Mininet & OpenFlow

24/11/2016

Firt steps: configure VM

PREREQUISITE: download and install the mininet VM from http://mininet.org/download/

THEN:

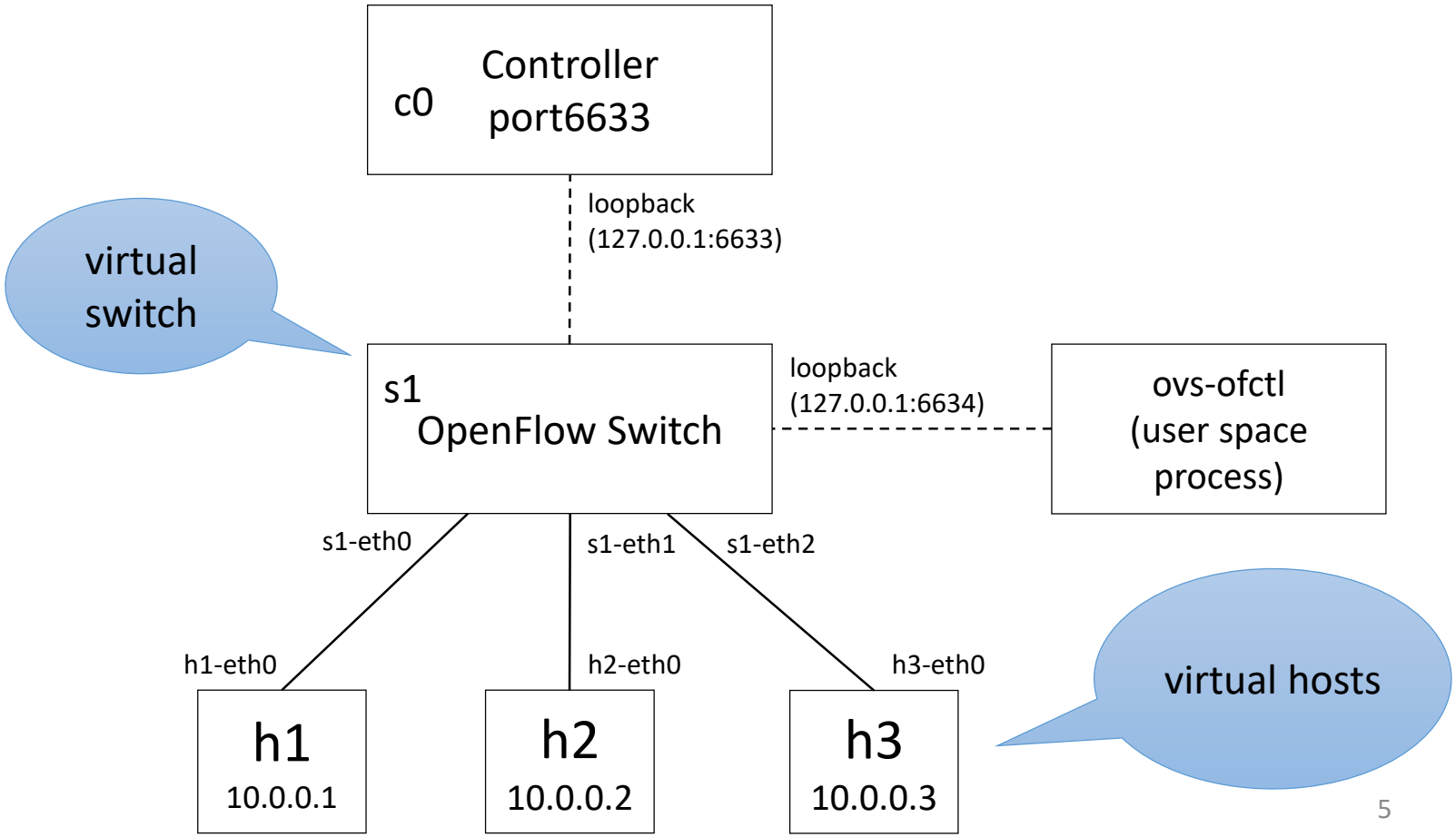
- Change network settings by enabling «bridge»
- Start the mininet VM
- From Host terminal(Ubuntu) launch:
  - ssh –Y mininet@<address_of_VM>
- Password is mininet

