SDN Controllers

POX

Ryu

OpenDaylight

OpenFlow
SDN Controllers

• The three concepts:
  – Programmability
  – Separation of the control and data planes
  – Management of ephemeral network state in a centralized control plane, regardless of the degree of centralization

• A software system or collection of systems that together provides above ideal concepts
SDN Controllers

- An idealized controller:

  - **Network Service Functions**
    - Slicing Manager
    - Topology Manager
    - Host Tracker
    - Switch Manager
    - Pwdg. Manager

  - **Network Orchestration Functions**

  - **Service Management Functions**

  - **Network Elements**

  - **Build Environment**

  - **HA/Clustering/State Storage**

  - **REST**

  - **OSGI FRAMEWORK**

  - **Network Services Platform API (NB APIs)**

  - **Java native function calls or RPC**

  - **API**

  - **REST/HTTP**

  - **Abstraction Layer**

  - **SB Protocol**

  - **PCEP**

  - **SB Proto Libraries**

  - **PCEP Libraries**

  - **OF x.y**

  - **OpenFlow Libraries**

  - **OpenFlow**

  - **PCEP**
SDN Controllers

• SDN controller provides:
  – Management of network state:
    • State management may involve a database, which serve as a repository for information derived from the controlled network elements
    • Purpose-driven data management processes or in-memory database
  – A high-level data model:
    • Captures the relationships between managed resources, policies and other services
  – A modern, often RESTful API:
    • Exposes the controller services to an application
    • Facilitates the controller-to-application operation
SDN Controllers

• SDN controller provides:
  – Management of network state:
    • State management may involve a database, which serve as a repository for information derived from the controlled network elements
    • Purpose-driven data management processes or in-memory database
  – A high-level data model:
    • Captures the relationships between managed resources, policies and other services
  – A modern, often RESTful API:
    • Exposes the controller services to an application
    • Facilitates the controller-to-application operation
SDN Controllers

• SDN controller provides:
  – A secure TCP control session between controller and associated agents in network
  – A standard-based protocol for the provisioning of application-driven network state on network
  – A device, topology, and service discovery mechanism
    • Path computation system and potentially other network-centric or resource-centric information service
SDN Controllers

- The current commercial SDN controllers:
  - Nicira: NVP
  - NEC: Trema
  - Big Switch Networks: Floodlight/BNC
  - Juniper/Contrail
  - Cisco XNC
SDN Controllers

- Founded in 2007, its network virtualization platform (NVP) was released in 2011
- NVP works with Open vSwitch (OVS), the hypervisor softswitch controlled by the NVP controller cluster
- Most of the OVS programming is achieved with a database-like protocol called the Open vSwitch Data Base Management Protocol (OVSDB)
- OVSDB provides strong management interface to the hypervisor
- OVSDB characteristics include following:
  - Uses JSON for schema format and wire protocol
  - Transactional
  - No-SQL
  - Persistency
  - Monitoring capability
  - Stores both provisioning and operational state

- The NVP controller is a cluster of servers that use database synchronization to share state
SDN Controllers

- NVP OVSDB interactions with virtual switches and 3rd party hardware

Virtual Network (VTEP Devices)
- OVS-1
- OVS-2
- ...

H/W VTEP Gateway Devices

1. Tunnel (VTEP) Provisioning
2. Tunnel to virtual Network binding
3. Tunnel to mac-address binding
4. Virtual Network statistics

NVP OVSDB interactions with virtual switches and 3rd party hardware
SDN Controllers

- A layer 2 or layer 3 gateway product converts Nicira STT tunnel overlays into VLANs (layer 2), VLAN-to-VLAN, or provides NAT-like functionality
- OVS together with gateways and the service nodes support redundant controller connections for high availability
- NVP Manager, the management server with a basic interface used mainly to troubleshoot and verify connections
SDN Controllers

• SDN Controller components
SDN Controllers

• NOX/POX
  – NOX:
    • NOX developed by Nicira and donated to the research community and hence becoming open source
    • Subsequently extended and supported via ON.LAB activity at Stanford with major contribution from UC Berkeley and ICSI
    • NOX provides a C++ API to OpenFlow and an asynchronous, event-based model
    • NOX is both a primordial controller and a component-based framework for developing SDN applications
SDN Controllers

