


Coursework: OpenFlow Router

In this coursework you will be expected to produce a Ryu-based OpenFlow controller that provides static routing and firewalling functionality.


 You are expected to have completed the recommended tutorials from the SCC365 GitHub prior to starting this coursework:

- [Mininet Tutorial](#)
- [Ryu Tutorial](#)

 You should also refer to the [network testing guide](#) throughout

Important Information

- 🕒 **Deadline:** Friday Week 20 (15:00)
 - ⚖️ **Weight:** 100% of the coursework grade
 - 🐍 **Language:** Python
-

 Throughout much of this spec you will find that many key terms will have hyperlinks to external sites. Please use these to ensure you fully understand the spec!


Overview

In the [Ryu Tutorial](#), you took an OpenFlow controller that provided the functionality of a simple [hub](#) and extended it to provide the functionality of a [learning switch](#) instead. In doing this, you had to implement logic that parsed packets to obtain information from their ethernet headers. With this information, specific OpenFlow matches and actions could be added to the device's [flow table](#).

For this coursework, you are expected to take the template controller given (a layer 2 learning switch with some extended helper functions and comments) and extend the logic to include static routing and firewalling.


Compared to the tutorial, the controller that should be implemented for this coursework differs in 2 key ways:

- Routing/firewalling makes changes to the packets as they flow through the device whereas switching does not (often referred to as network device transparency).
- Routing/firewalling uses data in higher packet layers to drive some decisions such as layers 3 and 4.

 **Note:** IP refers to IPv4 here. You can ignore IPv6 for this coursework!

What is a router?

A router's core functionality is to allow packets to be moved between [IP](#) networks/subnets. In the tutorial, the virtual network had IP devices only in the subnet `10.0.0.0/8` so all packets between these devices could be switched rather than routed.

 If you need to, refamiliarize yourself with subnets and cidr notation! Also, if you just want to quickly check what all the possible addresses are in a subnet, there is no shame in using a [subnet calculator](#).

As you might expect, understanding the concept of packet routing is vital to this coursework. This topic will be covered interactively in a lab session by TAs, and there are countless resources online that can help. Many of you will be familiar with the term "router" being the device in your house that does all the "routing", but in reality that device will likely do a lot more than just routing. Often those devices perform switching, firewalling, routing, and NATing (along with more depending on the device and the ISP). This coursework focuses on the actual task of **routing** (and does include firewalling in the later tasks).

Routing Example

A very basic representation of routing logic is given in here:

Assuming a pipeline that goes `h1 -> r1 -> r2 -> h2`. Where the `rX` devices are static routers and the `hX` devices are hosts.

1. To start, `h1` sends a packet (of `eth_type IP`) to `h2`. In the topology described, the packet is first processed by `r1`. To confirm the packet has been constructed correctly, `r1` should ensure that the destination MAC address of the packet matches that of the receiving interface/port on `r1`. The information needed for this comes from the packet's ethernet header and the routing device's [Interfaces Table](#). If the address does not match, `r1` will drop the packet.
2. Next, `r1` must check that it can route the packet. To do so it needs to check the destination IP address of the packet (via inspecting the IPv4 header). It compares the destination IP against the [Routing Table](#). If the value exists there, or the IP fits within a specified subnet, it can route the packet. It also uses this lookup to get the IP address of the next hop and the output port. If the destination IP/subnet is not in the Routing Table, the appropriate error ICMP packet is sent to the packet's source and the packet is dropped.
3. With the next hop IP found from the [Routing Table](#), it then can find the MAC address of the next hop using the [ARP Table](#). With the next hop MAC address found, it can then create an action to change the packet's destination MAC address to that of the one found when it sends the packet out.
4. Additionally, it should then create an action that changes the source MAC address of the packet when it sends it out to the MAC address associated with the output port. This is the output port found when performing the [Routing Table](#) lookup in step 2. To get the MAC address of the output port, another lookup of the [Interfaces Table](#) is used.
5. Yet another action should be created to decrease the packet's time to live (TTL) each time it sends the packet out.
6. With these actions in place, when executed the packet is then sent to the next hop: `r2` via the specified output port with the appropriate MAC address fields being modified in the ethernet header and the TTL field of the IP header being decremented by 1.
7. Next, `r2` follows the same steps as in `r1` however, when looking up the [Routing Table](#), it sees that the next hop is "*direct*". This means that the next hop is actually the packet's destination. All the other steps are repeated, however, the next hop IP address is the same as IP address found in the packet's IPv4 header.
8. The packet will then be forwarded to `h2` via the discovered output port 🎉