First sample commands

- `sudo mn -h`
- `sudo mn --topo single,8 --test pingall`
- `sudo mn --topo single,8 --test iperf`
- `sudo mn --topo linear,8 --test pingall`
- `sudo mn -c`

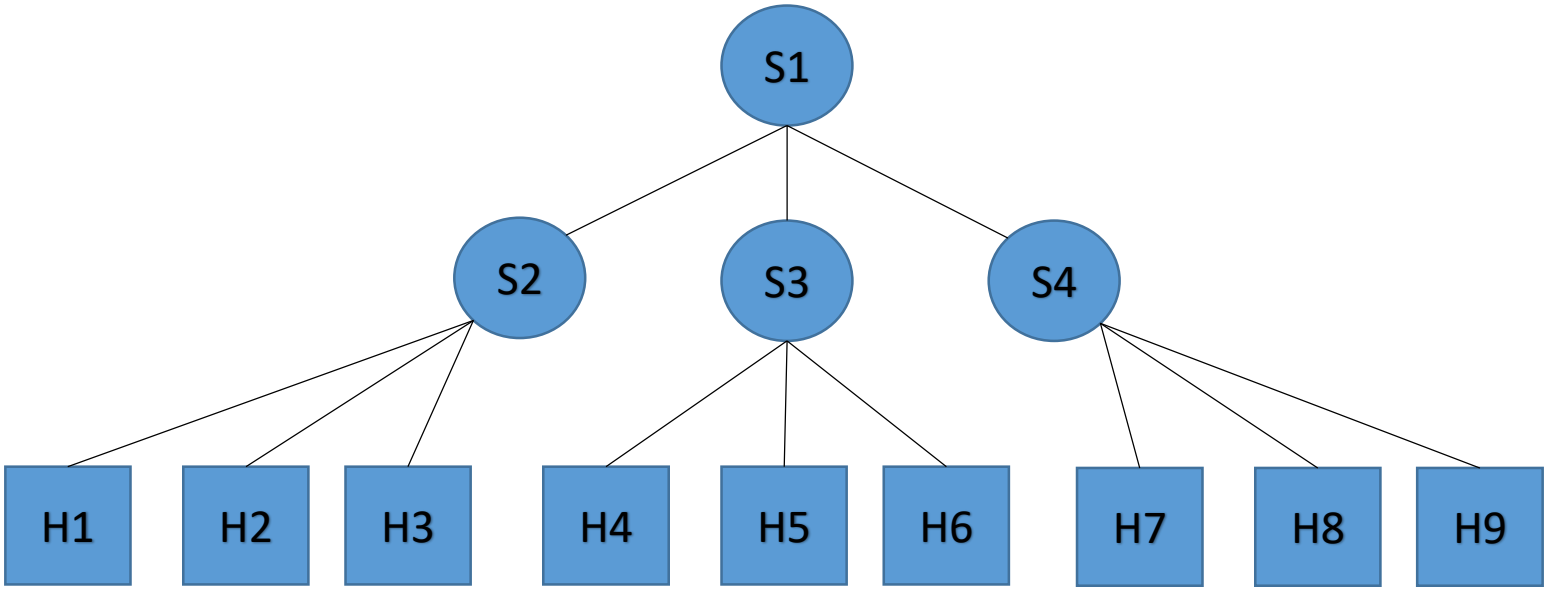
Setup 1: Mininet-based Single Switch

```
sudo mn --topo single,3 --switch ovsk --controller remote
```



First sample commands

- `sudo mn --topo tree,depth=2,fanout=3 --test pingall`
- `sudo mn --topo tree,depth=2,fanout=3 --link tc,bw=5,delay=40ms`