• NOX/POX
  – NOX:
    • NOX core provides helper methods and APIs for interacting with OpenFlow switches, including a connection handler and event engine
    • Additional components that use that API are available, including host tracking, routing, topology, and Python interface implemented as a wrapper for the component API
    • NOX often used in academic network research to develop SDN application:
      – SANE: An approach to representing the network as a filesystem
SDN Controllers

• NOX Architecture
SDN Controllers

• NOX/POX
  – POX:
    • The newer, Python-based version of NOX
    • Has a high-level SDN API including a query-able topology graph and support for virtualization
  • Advantages over NOX:
    – Has a Pythonic OpenFlow interface
    – Has reusable sample components for path selection, topology discovery, and so on
    – Runs anywhere and can be bundled with install-free PyPy runtime for easy deployment
    – Specifically targets Linux, Mac OS, and Windows
    – Supports the same GUI and virtualization tool as NOX
    – Performs well compared to NOX applications in Python
SDN Controllers

- **Trema**:  
  - An OpenFlow programming framework for developing an OpenFlow controller that was originally developed by NEC  
  - Provides basic infrastructure services as part of its core modules that support the development of user modules in Ruby or C  
  - Developers can individualize or enhance the base controller functionality by defining their own controller subclass object  
  - The core modules provide a message bus that allows the communication
SDN Controllers

- **Trema:**
SDN Controllers

• Trema:
  – The infrastructure provides a command-line interface and configuration filesystem for configuring and controlling applications, managing messaging and filters, and configuring virtual networks – via Network Domain Specific Language (DSL)
  – Trema-based OpenFlow controller can interoperate with any element agent that supports OpenFlow without require a specific agent
SDN Controllers

- Trema architecture and API interfaces
SDN Controllers

• Ryu:
  – Component-based, open source framework implemented entirely in Python
  – Components include:
    • OpenFlow wire protocol support
    • Event management
    • Messaging
    • In memory state management
    • Application management
    • Reusable libraries
  – Has an Openstack Quantum plug-in supports both GRE based overlay and VLAN
SDN Controllers

- Ryu architecture, APIs, applications
SDN Controllers

• Ryu:
  – Component-based, open source framework implemented entirely in Python
  – Components include:
    • OpenFlow wire protocol support
    • Event management
    • Messaging
    • In memory state management
    • Application management
    • Reusable libraries
  – Has an Openstack Quantum plug-in supports both GRE based overlay and VLAN
SDN Controllers

• Big Switch Networks/Floodlight:
  – Popular SDN controller contribution from Big Switch Networks to the open source community
  – Based on Beacon from Stanford University
  – An Apache-licensed, Java-based controller
  – Modular core architecture with components including:
    • Topology management
    • Device management (MAC and IP tracking)
    • Path computation
    • Generalized storage abstraction for state storage
SDN Controllers

- Floodlight architecture:
SDN Controllers

• Big Switch Networks/Floodlight:
  – Core module handles I/O from switches and translates OpenFlow messages into Floodlight events, creating an event-driven, asynchronous application framework
  – Floodlight incorporates a threading model that allows modules to share threads with other modules
  – Floodlight is Java/Jython centric
    • Jython: Python for the Java Platform
SDN Controllers

• Big Switch Networks/Floodlight:
  – Features:
    • Offers a module loading system that make it simple to extend and enhance
    • Easy to set up with minimal dependencies
    • Supports a broad range of virtual/physical OpenFlow switches
    • Can handle mixed OpenFlow and non-OpenFlow networks
    • Designed to be high-performance – core from a commercial product from Big Switch Networks
    • Support for OpenStack cloud orchestration platform
The SDN Stack

Monitoring/debugging tools
- oftrace
- oflops
- openseer

Applications
- ENVI (GUI)
- LAVI
- n-Casting
- ...

Controller
- NOX
- Beacon
- Trema
- Maestro
- ...

Slicing Software
- FlowVisor
- Console

FlowVisor

Commercial Switches
- HP, NEC, Pronto, Juniper... and many more

OpenFlow Switches
- Software Ref. Switch
- NetFPGA
- Broadcom Ref. Switch
- OpenWRT
- PCEngine WiFi AP
- Open vSwitch
NETWORK CONTROL PLANE
Network Control Plane

• The part of the router architecture
• It establishes the local data set used to create the forwarding table entries
• The data set used to store network topology is called routing information base (RIB)
  – Kept consistent (loop-free) through the exchange of information between other instance of control plane
Network Control Plane

• A forwarding table entries (FIB) are often mirrored between the control and data planes
  – FIB is programmed once the RIB is consistent and stable

