
Introduction

In lab 0, you have learned how to build and operate a layer-2 network using SDN. In this assignment, you will learn how to build and operate a layer-3 network using traditional distributed routing protocols (rather than SDN), how different networks managed by different organizations interconnect with each other, and how protocols, configuration, and policy combine in Internet routing.

More specifically, you will first learn how to set up a valid forwarding state within an autonomous system (AS) using OSPF, an intra-domain routing protocol (Part A). Then, you will learn how to set up valid forwarding state between different ASes, so that an end-host in one AS can communicate with a server in another AS via an intermediate AS (Part B). To do that, you will need to use the only inter-domain routing protocol deployed today: BGP. You will configure both OSPF and BGP through Quagga software routing suite [1], which runs on several virtual routers in your virtual machine (VM).

The rest of the document is organized as follows. We first give you a crash course on how to configure Quagga routers, then describe the setup you will have to use. Then we describe the two parts of the project: configuring OSPF within a domain, and configuring iBGP and eBGP. We conclude the document with how to test your code and how to submit the document.

Before you begin

All the files that you need for this lab are in the **`lab2`** sub-directory of your git repo. Before you begin the lab, you should move folder configs inside **`lab2`** to the **`/root`** directory. Once you have done this, the path to the configs file should be **`/root/configs`**. If you do not do this, you will get a lot of errors while working on the lab.

Here is a high-level overview of this lab (see figure below):

- In Part A, we give you the topology of a large ISP (Internet-2, which is the ISP that

connects higher education institutions in the US), with some of the details of the topology filled in and we ask you to configure all the routers and hosts in this topology.

- In Part B, we give you a multi-AS topology, where two ASs (East and West) are connected to Internet-2. The West AS is configured such that one of the routers is the static router you developed in Lab 1. We ask you to configure iBGP within Internet-2, and eBGP between East, West and Internet-2. Once you have done this, you will be able to download a webpage at a client host in West from a Web server running on East.

