HANDS-ON SDN

Introduction to Software-defined Networking
Block Course – 16-20 March 2015

David Koll

Detailed Course overview

Day	Morning Session 1	Morning Session 2	Afternoon Session 1	Afternoon Session 2
Mon	Introducing SDN I	Lecture Exercises	Introduction to Python	Python Exercises
Tue	Introducing SDN II	Lecture Exercises	Introduction to Python (cont.)	Python Exercises
Wed	Current Research in SDN	Paper Selection and Reading (Teams)	**Hands-On SDN I**	**SDN Exercises**
Thu	**Hands-On SDN II**	**SDN Exercises**	**Hands-On SDN III**	**SDN Exercises**
Fri	Presentation Prep / Exercises	Presentation Prep / Exercises	Wrap-Up & Free Slot	Presentations

Custom Topologies with Mininet Python API

Mininet offers some topologies!

Eg: single switch, linear, tree

What if you want to replicate your very own production network?

Create a custom topology!

Low-level API: Nodes and Links

```
h1 = Host( 'h1' )
h2 = Host( 'h2' )
s1 = OVSSwitch( 's1', inNamespace=False )
c0 = Controller( 'c0', inNamespace=False )
Link( h1, s1 )
Link( h2, s1 )
h1.setIP( '10.1/8' )
h2.setIP( '10.2/8' )
c0.start()
s1.start( [ c0 ] )
print h1.cmd( 'ping -c1', h2.IP() )
s1.stop()
c0.stop()
```

Mid-level API: Network Object

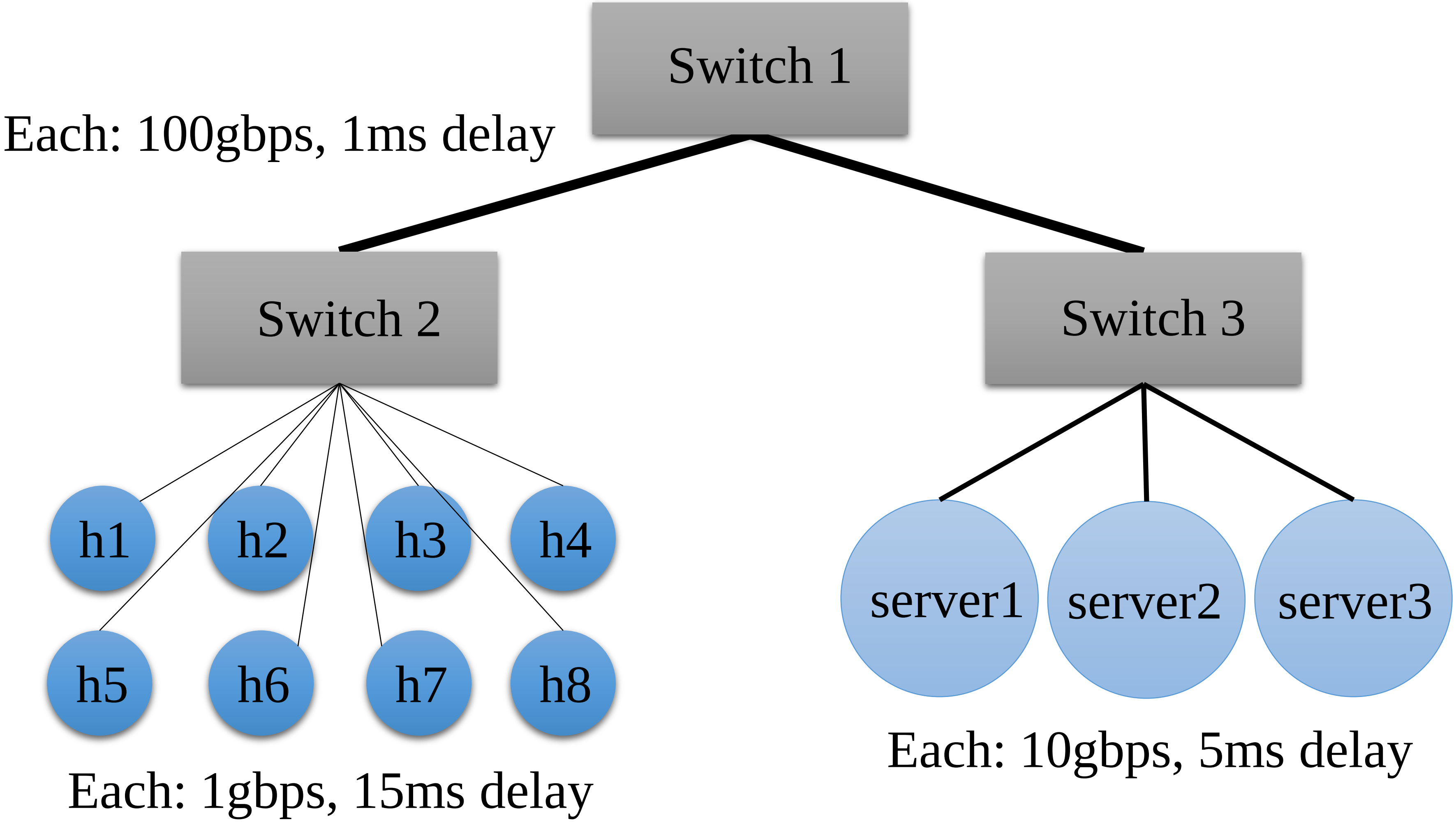
```
net = Mininet()
h1 = net.addHost( 'h1' )
h2 = net.addHost( 'h2' )
s1 = net.addSwitch( 's1' )
c0 = net.addController( 'c0' )
net.addLink( h1, s1 )
net.addLink( h2, s1 )
net.start()
print h1.cmd( 'ping -c1', h2.IP() )
CLI( net )
net.stop()
```