Custom Topologies

```
from mininet.topo import Topo

class MyTopo( Topo ):

        def __init__( self ):

                # Initialize topology
                Topo.__init__( self )

                # Add hosts and switches
                leftHost = self.addHost( 'h1' )
                rightHost = self.addHost( 'h2' )
                leftSwitch = self.addSwitch( 's3' )
                rightSwitch = self.addSwitch( 's4' )

                # Add links
                self.addLink( leftHost, leftSwitch )
                self.addLink( leftSwitch, rightSwitch )
                self.addLink( rightSwitch, rightHost )


topos = { 'mytopo': ( lambda: MyTopo() ) }
```

Custom Topologies

```
switch = self.addSwitch('s1')

# Python's range(N) generates 0..N-1
for h in range(n):
        host = self.addHost('h%s' % (h + 1))
        self.addLink(host, switch)
```

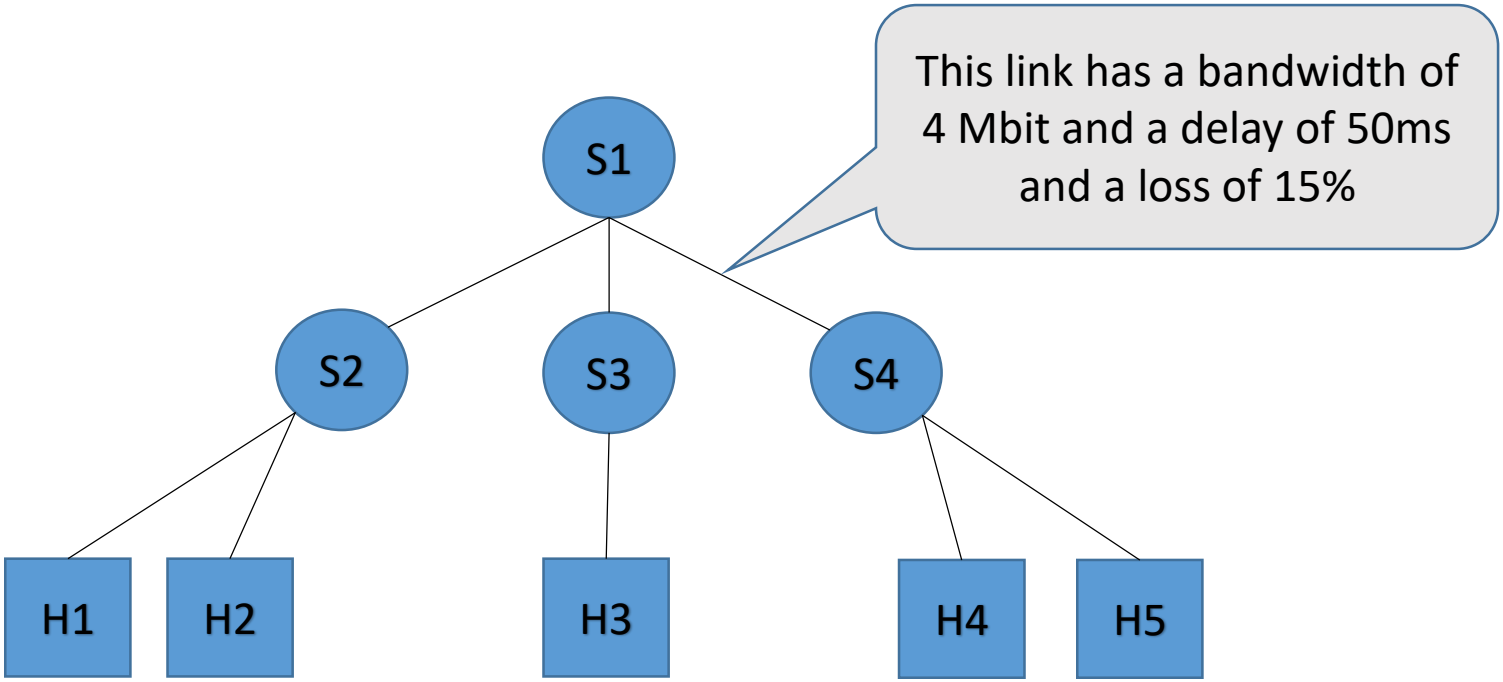
```
# Each host gets 50%/n of system CPU
host = self.addHost('h%s' % (h + 1), cpu=.5/n)

# 10 Mbps, 5ms delay, 10% loss, 1000 packet queue
self.addLink(host, switch, bw=10, delay='5ms',
                           loss=10, max_queue_size=1000, use_htb=True)
```

```
sudo mn --custom ~/mininet/custom/topo-2sw-2host.py
--topo mytopo –-link tc --test pingall
```