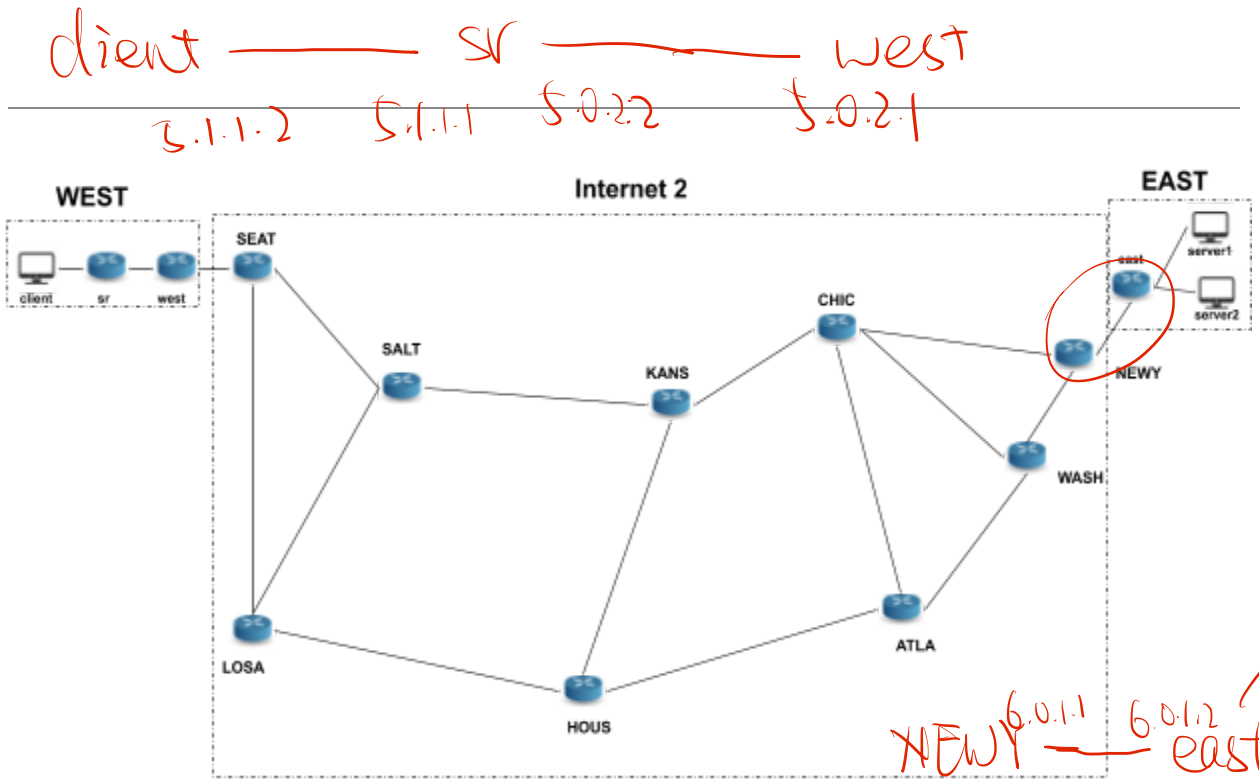


Figure 1: In this lab, you are given this topology, and asked to configure OSPF in Internet2, and BGP between EAST and Internet2 and WEST and Internet2

For Part A, before you begin configuring OSPF, you should, at a terminal window, type the following commands (inside the **`lab2`** directory):

```
$ sudo bash
$ killall ovs_controller
$ mn -c
$ python internet2.py
```

For Part B, before you begin configuring BGP, you should, at a terminal window, type the

following commands ((inside the **lab2** directory):

```
$ sudo bash
$ killall ovs_controller
$ mn -c
$ sh config.sh
$ python multiAS.py
```

Since you will be doing this often, you might want to create shell scripts for your convenience. Even if you do, it would help if you tried to understand what these commands do, so that you can debug any issues that might arise. The above commands will end inside mininet, so you should open another terminal to execute following commands. Or you could use detachable tools like screen or tmux.

Once you have started the topology, you should be able to log in to any node in the topology in order to configure it. For example, the Internet-2 topology contains a node called WASH. To access that node, you would use (inside the **lab2** directory: henceforth, any commands we specify for this lab should be run inside this directory, so we will omit this detail in our descriptions below) **sudo ./go_to.sh WASH**. Then, type **vtysh** to access the WASH router. You can configure it, as described below.

Configuring IP Routers

Traditional IP routers cannot be configured through OpenFlow. Instead, you need to configure them through a Command Line Interface (CLI). In this lab, we use an open-source routing protocol stack called Quagga. Quagga routers are configured through a CLI, which is similar to the CLIs used in real routers (e.g. from Cisco or Juniper).

In this section, we briefly describe how to configure a Quagga router. However, this is a very short introduction, we strongly encourage you to take a look at the official documentation to get more information.

The Quagga Command Line Interface

When you enter the Quagga CLI (see below for details how to access routers), you should see the following line:

```
router_name#
```

At any time when you are in Quagga CLI, you can use **?** to see all the possible commands that you can type at that point. For example, you should see the following (partial) output when you type **?** when you first enter the CLI.

```
router_name# ?
      clear Reset functions
      configure   Configuration from vty interface
      exit        Exit current mode and down to previous mode
      no          Negate a command or set its defaults
      ping        Send echo messages
      quit        Exit current mode and down to previous mode
      show        Show running system information
      traceroute  Trace route to destination
      write Write running configuration to memory, network, or terminal
```

Showing configuration: The command **show** will print various snapshots of the router configuration. To see what types of information can be shown, just type **show ?**. For example, **show running-config** will print the configuration that is currently running on the router. You can shorten the commands when there is no ambiguity. For instance, **show run** is equivalent to **show running-config**. Similar to the Linux terminal, you can also use auto-completion by pressing the tab key.

Switching to configuration mode: To configure your router, you must enter the configuration mode with **configure terminal** (**conf t** for the short version). You can verify that you are in the configuration mode by looking for the “config” prefix in your CLI prompt. You should see the following when you are in configuration mode.

```
router_name(config)#
```

If you want to cancel parts of the configuration, you can prefix the command you want to remove with **no**.

Configuring router interfaces

A router interconnects IP networks through several router interfaces. When a router receives a packet from one interface, it forwards it to another interface based on some pre-calculated forwarding decisions. Each interface must have an IP address configured and must be in a different subnet. To configure the IP address for an interface, you will first enter the configuration mode, and then specify the name of the interface you want to configure. For example, you can use the following commands to configure interface <*interface_name*> to 1.0.0.1/24.

```
router_name# conf t

router_name(config)# interface <interface_name>

router_name(config-if)# ip address 1.0.0.1/24
```

There are several ways to verify the current configuration has been updated correctly. You can use **`show run`** to examine the current router configuration. You can also show detailed information for interfaces by command **`show interface`**. To look at a specific interface, use **`show interface <interface_name>`**.