High-level API: Topology templates

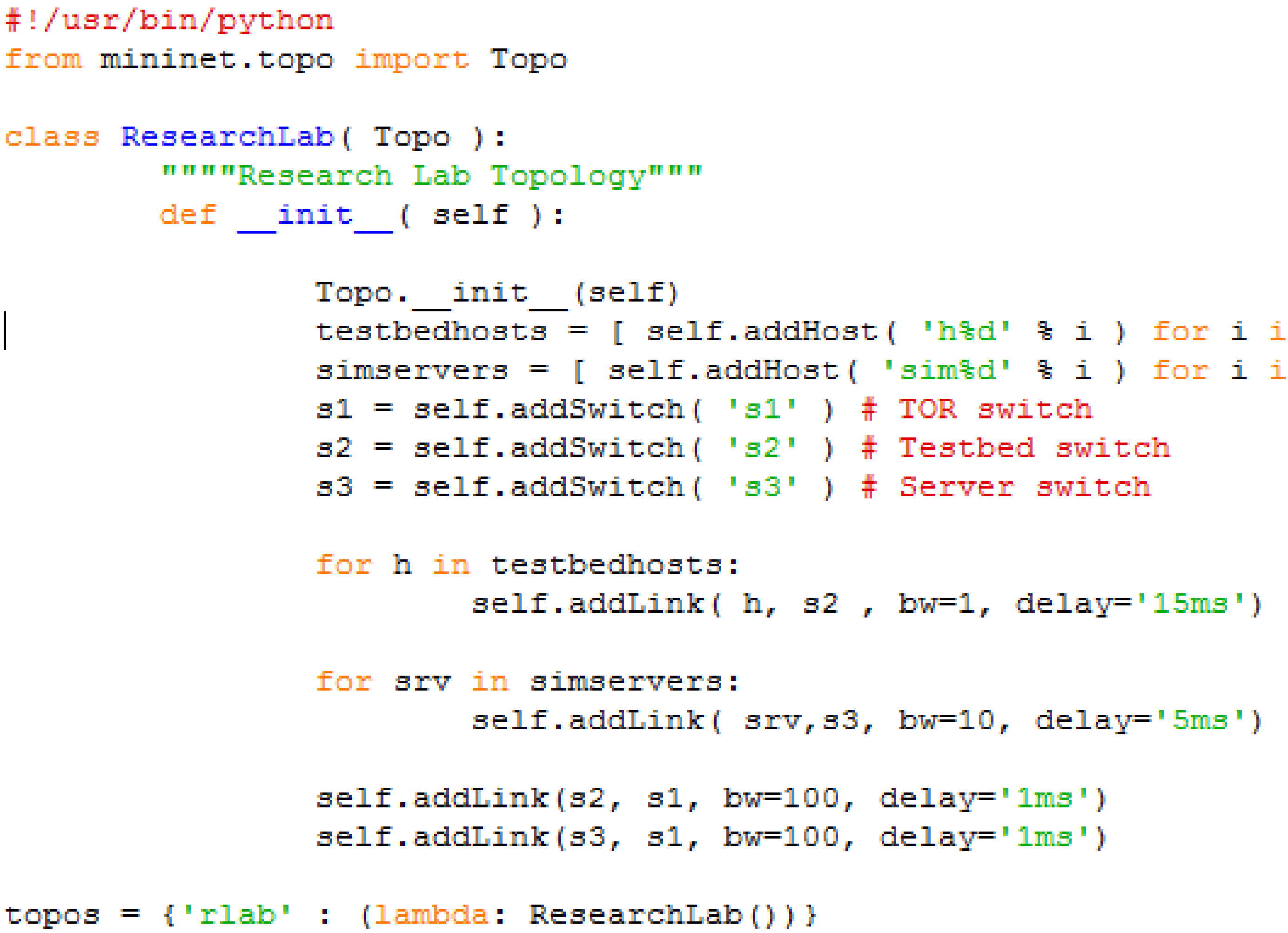
```
class SingleSwitchTopo( Topo ):
     "Single Switch Topology"
     def __init__( self, count=1):
          Topo.__init__(self)
          hosts = [ self.addHost( 'h%d' % i )
                    for i in range( 1, count + 1 ) ]
          s1 = self.addSwitch( 's1' )
          for h in hosts:
               self.addLink( h, s1 )

topos = {'singleswitch' : (lambda: SingleSwitchTopo())}
```