The Routing Table

The Routing Table contains entries with 3 key fields:

- **The target destination IP:** The IP of the final recipient

- **The next hop:** The IP address of the next hop, before final destination (or null if the destination is the same as the next hop)
- **The next hop port:** The port of the next hop (router), which the destination is routed via

An example pseudo entry might be:

```
The destination IP found in the packet "10.37.0.101" is not accessible
directly, however the next hop to this IP is "148.88.65.30" that is
found though Port 5
```

The ARP Table

The ARP table is normally populated using ARP requests and replies. This is effectively a table mapping IP addresses to MAC addresses. An example pseudo-entry might look like:

```
The host with the IP address of 192.168.1.17 has the MAC address of
09:fa:c3:6b:d1:82
```

The router uses this to find the MAC address of the next hop, given the Routing table only gives the IP address.

The Interfaces Table

There is one other data structure needed to perform static routing. Each device needs an Interfaces Table. This maps each of the device's ports to a MAC address and IP address. An example pseudo entry is the following:

```
Port 3 has the MAC address 09:fa:c3:6b:d1:82 and the IPv4 address
192.168.1.1
```

Static Routing

A router at boot-time knows only how to reach its immediate neighbors. Therefore, much like in a learning switch, it must create a sort of map, by learning which subnets exist in the network and how to reach them. Unlike the switch, it must also process layer 3 headers (mainly IP) and must have some idea of a next-hop router to reach subnets not directly connected. These discovery steps use protocols such as ARP, RIP and iBGP; which populate two information tables: the *ARP Table* and the *Routing Table*. A static router is given these tables pre-populated so that discovery messages are not needed.

Preparation

To develop the controller for this coursework, you are given some template files:

- The **router.py** template file does very little as is. It contains some empty functions and some utility functions that might be useful. It does however act as a layer 2 learning switch. You should add your code to this file to complete the coursework. You are free to edit this however you feel is necessary (e.g. you can delete lines etc...).
- The **topology.py** is a ready-made Mininet topology that contains multiple IP subnets and virtual OpenFlow devices that look for an external controller to provide routing functionality.
- Static data files:
 - **arp.json** file contains the JSON formatted ARP Tables that work with the provided topology.
 - **routing.json** file contains the JSON formatted Routing Tables that work with the provided topology.

- **interfaces.json** file contains the JSON formatted Interfaces Tables that work with the provided topology.
- **rules.json** file contains the JSON formatted firewall rules required for the final task of the coursework.

You should use these files to create your router controller. Although, you are free to change the provided code that sits within the router.py file, but do keep the file names the same and do not modify the contents of the other files.

The Topology

It is important to have a good understanding of the topology before you start this coursework! The topology we expect you to use is complicated for good reason, however it has been made for you and is described below in the tables and diagrams.



Note: This topology does not use `mn`, rather it uses the Mininet Python API.

If you are not comfortable using the Mininet network emulator, ensure you have completed the [Mininet Tutorial](#) and refer to the [network testing guide](#) for more information on how to use the command line tools needed to test the virtual network and in-turn, your controller.