Once you have configured an IP address and a subnet to an interface, the router knows that packets with a destination IP in this subnet must be forwarded to this interface. You can use the following commands to show the subnets that are directly connected to your router.

```
router_name# show ip route connected

      C>* 1.0.0.0/24 is directly connected, <interface_name>
```

We see that 1.0.0.0/24 is directly connected and reachable with the interface <*interface_name*>. At this stage, any packet with a destination IP that is not in a directly connected subnet will be dropped. If you want your router to know where to forward packets with an IP destination in a remote subnet, you must use routing protocols, such as OSPF or BGP.

Configuring OSPF

OSPF routers flood IP routes over OSPF adjacencies. For Quagga routers, they continuously (and automatically) probe any OSPF-enable interface to discover new neighbors to establish adjacencies with. By default, Quagga router will activate OSPF on any interface whose prefix is covered by a **`network`** command under the **`router ospf`** configuration. For instance, the following commands would activate OSPF on any interface whose IP address falls under 1.0.0.0/24 or 2.0.0.0/24:

```
router_name# conf t
router_name(config)# router ospf
router_name(config-router)# network 1.0.0.0/24 area 0
router_name(config-router)# network 2.0.0.0/24 area 0
```

OSPF has scalability issues when there is a large number of routers. To mitigate such issues, the router topology can be hierarchically divided into what is called “areas”. In this assignment, your network is small and you do not need more than one area: **you will only use the area 0**.

With OSPF, each link between two routers is configured with a weight, and only the shortest paths are used to forward packets. You can use the following commands to set the weight of a link connected to interface_name to 900:

```
router_name# conf t

router_name(config)# interface <interface_name>

router_name(config-if)# ip ospf cost 900
```

You can use the following command to check the OSPF neighbors of a router:

```
router_name# show ip ospf neighbor

    Neighbor ID Pri State Dead Time Address Interface RXmtL RqstL DBsmL
    1.0.0.2 1 Full/Backup 1.0.0.2 newy:1.0.0.1 0 0 0
```

```
2.0.0.2 1 Full/Backup 2.0.0.2 newy:2.0.0.1 0 0 0
```

We see that the router has established two OSPF session with two neighbors. The first one is connected via the interface 1.0.0.1 and its IP is 1.0.0.2. The second one is connected via the interface 2.0.0.1 and its IP is 2.0.0.2. Since you are now connected to two other routers through OSPF, they can send you information about the topology of the network. Let's take a look at the routes received by OSPF using the following command.

```
router# show ip route ospf
      O 1.0.0.0/24 [110/10] is directly connected, newy, 07:09:33
      O 2.0.0.0/24 [110/10] is directly connected, atla, 06:14:24
      O>* 10.104.0.0/24 [110/20] via 2.0.0.2, atla, 00:00:10
```

You can see that our router has learned how to reach the subnet 10.104.0.0/24. The O at the beginning of each line indicates that the router has learned this subnet from OSPF. To reach it, it must send the packets to its neighbor router 2.0.0.2. If you want to have more information about the routers of this OSPF area, you can use **`show ip ospf database`**.

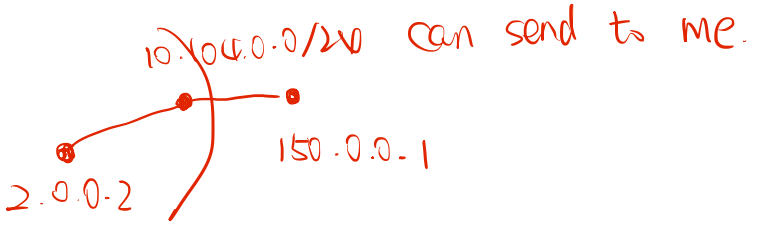
Configuring BGP

While OSPF is only used to provide IP connectivity within an AS, BGP can also be used to advertise prefixes between different ASes. Unlike OSPF, BGP routers will not automatically establish sessions. Each session needs to be configured individually. The following commands show you how to start a BGP process and establish two BGP sessions with neighboring routers. The integer following router bgp indicates your AS-number. Here, the local AS-number is 2.

```
router_name# conf t
router_name(config)# router bgp 2
router_name(config-router)# neighbor 150.0.0.1 remote-as 15
router_name(config-router)# neighbor 2.0.0.2 remote-as 2
```

bgp router-id <IP-Address>

By default, whenever the remote-as is different from the local number (here, 2), the BGP session is configured as an external one (i.e., an eBGP is established). In contrast, when the remote-as is equivalent to the local one, the BGP session is configured as an internal one. Here, rothe first session is an eBGP session, established with a router in another AS