Example Topology – Research Lab



Example Topology – Research Lab



```
#!/usr/bin/python
from mininet.topo import Topo

class ResearchLab( Topo ):
        """Research Lab Topology"""
        def __init__( self ):

                Topo.__init__(self)
                testbedhosts = [ self.addHost( 'h%d' % i ) for i in range( 1, 9) ]
                simservers = [ self.addHost( 'sim%d' % i ) for i in range( 1, 4) ]
                s1 = self.addSwitch( 's1' ) # TOR switch
                s2 = self.addSwitch( 's2' ) # Testbed switch
                s3 = self.addSwitch( 's3' ) # Server switch

                for h in testbedhosts:
                        self.addLink( h, s2 , bw=1, delay='15ms')

                for srv in simservers:
                        self.addLink( srv,s3, bw=10, delay='5ms')

                self.addLink(s2, s1, bw=100, delay='1ms')
                self.addLink(s3, s1, bw=100, delay='1ms')

topos = {'rlab' : (lambda: ResearchLab())}
```

```
sudo mn
--custom rlab.py
--topo rlab
--link=tc
```

The POX Controller

- Invoke with: ./pox.py [options] <component>

- <options> can be:
    - --verbose : display debugging info
    - --no-openflow: do not automatically listen for OpenFlow connections

- <components> are the real meat!
    - There are some basic components we will use for this class
    - Intention: developers will build their own components

The POX Controller - Components

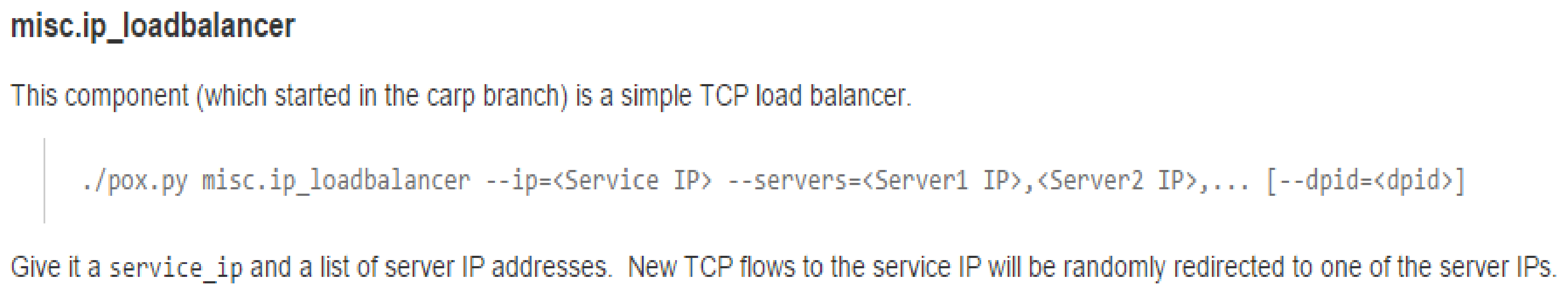
- Some stock components:
  - py
  - forwarding.hub
  - forwarding.l2_learning
  - forwarding.l2_pairs
  - forwarding.....

