Coursework: NAT Router

In this coursework you will be expected to produce:

- A Ryu-based OpenFlow controller that performs NAT routing

> 📚 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 19 (4pm)
- ✅ **Marking**: Book a marking slot to take place in Week 20
- ⚖ **Weight**: 50% of the coursework grade
- 🗣 **Language**: Python
- ⏳ **Estimated Time**: 6+ hours

Overview

Up until now, you have been producing transparent network devices that operate within a single IP subnet, in this case 10.0.0.0/8. This coursework will have you create a Router controller that allows network data to flow between different IP subnets. Specifically, you will be creating a Static NAT (Network Address Translation) Router Controller.

In short, a router is a network device that forwards packets between subnets, commonly found with features such as discovery and NAT. However, this coursework will have you create a router controller that gives the OpenFlow devices the functionality of a static NAT router. For this, you are given a topology that use the Mininet python API and creates a virtual network with 4 subnets.

This coursework will have you produce just 1 component:

- A Ryu-based static NAT router controller

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

But what makes it static?

A router at boot-time knows only how to reach its immediate neighbors. Thus, 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 be aware of layer 3 fields (mainly IP) and must have some idea of where devices that are not directly connected are best reached through (hopped to). 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 provided with those datasets at launch, so it doesn't actually need to handle any discovery (hence the *static* bit).

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`

Worked Routing Example

Below is a step-by-step example of routing logic. It assumes 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 first node this packet goes into is r1. To confirm the packet has been constructed correctly, r1 should ensure that the destination MAC address of the packet is that of one of r1's interfaces. If the MAC is not attached to any of r1's interfaces, the packet is dropped.

2. Next, r1 needs to 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 ICMP packet is sent to the packet's source.

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 destination MAC address when it sends the packet out.

4. Additionally, it should then create an action that changes the source MAC address when it sends the packet out to the MAC address associated with the output port.

5. Additionally++, it should create an action to decrease the packets time to live for when it sends the packet out.

6. The packet is then sent to the next hop: r2 via the specified output port.
7. Next, r2 follows the same steps as in router 2 however, when looking up the Routing Table, it sees 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 IP address found in the packets IPv4 header.

8. The packet will then be forwarded to h2 via the found output port 🎉

Network Address Translation

The proliferation of networked devices has resulted in an exhaustion of IPv4 addresses (Just think how many devices you use in your every day life that use packet technologies to transmit information). As a result, in recent years we experience a lack of public IPv4 address. IPv6 will hopefully fix this with a larger pool of IP addresses, but its deployment is still under development.

To address this problem, local networks use a special network device, called NAT, to reduce the number of public IPv4 addresses. In principle, a NAT device has two network interfaces, one connected to the local network and one connected to the Internet, each configured with an IP address. A NAT device allows a set of internal hosts to communication with Internet hosts, using a single public IP address. The device uses header information to map the local IP address and port number into a unique source port number on the public IP interfaces. Header information must be manipulated appropriately so that ensure that the end-hosts of the connection are unaware of the address translation. NAT devices are widely used nowadays and every home router offers such functionality by default. A big challenge with NAT devices is the directionality that they introduce on connectivity; an external host is never allowed to instantiate a connection to an internal host.

Upon the creation of a new flow, the NAT allocates a unique port number on the outgoing interface and stores the mapping information in a local lookup structure, so that subsequent packets can be processed appropriately. As a result, a NAT device contains a 'flow table' which stores the information (or instructions) on how to update a private-to-public packet with the correct source IP & port, and on how to update a public-to-private packet with the correct destination IP & port.

💡 **Tip**: You can test NAT in action on your own home network (in most circumstances).

1. Check what IPv4 address has been assigned to your device by your home router (often something like 192.168.X.X)
2. Next, run the following command from your terminal `curl https://am.i.mullvad.net/ip`. The 2 IP address *should* be different, the first being your internal IP, the second being your router's public IP.
3. As far as the global internet is concerned, all the devices connected to your router have the IP from step 2 thanks to NAT.

Private to Public Translation

When a packet from the private side passes through the NAT, the router rewrites the source IP and port of the packet. The source IP is changed to the router's IP on the network the packet is to be emitted on. If the NAT knows which session the packet belongs to, then the source port is replaced with the chosen source port for that flow. Otherwise, a source port is allocated by the NAT and the flow is tracked. The source port remains constant for packets belonging to the same session.