(150.0.0.1), while the second one is internal session (iBGP), established with a router within your AS (2.0.0.2). You can check the state of your BGP sessions using the following command.

```
router_name# show ip bgp summary
      Neighbor V AS MsgRcvd MsgSent TblVer InQ OutQ Up/Down State/PfxRcd
      2.0.0.2 4 2 3 6 0 0 0 00:01:16 0
      150.0.0.1 4 15 2009 1979 0 0 0 01:31:42 1
```

You can see that our two BGP sessions are up. Notice that if the State/PfxRcd column shows “Active” or “Idle” instead of a number, that indicates that the BGP session is not established.

You might find it useful to manage several BGP neighbors as a group. By doing so, you can apply configurations to the whole group rather than typing similar commands for every neighbor, which could be tedious. Check **peer-group** command in quagga documents for more information.

neighbor <name> peer-group
neighbor <ip-addr> peer-group <name>

After BGP sessions are established, you can advertise prefixes using **network** command. For example, the following command advertise a prefix 10.104.0.0/24.

```
router_name(config-router)# network 10.104.0.0/24
```

Notice that you will only use this command to start advertising prefixes that are directly connected to your network. For prefixes that can only be reached reach indirectly through other networks (e.g. prefixes advertised by your neighbor), you simply propagate the existing advertisements.

After you do that, your neighbor 150.0.0.1 will receive this advertisement and know that it can forward you all the packets with a destination IP in the subnet 10.104.0.0/24.

You can check the routes learned from other BGP neighbors using the following command:

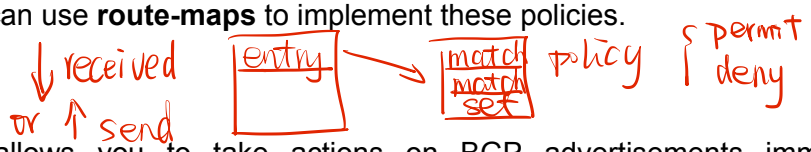
```
router_name# show ip route bgp

B>* 2.101.0.0/24 [20/0] via 2.0.0.2, interface used, 00:03:17
```

In this example, we can see that neighbor 2.0.0.2 has advertised one prefix: 2.101.0.0/24. The B at the beginning of the line indicates that the router has learned this prefix from BGP.

Configuring BGP policies

After you have BGP sessions running successfully, you might want to configure some BGP policies. For example, you may want to prefer one provider because it is cheaper than another

provider. For another example, you may want to avoid sending traffic to one AS for a particular prefix for security reasons. BGP offers several ways to configure such policies. The LOCAL-PREF attribute can help you influence outbound routing, while the MED attribute, AS-Path prepending, or selective advertisements can help you influence the inbound routing. In Quagga, you can use **route-maps** to implement these policies.



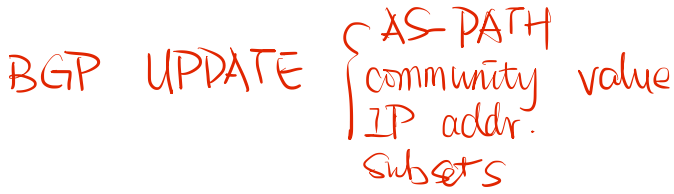
Route-map

Route-maps allows you to take actions on BGP advertisements immediately after an advertisement has been received, or right before being sent to a neighbor. A route-map is composed of route-map entries, and each entry contains three parts: **matching policy**, **match, and set**.

The matching policy can either be “permit” or “deny”. It specifies whether to permit or deny the routes that match the conditions in the match part. Intuitively, only permitted routes will go through the actions in the set part, while denied routes won’t be considered further. The match part is a boolean predicate that decides on which route to apply the actions to, and the set part defines the actual actions to take if a route matches.

