the following goals:

- On the headquarters network, you have two active DNS servers (using both the standard UDP service as well as the newer DNS-over-TLS standard). **hq1** provides DNS service to the public and **hq2** provides a private DNS service accessible only to the 5 corporate networks (i.e., the US, China, India, UK, and Headquarters network). Write a series of firewall rules that blocks the world (but allows the corporate networks) from accessing the DNS services on **hq2** and ensures that the DNS services on **hq1** are available to all.
- On the headquarters network, the host **hq3** acts as a VPN server that connects to each of the other sites (hosts **us3**, **uk3**, **in3**, and **cn3**) using the OpenVPN server (standard ports – both TCP and UDP). Create a set of firewall rules that will only allow the 4 offsite hosts (**us3**, **uk3**, **in3**, and **cn3**) access to the **hq3** OpenVPN server. No other hosts in the world should be able to access the OpenVPN server on **hq3**.
- Only the hosts on the Headquarters network should be pingable by the world. However, each host on a particular subnet (subnet list = the 5 different networks defined in the topology file) should be able to ping each other (i.e., **us1** should be able

to ping **us5**, but not **in1**). (Note that there are many different methods to reach this objective)

- One of the main routes for ransomware to enter a corporate network is through a Remote Desktop connection with either an insecure version of the server protocol or via leaked or weak credentials (using either the Microsoft Remote Desktop protocol or the Virtual Networking Computing (VNC) protocols). Write a set of firewall rules that will block any host in the world from connecting to a Remote Desktop connection on any of the 5 corporate networks (i.e., the US, China, India, UK, and Headquarters network) using the standard ports for the protocols.
- The servers located on hosts **us3** and **us4** run a micro webservice on TCP and UDP Port 8500 that processes financial information. Access to this service should be blocked from hosts **uk2**, **uk3**, **uk4**, **uk5**, **in4**, **in5**, **us5**, and **hq5**. (Hint: Note the IP Addresses for these hosts and you may use the smallest subnet mask that handles the listed hosts using CIDR notation).
- GT-SDN Corporation has banned the use of NetBIOS over TCPIP for the use of file and printer sharing. Write a series of rules that blocks NetBIOS over TCPIP to and from any host in the world (i.e., you can use a block for the world for this). Use the standard ports and protocols for NetBIOS over TCPIP.
- Block the ingress of the GRE IP Protocol into any of the 5 corporate networks (i.e., the US, China, India, UK, and Headquarters network). GRE is an encapsulation protocol primarily used in PPTP-based VPN connections. You only need to block the IP protocol and not any ports associated to PPTP-based VPN connections.

## Part 7: Error Conditions and Helpful Tips

- On the topology terminal window, if you get an error message that states “Unable to contact Remote Controller”, it means that the POX controller had crashed and normally shows that there is a bug in your implementation code.
- Pay attention to the Fields ignored due to unspecified prerequisites warning. If you see this rule, you have issues with missing prerequisites in your implementation code and that the specified field will be ignored. **See Appendix A Match Structure for a discussion of the prerequisites**
- Watch for type mismatches.

- If you get the following error message, please run the cleanup.sh utility: "Exception: Error creating interface pair (s1-eth0,hq1-eth0): RTNETLINK answers: File exists"
- If you get a struct.pack or struct.unpack error message, take a look at [https://github.com/att/pox/blob/7f76c9e3c9bc999fcc97961d408ab0b71cbc186d/pox/openflow/libopenflow\\_01.py](https://github.com/att/pox/blob/7f76c9e3c9bc999fcc97961d408ab0b71cbc186d/pox/openflow/libopenflow_01.py) for more information. Also, the struct.pack error might reference how to fix (i.e., not an integer, EthAddr(), etc).
- A struct.unpack error may be ignored if it occurs on your GRE rule.
- If you use Visual Studio code, add the following to your workspace settings:  
"python.autoComplete.extraPaths": [ "/home/mininet/pox/" ],

## What to Turn In

You need to submit your copy of `packetcapture.pcap`, `sdn-firewall.py` and `configure.pol` from your project directory using the instructions from the Piazza Post “How to Submit / Zip Our Projects” (#33 for Spring 2021). To recap, zip up the two files using the following command, replacing `gtlogin` with your GT Login that you use to log into Canvas:

```
zip gtlogin_sdn.zip packetcapture.pcap configure.pol sdn-firewall.py
```

You may also include an additional text file if you have comments, criticisms, or suggestions for improvement for this project. If you wish to provide this information, add it to your ZIP file with the name `comments.txt`. This is completely optional.