Running the Network

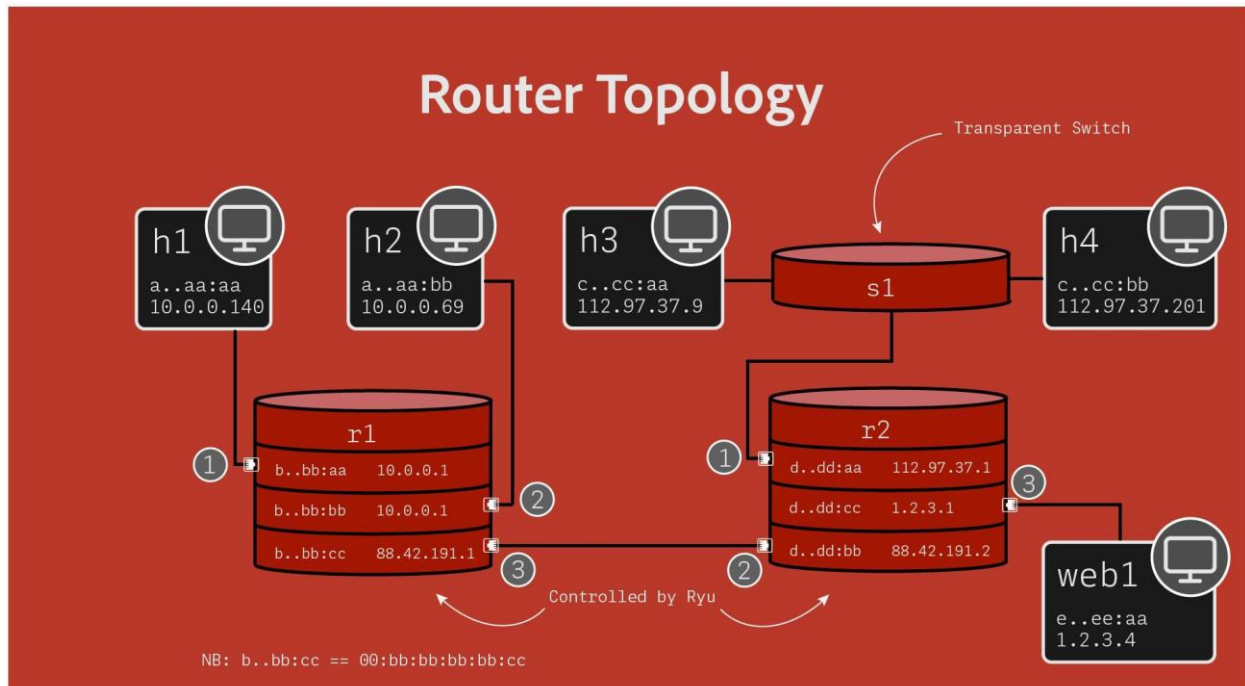
In the tutorials, the networks were somewhat simple thus they could use the Mininet command line tool `mn` coupled with a brief Python description. Due to the complexity of the topology, this uses the Mininet Python API to describe the network. Still, it is recommended to read through the Python file `topology.py` to try and get an understanding of the topology you will be testing your work on. This section of the coursework specification does also explain the topology.

To run the topology, the following command can be run from a terminal where the current working directory contains the `topology.py` file.

```
sudo python3 topology.py
```

As the topology is for a router coursework, it contains different subnets where previously all the hosts have had `10.0.0.0/8` IP addresses. This topology overall is somewhat more complex than the ones you have created for tutorials. It is recommended that you read the following information about the topology and read through the `topology.py` file.

Understanding the Topology



- **Controllers:** This topology uses 2 separate controllers, where switches are connected to `cs1` (providing default learning switch functionality) and "routers" are connected to `cr1` (a remote controller to be found at `127.0.0.1:6633`).
- **Datapaths:** `s1`, `r1`, `r2` are all datapaths that use Open vSwitch, however the separation in functionality comes only from the controller logic.
- **Subnets:** When working with routing, it can be easier to think of ports (interfaces) rather than nodes (hosts/switches). This is because a port typically exists only in one subnet, whereas a node can be a part of many. For example, `r2` has 3 ports: each port is in only one subnet, but all 3 ports are in different subnets.
- **CIDR Notation:** From 2nd year you should be familiar with CIDR notation (normally looks something like `10.0.0.0/8` to say the network is `10.0.0.0` with a netmask of `255.0.0.0`). The data associated with this topology requires the use of that knowledge.