Let’s take a look on what you can match on:

```
router_name# conf t
router_name(config)# route-map <MY_ROUTE_MAP> permit 10
router_name(config-route-map)# match ?
      as-path            Match BGP AS path list
      community          Match BGP community list
      interface          Match first hop interface of route
      ip                 IP information

      metric             Match metric of route
      origin             BGP origin code
      peer               Match peer address

      ...                ...
```



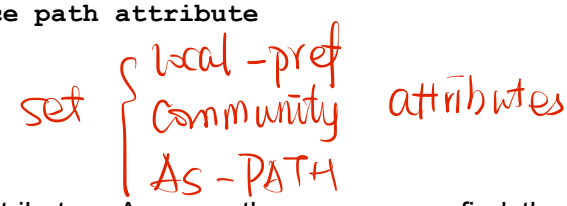
As you can see, you can match pretty much any attribute contained in a BGP UPDATE including: AS-PATH, community value, IP addresses or subnets, etc.

A BGP community can be seen as a label or a tag that can be attached to any route. You can use this attribute in both the match part and the set part of a route-map. As a convention, a community is often expressed as two separate integers, with the first one identifying the AS number that defined the community (either to use in routes internally within the AS, on routes

the AS passes to neighbors, or for neighbors to attach to routes passed to the AS). More than one community can be attached to a route.

Now let's take a look on what you can do with the matched routes using the set part.

```
router_name# conf t
router_name(config)# route-map <MY_ROUTE_MAP> permit 10
router_name(config-route-map)# set ?
      as-path            Transform BGP AS-path attribute
      community          BGP community attribute
      ip                 IP information
      metric             Metric value for destination routing protocol
      local-pref         BGP local preference path attribute
      origin             BGP origin code

      ...                ...
```



This set part enables you to modify any route attributes. Among others, you may find the following attributes most useful in this project: local-pref, community attributes, AS-path (to perform AS-path prepending.) Notice that these operations will only be applied on routes that match the condition specified in match part, if you use the permit clause.

A route-map can contain multiple entries. The order in which entries are processed is given by a sequence number (in the previous example, 10). The entry with the lowest sequence number is executed first. For example, if there are two entries for a route-map, with sequence number 10 and 20 respectively, a route will be first examined using the entry with sequence number 10, and then using the entry with sequence number 20. Notice that by default, if a route matches an entry, it won't be considered for the following entries with higher sequence numbers. Please check the official Quagga document to learn more about the default behaviors and how to change that if needed.

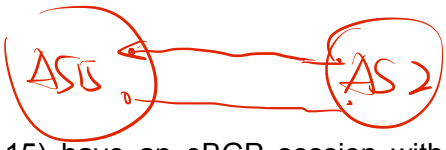
Important: Whenever you use a route-map, an entry that denies everything is implicitly added with the largest sequence number. That is to say, any routes that do not match in the previous entries would by default be denied. You need to add an additional entry to permit any routes to change this behavior.

Applying route-maps to BGP sessions

Once you have defined a route-map, you can apply it to a BGP session. A route-map can either be applied on incoming routes or on outgoing routes. For example, you can use the following commands to apply a route-map to the outbound advertisements to a BGP neighbor 150.0.0.1.

```
router_name# conf t
router_name(config)# router bgp 2
router_name(config-router)# neighbor 150.0.0.1 route-map <MY_ROUTE_MAP> out
```

The keyword ***out*** at the end of the last line means that this route-map is applied to the routes your router advertises. The keyword ***in*** would have applied the route-map to the routes the neighbor is advertising to your router.



Example

Let's assume multiple routers in your AS (AS 15) have an eBGP session with one of your provider (AS 2). However, you strongly prefer your provider to send traffic to you through one of them. You have agreed with your provider to use a BGP community value equal to 15:100 (15 is your AS number) to indicate the path you prefer.

The following commands show you how you can configure the route-map and apply it to the preferred router to tag any outgoing routes with such community.

```
router_name# conf t
router_name(config)# route-map <COMMUNITY_VALUE> permit 10
router_name(config-route-map)# set community 15:100
router_name(config-route-map)# exit
router_name(config)# router bgp 15
router_name(config-router)# neighbor 2.0.0.2 route-map <COMMUNITY_VALUE> out
```

} set route map

} deploy route map

Your provider can check that your prefixes have been advertised with the BGP community value with this command: **`show ip bgp community 15:100`**. The prefixes listed are the prefixes with the community value **`15:100`**.

Tip: Sometimes it takes time for BGP to converge, so you might not see the results corresponding to your changes immediately. You can use `clear ip bgp *` to force it to converge faster.