As an example, suppose there is a NAT between a private network (`192.168.0.0/24`) and the internet, where the router has the address 192.168.0.1 on the private side, and `23.58.23.4` on the internet facing side:

some TCP and UDP packets on the private side may have the following fields:

Ether Type	IpProto	Source IP	Source Port	Destination IP	Destination Port
0x0800	0x06	192.168.0.4	54833	8.8.8.8	80
0x0800	0x11	192.168.0.4	54833	8.8.8.9	53
0x0800	0x06	192.168.0.5	21344	8.8.8.8	80

After passing through the NAT into the internet, they would have the following fields:

Ether Type	IpProto	Source IP	Source Port	Destination IP	Destination Port
0x0800	0x06	23.58.23.4	34324	8.8.8.8	80
0x0800	0x11	23.58.23.4	52732	8.8.8.9	53
0x0800	0x06	23.58.23.4	61234	8.8.8.8	80

Public to Private Translation

When a packet from the public side enters the NAT, the NAT checks to see if it knows about the session the packet belongs to. If no session can be found, the packet should not be modified by the NAT. Alternatively, the destination IP and port should be replaced with the correct values such that the packet makes its way to the device the session belongs to.

As an example, suppose there is a NAT between a private network (`192.168.0.0/24`) and the internet, where the router has the address `192.168.0.1` on the private side, and `23.58.23.4` on the internet facing side:

some TCP and UDP packets on the public side may have the following fields:

Ether Type	IpProto	Source IP	Source Port	Destination IP	Destination Port
0x0800	0x06	8.8.8.8	80	23.58.23.4	34324
0x0800	0x11	8.8.8.9	53	23.58.23.4	52732
0x0800	0x06	8.8.8.8	80	23.58.23.4	61234

After passing through the NAT into the private network, they would have the following fields:

Ether Type	IpProto	Source IP	Source Port	Destination IP	Destination Port
0x0800	0x06	8.8.8.8	80	192.168.0.4	54833
0x0800	0x11	8.8.8.9	53	192.168.0.4	54833
0x0800	0x06	8.8.8.8	80	192.168.0.5	21344

Keeping track of flows

To perform its job, a NAT needs to keep track of ongoing sessions for TCP and UDP sessions, this can be done by keeping track of the five-tuple of expected public->private and private->public packets. (*reminder: A five-tuple is a tuple composed of: ip protocol, source ip, destination ip, source transport port, destination transport port*).

An example flow table storing the information required to perform NATting of the packets shown above would be:

IpProto	Private IP	Inside Local	Remote Port	Destination IP	Outside Local
0x06	192.168.0.4	54833	80	8.8.8.8	34324
0x11	192.168.0.4	54833	53	8.8.8.9	52732
0x06	192.168.0.5	21344	80	8.8.8.8	61234

> 📝 **Note**: In the circumstance of an SDN controller submitting flow mods to a device, the NAT rules should be realized by using flow mods. Having every packet sent to the controller to look up information from a NAT table would be very slow.

Tasks

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

Provided Files

There are four provided files on Moodle. 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.

- The `router.py` template file does very little as it. It contains some empty functions and some utility functions that might be useful.
- The `arp.json` file contains the ARP tables that work with the provided topology.
- The `routing.json` file contains the routing tables that work with the provided topology.
- The `topology.py` file contains a 4 subnet Mininet topology that is described further below.

The Topology

The topology has been created for you this time, but it is still important to have a good understanding of it. The topology file is contained within `nat-router.tar`, an archive that can be downloaded from Moodle.