Exercise 1

- Build the following topology, execute a ping between all the hosts and measure the bandwidth between host 1 and host 4



Pox Controller

- POX is an open platform for the rapid development and prototyping of **network control software**

- Pox architecture is ‘‘component based’’

- Ex: `./pox.py samples.pretty_log forwarding.l2_learning`

- Some stock components:
  - openflow.of_01 *(usually started automatically)*
  - forwarding.hub
  - forwarding.l2_learning
  - forwarding.l2_pairs
  - forwarding.l2_multi
  - openflow.spanning_tree
  - openflow.discovery
  - **misc.of_tutorial** → the component we will customize in this lab
  - …

Packets in POX

- POX generally works with `ethernet` packets
  - Which often contain `ipv4` packets…
    - (which often contain `tcp` packets...)
- Some of the packet types supported by POX:
  - ethernet, arp, ipv4, icmp, tcp, udp, dhcp, dns…
- Most packets have some sort of header and some sort of a payload
  - A payload is another type of packet

Ethernet packets in POX

- Class ethernet
  - defined in ~/pox/pox/lib/packet/ethernet.py
- Attributes:
  - dst (EthAddr)
  - src (EthAddr)
  - type (int)
  - effective_ethertype (int)
  - payload (for example an ipv4 packet…)
- Constants:
  - IP_TYPE, ARP_TYPE, VLAN_TYPE, …
- Example: packet.src, packet.IP_TYPE

The Event System

- Event Handling in POX fits into the publish/subscribe paradigm
  - Certain objects publish events and others subscribe to specific events on these objects
- In other words: we'd like a particular piece of code to be called
- Ex: `chef.addListenerByName("SpamFinished", spam_ready)`

The object that raises the event

The name of the event

The function handling the event

The Event System

- Ex: object chef raises two events, SpamStarted and SpamFinished

```
class HungryPerson (object):
  """ Models a person that loves to eat spam """

  def __init__ (self):
    chef.addListeners(self)

  def _handle_SpamStarted (self, event):
    print "I can't wait to eat!"

  def _handle_SpamFinished (self, event):
    print "Spam is ready!  Smells delicious!"
```

Example: empty controller

- Let's go to the code and see the events ConnectionUp and PacketIn!

- ConnectionUp: fired in response to the establishment of a new control channel with a switch

- PacketIn: Fired when the controller receives an OpenFlow Packet-In message from a switch
  - Attributes:
    - `port` (`int`): number of port the packet came in on
    - `data` (`bytes`): raw packet data
    - `parsed` (`packet subclasses`): packet's parsed version
    - `ofp` (`ofp_packet_in`): OpenFlow message which caused this event

Packet-In message in POX

- The POX object type is `ofp_packet_in`
- Attributes:
  - `in_port` (`int`): number of port the packet came in on
  - `data` (`bytes`): raw packet data
  - `buffer_id` (`int`): ID of the buffer in which the packet is stored at the switch
  - …

Packet-Out message in POX

- The POX object type is `ofp_packet_out`

attribute	type	default	notes
in_port	int	OFPP_NONE	Switch port that the packet arrived on, if resending a packet
data	bytes / ethernet / ofp_packet_in	''	The data to be sent. If you specify an `ofp_packet_in` for this, `in_port`, `buffer_id`, and `data` will all be set correctly – this is the easiest way to resend a packet.
buffer_id	int/None	None	ID of the buffer in which the packet is stored at the switch. If you're not resending a buffer by ID, use None
actions	list of ofp_action_XXXX	[ ]	An action or a list of actions

