Workshop in Communication Networks

# Exercise 2 – Layer 3 – Software Defined Router

## General Guidelines:

Submission deadline is **Sunday, May 4, 23:55 moodle server time**
Submit your answers as TXT and Python files, packed into a single ZIP file.
Pack your files in a ZIP file named ex2-YourName1-YourName2.zip,
for example: ex2-LoisLane-ClarkKent.zip
Post the ZIP file in the submission page in course website

- No late submission will be accepted!
- Document your code. Place your names at the top of every source file, as well as in the TXT files.
- Make sure your files have EXACTLY the requested file names.
- We may give a bonus for exceptional works. However, do not make anything more complex than required.

**In this exercise we will add to our controller the ability to control SDN Layer 3 routers, in addition to Layer 2 switches. The controller will compute shortest paths using Dijkstra's Algorithm and set the routing tables of the routers accordingly.**
**The network and the weights of the links are configured in a configuration file that is loaded to the controller.**

**Important: You are not allowed to use any predefined POX modules in this exercise, except for the core modules that are already imported in the template file `of_router_imports.py`**

The grading for this exercise will be mainly automatic. If your code does not work as expected, the grade will be very low.

Your submission ZIP file should contain the following files:

- Code – named **of_router.py**
- If you make any change to **utils.py** from exercise 1 – add it to the ZIP as well
- A text file named **description.txt** that *briefly* describes how you implemented each part
- If you do the bonus in part 1: a text file named **bonus.txt** that *briefly* describes what you did.

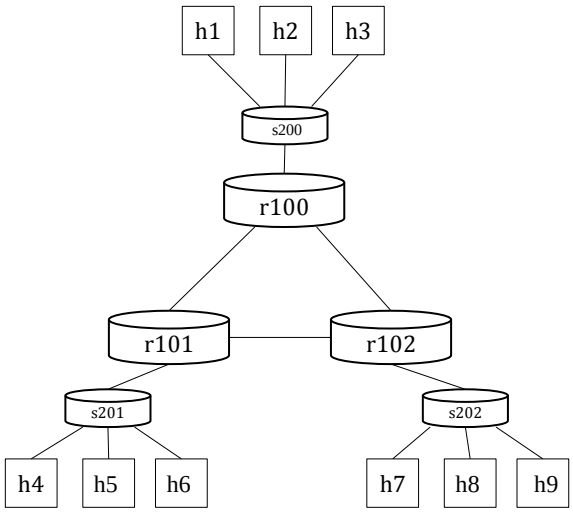
*Your code must include log messages for a small set of important events. See the complete list in Appendix A. This will help you track your controller, and will help us grade it.*

## Prerequisites

In order to complete this exercise you must have your learning switch code from Exercise 1 working.
***It is enough to have Part 1 of Exercise 1 as the basis for this exercise.***

Useful links for this exercise:
OSPF | Dijsktra's Algorithm | ARP | ICMP | Python Tuples

Assumed Topology
A sample topology is given in the attached topologies.py file, named **WanTopology**. It is described in the figure. Some of the OpenFlow



switches are named r1XX (where 1XX is also their DPID), and are supposed to act as Layer 3 routers. Other switches are names s2XX (where 2XX is also their DPID), and are supposed to act as Layer 2 switches (using your exercise 1 code). Your code should work with other topologies as well.
However, we assume that no loops exist in LANs (between Layer 2 switches). Loops may exist between Layer 3 routers, but they should be avoided by the shortest path mechanism of the routing algorithm and not by creating a spanning tree.

Running Mininet with Layer 3 Settings
The topology represents a wide area network with different addressing at each subnet. All subnet masks are 255.255.255.0 (/24 addresses). If a host sends a packet to an address outside of its subnet, it should destine it to the MAC address of its ***default gateway***. The IP address of the default gateway is set manually for each host by running a mininet script during mininet initialization.