Now your provider can match on this tag to set the local-preference attribute accordingly and force traffic to exit via the chosen exit point. The following commands show you how your provider can do that.

```
router_name# conf t
router_name(config)# ip community-list standard <TAG> permit 15:100
router_name(config)# route-map <LOCAL_PREF> permit 10
router_name(config-route-map)# match community <TAG>
router_name(config-route-map)# set local-preference 1000
router_name(config-route-map)# exit
router_name(config)# router bgp 2
router_name(config-router)# neighbor 15.0.0.2 route-map <LOCAL_PREF> in
```

You can check that the local-preference has been correctly set with the following command:

```
router# show ip bgp
      Network Next Hop Metric LocPrf Weight Path
      *> 10.104.0.0/24 1.0.0.1 0 100 0 2 i
      *> 15.1.0.0/24 150.0.0.1 0 1000 0 15 i
```

The prefix advertised from AS 15 (15.1.0.0/24) with the community value 15:100 now has a local-preference value equals to 1000. In contrast, another prefix (10.104.0.2/24) without the community value have the default local-preference value 100.

This section only sketched the basics for route-maps. In this project, one of the learning goal is to get yourself more familiar with route-maps in order to implement the correct BGP policies.

Details on Configuring Quagga

After following the instructions here, you can then use the script *go_to.sh* to switch to a router or a host. For example, with the following command, you will access Internet-2's router LOSA.

```
> ./go_to.sh LOSA
```

LOSA is a router, to access the CLI of LOSA, just use the following command.

```
LOSA> vtysh
```

Type ‘exit’ to leave the CLI, and another ‘exit’ to leave the router.

From your VM, you can also go to a host. For example, if you want to go to the host which connects to router SEAT, you can use the following command.

```
> ./go_to.sh SEAT-host
```

When you are in a host, you can use **`ifconfig`** to see the interface which connects to the router. In this case, the name of the interface is seat.

Important: When you want to configure your router, **DO NOT** edit the config files directly. Instead, always use Quagga CLI.

Important: The current running configurations are not automatically synchronized with the configuration files you can find in the directory configs. **You must synchronize them manually**. To write your current configuration to the configuration files, you can use the following command. You will have to do that for each router.

```
router_name# write file
```

We recommend you to regularly save your configuration (the directory configs) to your own machine, since all your configuration will be reset if we reboot your VM. In case of a reboot, you can quickly put back your configurations in the routers by copy/pasting your configurations into the Quagga CLI. However, see below: in the lab, we ask you to write a Python script to load the saved configurations automatically to each router in the topology.

The Lab: Part A

In Part A, we ask you to configure OSPF in Internet2 (see Figure 1). In the next part, you will configure BGP at the border routers, and then be able to send traffic end-to-end. Thus, Part B depends on Part A.

Getting Started

To configure OSPF, you need two things: (a) the network topology, and (b) network configuration information (IP addresses, link costs etc.). We provide you with the code that creates a network topology in Mininet (more precisely, in MiniNExT). You can start this topology by typing the set of commands below in a terminal:

```
$ sudo bash
$ killall ovs_controller
```

```
$ mn -c
$ python internet2.py
```

Once you do this, you can log in to a router using the commands described above. After logging in to the router, you should be able to configure OSPF on Internet2, discussed below.

To configure OSPF, however, you will need configuration information. This configuration information is in the **`netinfo`** directory and is common both to Part A and Part B. This contains the following files:

- **`hosts.csv`**: This lists the hostnames of all the hosts in i2 (which is relevant for this part) and in East and West (relevant for Part B).
- **`routers.csv`**: This lists all the routers in each network (i2, East, West) and their loopback addresses. You will need these addresses for iBGP configuration.
- **`links.csv`**: This lists all the links in each network, together with interface names, IP addresses and link costs.
- **`asns.csv`**: This lists the AS numbers that you should use for each of the 3 ASes. (Used in Part B when you configure BGP).

You should read these files carefully. It might help to visualize these numbers by printing out Figure 1 on a sheet of paper and then annotating IP addresses etc. This annotation will help you avoid misconfigurations, and will also help you interpret your results (e.g., understanding if the traceroute took the correct path).

Configuring OSPF

You are now ready to configure OSPF on Internet2. You should follow these two steps.

Step 1

Configure OSPF to enable end-to-end connectivity between all the hosts inside your AS. Before configuring OSPF, you will have to configure all the IP addresses for each interface of your routers and hosts.