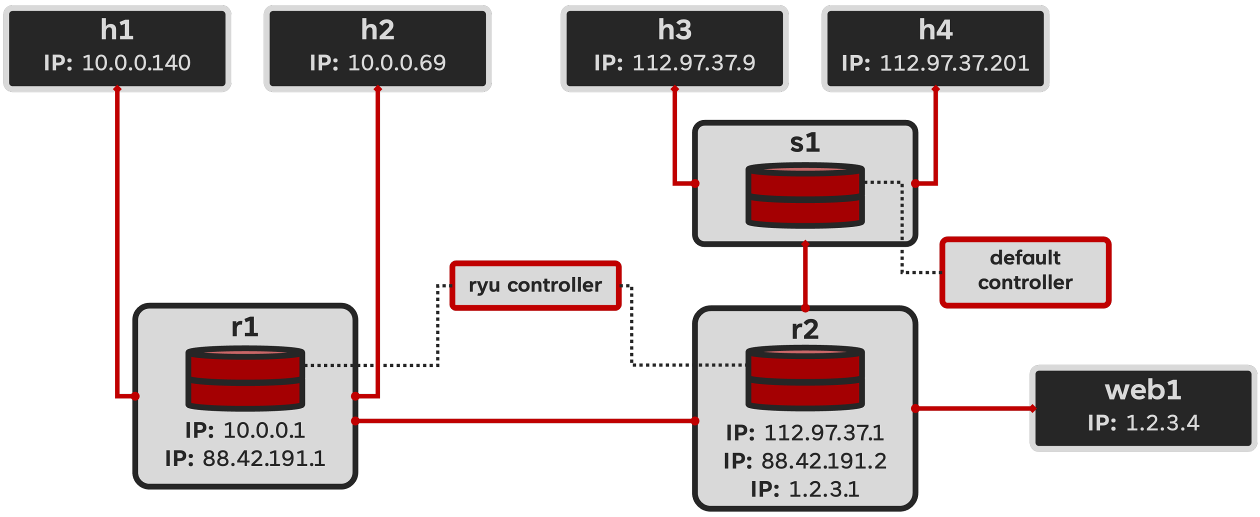
> 📝 **Note**: This topology does not use `mn`, rather it uses the Mininet Python API.

To start the topology:

```
sudo python3 topology.py
```

As the topology is for a router coursework it contains different subnets, whereas previously all the hosts have had `10.0.0.X` IP addresses. This topology overall is somewhat more complex than the ones you have created. 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	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`

Also to help test your controller, each node is running a web server on TCP port 80 that will return the perceived IP address of the caller. 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.

Task 1: Interface tables

[**15 marks**]

For the most part, the static data needed to build a suitable static router controller for the given topology is complete. It does not however contain the Interface Tables. With a combination of OpenFlow messages (port description requests) sent to datapaths and the provided ARP data, the interface tables can be built by the controller.

Each table entry should have 3 fields:

- ***port*** - the port number.
- ***hw*** - the MAC address of that given port.
- ***IP*** - the IPv4 address associated with that port.

Task 2: Routing

[40 marks]

You should provide the datapaths that register with the controller the ability to route packets between subnets. You should also be adding the appropriate packet 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 routing should use the data loaded in Part 1.

This task can be implemented in many ways, some efficient, some not so much. You will be marked not only on the function of this task, but also the efficiency and quality of your solution. A good start to tackling this in an efficient way would be to install flow-mods and from there, look at what you are matching on (hint: masked matches can exist!).

Task 3: ICMP

[20 marks]

Routers are non-transparent network devices! And one of the features 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 correctly built 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, 8/0, 11/0.

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.

Task 4: NAT

[25 marks]

For this final task of the coursework you must implement a NAT functionality in your router code. If you inspect the `routing.json` file, you will notice that each route entry contains a boolean field, named **nat**. Your implementation should use this field to decide if NAT functionality should be applied to traffic for specific subnets. Your implementation must be able to perform NAT on TCP and UDP traffic, while all other protocols can be ignored by the program. In addition, your implementation should allow connection to be established only when the first packet is sent from internal host.

Once a packet is received from an internal host to a NATted network subnet, your implementation should select a unique port number on the interface through which the packet will be transmitted. The source IP and port number should be modified accordingly. The selection of the next hop should be the same as in previous tasks. Your implementation should create the required flow mods to ensure that the header fields of the packets flowing on both directions are modified accordingly to ensure that both end-point are agnostic to the NAT operation and packets are forwarded successfully.

> 💡 **Hint**: In order to simplify the logic of your code, we suggest you to install all the required flow mods to process traffic on both directions when a new outgoing connection is created on the NAT. It is safe to assume that the topology and the next hop will not change during the operation of your NAT.

Submission and Marking

Submit your router controller to Moodle **before 4pm on Friday week 19**. This should be a zip or tar of all your files:

- `router.py`: your ryu based router controller
- `requirements.txt`: your required python modules
- `*.py`: any other python files you've split your controller across
- `controller.Dockerfile`: should you be using docker and edit the Dockerfile

Remember, test your controller in the Docker image before submitting, as it will need to run in the container when marking!

Marking is again a book-a-session approach. These sessions will be bookable near to the submission date.

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 `OFPInstructionGotoTable` *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 `OFPInstructionGotoTable` action.

Note: You'll need to adapt any helper methods such that the `OFPInstructionGotoTable` can be used as well as the `OFPInstructionActions` 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`.