• The view of the network topology in RIB can be:
  – Manually programmed
  – Learned through observation
  – Built from pieces of information gathered
  – Combination of above
Network Control Plane

Typical Network

Control Plane

Switch

FIB

RIB

1

2

3

4

C

A

B
Network Control Plane

• Packets received by node A forwarded to node B
  – Control plane and data plane running on separate processor
  – Packets received on the input port of the line card where the data plane resides
  – If the packet comes from an unknown MAC address, it is redirected (4) to the control plane to be learned, processed, and later forwarded
Network Control Plane

• A packet delivered to the control plane:
  – The information contained is processed and possibly result in an alteration of the RIB (new route is learned)

• Control plane returns the packet (C) to the data plane (2), and forward the packet
  – FIB is programmed in step (C)

• The same packets processing happens in the node B
Network Control Plane

- Layer 2 control plane focuses on hardware or physical layer addresses (MAC)
- Layer 3 control plane built to facilitate network layer addresses (IP protocol)
- Scaling concerns:
  - Layer 2 and 3 are merged due to layer 2 doesn’t scale well for large number hosts
  - End hosts moving between networks, resulting massive churn of forwarding tables
Network Control Plane

• In layer 2 network, forwarding focuses on MAC addresses reachability:
  – Primary deal with MAC address storing
  – Hard to manage due to large number of end hosts

• Layer 3 network focuses on IP address reachability:
  – Primary concerns destination IP prefix for both unicast and multicast
  – Used to segment or stitch together layer 2 domains to overcome scale problem
Network Control Plane

• Layer 2 bridges that represent some sets of IP subnetworks are typically connected together with a layer 3 router
• Layer 3 routers are connected to form a larger network
• Protocols blurring these lines:
  – Multiprotocol Label Switching (MPLS)
  – Ether Virtual Private Network (EVPN)
  – Locator/ID Separation Protocol (LISP)
Network Control Plane

- Multiprotocol Label Switching Protocol:
  - Combines the best part of layer 2 forwarding and best part of layer 3 IP routing = extremely fast-packet forwarding
  - Directs data from one node to next based on short path labels rather than long network addresses
    - Avoiding intensive lookups in a routing table
  - Labels used to identify paths between distant nodes rather than endpoints
  - It can encapsulate packets of various network protocols
Network Control Plane

• Multiprotocol Label Switching Protocol:
  – Packet forwarding decisions are made only on the contents of the label
    • Allows one to create point-to-point circuits on any type of transport medium, and any protocols
  – It operates at the layer between layer 2 and layer 3 (layer 2.5 protocol)
  – Designed to provide a unified data-carrying service for both circuit-based/packet-switching clients
  – Can be used to carry different kinds of traffic: IP, ATM, SONET, and Ethernet
Network Control Plane

• MPLS Operation:
  – It works by prefixing packets with an MPLS header, containing one or more labels
  – Each label contains 4 fields:
    • 20 bits label value
    • 3 bits experimental (QoS and ECN)
    • 1 bit bottom of stack flag
    • 8 bits TTL
Network Control Plane

• MPLS Operation:
  – MPLS-labeled packets are switched after a label lookups instead of looking into IP table (RIB)
    • Faster due to this can be done within the switched fabric other than CPU
  – MPLS packets are routed using label switch router (LSR)
    • Uses label as the index to determine next hop
  – A label edge router (LER) operates at the edge of an MPLS network
    • Labels the IP datagram into MPLS domain
Network Control Plane

• MPLS Operation:
  – Labels are distributed between LERs and LSRs using the label distribution protocol (LDP)
    • It is used to build and maintain label-switched path (LSP) databases
  – LSRs regularly exchange label and reachability information with each other to build a complete network topology
Network Control Plane

• Ethernet Virtual Private Network:
  – A family of methods for utilizing the beauty of MPLS to create VPNs
  – An attempt to solve the layer 2 scale problem by effectively tunneling distant layer 2 bridges together over an MPLS
  – Layer 2 addressing and reachability information exchange over the tunnel does not contaminate the scale of layer 3 networks
Network Control Plane

- Locator/ID Separation Protocol:
  - A map-and-encapsulate protocol
  - Combines two functions in one domain:
    - **Routing locator**: where a client is attached to the network
    - **Identifier**: who the client is
  - Both identifiers and locators can be arbitrary elements
    - IP addresses
    - A MAC address
    - GPS coordinates
Network Control Plane