**Please check your submission after uploading.** As usual, we do not accept resubmissions past the stated deadlines.

## What you can and cannot share

Do not share the content of your `sdn-firewall.py`, `configure.pol`, or `packetcapture.pcap` with your fellow students, on Ed Discussions, or elsewhere publicly. You may share any new topologies, testing rulesets, or testing frameworks, as well as packet captures that do not address the requirements of Part 4b.

Do not post your code in public repositories on Github.

## Rubric

For the Summer 2021 Semester, this project is worth a total of 150 points which is distributed in the following fashion:

- 10 points for submitting a version of `sdn-firewall.py` that indicates effort was done.
- 10 points for submitting a version of `configure.pol` that indicates effort was done.
- 30 points for submitting a version of `packetcapture.pcap` that indicates effort was done.
- 35 points - Your `configure.pol` file will be tested by a known good firewall implementation and by your `sdn-firewall.py` file with a series of unit tests to make sure that the rules were implemented. The higher of the two grades will be used (thus, you will not be penalized if your `sdn-firewall.py` file is not complete or has issues).
- 15 points – This is just a test of your provided `sdn-firewall.py` and `configure.pol` to ensure that your code is working (i.e., it tests your implementation, and not your ruleset)
- 50 points – The final portion of the grade consists of testing your `sdn-firewall.py` file with a different grading ruleset and/or topology to make sure that your code is robust enough to handle any valid configuration file.

If your code crashes from a simple clerical error, a deduction of 20 points for each edit (that resolves the crash) will be done vs losing most of the 15, 35, or 50 points for your code.

## Appendix A: POX API Excerpt

This section contains an excerpt from the POX Manual. You should not need to use any other POX objects for this project. TA Comments are highlighted. Everything on these pages is important to complete the project.

Excerpted from: <https://noxrepo.github.io/pox-doc/html/>

### ofp\_flow\_mod - Flow table modification

```
class ofp_flow_mod (ofp_header):
    def __init__(self, **kw):
        ofp_header.__init__(self)
        self.header_type = OFPT_FLOW_MOD
        if 'match' in kw:
            self.match = None
        else:
            self.match = ofp_match()
        self.cookie = 0
        self.command = OFPFC_ADD
        self.idle_timeout = OFP_FLOW_PERMANENT
        self.hard_timeout = OFP_FLOW_PERMANENT
        self.priority = OFP_DEFAULT_PRIORITY
        self.buffer_id = None
        self.out_port = OFPP_NONE
        self.flags = 0
        self.actions = []
```

- cookie (int) - identifier for this flow rule. (optional)
- command (int) - One of the following values:
- OFPFC\_ADD - add a rule to the datapath (default)
- OFPFC\_MODIFY - modify any matching rules
- OFPFC\_MODIFY\_STRICT - modify rules which strictly match wildcard values.
- OFPFC\_DELETE - delete any matching rules
- OFPFC\_DELETE\_STRICT - delete rules which strictly match wildcard values.
- idle\_timeout (int) - rule will expire if it is not matched in 'idle\_timeout' seconds. A value of OFP\_FLOW\_PERMANENT means there is no idle\_timeout (the default).
- hard\_timeout (int) - rule will expire after 'hard\_timeout' seconds. A value of OFP\_FLOW\_PERMANENT means it will never expire (the default)
- priority (int) - the priority at which a rule will match, higher numbers higher priority. Note: Exact matches will have highest priority. In general, you will need to adjust this to make Allow work. Maximum is 32767. Do NOT use the Rulenum as the priority.
- buffer\_id (int) - A buffer on the datapath that the new flow will be applied to. Use None for none. Not meaningful for flow deletion.
- out\_port (int) - This field is used to match for DELETE commands.OFPP\_NONE may be used to indicate that there is no restriction. DO NOT USE THIS.



- flags (int) - Integer bitfield in which the following flag bits may be set:
- OFPFF\_SEND\_FLOW\_REM - Send flow removed message to the controller when rule expires
- OFPFF\_CHECK\_OVERLAP - Check for overlapping entries when installing. If one exists, then an error is send to controller
- OFPFF\_EMERG - Consider this flow as an emergency flow and only use it when the switch controller connection is down.
- actions (list) - actions are defined below, each desired action object is then appended to this list and they are executed in order.
- match (ofp\_match) - the match structure for the rule to match on (see below).

See section of 5.3.3 of OpenFlow 1.0 spec. This class is defined in `pox/openflow/libopenflow_01.py`.

