SDN Hands-On
Corso di Tecnologie di Infrastrutture di Reti

Martin Klapez

Department of Engineering *Enzo Ferrari*
University of Modena and Reggio Emilia

Overview

Slides at: netlab.unimore.it/wp-content/uploads/2024/05/sdn.pdf

- ⋆ Software-Defined Networking (SDN) in a nutshell
- ⋆ Hub vs L2 Learning Switch
- ⋆ Assignment: a custom SDN controller
- ⋆ Assignment: NFV

Become familiar with basic OpenFlow concepts

SDN in a nutshell:

- Traditionally , network hardware did both the decision-making and the actual work.
- It was not programmable or, if it was, you had to tinker with vendor-specific APIs, directly on the hardware itself.
- So, for instance, a switch had to calculate where to forward incoming packets, and then had to actually send them. Direction & Execution.

Become familiar with basic OpenFlow concepts

SDN in a nutshell:

- Someone got tired by this lack of flexibility and saw the potential of applying software abstractions to networking, starting the trends of SDN and Network Function Virtualization (NFV).
- SDN: "Now" (not really, the majority of the deployed network hardware still operates in the old way), *decision-making* is performed by software, wrote with modern programming languages, that run on general-purpose hardware (*Control Plane*), and *execution* is carried on by 'dumb' but fast specialized hardware that just follow the rules sent to them (*Data Plane*).
- NFV: functions that were performed by specialized hardware, e.g., a firewall, can be now performed through software, and they can be chained through APIs.

Become familiar with basic OpenFlow concepts

L1 Hub.

A L1 Hub has the job of forwarding a packet to the destination.

It works this way:

- ⋆ It receives a packet from a port x.
- ⋆ It floods the packet on all the *other* ports.

The intended destination will eventually receive the packet. Inefficient but simple.

Become familiar with basic OpenFlow concepts

L2 Learning Switch.

Same job, but it works this way:

- ⋆ It has a table to keep track of [*mac&* : *port*] associations.
- ⋆ It receives a packet from a port x.
- ⋆ It puts in the table [*mac&* : *port*].
- ⋆ It looks in the headers of the packet for the destination *mac&*.
- ⋆ If it finds the destination *mac&* in the table, it forwards the packet to the corresponding port.
- ⋆ Otherwise, it floods the packet as a hub, but as soon as it receives a packet from the destination *mac&* (a TCP ACK for instance), it tracks its port in the table.

Become familiar with basic OpenFlow concepts

Remove the POX folder if it exists already and clone the POX repo

```
> sudo rm -r pox
> git clone https://github.com/noxrepo/pox.git
> cd pox
> git checkout 7030909
```

(if you get a permission issue) > export GIT_SSL_NO_VERIFY=1

To start the POX Hub module:

```
> ./pox.py forwarding.hub &
```

To start the default POX L2 Learning Switch module:

```
> ./pox.py forwarding.l2_learning &
```

(you can add "--unthreaded-sh" after ./pox.py if it doesn't work)

Become familiar with basic OpenFlow concepts

To start Mininet and attach to the POX controller:

```
(2 hosts)     > sudo mn --controller remote
(25 hosts)    > sudo mn --controller remote --topo tree,depth=2,fanout=5
```

To check the attainable data rates:
start an iPerf server on h2, in background

```
mininet > h2 iperf -s &
```

and start an iPerf client on h1

```
mininet > h1 iperf -c h2
```

Now, let's check the performance.

Become familiar with basic OpenFlow concepts

By default, nodes and links in Mininet are supposed **not** to be constrained, so, “infinite” bandwidth and “infinitesimal” latency.

This and the commands above mean that the data rate you see:

- ⋆ Is bounded by how fast your system can do things, let’s simplify by saying CPU-bounded. A faster CPU will give you faster transfers.
- ⋆ Is measured on the client. Don’t ever do this. Measure it on the server to get the actual goodput, especially with iPerf, or you’ll also include retransmissions in your figure. However, it’s ok for our purpose here.

Become familiar with basic OpenFlow concepts

behind the scenes

When a switch receives a packet that doesn't satisfy any of the rules in its forwarding table, it sends that packet to the controller **with a specific OpenFlow command** $\rightarrow$ *"S: What should I do with this?"*

Then, the controller decides what the switch should do with that packet and instructs the switch to do so. This usually happens through **an OpenFlow command that instructs the switch to remember the rule** $\rightarrow$ *"C: From now on, do x with that and similar packets"*

Become familiar with basic OpenFlow concepts

This is the default behavior, and it is so for performance reasons.

If a switch already has a rule specifying what to do with an inbound packet, it simply forwards it.

When compared with the direct forwarding, the ping-pong between switch and controller needs $\sim$ an order of magnitude more time.

Assignment: a custom SDN controller

The default POX L2 Learning Switch controller module installs rules on the switch. Your job is to modify it so that:

1. The controller installs no rule
2. Each packet that reaches the switch is sent to the controller
3. The controller, literally each time, explicitly tells the switch in which port to forward the packet to.

Relevant documentation:
https://noxrepo.github.io/pox-doc/html/#openflow-messages

To shutdown the POX controller currently in execution: (outside Mininet)

```
> sudo pkill python
```

Assignment: a custom SDN controller

hints

- ⋆ You need to know the existence of 3 OpenFlow messages and what they do: *Packet-In*, *Flow-Mod*, and *Packet-Out*
- ⋆ You can do all the three points above in one shot
- ⋆ It's enough to modify 1 (yes, ONE) line of code
  (but be careful, you might find two lines with flow_mod...)

Assignment: NFV

With this topology:

```
> sudo mn --controller remote --topo tree,depth=2,fanout=5
```

your job is to build a Software-Defined firewall with POX, by modifying or creating form scratch a POX module. The firewall should:

1. mutually block traffic between h1 and h2
2. mutually block traffic from h1 and h25
3. block traffic from h6 to h1 but not the other way around