To run mininet on the sample topology use the following command line:

```
sudo mn --custom ./topologies.py --topo WanTopology --switch ovsk
   --controller remote --pre ex2_cli_script --ipbase='10.0.0.0/24'
   --mac
```

**Part 1 – Represent a Routing Table** (10 points)
Create a new class named **RoutingTable**. The class represents a logical routing table that is stored in the controller for each router.
The class should have the following methods:

- __init__(self): constructor. Initializes fields, etc. No parameters but self.
- add(self, address, mask, destination): Adds an address with a specified subnet mask and destination port to the routing table. Parameters:
    - address: IP address (String of four integers in [0,255], such as "10.0.0.1")
    - mask: Subnet mask (String of four integers in [0,255], such as "255.255.255.0")
    - destination: Either a port number, or some other value, as you wish to have in the table.

  The method does not return any value. If an entry exists for the same *masked address*, then the old entry is replaced by the new entry.
- lookup(self, address): Looks up a given address in the table. Parameter address is a string of an IP address (as in add). If there is a corresponding entry in the table (with regard to address and subnet mask), then the destination of that entry is returned. Otherwise, the method returns None.
  The lookup process should be "dumb" – it should just iterate and find the corresponding entry, no smart logic is required.
  Bonus: Add smart logic to find the address in O(1) instead of O(number of entries).
- __str__(self): Returns a string representation of the routing table. This should be multi-line representation that should start with '\n' so that everything will be printed in new lines.

**Part 2 – Read the Configuration File** (10 points)
Create a new class named Network. This class should be a singleton and it should read the network configuration file (named *config*) once created. It should assume that the file is always under /home/mininet, and that it is always named *config*. The configuration file contains information on each router and on each link between two routers. The configuration file is a plain text file, structured as a set of blocks. Each block, which is a set of consecutive lines, either describes a router or a link. Blocks are separated with an empty line. See the attached *config* file.

Router Block

A block that describes a router starts with a line like:

```
router 100
```

which means that this block describes router r100 in the topology.
Then, the next line tells the number of ports in this router:

```
ports 3
```

which means that this router has three ports and details about these three ports is coming ahead.
Each port is represented as a "sub-block" with four lines (line order may change after the first one):

```
port 1
ip 10.0.0.1
mask 255.255.255.0
mac AA:00:00:00:00:01
```

This sub-block, which is not separated by an empty line as it is part of the router block, means that the port number is 1, the IP of the port is 10.0.0.1, its subnet mask is 255.255.255.0 and its MAC address is AA:00:00:00:00:01.

Link Block

A link block starts with a line that contains one word:

```
link
```

Then, the following three lines may appear (not necessarily in this order):

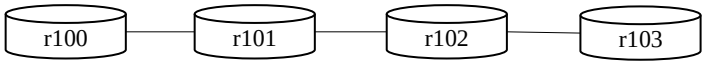
```
left 100,2
right 101,2
cost 1
```

These lines mean that this link connects router r100, port 2 with router r101, port 2, and that the cost of this link (in terms of link preference for shortest path computation) is 1. Cost can be any positive number.

**Part 3 – Compute Shortest Path**

A.

Add a method called **compute_dijkstra(self, src_router)** to class Network.
The method computes the shortest path from src_router (a DPID) to all other routers in the network and returns a dictionary (i.e. {}) that maps each router (DPID) to the corresponding next hop router (DPID) that have to be taken in order to reach the destination router.
For example, in the following network:




Calling self.compute_dijkstra(100) will return { 101: 101, 102: 101, 103: 101 } as r101 is the next hop from r100 to all other routers.
You may use the **priority_dict** class from priority_dict.py for this implementation.

B.

Add a method called **compute_ospf(self)** to class Network.
The method returns a next hop router dictionary for all routers: given a router (DPID), the dictionary returns the next hop dictionary computed by compute_dijkstra for this DPID.
(It simply means a dictionary of dictionaries for all routers)

Example with the network from Part 3.A.:

```
res = self.compute_ospf()
print res[100][102]
              # prints 101 as in order to get from r100 to r102 the next hop is r101
```

**Part 4 – Create Routing Tables**

Add a method called **get_routing_table(self, router)** to class Network.
This method creates and returns a complete routing table (using your RoutingTable object), for all existing subnets in the network, for a given router. This includes subnets of the given router that should be destined at the correct port of the router.
Specifically, it should, for each subnet (IP+mask) in the entire network:

1. Find the destination router that corresponds to this subnet.
2. If destination router is this router, add an entry to the table with the right output port for this subnet (and continue to next subnet).
3. Find next hop for destination router using OSPF results. If no such next hop, continue to next subnet (this means that the destination part of the network is unreachable from this router).
4. Find the port that leads to the next hop router and add an entry to the table with this port for this subnet (and continue to next subnet).