Subnet/Interface Table

Subnet	Interfaces
10.0.0.0/24	h1-eth0, h2-eth0, r1-eth1, r1-eth2
88.42.191.0/24	r1-eth3, r2-eth2
112.97.37.0/24	h3-eth0, h4-eth0, r2-eth1 (s1 exists transparently in here)
1.2.3.0/24	web1-eth0, r2-eth3

Interface to MAC Address Table


Interfaces	MAC Addresses
h1-eth0	00:aa:aa:aa:aa:aa
h2-eth0	00:aa:aa:aa:aa:bb
h3-eth0	00:cc:cc:cc:cc:aa
h4-eth0	00:cc:cc:cc:cc:bb
r1-eth1	00:bb:bb:bb:bb:aa

Interfaces	MAC Addresses
r1-eth2	00:bb:bb:bb:bb:bb
r1-eth3	00:bb:bb:bb:bb:cc
r2-eth1	00:dd:dd:dd:dd:aa
r2-eth2	00:dd:dd:dd:dd:bb
r2-eth3	00:dd:dd:dd:dd:cc
web1-eth0	00:ee:ee:ee:ee:aa

To help you test your controller, each node is running a web server on TCP port 80 that will respond to HTTP GET request with a response that contains the perceived IP address of the client. For example, if h3 calls h4 (**mininet>h3 curl h4**), the IP address that h4 thinks is the source of the call will be returned in `json` format.

Tasks

This coursework is split into 4 tasks, however not every task is dependent on the completion of all the previous tasks. So if you get stuck on a task, it is possible to move on. You *should* try and spread these tasks out across the time that you have to complete this coursework.


 **From the TAs:** We know every course asks you to start courseworks well in advance of deadlines, but this is a fairly large coursework covering many new concepts. The only times you can be assured TAs will be answering questions is during lab hours!

Task 1: Routing Fundamentals

[30 marks]

You should implement the controller logic that will allow the OpenFlow datapaths (switches) to route packets between subnets. Your implementation is expected to perform the required packet header validations. For example, a router should not forward a packet if the packet did not have the destination MAC of a port on the router.

This task can be implemented in many ways, some efficient, some not so much. You will be marked not only on the correctness of this task, but also for the efficiency and quality of your solution. A good start point to tackle this efficiently, would be to install flow-mods and think how you can keep the installed number of flows at a minimum by using wildcards

 **Tip:** Masked matches are supported in OpenFlow and if used, can make your solution far more optimal than going ahead without them.

Task 2: ICMP

[30 marks]

Routers are non-transparent network devices! And one of the protocols they typically support is ICMP Echo (ping). The devices connected to your controller should be able to appropriately handle ICMP Echo Requests by sending back a valid ICMP response.

Not only should a router use ICMP for echo, but also for errors. Below is a list of ICMP type/code values that your controller's routers should use appropriately: 0/0, 3/0, 3/1, 3/6, 3/7.

For example, if a packet enters a router that is destined for an IP within a subnet unknown to it, it should respond with the ICMP packet with type/code 3/6.

In order for applications to properly recognise the response from a router, you will need to construct your packets carefully and spent some time to understand what is the payload for each ICMP message type. Guidance for this is available in [RFC792](#). (hint: the latter part of this task is the hard bit!).

Task 3: Firewall

[30 marks]

As mentioned in the [overview](#), a typical home "router" does a lot more than just routing. One of the most common extra features is firewalling. Your controller should ingest a JSON formatted set of firewall rules and enforce the parsed rules in the packet forwarding logic.

Rule format

The rules JSON when parsed into a Python variable, it will likely be a dictionary of the same structure file is structured like so:

Outer Structure

The outer structure is an object where each key is a 16 char datapath ID (e.g. 0000000000000002) and each has a key that is an array of **inner objects**. The **inner objects** in the array should only be applied to the datapath indicated by the **outer structure**'s key.

```
{
  "datapath_id": [
    <inner objects>
  ]
}
```

Inner Object

The inner object could be simply called a rule. This describes what packets should be dropped in the form of a set of packet headers.

```
{
  "id": <number>,
  "description": "<short description of the rule>",
  "allow": <true | false>,
  "priority": <priority of the rule>,
  "match": {
    "eth_src": "<src mac address>",
    "eth_dst": "<src dst address>",
    "ethertype": "<layer 3 protocol hex code>",
    ...
  }
  ...
}
```


Logic

A firewall system is not only a ruleset, but by the logic that implements them. For this task, your firewall should implement the rules using the logic given here. Importantly, the firewall implementation is expected to **block all traffic by default**. Even if your routing logic works as outlined

in Task 1, by adding in your firewall logic, your router should not forward any packets unless a given rule explicitly allows for it.