• Locator/ID Separation Protocol:
  – Originally conceived to address Internet scaling issue:
    • IP addresses denote both location and identity
    • Overloaded IP address makes efficient routing impossible
    • IPv6 does not fix this issue
    • Routers require more expensive memory to hold the Internet routing table
    • Expensive for network builders/operators
    • Replacing equipment for the wrong reason (hold the routing table rather than implementing new features)
Network Control Plane

• Locator/ID Separation Protocol:
  – LISP creates 2 name spaces:
    • **EID** *(Endpoint Identifier)* – The host IP address:
      – Used inside of sites and end-site addresses for hosts and routers
      – EIDs go in DNS records – same as today
      – Generally not globally routed on underlying infrastructure
    • **RLOC** *(Routing Locator)* – The LISP router IP:
      – Used in the core and infrastructure addresses for LISP routers and ISP routers – Routed like today
      – Hosts do not know about them
      – Globally routed and aggregated along the Internet connectivity topology
Network Control Plane

• Locator/ID Separation Protocol:
  – EID packets are encapsulated in RLOC packets and forwarded over the Internet
  – Network-based map-and-encapsulate:
    • No changes in hosts, DNS, or core infrastructure
    • New mapping service required for EID-to-RLOC
Network Control Plane

• Locator/ID Separation Protocol:
Network Control Plane

- LISP header format:
Network Control Plane

- LISP Mapping System:
  - LISP separates “where” and “who”:
    - Network routers responsible for looking up the mapping between endpoint ID (EID) and routing locator (RLOC)
    - The mapping process is invisible to end-hosts
    - Mappings are stored in a distributed database called the mapping system
    - RLOC: An IPv4 or IPv6 address of an egress tunnel router (ETR), the output of an EID-to-RLOC lookup
    - EID: An IPv4 or IPv6 address used in the source/destination fields of 1st LISP header
EID-to-RLOC Mapping System

LISP Delegated Database Tree (LISP-DDT) ➔ Similar to DNS

LISP DDT is a distributed database which embodies the delegation of authority to provide mappings from LISP Endpoint Identifiers (EIDs) to Routing Locators (RLOCs). It is a statically-defined distribution of the EID namespace among a set of LISP-speaking servers, called DDT nodes. Each DDT node is configured with an EID-prefix for which it is authoritative plus information, including the RLOCs for Map Servers or other DDT nodes for each defined more-specific EID-prefix of the authoritative prefix.
Network Control Plane

- **LISP Mapping System:**
  - **LISP network elements:**
    - Ingress Tunnel Router (**ITR**):
      - Finds EID to RLOC mapping
      - Encapsulate to Locators at source site
    - Egress Tunnel Router (**ETR**):
      - Owns EID to RLOC mapping
      - Decapsulates at destination site
  - **LISP forwarding:**
MORE ON MININET

> sudo mn

- controllers
- switches
- hosts
Mininet – Create Learning SW

• Controller choice: POX (Python)
  – The Python-based SDN controller
  – We do not need the reference controller anymore so we can kill it from SSH:

```
$ sudo killall controller
```

  – We should need to make sure everything is clean by running:

```
mininet> exit
$ sudo mn -c
$ sudo mn -topo single,3 --mac --switch ovsk --controller remote
```
Mininet – Create Learning SW

• Controller choice: POX (Python)
  – Then you need to download the POX code from the POX repository on github to VM:
    $ git clone http://github.com/noxrepo/pox
    $ cd pox
  – Now you can try to run a basic hub example:
    $ ./pox.py log.level --DEBUG misc.of_tutorial
  – The above command enables POX verbose logging and start the of_tutorial component
Mininet – Create Learning SW

• Controller choice: POX (Python)
  – The switches may take longer time to connect since the OpenFlow switch losses its connection to a controller, up to a maximum of 15 seconds
  – When the switch connects, POX will display something as follow:

   /INFO:openflow.of_01:[Con 1/1]
    Connected to 00-00-00-00-00-00-01
    DEBUG:samples.of_tutorial:Controlling [Con 1/1]
Mininet – Create Learning SW

• Benchmark hub controller w/ipperf
  – Let’s benchmark the provided of_tutorial hub by running:

```
mininet> pingall
```
  – This will check connectivity of all virtual host, then we can run:

```
mininet> iperf
```
  – You should see that every packet goes up to the controller now
Mininet – Create Learning SW