**Part 5 – Handle ARP**

A router uses ARP to determine the MAC address of destinations in its own subnets. For example, when r100 receives a packet for h1 it should send it on the port the goes to s200, with the MAC address of h1 as Layer 2 destination address. For that, it needs to know the MAC address of hosts it sends packets to in this subnet. In addition, when hosts send packets outside of their own subnet, they send them to their predefined *default gateway* (as we define in the mininet preload script). These packets are sent in Layer 2 and thus the hosts should find the MAC address of their default gateway, so the router should reply to ARP requests for its own IPs.

A. ARP Cache and Pending Request Management

When a router receives a packet it should forward to its own subnet, it first looks up its Layer 2 destination MAC address by sending an ARP request to this network. Meanwhile, the packet is buffered until an ARP reply arrives.

Step 1:
In Tutorial.__init__, initialize the following fields to your Tutorial class:
self.arp_cache = {}
self.waiting_arp_requests = {}
These fields are dictionaries that map an IP address (a string) to a tuple:
**waiting_arp_requests** maps an IP address to a 3-tuple of (packet, packet-in, timestamp) where packet and packet_in are the data packet and its enclosing openflow message that are to be buffered until an ARP reply arrives, and timestamp is the time the corresponding ARP request was sent.
**arp_cache** maps an IP address to a 2-tuple of (mac_address, timestamp) where mac_address is the MAC address associated with the IP address according to the ARP reply, and timestamp is the time the ARP reply was received.

Step 2:
Create the method **clean_arp_cache(self)** in class Tutorial:
This method should clean both **waiting_arp_requests** and **arp_cache** from entries whose timestamp is too old. Specifically:

- If an ARP request was not answered in more than 3 seconds, it should be deleted from **waiting_arp_requests**. The arp_cache should be updated such that the IP address in the request is unreachable (mapped to None instead of to a valid MAC address). The buffered packet (and packet_in) should be handled again by calling **self.handle_ip(…)** (implemented later) as if they came only now (since the ARP cache knows that destination is unreachable, handle_ip will issue *destination host unreachable* ICMP message).

- If an entry was in the ARP cache for more than 3600 seconds, or if this entry corresponds to an unreachable host and it has been there for more than 5 seconds, this entry should be removed from **arp_cache**.

Finally, add a timer in Tutorial.__init__ for each instance that handles a router (not for switch instances) that calls **self.clean_arp_cache()** every 3 seconds.

B. Handle Incoming ARP requests and responses

Add the method **handle_arp(self, packet, packet_in)** to class Tutorial:
This method will be called when an incoming packet is an ARP packet. Parameter **packet** is the Layer 2 packet that encloses the ARP packet and **packet_in** is the enclosing openflow message. **packet.payload** is the ARP packet. It has the following fields:

- opcode: the ARP opcode of this packet (arp.REQUEST, arp.REPLY, and more)
- protosrc: the source IP this ARP message was sent from (IPAddr object)
- protodst: the destination IP this ARP message was sent to (IPAddr object)
- hwsrc: the source MAC address this ARP message was sent from (EthAddr object)
- hwdst: the destination MAC address this ARP message was sent to (EthAddr object)

You can also build and send your own ARP packet as follows:

```
my_arp = arp()
// set my_arp fields as described above…
my_ether = ethernet()
my_ether.type = ethernet.ARP_TYPE
my_ether.dst = …destination MAC address (EthAddr object)…
my_ether.src = …source MAC address (EthAddr object)…
my_ether.payload = my_arp
// send my_ether to switch with a packet_out openflow message
```

You should check the ARP type of the incoming packet and act accordingly:

- If it is an ARP request, send an ARP reply
  An ARP request to the router will include the IP address of the port the request was sent to. The router may also receive ARP requests to other hosts/routers in the network and it should ignore them. You should check that the destination IP address in the ARP request matches the IP of the port the request arrived at (packet_in.in_port) using your Network class. If it matches, send an ARP reply with this port virtual MAC address ***as defined in the configuration file*** (and not as assigned by mininet).
- If it is an ARP reply, cache it and check **waiting_arp_requests**:
  If this message is an ARP reply that was sent to the MAC address that we set to this port in the configuration file, then it is a reply to an ARP request this router sent on this port (verify that using your Network class). In this case, check whether there is a waiting ARP request in **waiting_arp_requests** for the source IP address of this reply. If not, then the request was already timed out and the response should be ignored. Otherwise:
  1. Cache the reply in **arp_cache**
  2. Remove the pending request from **waiting_arp_requests**
  3. Call **self.handle_ip(…)** to handle the pending packet that was buffered in **waiting_arp_requests**.

C. Send ARP requests:

Add the method **send_arp_request(self, req_ip, port)** to class Tutorial:
(You may change the parameters of this method if you want to)
Create an ARP request message similarly to the ARP response creation and send it to the specified port. Note that ARP requests are broadcasted on Layer 2, by sending them to MAC address FF:FF:FF:FF:FF:FF.

**Part 6 – Send ICMP Messages**

Routers send ICMP messages in several cases. Most noted is the ICMP ECHO mechanism that is used for Ping, but ICMP is also used to report network-level errors. Use the code attached in **send_icmp.py** file to create a method that sends ICMP messages with the given type and code.

**Part 7 – Handle IP packets**

Add the method **handle_ip(self, packet, packet_in)** to class Tutorial:
This method will be called when an incoming packet is an IP packet. Parameter **packet** is the Layer 2 packet that encloses the IP packet and **packet_in** is the enclosing openflow message.
**packet.payload** is the IP packet. It has the following fields:

- srcip: Sorce IP address (IPAddr object)
- dstip: Destination IP address (IPAddr object)
- protocol: Layer 3 protocol (such as ipv4.TCP_PROTOCOL, ipv4.UDP_PROTOCOL, ipv4.ICMP_PROTOCOL)
- ttl: the time-to-live value of the IP packet (int)

A note on TTL: a router should decrement the TTL value of each packet it forwards. If the TTL is decremented to 0, then the packet should be dropped and an ICMP TTL Expired message should be sent to sender. Due to OpenFlow 1.0 limitations, we will not be able to completely implement this behavior, but we will try to do our best as described below.

This method should:

A. Compute the routing table of the router, if not computed already, by calling Network().get_routing_table(…).
B. Find the destination port for the destination IP address from the routing table.
C. If the destination port is not None, it is not the same port the packet came on (packet_in.in_port), and TTL is 2 or more, then:
    a. If the destination IP is in the subnet of the destination port (for example, r100 receives a packet to h1), then we should find the MAC address of the destination by checking **arp_cache**:
        i. If the destination IP address is in arp_cache and it is mapped to a valid MAC address, then let **dest_mac** be this MAC address (we will use it soon), and continue to step b.
        ii. If the destination IP address is in arp_cache but it is mapped to None, it means that the destination host is unreachable (recall Part 5.A. Step 2). The controller should initiate an ICMP message with type=3, code=1 (Destination Host Unreachable) to the sender on the ingress port. Do not continue to step b in this case.
        iii. If the destination IP is not in arp_cache at all, then the packet (and its corresponding packet_it) should be added to waiting_arp_requests , and an ARP request for this address should be initiated to the destination port. Do not continue to step b in this case.
    b. Create a flow_mod message to the router, with a match for the following fields:
    dl_type = ethernet.IP_TYPE (match packet of Layer 2 type = IP)
    in_port = packet_in.in_port
    nw_dst = …destination IP address (IPAddr object) …
    Attach the current buffer_id and raw data to the flow_mod as we did for switches.
    Create the following actions and add them to the flow_mod action list:
        - If **dest_mac** is defined (from step a.i.), then add an action to set data layer (Layer 2) destination address to **dest_mac** to the list of actions.
        Example for creating such an action:
        `action_set_dl_dst = of.ofp_action_dl_addr.set_dst(EthAddr(dest_mac))`
        - Add an action to decrement TTL by 1:
        `action_dec_ttl = nx.nx_action_dec_ttl()`

(This is an extension to OpenFlow 1.0 added by Nicira to POX and Open vSwitch. It may not be supported by other switches!)
- Output the packets to the destination port (you know how to create this type of actions already).