**TA Note:** In the example below, the `self.connection.send(msg)` is already done for you in the project. For your implementation, assume `rule = msg`.

### Example: Installing a table entry

```
# Traffic to 192.168.101.101:80 should be sent out switch port 4

# One thing at a time...
msg = of.ofp_flow_mod()
msg.priority = 42
msg.match.dl_type = 0x800
msg.match.nw_dst = IPAddr("192.168.101.101")
msg.match.tp_dst = 80
msg.actions.append(of.ofp_action_output(port = 4))
self.connection.send(msg)

# Same exact thing, but in a single line...
self.connection.send( of.ofp_flow_mod( action=of.ofp_action_output( port=4 ),
                                     priority=42,
                                     match=of.ofp_match( dl_type=0x800,
                                                         nw_dst="192.168.101.101",
                                                         tp_dst=80 )))
```

### Match Structure

OpenFlow defines a match structure – `ofp_match` – which enables you to define a set of headers for packets to match against. You can either build a match from scratch, or use a factory method to create one based on an existing packet.

The match structure is defined in `pox/openflow/libopenflow_01.py` in class `ofp_match`. Its attributes are derived from the members listed in the OpenFlow specification, so refer to that for more information, though they are summarized in the table below.

`ofp_match` attributes:

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

Attributes may be specified either on a match object or during its initialization. That is, the following are equivalent:

```
my_match = of.ofp_match(in_port = 5, dl_dst = EthAddr("01:02:03:04:05:06"))
#.. or ..
my_match = of.ofp_match()
my_match.in_port = 5
my_match.dl_dst = EthAddr("01:02:03:04:05:06")
```

## Partial Matches and Wildcards

Unspecified fields are *wildcarded* and will match any packet. You can explicitly set a field to be wildcarded by setting it to `None`.

Note

*Info:* While the OpenFlow `ofp_match` structure is defined as having a `wildcards` attribute, *you will probably never need to explicitly set it when using POX* – simply don't assign values to fields you want wildcarded (or set them to `None`).

IP address fields are a bit trickier, as they can be wildcarded completely like the other fields, but can also be *partially* wildcarded. This allows you to match entire subnets.

There are a number of ways to do this. Here are some equivalent ones:

```
my_match.nw_src = "192.168.42.0/24"
my_match.nw_src = (IPAddr("192.168.42.0"), 24)
my_match.nw_src = "192.168.42.0/255.255.255.0"
my_match.set_nw_src(IPAddr("192.168.42.0"), 24)
```

In particular, note that the `nw_src` and `nw_dst` attributes can be ambiguous when working with partial matches – especially when reading a match structure (e.g., as returned in a `flow_removed` message or `flow_stats` reply). To account for this, you may use the unambiguous `.get_nw_src()`, `.set_nw_src()`, and the destination equivalents. These return a tuple such as `(IPAddr("192.168.42.0"), 24)` which includes the number of matched bits – the number that would follow the slash in CIDR-style representation (`192.168.42.0/24`).

TA Note: What isn't very clear by this documentation is that `nw_*` is expecting a network address. If you are calling out an IP Address like `10.0.1.1/32`, it is an acceptable response to `nw_*`. However, if you are calling out a subnet like `10.0.1.0/24`, the IP address portion of the response MUST BE the Network Address.

From Wikipedia: IP addresses are described as consisting of two groups of [bits](#) in the address: the [most significant bits](#) are the [network prefix](#), which identifies a whole network or [subnet](#), and the [least significant](#) set forms the *host identifier*, which specifies a particular interface of a host on that network. This division is used as the basis of traffic routing between IP networks and for address allocation policies. ([https://en.wikipedia.org/wiki/Classless\\_Inter-Domain\\_Routing](https://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing))

Thus for a `/24` network, the first 24 bits of the address comprises the network address. Thus, it would be `10.0.1.0`. For a `/25` network, there would be two networks in the `10.0.1.x` space – a `10.0.1.0/25` and a `10.0.1.128/25`.

Your implementation code does NOT need to convert the given IP Address into a network – you can assume that any given address in a possible configuration file must be valid. However, your `configure.pol` file MUST be using the proper form if you are using a CIDR notation other than `/32`. Why would you do this? To reduce the number of rules needed. You may use this for the 5<sup>th</sup> rule from Part 6.