For routers, there are several interfaces we already created for you, so you only need to configure the correct IP address to corresponding interfaces using the commands in this section. You can view all the existing router interfaces using the command **`show interface`** in Quagga CLI. For hosts, you will also have to configure a default gateway (Internet2 has one host connected to each router). For example, if you want to configure the IP address and the default gateway for the host connected to NEWY router, you can use something like the following commands (the actual IP address may be different, you will have to look at the files in the **`netinfo`** directory):

```
> sudo ifconfig newy 4.101.0.1/24 up
> sudo route add default gw 4.101.0.2 newy
```

Make sure that each router and host can ping its directly connected router before you start the OSPF configuration. After you have configured OSPF for each router, test host-to-host connectivity with pings. Do not move on to other steps of the assignment before you verify that every host can ping every other hosts.

Step 2

Assign the OSPF weights based on Figure 1. Use traceroute between each pair of hosts attached to the Internet2 routers, to verify that the paths with smallest costs are used.

What to expect

If your configuration is working correctly, you should be able to:

- Ping or traceroute from any router to any other router or host
- Ping or traceroute from any host to any other host or router

Be careful: these will work only after OSPF has converged. How can you determine if OSPF has converged?

Scripting Configurations

Once you have a working configuration, we would like you to write the following two Python scripts (**Note:** it is important that your script names match the names suggested below, and these names should be placed in the **`lab2`** folder).

- **`load_configs.py:`** This script should automate the loading of the configs for all the routers in Internet2. When you save the configs in Quagga using the **`write file`** Quagga saves the configs in files with the **`.sav`** extension. For example, for each router, it will save configuration files named **`zebra.conf.sav`** and **`ospfd.conf.sav.`** Your script should read these files and load them automatically to each router appropriately. To do this, we give you two hints:
    - Look at the code for **`go_to.sh`** to understand how to run a command at a router.
    - Look at the manual for **`vtysh`** to understand how to pass commands to **`vtysh`**.
- **`config_i2_hosts.py:`** This script should automate configuring each Internet2 host's interface and configuring the default route.

Submitting and Testing

You are required to submit the following files for Part A. To do this, do a **`git add`** for these

files, then **`commit`** and **`push`** them to your github repo:

- The saved configs for each router. You will save these in a directory called **`configs`** within the **`lab2`** directory. Inside the **`configs`** directory, there should be a directory for each router (e.g., **`LOSA`**, **`WASH`**), and these should contain **`zebra.conf.sav`** and **`ospfd.conf.sav`** files for that router.
- The scripts **`load_configs_i2.py`** and **`config_hosts_i2.py.`** If you have done everything correctly, we should be able to type the following commands in the lab2 folder to load the configs to each router and to configure the host interfaces.

```
$ python load_configs.py configs
$ python config_i2_hosts.py
```

To test if you have implemented everything correctly, you can run the following command. This command automatically runs a bunch of tests on your implementation. Note that you should quit all the mininet terminals in your other screens initiated by internet2.py. Otherwise the tester won't start correctly. ***If you pass all of these tests, you will get a full grade for Part A.***

```
$ sudo python ../tester/generic_tester.py partA.xml
```

Look inside the **`partA.xml`** file to figure out what tests we plan to run on your code. It might be a good idea to start testing your code early, don't wait until the last minute to do so. Running these tests take time, so please be patient; in particular, the testing script may wait a minute or more for the network to converge before running the tests. At the end of the run, the tester will print your score for Part A.

The Lab: Part B

In this part, you are asked to configure BGP between EAST and Internet2 and WEST and Internet2 (see Figure 1). You will also need to configure iBGP inside Internet2.

Getting Started

As with Part A, to configure BGP, you need two pieces of information: (a) the network topology, and (b) the IP addresses and other configuration information. In this part also, we have pre-built the topology for you. We have also already configured all routers **within** the two ASs EAST and WEST. To start the topology for this part, type the following commands, which you might want to automate in a shell script.

```
$ sudo bash
$ killall ovs_controller
$ mn -c
$ sh config.sh
$ python multiAS.py
```

To configure BGP, you can use the information in the **`netinfo`** directory, as described above. In addition to configuring BGP, you will also use the OSPF configuration from Part A for testing.

Configuring BGP

Step 1

Configure internal BGP sessions (iBGP) between all pairs of routers (full-mesh) in Internet2. Please use the interface that connects to a host when specifying a BGP neighbor. These addresses are listed in **`routers.csv`** in the **`netinfo`** directory. You might need to specify using this particular interface when sending announcements to other BGP routers (hint: there is a command to do so). Verify that each of your routers does have an iBGP session with all the other routers with the command **`show ip bgp summary`**.