  - openflow.webservice
    - Creates a webinterface to interact with OpenFlow

  - openflow.of_01
    - Communicates with OpenFlow 1.0 switches

The POX Controller - Components

- Developing your own components:
  - If you are interested:
    - https://openflow.stanford.edu/display/ONL/POX+Wiki#POXWiki-DevelopingyourownComponents



misc.ip_loadbalancer

This component (which started in the carp branch) is a simple TCP load balancer.

```
./pox.py misc.ip_loadbalancer --ip=<Service IP> --servers=<Server1 IP>,<Server2 IP>,... [--dpid=<dpid>]
```

Give it a `service_ip` and a list of server IP addresses. New TCP flows to the service IP will be randomly redirected to one of the server IPs.

- In general: POX wiki a good place to look for help
  - https://openflow.stanford.edu/display/ONL/POX+Wiki

POX Event Creation

- Recall: Components produce events, you write handlers for these events (**`pox.lib.revent`**)
- A class can raise events (has to inherit from `EventMixin` class)

```
class PacketHandler (EventMixin):
  """
  Class modeling a packet handler
  """
  _eventMixin_events = set([
    packet_in,
        ...
  ])
```

```
handler = PacketHandler()
handler.raiseEvent(packet_in, "Generic")
```

POX API – Writing Handlers

- When writing or modifying components (you will do the later in this course), POX offers some helpful API.
  - E.g.: API for packet handling: **`pox.lib.packet`** import pox.lib.packet

Example: Get L2 source and destination from a packet

```
def _handle_PacketIn(self, event):
        packet = event.parsed # POX is based on events!
        src_of_packet = packet.src #returns an EthAddr
        dst_of_packet = packet.dst #also returns an EthAddr
```

POX API – Writing Handlers

- When writing or modifying components (you will do the later in this course), POX offers some helpful API.
  - E.g.: API for packet handling: **`pox.lib.packet`** import pox.lib.packet

```
Example: Get L2 source and destination from a packet

def _handle_PacketIn(self, event):
        packet = event.parsed # POX is based on events!
        src_of_packet = packet.src #returns an EthAddr
        dst_of_packet = packet.dst #also returns an EthAddr
```

POX – Writing Handlers

- When writing or modifying components (you will do the latter in this course), POX offers some helpful API.
  - E.g.: API for packet handling: **`pox.lib.packet`**

`Example: Get source IP from a packet`

```
def _handle_PacketIn(self, event):
      "check if packet is an IP packet"
      packet = event.parsed
      ip = packet.find('ipv4') #check if packet is IP
      if ip is None: #packet is not IP
            return
      print "Source IP: ", ip.srcip
```

POX Listening for Events

```
inhandler.addListener(packet_in, _handle_packetIn)

inhandler.addListenerByName(“packet_in“, _handle_packetIn)


def launch ():
      """
      Starts the component
      """
      def start_switch (event):
             log.debug("Controlling %s" % (event.connection,))
             Tutorial(event.connection)
             inhandler.addListenerByName(“packet_in", \
                                       handle_packetIn)
```