OpenFlow actions in POX

- `ofp_action_output`: Forward packets out of a port
- `Ex: of.ofp_action_output(port = 4)`

Output port for the packet

Reference to the object that manages the OpenFlow protocol

Possible values for ‘‘port’’:

- **OFPP_IN_PORT**: Send back out the port the packet was received on
- **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
- **OFPP_ALL**: output all openflow ports except the in port
- **OFPP_NONE**: Output to no where
- …

OpenFlow messages in POX

```
""" Instructs the switch to resend a packet that
it had sent to us. "packet_in" is the ofp_packet_in object
the switch had sent to the controller due to a table-miss. """

msg = of.ofp_packet_out()
msg.data = packet_in

# Add an action to send to the specified port
action = of.ofp_action_output(port = out_port)
msg.actions.append(action)

# Send message to switch
self.connection.send(msg)
```

Example: `of_tutorial.py`

- Let's go to the code and see the OpenFlow tutorial!
- You can find the code here:

~/pox/pox/misc/of_tutorial.py

- To start the controller, type in the ~/pox folder:

```
./pox.py misc.of_tutorial samples.pretty_log
```

Exercise 2

- Modify the `of_tutorial.py` to implement the behavior of a learning switch using the OpenFlow message `Packet-Out`

IP packets in POX

- `Class ipv4`
  - defined in ~/pox/pox/lib/packet/ipv4.py
- Attributes:
  - `dstip (IPAddr)`
  - `srcip (IPAddr)`
  - `protocol (int)`
  - `payload (for example a TCP packet…)`
- Constants:
  - `TCP_PROTOCOL, UDP_PROTOCOL, …`
- `Example: packet.srcip, packet.TCP_PROTCOL`

TCP packets in POX

- Class tcp
  - defined in ~/pox/pox/lib/packet/tcp.py
- Attributes:
  - dstport (EthAddr)
  - srcport (EthAddr)
  - SYN (bool)
  - FIN (bool)
  - ACK (for example an ipv4 packet…)
  - …
- Example: packet.srcport

Exercise 3

- Develop a firewall that allows only
  - ARP packets
  - TCP packets over IP packets, but only if:
    - directed to host 10.0.0.1 (port 80)
    - host 10.0.0.1 is the source

Flow-Mod message in POX

- The POX object type is `ofp_flow_mod`
- It is used to add/delete/modify flow table entries
- Attributes:
  - `command` (`int`): default is add a rule
  - `idle_timeout` (`int`): rule expire time, default is unlimited
  - `match` (`ofp_match`): the match structure for the rule to match on
  - `actions` → see Packet-Out
  - `data` → see Packet-Out

ofp_match structure

- Defines a set of headers for packets to match against
- You can either build a match from scratch or create one based on an existing packet
- Attributes:
  - `priority:` matching precedence of the flow entry
  - `in_port`: port number the packet arrived on
  - `dl_src`: ethernet source address
  - `dl_dst`: Ethernet destination address
  - `tp_src`: TCP/UDP source port
  - `tp_dst`: TCP/UDP destination port
  - …
- Or you can use `ofp_match.from_packet(<packet>)`

Example

```
# 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.match.dl_type = 0x800
msg.match.nw_dst = IPAddr("192.168.101.101")
msg.match.nw_proto = 6 #TCP protocol
msg.match.tp_dst = 80

msg.actions.append(of.ofp_action_output(port = 4))

self.connection.send(msg)
```

```
# Create a match from an existing packet

# One thing at a time...
msg = of.ofp_flow_mod()

msg.match = of.ofp_match.from_packet(packet)
msg.actions.append(of.ofp_action_output(port = 4))
msg.data = packet_in

self.connection.send(msg)
```

Exercise 4

- Modify the `of_tutorial.py` to implement the behavior of a learning switch using the OpenFlow message `Flow-Mod`