For each iBGP neighbors, please also add this command and think about why it is essential for iBGP neighbors.

neighbor <ip-addr> remote-as <AS-number>

```
router_name(config-router)# neighbor <ip_address> next-hop-self
```

Step 2

In Figure 1, determine where in the topology e-BGP needs to be configured (i.e., between which routers). Configure e-BGP between them. In this lab, we do not apply any routing policies, so you need not configure any policies. However, you should configure e-BGP so that only aggregated prefixes (not /24s) that cover the entire AS are advertised.

What to expect

If you have configured e-BGP correctly, you should be able to ping from **`client`** in WEST to **`server1`** or **server2** in EAST. For this to work, you will need to **copy** the simple router executable file (**`sr`**) that you developed in Lab 1 into the **`lab2`** folder. (**Important:** Lab 2 depends on Lab 1!). Then, you will need to run the following commands on the terminal, *after routing has converged* (how do you know if routing has converged?):

```
$ screen -S pox -d -m ~/pox/pox.py cs144.ofhandler cs144.srhandler
$ expect pox_expect
$ screen -S sr -d -m ./sr
```

As always, you might want to put these commands in a shell script because you might be using these often. Before you do so, try to understand what these commands are doing.

Once you have ping working, you should try traceroute from **`client`** in WEST to **`server1`** or **server2** in EAST. In addition, we have configured web servers on **`server1`** and **`server2`**, so you should be able to use HTTP to download files from these servers at *any node in the*

network not just **`client`**. Having completed Lab 1, you should be able to figure out which files can be downloaded from these two servers.

Scripting Configurations

Once you have a working configuration, we would like you to write the following Python script (**Note:** it is important that your script names match the names suggested below, and these names should be placed in the **`lab2`** folder).

- **`load_configs_multiAS.py:`** This script should automate the loading of the **OSPF and BGP** configs for all the routers in Internet2. When you save the configs in Quagga using the **`write file`** Quagga saves the configs in files with the **`.sav`** extension. For example, for each router, it will save configuration files named **`zebra.conf.sav`**, **`ospfd.conf.sav`** and **`bgpd.conf.sav`**. Your script should read these files and load them automatically to each router appropriately. To do this, see the hints we gave you for Part A. This script will be called with:

  ```
  $ python load_configs_multiAS.py configs_multiAS
  ```

Testing and Submitting

You are required to submit the following files for Part B. To do this, do a **`git add`** for these files, then **`commit`** and **`push`** them[1] to your github repo:

- The saved BGP and OSPF configs for each router. You will save these in a directory called **`configs_multiAS`** within the **`lab2`** directory. Inside the **`configs_multiAS`** directory, there should be a directory for each router (e.g., **`LOSA`**, **`WASH`**, **`east`**), and these should contain **`zebra.conf.sav`**, **`ospfd.conf.sav`** and **`bgpd.conf.sav`** files for that router.
- The script **`load_configs_multiAS.py`**. If you have done everything correctly, we should be able to type the following commands in the lab2 folder to load the configs to each router and to configure the Internet2 host interfaces (this script is from Part A).

  ```
  $ python load_configs_multiAS.py configs_multiAS
  $ python config_hosts_i2.py
  ```

To test if you have implemented everything correctly, you can run the following command. This command automatically runs a bunch of tests on your implementation. ***If you pass all of these tests, you will get a full points for Part B.***

```
$ python ../tester/generic_tester.py partB.xml
```

[1] Of course, **you should be frequently committing your work to the git repo as you develop your project to avoid losing your work.**

Look inside the **`partB.xml`** file to figure out what tests we plan to run on your code. It might be a good idea to start testing your code early, don't wait until the last minute to do so. Also, when running this automated tester, please be aware that some tests might take several tens of seconds or more, especially those that traceroute through your simple router. At the end of the run, the tester will print your score for Part B.

References

[1] Quagga Routing Suite. [Online]. Available: http://www.nongnu.org/quagga/
[2] Quagga Routing Suite - Documentation. [Online]. Available: http://www.nongnu.org/quagga/docs/docs-info.html
[3] Tmux, a terminal multiplexer. [Online]. Available: https://tmux.github.io

[4] Internet2. [Online]. Available: http://www.internet2.edu
[5] Nping. [Online]. Available: https://nmap.org/nping/