POX Component - Example

Demo of Example Code

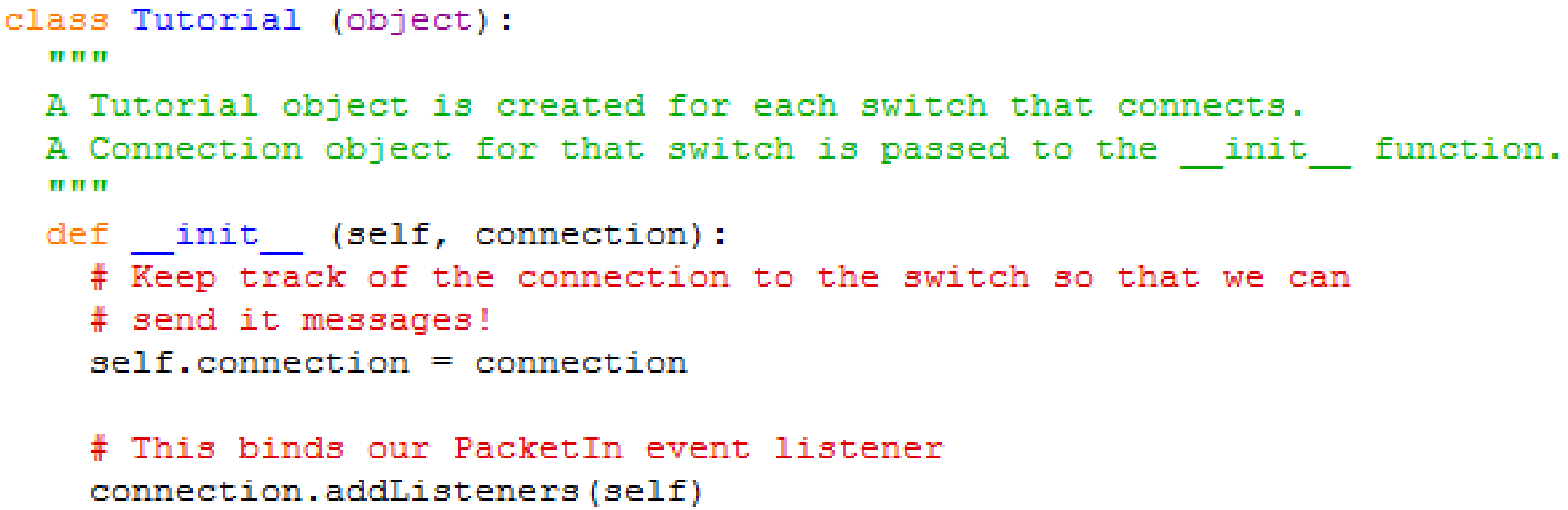
POX and Openflow

- Usually, switches connect to POX automatically via OpenFlow
  - Exception: `no-openflow` option (see previous slides)
- So – how do we communicate with them?
- Standard POX component for OpenFlow: **`openflow.of_01`**

POX – Connection Elements

- Upon connecting to POX, a switch is associated with a `Connection` object
- Use that object‘s `send()` method to send messages to the switch
- `Connection` object will raise events on the corresponding switch
  - Create **event handlers** for events you are interested in

POX – Connection Elements



```
class Tutorial (object):
  """
  A Tutorial object is created for each switch that connects.
  A Connection object for that switch is passed to the __init__ function.
  """
  def __init__ (self, connection):
    # Keep track of the connection to the switch so that we can
    # send it messages!
    self.connection = connection

    # This binds our PacketIn event listener
    connection.addListeners(self)
```

In Practice

- Launch our component.
- Add one event listener for PacketIn

```
from pox.core import core
import pox.openflow.libopenflow_01 as of

log = core.getLogger()

def launch ():
    “Starts the Component“
    core.openflow.addListenerByName("PacketIn",
                   _handle_packetin)

    log.info("Switch running.")
```

In Practice

- Write packet handler (here: flood packet)

```
def _handle_packetin (event):
    “Handle PacketIn“
    packet = event.parsed
    send_packet(event, of.OFPP_ALL) #broadcast
```

In Practice

- Write send_packet method (simplified)

```
def send_packet (event, dst_port):
     “Instructs switch to send packet via dst_port“
     msg = of.ofp_packet_out(in_port=event.ofp.in_port)
     msg.data = event.ofp.data
     msg.actions.append(of.ofp_action_output(port = dst_port))

     event.connection.send(msg)
```

POX OpenFlow Events

- ConnectionUp / ConnectionDown
- PortStatus – indicates a change in ports on switch
- FlowRemoved – e.g. on timeout of a flow entry
- PacketIn – on packet in – can for instance indicate a table miss

Direction: Switch → Controller

POX OpenFlow Messages

- ofp_packet_out – instruct a switch to send out a packet
- ofp_flow_mod – instruct a switch to modify a flow table
- ofp_stats_request – request statistics from switch