Send the flow_mod message to the router.

D. If the destination port is the same port the packet came on (packet_in.in_port), and TTL is 2 or more, then this message was either sent to another host/router in this subnet, or was sent to the router itself:
   a. If the destination IP address is the IP address of the port the packet came on (check with your Network class), the packet was indeed sent to the router itself. The router may only handle IP messages of type ICMP. Therefore, check whether the protocol of the IP packet is ipv4.ICMP_PROTOCOL:
      i. If it is, and it is an ICMP ECHO Request (`packet.payload.payload.type == 8`), then send an ICMP reply.
      ii. If the packet is of another type of ICMP, ignore it, as we do not handle other ICMP types in this exercise.
      iii. If the packet is not an ICMP type, send an ICMP Unreachable Port error message (type=3, code=3), as this port is unreachable for such IP messages.
   b. If the destination IP address is not the IP address of the port the packet came on, then this packet, and any further messages to this destination IP address, should be dropped. Create a flow_mod message just as you did in step C.b. but with no actions at all (which means – drop) and send it to the router.

E. If TTL is less than 2, packet should be dropped and an ICMP TTL Expired (type=11, code=0) message should be sent to the source IP address.
   Note: although we would like this to always happen when TTL is less than 2, it will not happen after we install flows to routers, as controller will not get these packets anymore. An OpenFlow switch cannot match on the TTL field and cannot initiate ICMP messages, so we cannot install a rule with higher priority to catch this case. For this reason, traceroute cannot work properly in our network.

F. If the destination port is None (unknown), the destination network is unreachable from this router. Send an ICMP Destination Network Unreachable (type=3, code=0) message to the source IP address.

**Part 8 – Act Like Router**

A.

Add the method **act_like_router(self, packet, packet_in)** to class Tutorial:
This method will be called instead of act_like_switch(…) when the controlled OpenFlow switch is intended to be a Layer 3 router.
This message should inspect the Layer 2 type of the packet it received:

- If packet.type == ethernet.IP_TYPE:
  - Call self.handle_ip(…) from Part 7 to handle this packet
- If packet.type == ethernet.ARP_TYPE:
  - Call self.handle_arp(…) from Part 5 to handle this packet
- Otherwise:
  - Ignore this packet

B.

Change method _handle_PacketIn(…) in class Tutorial to call act_like_router(…) or act_like_switch(…) according to the DPID of the OpenFlow switch: DPID between 100 to 199 is a Layer 3 router, while DPID between 200 and 299 is a Layer 2 switch (other DPIDs are not expected and should be logged and ignored).

**Part 9 – Testing**

First of all, test that your switch behavior was not damaged due to your changes by testing connectivity between hosts in the same subnet (e.g. h1, h2, and h3).
Then, test the connectivity between hosts on different subnets. Use your logging to see how controller installs rules and to find any possible Layer 3 forwarding problem. In the sample topology, all link weights are 1 and thus you should expect direct routing. However, if you change the costs in the configuration files you should be able to see how routes change accordingly.

**Make sure your controller has the required logging as listed in Appendix A!**

**Make sure you document your work in a separate file – description.txt, and attach it to your submission!**

**Appendix A – Required Logging**

The following events must print some organized log messages in the DEBUG log level (using `log.debug(`*`message`*`)`):

For switches:

- All required logging from exercise 1

For each router (always state the router name or DPID):

- Sending ARP request (state destination IP address and the port you send it on)
- Sending ARP reply (state destination MAC address and the port you send it on)
- Received an ARP reply (state source MAC address, source IP address and input port)
- Sending an ICMP message (state type, code, destination IP address, output port)
- IP packet received (state source IP, destination IP, TTL)
- Destination host for IP packet is unreachable
- Destination host MAC address is unknown and an ARP request was sent to find it
- A flow record is set on a router (state matching fields and values, actions and values), including *Drop* rules
- ICMP ECHO Request received (state source IP address)
- Destination port for IP packet is unreachable (when non-ICMP packet is sent to router)
- TTL exceeded
- Destination network is unreachable