Note that some fields have *prerequisites*. Basically this means that you can't specify higher-layer fields without specifying the corresponding lower-layer fields also. For example, you can not create a match on a TCP port without also specifying that you wish to match TCP traffic. And in order to match TCP traffic, you must specify that you wish to match IP traffic. Thus, a match with only `tp_dst=80`, for example, is invalid. You must also specify `nw_proto=6` (TCP), and `dl_type=0x800` (IPv4). If you violate this, you should get the warning message 'Fields ignored due to unspecified prerequisites'. For more information on this subject, see the FAQ entry "I tried to install a table entry but got a different one. Why?".

## ofp\_match Methods

Method	Description
<code>from_packet(packet, in_port=None, spec_frags=False)</code>	Class factory. See "Defining a match from an existing packet" below.
<code>Clone()</code>	Returns a copy of this <code>ofp_match</code> .

Method	Description
Flip()	Returns a copy with its source and destinations reversed.
Show()	Returns a large string representation.
Get_nw_src()	Returns the IP source address and the number of matched bits as a tuple. For example: (IPAddr("192.168.42.0", 24). Note that the first element of the tuple will be None when the second is 0.
Set_nw_src(IP and bits)	Sets the IP source address and the number of bits to match. The arguments can either be two arguments (one for IP and one for bit count), or a tuple in the format used by get_nw_src().
Get_nw_dst()	Same as get_nw_src() but for destination address.
Set_nw_dst(IP and bits)	Same as set_nw_src() but for destination address.

## Defining a match from an existing packet

There is a simple way to create an exact match based on an existing packet object (that is, an ethernet object from `pox.lib.packet`) or from an existing `ofp_packet_in`. This is done using the factory method `ofp_match.from_packet()`.

```
my_match = ofp_match.from_packet(packet, in_port)
```

The `packet` parameter is a parsed packet or `ofp_packet_in` from which to create the match. As the input port is not actually in a packet header, the resulting match will have the input port wildcarded by default when this method is called with a packet. You can, of course, set the `in_port` field later yourself, but as a shortcut, you can simply pass it in to `from_packet()`. When using `from_packet()` with an `ofp_packet_in`, the `in_port` is taken from there by default.

Note that you can set fields of the resultant match object to `None` (wildcarding them) if you want a less-than-exact match.

## Example: Matching Web Traffic

As an example, the following code will create a match for traffic to web servers:

```
import pox.openflow.libopenflow_01 as of # POX convention
import pox.lib.packet as pkt # POX convention
my_match = of.ofp_match(dl_type = pkt.ethernet.IP_TYPE, nw_proto = pkt.ipv4.TCP_PROTOCOL, tp_dst = 80)
```

## OpenFlow Actions

OpenFlow actions are applied to packets that match a rule installed at the datapath. The code snippets found here can be found in `libopenflow_01.py` in `pox/openflow`.

## Output

Forward packets out of a physical or virtual port. Physical ports are referenced to by their integral value, while virtual ports have symbolic names. Physical ports should have port numbers less than 0xFF00.

Structure definition:

```
class ofp_action_output(object):
    def __init__(self, **kw):
        self.port = None # Purposely bad -- require specification
```

- port (int) the output port for this packet. Value could be an actual port number or one of the following virtual ports: (Actual port number refers to the physical port number on the switch and not the application port)
- OFPP\_IN\_PORT - Send back out the port the packet was received on. Except possibly OFPP\_NORMAL, this is the only way to send a packet back out its incoming port. This refers to the physical port, not the application port.
- OFPP\_TABLE - Perform actions specified in flowtable. Note: Only applies to ofp\_packet\_out messages.
- OFPP\_NORMAL - Process via normal L2/L3 legacy switch configuration (if available – switch dependent)
- OFPP\_FLOOD - output all openflow ports except the input port and those with flooding disabled via the OFPPC\_NO\_FLOOD port config bit (generally, this is done for STP)
- OFPP\_ALL - output all openflow ports except the in port.
- OFPP\_CONTROLLER - Send to the controller.
- OFPP\_LOCAL - Output to local openflow port.
- OFPP\_NONE - Output to no where.

Think carefully about the definitions given above for output actions. Remember that if you match a packet, no action will be done unless you set an output action.

## Example: Sending a FlowMod

To send a flow mod you must define a match structure (discussed above) and set some flow mod specific parameters as shown here:

```
msg = ofp_flow_mod()
msg.match = match
msg.idle_timeout = idle_timeout
msg.hard_timeout = hard_timeout
msg.actions.append(of.ofp_action_output(port = port))
msg.buffer_id = <some buffer id, if any>
connection.send(msg)
```

Using the connection variable obtained when the datapath joined, we can send the flowmod to the switch.