Direction: Controller → Switch

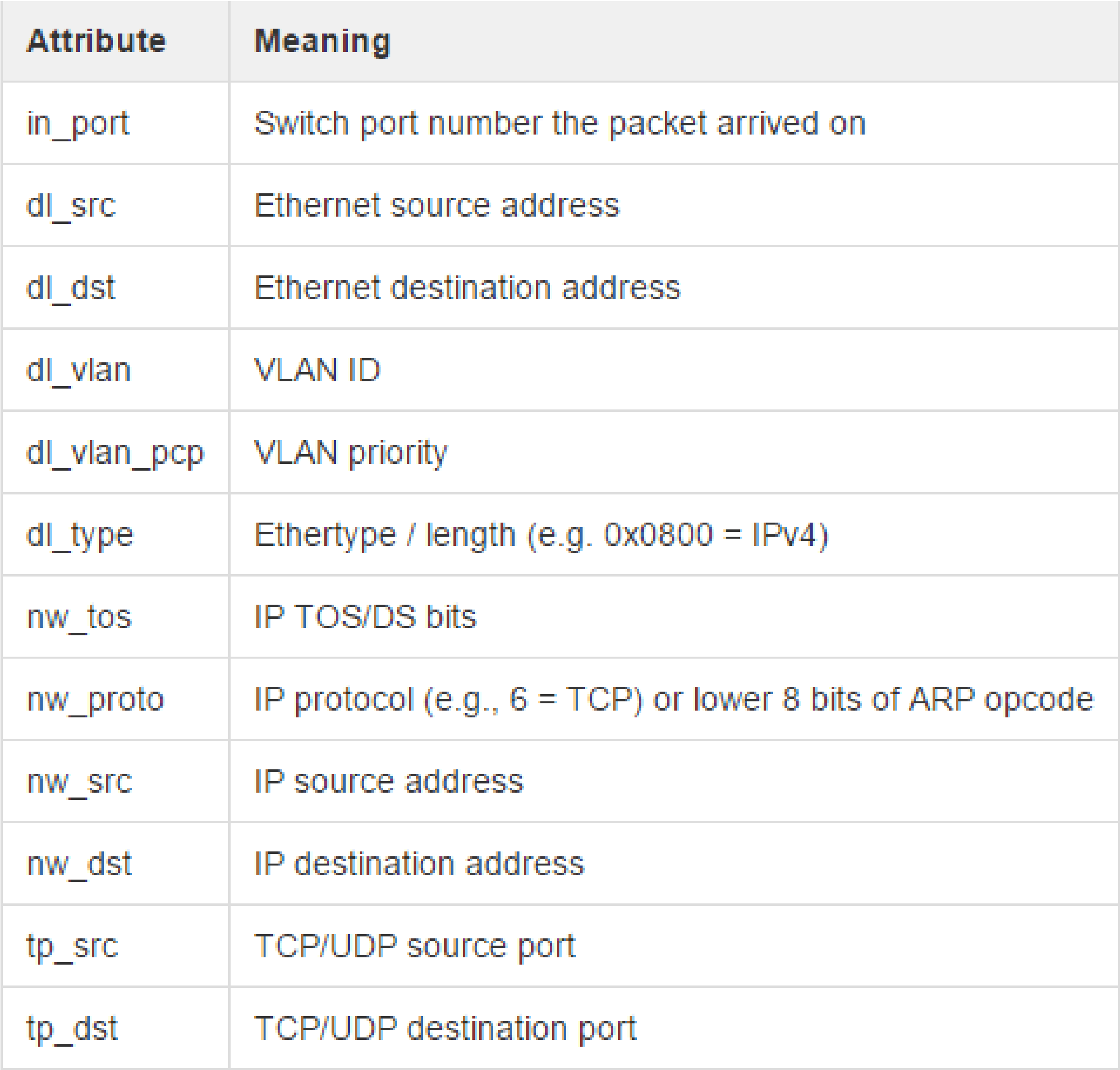
ofp_flow_mod

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

msg = of.ofp_flow_mod()
msg.priority = 42
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)

# create ofp_flow_mod message to delete all flows
msg = of.ofp_flow_mod(command=of.OFPFC_DELETE)
self.connection.send(msg)
```

Match fields



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

Match fields

Switch Port	MAC src	MAC dst	Eth type	VLAN ID	IP Src	IP Dst	IP Prot	TCP sport	TCP dport	Action
*	*	00:1f:..	*	*	*	*	*	*	*	port6

Actions

- Output – outputs a packet on a certain port
- Set VLAN ID
- Set Ethernet Src/Dst Address
- Set IP Src/Dst Address…

Actions

```
#sending an out packet

msg = of.ofp_packet_out(in_port=of.OFPP_NONE)
msg.actions.append(of.ofp_action_output(port = outport))
msg.buffer_id = <some buffer id, if any>
connection.send(msg)
```

Putting it Together

Demo Code

In Practice

- Code on previous slides implemented a hub behaviour

- Exercise: modify hub behaviour to learning switch behaviour

POX and Openflow

- More details: Best to read POX wiki:
  - https://openflow.stanford.edu/display/ONL/POX+Wiki#POXWiki-OpenFlowinPOX

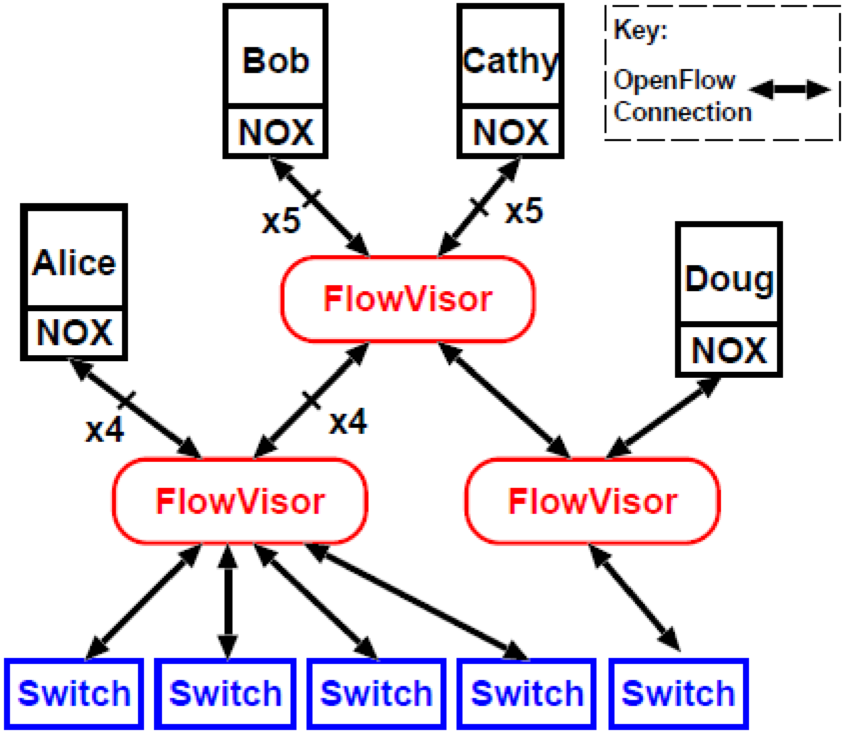
Exercise!

Time for Exercise 7

FlowVisor

- Exercise 5: You have already installed FlowVisor

- Recall: FlowVisor is an extra layer between controllers and switches



Taken rom: Sherwood, et al. *"Flowvisor: A network virtualization layer." OpenFlow Switch Consortium, Tech. Rep* (2009).

FlowVisor

- Basic procedure:
    - Create and start your network topology with Mininet
    - Connect Flowvisor to switches on standard port
    - Slice network with Flowvisor
    - Connect Controllers to Flowvisor slices

FlowVisor

- Basic procedure:
  - Create and start your network topology with Mininet
  - Connect Flowvisor to switches on standard port
  - Slice network with Flowvisor
  - Connect Controllers to Flowvisor slices

Connecting FlowVisor

- FlowVisor operates outside of Mininet!

```
$ sudo /etc/init.d/flowvisor start
```

(see demo)

- Afterwards: use flowvisor control (command: `fvctl`) to slice

Slicing the Network with FlowVisor

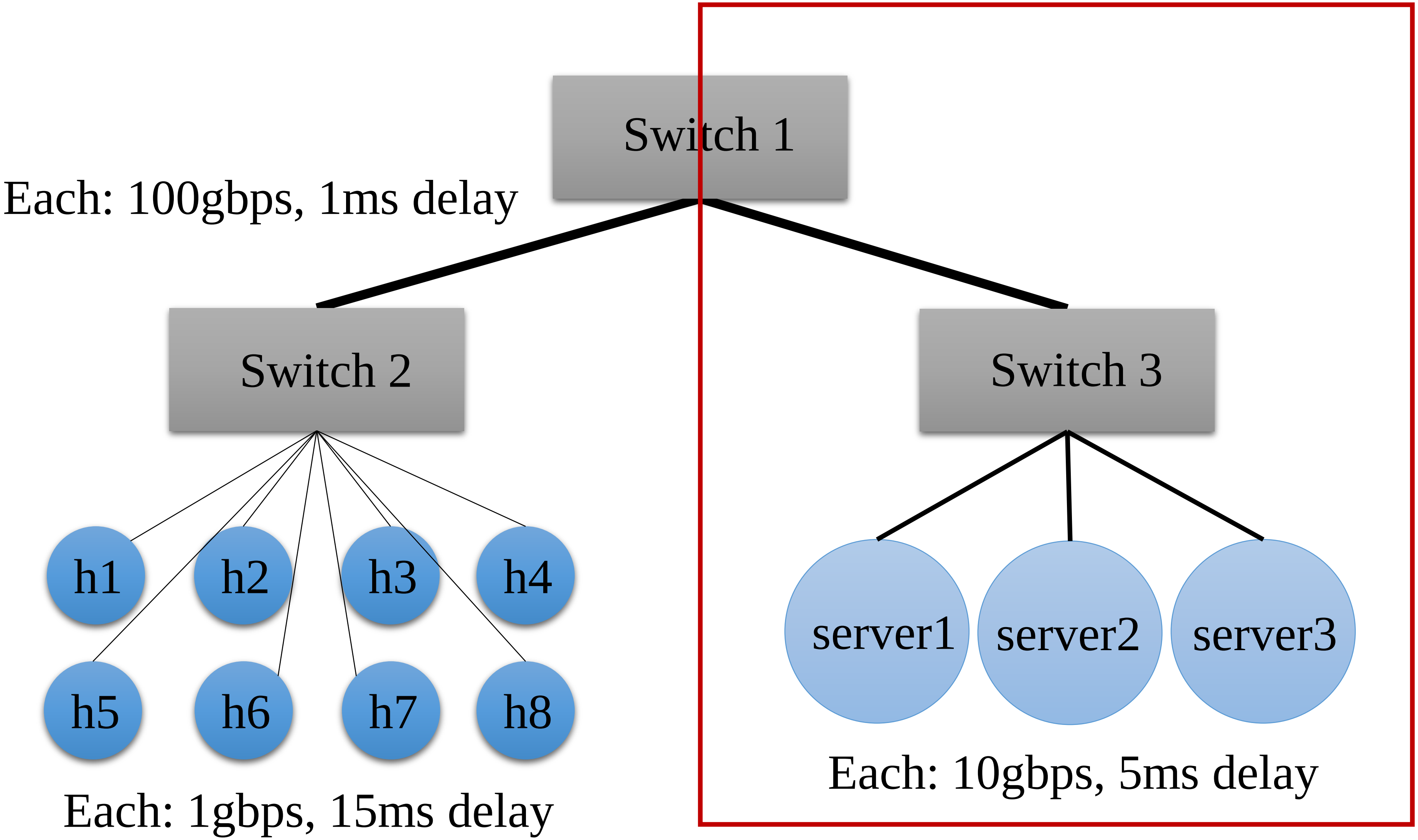
- First: enable topology controller

```
$ fvctl –f /dev/null set-config --enable-topo-ctrl
$ sudo /etc/init.d/flowvisor restart
```

(see demo)

- -f /dev/null option: -f points to pwd file – in our case: empty pw

Let’s slice the research lab



Slicing the Network with FlowVisor

- Want to create slice for servers. Have a look at topology:

```
$ fvctl –f /dev/null list-slices
$ fvctl –f /dev/null list-flowspace
$ fvctl –f /dev/null list-datapaths
$ fvctl –f /dev/null list-links
```

(see demo)

Slicing the Network with FlowVisor

- Add slices with

```
fvctl add-slice [options] <slicename>
                <controller-url> <admin-email>
```

```
$ fvctl –f /dev/null add-slice servers
                   tcp:localhost:10001 admin@servers
```

(see demo)

Add Flowspaces

- Add flowspaces with

```
fvctl add-flowspace [options] <flowspace-name> <dpid>
                      <priority> <match> <slice-perm>
```

```
$ fvctl –f /dev/null add-flowspace switch1-port2
                   1 1 in_port=2 servers=7
```

- Permissions: Bitmask
  - `1=DELEGATE, 2=READ, 4=WRITE`

(see demo)

Connect Controllers

- Start controller and connect to FlowVisor

(see demo)

Test Slicing

- Servers should be able to ping each other, but not any hosts

(see demo)

Exercise!

Time for Exercise 8