• Open hub code and modify it
  – Let’s stop the tutorial hub controller using Ctrl-C in SSH terminal
  – We will modify pox/misc/of_tutorial.py. You can open this file in your favorite editor
  – The current code calls `act_like_hub()` function from the handler for packet_in messages to implement switch behavior, you can also try `act_like_switch()` function to explore more about POX
Example: Controller App

```python
def _handle_PacketIn (self, event):
    """
    Handles packet in messages from the switch.
    """

    packet = event.parsed # This is the parsed packet data.
    if not packet.parsed:
        log.warning("Ignoring incomplete packet")
        return

    packet_in = event.ofp # The actual ofp_packet_in message.

    # Comment out the following line and uncomment the one after
    # when starting the exercise.
    self.act_like_hub(packet, packet_in)
    #self.act_like_switch(packet, packet_in)

def launch ()�:
    """
    Starts the component
    """

def start_switch (event):
    log.debug("Controlling %s" % (event.connection,))
    Tutorial(event.connection)
    core.openflow.add_listenerByName("ConnectionUp", start_switch)
```
Mininet – Create Learning SW

• Get to know about Python:
  – A dynamic, interpreted programming language, no separate compilation step
  – Uses indentation rather than “{ }” and “;” to delimit code
  – Dynamic typed. No need to pre-declare variables, types are automatically handled
  – Has built-in hash tables – Dictionaries, and vectors – Lists
  – Object-oriented and introspective, easily print the member variables and functions at run time
Mininet – Create Learning SW

• Python common operations:
  – To initialize a dictionary:
    mactable = {}
  – To add an element to a dictionary:
    mactable[0x123] = 2
  – To check for dictionary membership:
    if 0x123 in mactable:
      print ‘element 2 is in mactable’
    if 0x123 not in mactable:
      print ‘element 2 is not in mactable’
• Python common operations:
  – To print a debug message in POX:
    \[\text{log.debug('saw new MAC!')}\]
  – To print an error message in POX:
    \[\text{log.error('unexpected operation')}\]
  – To print member variables and functions:
    \[\text{print dir(object)}\]
  – To comment a line of code:
    \[\text{# Prepend comments with a #; no // or /**/}\]
Mininet – Create Learning SW

• Python common operations:
  – Sending OpenFlow messages with POX:
    
    ```python
    connection.send(...)  # send an OpenFlow msg
    ```

  • `ofp_action_output` class:
    – An action for use with `ofp_packet_out` and `ofp_flow_mod`, which specifies a switch port that you wish to send the packet out of
    – Example: Create an output action that would send packets to all ports:
      
      ```python
      out_action=of.ofp_action_output(port=of.OFPP_FLOOD)
      ```
Mininet – Create Learning SW

• Python common operations:
  – Sending OpenFlow messages with POX:
    ```python
    connection.send(...) # send an OpenFlow msg
    ```
  • `ofp_match` class:
    – It describes packet header fields and an input port to match on
    – Some notable fields of ofp_match objects are:
      » `dl_src`: The data link layer (MAC) source address
      » `dl_dst`: The data link layer (MAC) destination addr
      » `in_port`: The packet inputs switch port
    – Create a match that matches packets arriving on port 3
    ```python
    match = of.ofp_match()
    match.in_port = 3
    ```
Mininet – Create Learning SW

• Python common operations:
  – Sending OpenFlow messages with POX:

  • ofp_packet_out OpenFlow message:
    – It instructs a switch to send a packet, which might be constructed at the controller, or might be the one switch received, buffered, and forwarded to the controller, those fields are:
      » buffer_id: The ID of the buffer you wish to send
      » data: Raw bytes you wish the switch to send
      » actions: A list of actions to apply
      » in_port: The port number for the packet initially arrived on if you sending by buffer_id, otherwise OFPP_NONE
Mininet – Create Learning SW

• Sending OpenFlow messages with POX:
  • `ofp_packet_out` OpenFlow message:
    – Example: The `send_packet()` method:

```python
def send_packet(self, buffer_id, raw_data, out_port, in_port):
    msg = of.ofp_packet_out()
    msg.in_port = in_port
    if buffer_id != -1 and buffer_id is not None:
        msg.buffer_id = buffer_id
    else:
        if raw_data is None:
            return
        msg.data = raw_data
    action = of.ofp_action_output(port=out_port)
    msg.actions.append(action)
    self.connection.send(msg)
```
Mininet – Create Learning SW