For any given rule:

- The `id` and `description` fields are simply there for human readability (for example, discussing the logic of a given rule with TAs).
- The `match` field contains OpenFlow match fields that, if a packet on the related datapath matches, should be impacted by the rule. Take care when using these fields. OpenFlow demands that packet match fields be built from layer 2 up. For example, to have fields from an IPv4 header in the match (such as `ipv4_src`), the match must also specify that the packet has an IP header by giving the `ethertype` field to be IPv4.
- The `allow` field specifies how a matching packet should be handled. If `true`, the packet should be routed as normal. Otherwise, the packet should be dropped. When allowing a packet, the logic implemented in [task 1](#) should apply.
- The `priority` field is an int that defines the order in which rules on a given datapath should be checked against each packet. Packets are checked against rules with the larger int value first, going down in priority until it matches some rule.

 **Tip:** The use of multiple [OpenFlow tables and GoTo](#) commands can make this task a little more straightforward. Although, grasping how OpenFlow tables can be used can be challenging itself.

We are not expecting you to memorize all [ethernet type](#) and [IP protocol](#) values for this! You can use the linked wiki pages to look those up.

Task 4: TTL

[10 marks]

To limit how long a packet can be in the network, the IPv4 header contains a TTL (Time To Live) field that is decremented by each router it passes through. If a router sees a packet with a TTL ≤ 1 , it should simply drop the packet. Paired with this is the TTL Exceeded ICMP response. When dropping a packet with an expired TTL, the TTL expired ICMP packet is sent to the IP that sent the dropped packet.

This functionality enables a tool called [traceroute](#). This tool allows a user to see the IP address of each hop a packet takes to a given destination. It does this by sending packets with incrementing TTLs (starting at 1) to the destination and displays the source IP of the TTL expired ICMP packet.


So, your router should decrement the TTL of all the packets it forwards. It should also ensure that it drops packets with an invalid TTL, also then sending the appropriate ICMP packet (11/0) back to the original sender. This is testable using traceroute. If your router implements this you will be able to see the IP of each router that the packet goes through on the way to a given destination. Again, this task will be marked heavily on the efficiency and elegance of your solution.

Submission and Marking

Submit your router controller to Moodle **before 3pm on Friday week 20**. This should be a zip or tar of your working directory, and must include at a minimum:

- `router.py`: your ryu based router controller
 - `requirements.txt`: your required python modules (if any)
 - `*.py`: any other python files you've split your controller across
-

Useful OpenFlow features

 **Note:** You are not required to use these features of Ryu/OpenFlow, however we feel that using them will help to make your controller code and logic much more concise!

Tables

OpenFlow flow mods are attached to tables, by default all flow mods are added to table 0, which is also the table of flow mods that are used to process packets when they enter the device.


Normally only table 0 is ever used, however if the `OFPIinstructionGotoTable` *instruction* is used instead of the `OFPActionOutput` *action* the packet will proceed to be processed by the flow mods contained in the table specified by the `OFPIinstructionGotoTable` action.

Note: You'll need to adapt any helper methods such that the `OFPIinstructionGotoTable` can be used as well as the `OFPIinstructionActions` instruction, which applies actions. Check the docs of `OFPFlowMod` for info.

By using tables, you'll be able to break up operations that are independent and should be applied together with other operations.

Cookies

Cookies are a method of tracking which flow mods have applied to the packet by attaching a numeric value. When a packet matches a flow mod, the cookie value of the flow mod is assigned to the packet. This value can be read in packet in events (`OFPPacketIn`) to take actions based on the cookie value.

 **Note:** As with tables, you'll have to adapt any helper methods to pass through a value for `cookie` to `OFPFlowMod`.