• Sending OpenFlow messages with POX:
  • `ofp_flow_mod` OpenFlow message:
    – It instructs a switch to install a flow table entry
    – Flow tables entries match some fields of incoming packets, and executes some list of actions on matched packets, some notable fields are:
      » `idle_timeout`: Number of idle seconds before the flow entry is removed. Default: No idle timeout
      » `hard_timeout`: Number of seconds before the flow entry is removed. Default: No timeout
      » `actions`: List of actions to perform on matched one
      » `priority`: When using non-exact matches, this specifies the priority for overlapping matches
      » `buffer_id`: The ID of a buffer to apply the actions to
      » `in_port`: If using a buffer_id, it’s the associated port
      » `match`: An `ofp_match` object. Default: match all
• Sending OpenFlow messages with POX:

  • `ofp_flow_mod` OpenFlow message:

    – Example: Create a flow_mod that sends packets from port 3 to port 4

    ```
    fmod = of.ofp_flow_mod()
    fmod.match.in_port = 3
    fmod.actions.append(of.ofp_action_output(port = 4))
    ```
Mininet – Create Learning SW

• Parsing packets with the POX packet libraries
  – It is used to parse packets and make each protocol available to Python
  – It can also be used to build packets for sending
  – Libraries location:
    • pox/lib/packet/
  – Each protocol has a associated parsing file
  – Use dot notation to extract data: packet.src
  – See all members: print dir(packet)
Mininet – Create Router

• Create a static layer-3 switch
  – The router will be completely static with no BGP or OSPF, and you do not need to send or receive updates for route table
  – Each network node will have a configured subnet, the node acts as a switch if a packet is destined for a host within that subnet
  – If a packet is destined for known next hop IP address, it should modify the layer-2 destination and forward the packet to the correct port
• Create a static layer-3 switch
  – We will need different network topology

  There is an example custom topology we can utilize to build our own network at:
  ~/mininet/custom/topo-2sw-2host.py
• Create a static layer-3 switch
  – First, copy the example to a new file:
    $ cp ~/mininet/custom/topo-2sw-2host.py mytopology.py
  – Run the custom topology in mininet:
    $ sudo mn --custom mytopology.py --topo mytopo --mac
  – Then in the Mininet console, run:
    mininet> pingall
  – Now we can modify the topology file to match the topology we want to create
Example: Custom Topology

"""Custom topology example

Two directly connected switches plus a host for each switch:

    host --- switch --- switch --- host

Adding the 'topos' dict with a key/value pair to generate our newly defined
topology enables one to pass in '--topo=mytopo' from the command line.
"""

from mininet.topo import Topo

class MyTopo( Topo ):
    "Simple topology example."

def __init__( self ):
    "Create custom topo."

# 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() ) }
Mininet – Create Router

• Create a static layer-3 switch
  – Set up hosts:
    • Set up IP configuration on each virtual host to force each one to send to the gateway for destination IPs outside of their configured subnet
    • You will need to configure each host with a subnet, IP, gateway, and netmask
    • Do not assign IP addresses to the interface of switches s1 and s2, using OpenFlow if you need to handle traffic “to” or “from” the switch
Mininet – Create Router

• Create a static layer-3 switch
  – Using commands via Mininet CLI

mininet> h1 ifconfig h1-eth0 10.0.1.1/24
mininet> h1 route add default gw 10.0.1.1
mininet> h1 route -n
mininet> h2 ifconfig h2-eth0 10.0.1.2/24
mininet> h2 route add default gw 10.0.1.1
mininet> h2 route -n
mininet> h3 ifconfig h3-eth0 10.0.2.3/24
mininet> h3 route add default gw 10.0.2.1
mininet> h3 route -n
Mininet – Create Router

• Create a static layer-3 switch
  – Testing the router:
    • If the router work properly:
      – Try to send from 10.0.1.2 to an unknown address range
        like 10.89.0.1 should yield an ICMP destination
        unreachable message
      – Packets sent to host with a known address range
        should have their MAC dst field changed to that of the
        next-hop router
      – The router should be pingable, and should generate an
        ICMP echo reply
      – Packets sent to a host on the same subnet should be
        treated like before
Further Resources on Mininet

• Mininet in the Classroom (Computer Networks Course):
  https://cs.wmich.edu/~alfuqaha/spring14/cs5550/

• Multipath Routing:

• Select Group Type: