BGP Lab: Building an Internet Simulator

Copyright © 2020 by Wenliang Du.
This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. If you remix, transform, or build upon the material, this copyright notice must be left intact, or reproduced in a way that is reasonable to the medium in which the work is being re-published.

1 Overview

Border Gateway Protocol (BGP) is the standard exterior gateway protocol designed to exchange routing and reachability information among autonomous systems (AS) on the Internet. It is the "glue" of the Internet, and is an essential piece of the Internet infrastructure. It is also a primary attack target, because if attackers can compromise BGP, they can disconnect the Internet and redirect traffics.

Because of the complexity of BGP, it is hard to do everything in a single lab. Therefore, we have developed a series of labs related to BGP. This lab is the first in the series, and it is the basis for all the other BGP labs. The goal of this lab is to help students understand how BGP "glues" the Internet together, and how the Internet is actually connected. In this lab, we guide students to build a small-scale Internet. We call this Internet the Internet Simulator (or simply Simulator in short). This simulator will be the basis for all other BGP labs, as well as for some non-BGP labs that depends on the Internet.

In Tasks 1 to 3, we will show students how to build an Internet Simulator step by step. Most of the files are already provided in these tasks. Once students understand the building process, in Task 4, they will be asked to expand this simulator by including more autonomous systems, BGP routers, and Internet exchange points. This lab covers the following topics:

- How the BGP protocol works
- BGP configuration
- Routing
- Internet Exchange Point (IXP)

Videos. Detailed coverage of the BGP protocol can be found in Section 10 of the SEED Lecture at Udemy, *Internet Security: A Hands-on Approach*, by Wenliang Du. See details at `https://www.handsonsecurity.net/video.html`. The lecture was recorded before this lab was developed; it focuses mostly on the theory part, i.e., explaining how the BGP protocol works. This lab provides the practical part.

Lab environment. This lab has been tested on our pre-built Ubuntu 20.04 VM, which can be downloaded from the SEED website. Since we use containers to set up the lab environment, this lab does not depend too much on our SEED VM. You can do this lab using other VMs or physical machines.

Customization by instructor. It should be noted that Task 4 requires the instructor to make a random choice (to deter plagiarism).

Acknowledgment. This lab was developed with the help of Honghao Zeng, a graduate student in the Department of Electrical Engineering and Computer Science at Syracuse University. The SEED project is funded by the US National Science Foundation.

2 Conventions

In our Internet setup, we will use a lot of numbers. They are in two categories: autonomous system number (ASN) and IP address (for both network and host). In the real world, ASN and IP addresses have no particular relationship, and assignment of these numbers do not have specific patterns. In our setup, in order to make it easy to remember those numbers, we do use several conventions.

Autonomous System Number assignment. We mainly have four categories of autonomous systems, stub AS, Internet Exchange, small transit AS , and large transit AS. Here is the convention for the ASN assignment:

- `1`: Reserved.
- `2 - 19`: For large transit ASes.
- `20 - 49`: For small transit ASes.
- `50 - 99`: Reserved.
- `100 - 149`: For Internet Exchange.
- `150 - 254`: For stub ASes.

IP prefixes assigned to AS. In the real world, there is no relationship between ASN and IP prefix. In our simulator, we create an artificial relationship between them, so we can easily find the network address for any given ASN.

- For AS number `X`, its network prefix is `10.X.0.0/16`.
- If this AS has multiple subnets, the subnet ID will be `10.X.1.0/24`, `10.X.2.0/24`, and so on.

IP address assigned to machine. When we assign IP addresses to routers and hosts, we use the following convention on the IP address. We only apply the convention to the last 8 bits of IP address.

- `71 - 99`: For normal hosts. The assignment starts from `71`.
- `200 - 254`: For BGP routers. The assignment starts from `254` and goes backward.

3 Task 1. Building Containers for Autonomous Systems

To build an Internet simulator, we need to run many machines, some serving as routers and some as hosts. We will use Docker container to achieve that, running each machine as a container. In this task, we will create two autonomous systems, `AS150` and `AS151`. Inside each AS, we will create a network, a BGP router, and a normal host. Their IP addresses are specified in the following. We will use the container technology to do this. All the files needed for this task can be found in the `Base_Images` and `Task1_AS` folders.

```
AS150:
  Network:     10.150.0.0/24
  BGP router:  10.150.0.254
  Host:        10.150.0.71

AS151:
  Network:     10.151.0.0/24
  BGP router:  10.151.0.254
  Host:        10.151.0.71
```

3.1 The Base Container Images

The docker images in this lab are all based on two base images, one for the host and the other for the router. Both images are pretty much the same, except the host image comes with the `nginx` web server, while the router image comes with the `bird` BGP server. We have already built the base images and made them available from the Docker Hub.

```
Host:  handsonsecurity/seed-server:nginx
BGP:   handsonsecurity/seed-server:bgp
```

Although we can directly build our container images using the above base image from the Docker Hub, to reduce the number of access to the Docker Hub (the company puts a limit on how many accesses one can make during a six-hour time window), we first build a local base image using the one from Docker Hub, and then use the local base image to build the rest of the container images in this lab.

3.2 Containers for Host

The following is the content of the `Dockerfile` for host container. It is built upon the base image described above: `seed_base_host` refers to the base image. This is the name given in the docker compose file, which will be discussed later. When building the image, we copy a web page file (`index.html`) to the web folder.

```
FROM seed_base_host

COPY index.html /var/www/html/

CMD ip route change default via ${DEFAULT_ROUTER} dev eth0 \
    && service nginx start \
    && tail -f /dev/null
```

The `CMD` entry specifies the command that will be executed when the container starts. We have included three commands. The first one sets the default route based on the `DEFAULT_ROUTER` environment variable, which will be passed to the container when we will run it. The second command starts the `nginx` web server. The third command (`tail`) will block, preventing the command from finishing. If the command finishes, the container will be immediately shut down.

3.3 Containers for BGP Router

The `Dockerfile` for the BGP router is built on the `seed_base_router` base image built previously. It copies the bird configuration file to the container. We are using this same folder to build different docker images, each with a different bird configuration file. All the configuration files are stored inside the folder, but which one is copied depends on the value of the `BIRD_CONF` argument. The value will be set in the Compose file discussed later.

```
FROM seed_base_router
ARG BIRD_CONF

# Copy the bird configuration file
COPY ${BIRD_CONF} /etc/bird/bird.conf

# Delete the default routing entry and start BGP server
CMD ip route del default ; mkdir -p /run/bird && bird -d
```

3.4 Docker Compose

In this task, we only have four containers, so manually managing and running them is doable. However, as we get to the later tasks, the number of containers will significantly increase. For a more complicated setup, having 20 to 30 containers is not uncommon. In addition to these containers, we also have to create a number of networks, and assign IP addresses to each container. Manually managing all of these will be quite difficult.

Docker provides a tool called Compose, which is for defining and running multi-container Docker applications. With Compose, we use a YAML file to configure our containers, such as creating networks, assigning IP address to containers, configuring containers, etc. Then, with a single command, we can create and start all the containers from our configuration.

The `docker-compose.yml` file. There are two main sections in a compose file. The `services` section lists all the containers that we want to build, while the `networks` section lists all the networks that we need to create. The following example lists the two networks needed in this task, while the service entries are omitted.

```
version: "3"

services:
    ... (omitted, will be discussed later) ...

networks:
    as150-net:
        ipam:
            config:
                - subnet: 10.150.0.0/24
        name: seed-as150-net

    as151-net:
        ipam:
            config:
                - subnet: 10.151.0.0/24
        name: seed-as151-net
```

The base image. The first two service entries are the base container images that we want to build first. They are the bases for the other containers. The `image` entry specifies the name of the image. These names are already built into the `FROM` entry in the `Dockerfile` of each container, so if we change the names here, we need to change all of the `Dockerfile` files. These two containers do not play any role in the lab, and they will immediately exit after starting. Only their images are used as the basis for the other containers. As we mentioned before, we do this to avoid doing too many image pulls from the Docker Hub, which has set limits for users.

```
seed_base_router:
   build: ../Base_Images/router
   image: seed_base_router
   container_name: seed-base-router
   command: " echo ’exiting ...’ "    ← The container exits immediately
```

```
seed_base_host:
   build: ../Base_Images/host
   image: seed_base_host
   container_name: seed-base-host
   command: " echo ’exiting ...’ "  ← The container exits immediately
```

Host container. We place one host in each autonomous system. The following service entry specifies the host container for `AS150`.

```
as150_host:
   build: ./host
   image: seed-as-common-host             ← Name of the image
   container_name: as150-host-10.150.0.71 ← Name of the container
   environment:
         DEFAULT_ROUTER: 10.150.0.254     ← Used in Dockerfile’s CMD entry
   cap_add:
         - ALL
   networks:
         as150-net:
           ipv4_address: 10.150.0.71
```

We provide some further explanation in the following:

- `build: <folder name>`: This entry indicates the container's folder name, and will use the `Dockerfile` inside the folder to build the container.
- `container_name`: This entry specifies the name for the container. For convenience, we include the IP address of the container in the name, so we can easily know a container's IP address from its name.
- `environment`: We specify a environment variable called `DEFAULT_ROUTER`. As we have discussed before, this environment variable is used by the `CMD` entry of the `Dockerfile`.
- `cap_add`: this entry assigns capabilities to container. In our setup, we assign `ALL` the capabilities to the containers. This is because we need to conduct some operations inside container that are privileged. The root account inside a container does not have the same privilege as the root in the host machine. Granting all the capabilities to a container gives most of the real-root's privileges to the root inside the container.[1]
- The `networks` entry: It specifies the name of the networks that this container is attached to, along with the IP address assigned to the container. You can attach multiple networks to a container. In Tasks 2 and 3, you will find out that we attach two networks to the router containers (not for this task).

Router. In this task, for each autonomous system, we provide one BGP router. The following is the BGP router container for `AS150`.

```
as150_router:
   build:
       context: ./router
       args:
           BIRD_CONF: as150_bird.conf  ← Used in Dockerfile
```

[1] There are still something that the container-root cannot do, such as changing the `sysctl` parameters.

```
   image: as150-router
   container_name: as150-router-10.150.0.254
   cap_add:
       - ALL
   sysctls:
           - net.ipv4.ip_forward=1     ← Enable IP forwarding
   networks:
         as150-net:
           ipv4_address: 10.150.0.254
```

Most of entries are the same as those in the host container. We provide some further explanation in the following:

- The `args` entry: In this entry, we define an argument called `BIRD_CONF`. This argument is used in the `Dockerfile`, it specifies which of the bird configuration files should be copied into the container image.
- The `sysctls` entry: Since this container is used as a router, we need to enable its IP forwarding; otherwise, it will not forward packets.

3.5 Running and Testing the Simulator

We can use the following commands to build the containers first, and then use the `up` command to start all the containers. To stop and delete all the running containers, we can use the `down` command. Run these command inside the same folder where you put `docker-compose.yml`, because by default, they will look for this configuration file in the current folder.

```
// Build the containers
$ docker-compose build

// Start all the containers
$ docker-compose up

// Stop all the containers
$ docker-compose down
```

All the containers will run in the background. If we need to run some commands in a container, we just need to first find out this container's ID using the "`docker ps`" command, and then use the "`docker exec`" command to run a `bash` shell inside the container. If we use the `-it` option, we will get the interactive shell (a root shell). See the following.

```
$ docker ps
CONTAINER ID       NAMES
55ffedc5cc37       as150-router-10.150.0.254   ...
06d1693ad45f       as151-host-10.151.0.71      ...
debedefdfe18       as151-router-10.151.0.254   ...
265c098a6e19       as150-host-10.150.0.71      ...

$ docker exec -it 265 /bin/bash
#
```

For convenience. The printout of the "docker ps" command is quite long. Most of the information in the printout is not very useful to us. We can use the --format option to limit the fields in the printout. That will make the command quite long. Since we are going to use this command very frequently, we have created the following alias in .bashrc in the SEED VM. Moreover, we are also going to run "docker exec" very frequently, so we also want to create an alias for this command. Since it needs to take an argument, we add a shell function definition in the .bashrc file.

```
alias dockps=’docker ps --format "{{.ID}}  {{.Names}}" | sort -k 2’
docksh() { docker exec -it $1 /bin/bash; }
```

The "sort -k 2" command will sort the output based on the 2nd column. With the above alias and function, we can simply do the following to list all the running containers.

```
$ dockps
265c098a6e19  as150-host-10.150.0.71
55ffedc5cc37  as150-router-10.150.0.254
06d1693ad45f  as151-host-10.151.0.71
debedefdfe18  as151-router-10.151.0.254
```

With the IDs, we can now use the docksh function to run bash on a selected container. There is no need to type the entire ID string; just type the first few characters, as long as they are unique.

```
$ docksh 26
root@265c098a6e19:/#
```

Testing (Your Task). Please get into the host in AS150. Try to ping the router in the same AS, and also ping the host in AS151. Please provide your observation (screenshots are required).

```
# ping 10.150.0.254
# ping 10.151.0.71
```

4 Task 2. Internet Exchange Point and Peering

From the previous task, we can see that although the containers can reach the machine within the same autonomous system, they cannot reach the machine in the other autonomous system. This is because we have not connected these two ASes yet. In this lab, we will connect them, so they can reach each other.

4.1 Task 2.1. Connecting the Cable

Two autonomous systems can either connect directly or indirectly (i.e., through another autonomous system). In this task, we will connect two ASes directly. That means, their networks need to be physically connected. This is called *peering*. Peering can be done privately at a data center where two ASes connects with each other. Peering can also be done at a public place, which provides facilities for many ASes to peer with one another. Such a public place is called Internet Exchange Point (IXP) or simply called Internet Exchange (IX).

Internet Exchange in the real world could be quite complicated, but at its core is just a switch (a LAN). Autonomous systems who want to peer with others in this facility will connect their BGP routers directly to this LAN. In this task, we will set up an Internet Exchange in our simulator, and peer AS150 with AS151 inside it.

To do that, both `AS150` and `AS151` need to place a BGP router inside the IX. They will be connected to the LAN provided by the IX. For this IX, the network is `10.100.0.0/24`. These BGP routers also connect to their own networks, so each of them has two network interfaces and two IP address. They are specified in the Compose file. For convenience, we list their IP addresses in the following:

```
Router/Host      Interface 1         Interface 2
---------------------------------------------------
AS150 Router     10.150.0.254        10.100.0.150
AS151 Router     10.151.0.254        10.100.0.151
```

The BGP configuration file for each BGP router is placed inside the `ix100` folder. They will be used when we build the container images for the BGP routers.

Task (Your Task). Run this simulator, and `ping` the `AS151` host from the `AS150` host, report your observation, and provide the screenshots. Use the `mtr` command to do a traceroute and see where your packets go.

```
# mtr -n 10.151.0.71
```

4.2 Task 2.2. Peering Directly

You will find out that the hosts inside `AS150` and `AS151` still cannot reach each other, even though at the hardware level, their networks are connected via the `IX100` internet exchange. Routers in these two ASes still do not know the networks inside the other AS, or the networks that can be reached via the other AS.

This is because we are still missing the software component of the internet exchange. That is BGP, which is considered as the "glue" of the Internet. In this task, we will set up BGP, so `AS150` and `AS151` can exchange information about their networks and the networks that they can reach. The information will then be used by routers to route packets.

The software that we use for BGP is called BIRD. It already starts in each of the router containers. BIRD takes the configuration file `bird.conf` from the `/etc/bird/` folder. The following is the content of the configuration file for `AS150`'s BGP router.

Listing 1: The `bird.conf` file for `AS150`

```
protocol device {
}

protocol direct {
    interface "eth0";
}

protocol kernel {
    import none;
    export all;
}
```

To define a routing protocol in BIRD, we use the `protocol` keyword. BIRD supports many different routing protocols, like BGP and OSPF. A `protocol` can import or export routes to the BIRD's routing table. For this lab, we will use either `all` or `none`. In future BGP labs, we will introduce filters here, so you can set up rules to decide what routes can be imported or exported.

- The `device protocol` is not a real routing protocol. It doesn't generate any route, nor does it accept any route. It is only used to get information about the network interfaces from the kernel. Without the `device protocol`, BIRD will know nothing about the network interfaces; it will not even be able to run BGP, as it does not know how to reach BGP peer's IP address. Therefore, this block is mandatory.

- The `direct protocol` is for generating routes for the directly connected networks from a list of interfaces. In the `direct protocol` block, the *interface* keyword is used to generate routes from an interface. In this case, `AS150`'s bgp router uses `eth0` to connect to `AS150`'s internal network (`10.150.0.0/24`). Therefore, this `direct protocol` block will generate a routing entry for `10.150.0.0/24`. BGP will announce this network prefix to the peers.

- The `kernel` protocol is not a real routing protocol like BGP or OSPF. Instead of communicating with other routers in the network, it connects BIRD's routing table to kernel's routing table. It is the kernel's routing table that is used in the actual routing, not BIRD's routing table. This protocol is used to set the kernel routing table using the routes received from BGP peers. Here `import none` means BIRD's routing table will not import anything from the kernel's routing table; `export all` means BIRD's routing table will export all the routes to the kernel's routing table. This is how BGP routers set the routing table using the data collected from the BGP protocol.

Adding the BGP protocol to `bird.conf`. So far, we have shown how routes are generated and how BIRD's routing table can be used to set the kernel's routing table, but we have not told BIRD to run the BGP protocol. We add the following entry to `bird.conf` to run the BGP protocol on each BGP router:

```
protocol bgp {
    import all;
    export all;

    local    10.100.0.150 as 150;
    neighbor 10.100.0.151 as 151;    ← BGP peer
}
```

The `local` option in the BGP protocol sets the local IP address and ASN of the BGP session. The syntax is "`local [ip_address] as <asn>`". The `[ip_address]` part is optional, but it makes the configuration looks clearer and can prevent selecting the wrong IP address for the BGP session when there are multiple IP addresses on the router. This IP address should be the one on the IX's network.

The `neighbor` option in the BGP protocol sets the neighbor IP address and ASN of the BGP session. This is the actual peering part. In this example, we set up a BGP session between `AS150` and `AS151`, so they can exchange route information using the BGP protocol. Similarly, the IP address here should be the one on the IX's network.

Testing (Your Task). Please add the "`protocol bgp`" block to both `AS150`'s and `AS151`'s BGP routers inside `IX100`, so these two routers can establish a BGP session between themselves, and start exchanging route information. Start all the containers, and test whether the hosts inside these two autonomous systems can reach each other. If you have done everything correctly, they should be able to reach each other. Please provide screenshots and explanation.

4.3 Task 2.3. Peering via Route Server

In a public IXP, autonomous systems want to peer with many other autonomous systems. Let's say we have N autonomous systems, and they want to peer with one another. If we use the approach from the previous task, each pair of ASes needs to set up a peering relationship. That will be quite complicated.

Most IXPs provide a mechanism to simplify this. They provide a special server called *route server*. All these N autonomous systems will only need to peer with this route server. When the router server receives a route from a participant over BGP, it re-distributes the routes to all other connected participants. The route server function pretty much like multicast: any BGP route sent to the router server will be received by everybody that peers with the route server.

It should be noted that route server is not a real BGP peer, and its behavior is different from a real BGP peer. Most importantly, it does not add its own ASN to the path, nor does it change the nexthop attribute of the route. It is transparent to other BGP routers, and will not affect the outcome of the BGP protocol. Peering via a route server is equivalent to peering directly; its main purpose is solely to make peering among many BGP routers easier.

In this task, we will modify the peering between AS150 and AS151, so they peer via a route server provided by IX100. We only need to make one line of change. The route serve's IP address in IX100 is 10.100.0.100.

```
neighbor 10.100.0.100 as 100;     ← Peer with the router server
```

BIRD configuration for router server. The router server needs to specify peers in its BIRD configuration file. It should have an "protocol bgp" entry for each of its peer. Inside this entry, we added the "rs client" option, which tells BIRD not to prepend the IX's own AS number to the AS path, and not to change the nexthop attribute of the AS path. This way, no information of the router server will be added to the AS paths exchanged among its peers.

All these "protocol bgp" entries will have the same content, except for the neighbor option. In the BIRD configuration, we can use template to simplify the configuration. We create a BGP template called rs_peer, and use it as the basis for all the "protocol bgp" entries.

```
template bgp rs_peer {
    import all;
    export all;
    rs client;
    local 10.100.0.100 as 100;
}

protocol bgp from rs_peer {
    neighbor 10.100.0.150 as 150;     ← Peer with AS150
}

protocol bgp from rs_peer {
    neighbor 10.100.0.151 as 151;     ← Peer with AS151
}
```

Testing (Your Task). Please modify the "protocol bgp" block to both AS150's and AS151's BGP routers inside IX100. Please also add the corresponding "protocol bgp" block to the router server. Then start all the containers, do a traceroute from AS150's host to AS151's host, and explain your observation. More specifically, please compare the result with the one in the previous task, where we peer these two ASes directly, without using the route server.

5 Task 3. Transit Autonomous System

So far, if two ASes want to connect, they will peer with each other at an Internet exchange point. The question is how two ASes in two different locations get connected to each other. It is hard for them to find a common location to peer. To solve this problem, a special type of AS is needed.

This type of AS have BGP routers in many Internet Exchange Points, where they peer with many other ASes. Once packets get into its networks, they wll be pulled from one IXP to another IXP (typically via some internal routers), and eventually hand it over to another AS. This type of AS provides the transit service for other ASes. That is how the hosts in one AS can reach the hosts in another AS, even though these ASes are not peers with each other. This special of AS is called *Transit AS*. In this task, we will add a transit AS to our Internet Simulator.

5.1 Task 3.1. Peering at Multiple Internet Exchange Points

Here is the topology. `AS2` is a transit AS. It peers with `AS150` at `IX100`, and with `AS152` at `IX101`. Its goal is to provide transit services to these two ASes, so they can reach each other through `AS2`. The peering relationship of `AS150` and `AS151` stays the same. The peering relationships are depicted in Figure 1. All the files for this task can be found from the `Task3_Transit` folder.

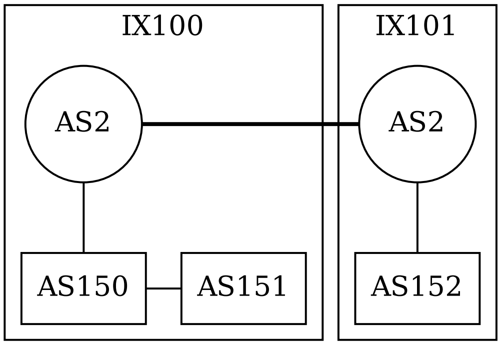

Figure 1: Internet Simulator for Task 3

We need two BGP routers for `AS2`. For the sake of simplicity, we assume that these two routers are on the same LAN (`10.2.0.0/24`) . In the real world, that is usually not the case. The network and IP address assignment are described in the following (information for `AS150` and `AS151` does not change).

```
Router/Host              Interface 1      Interface 2
------------------------------------------------------
AS2's Router @IX100      10.2.0.254       10.100.0.2
AS2's Router @IX101      10.2.0.253       10.101.0.2

AS152's Router @Ix101    10.152.0.254     10.101.0.152
AS152's Host             10.152.0.71          --
```

Testing. Start all the containers. Try to `ping 152`'s host from `AS150`'s host, and also run a traceroute between these two hosts. Please show your observation.

Run the `"ip route"` command on `AS2`'s BGP router, and see whether you see the entries to all the ASes (`AS2` connects to all the ASes in our topology). Here is a sample result on `AS2`'s router inside `IX100`. You should show your own results on both `AS2`'s BGP routers.

```
# ip route
10.2.0.0/24 dev eth0 proto kernel scope link src 10.2.0.254
10.100.0.0/24 dev eth1 proto kernel scope link src 10.100.0.2
10.150.0.0/24 via 10.100.0.150 dev eth1 proto bird
10.151.0.0/24 via 10.100.0.151 dev eth1 proto bird
```

From the above result, you can see that `AS2`'s BGP router inside `IX100` does not know how to route to `AS152`'s network `10.152.0.0/24`, even though `AS2` does peer with `AS152`, but the peering is in a different IXP. Although `AS2` has already laid the cable to connect the BGP routers at these two IXPs, they are not exchanging information. We are missing an important step.

5.2 Task 3.2. Adding Internal BGP Peering

Just like the peering of BGP routers from different autonomous systems, for the BGP routers in the same autonomous systems to exchange information, they also need to peer with each other and run the BGP protocol to exchange information. The BGP protocol conducted by the BGP routers inside the same AS is called IBGP (Internal BGP).

When we establish a BGP session between two routers with the same ASN, it will be considered as an IBGP session, and when the session is between two routers with different ASNs, the session is considered as an EBGP session (External BGP). Therefore, the way to define an IBGP session is the same as defining an EBGP session.

Let's add the following peering to `AS2`'s BGP routers in both `IX100` and `IX101`. In our setup, their IP addresses are `10.2.0.254` for the router in `IX100` and `10.2.0.253` for the router in `IX101`. The following example is the entry for the router in `IX100`. You need to make some changes for the other router. In the example, we give this BGP session a name called `ibgp`, but you can use any name.

```
protocol bgp ibgp {
    import all;
    export all;

    local    10.2.0.254 as 2;
    neighbor 10.2.0.253 as 2;
}
```

Notes. It should be noted that IBGP session has several different behaviors than the EBGP session.

- In IBGP sessions, when sending routes to peers, routers will not prepend their own ASN in the `AS_PATH`, and the `nexthop` attribute will not be altered either. **In your lab report, please explain why.**

- BGP routers will not forward the information collected from one IBGP peer to another; otherwise, there will be a loop. This is because IBGP does not add their own information to the AS path, BGP routers will not be able to know whether their peers already know the AS path or not. If forwarding is enabled, a BGP router will keep forwarding information to their peers, creating loops. Because there is no forwarding, typically, inside an AS, all IBGP peers will peer with one another.

Testing (Your Task). After making the required changes, run the Internet Simulator. Run the "`ip route`" command on `AS2`'s BGP router, and see whether you see the entry to all the ASes (`AS2` connects to all the ASes in our topology). Here is a sample result on `AS2`'s router inside `IX100`. You should show your own results on both `AS2`'s BGP routers.

```
# ip route
10.2.0.0/24 dev eth0 proto kernel scope link src 10.2.0.254
10.100.0.0/24 dev eth1 proto kernel scope link src 10.100.0.2
10.150.0.0/24 via 10.100.0.150 dev eth1 proto bird
10.151.0.0/24 via 10.100.0.151 dev eth1 proto bird
unreachable 10.152.0.0/24 proto bird
```

This time, we do see an entry to `AS152`'s network. That is a progress compared to the previous experiment, but the entry says "unreachable". Let's figure out why. Due to the IBGP session, now `AS2`'s BGP router at `IX100` has received the information about `AS152`'s networks. That is why we see the entry in the kernel's routing table.

In Task 3.5, you will learn a very useful utility `birdc`. If you run this utility to see the route information obtained from the IBGP session, you will see the following:

```
# birdc
bird> show route all
...
10.152.0.0/24   unreachable [ibgp 18:52:19 from 10.2.0.253] * (100/-) [AS152i]
   Type: BGP unicast univ
   BGP.origin: IGP
   BGP.as_path: 152
   BGP.next_hop:  10.101.0.152
   BGP.local_pref: 100
```

Pay attention to the `next_hop` attribute. This information comes from the BGP router in a different IXP. As we mentioned in the note above, in IBGP sessions, routers will not change the `next_hop` attribute (in EBGP sessions, this attribute will be updated because the next hop changes). This attribute still refers to the router located inside the `IX101` network (`10.101.0.152`), but the BGP router in `IX100` is not connected to that network; it does not know how to reach the `10.101.0.0` network. That's why it shows `unreachable`. It needs to find out which router can help forward packets to this destination network.

5.3 Task 3.4. Adding Internal Routing

Routers inside an autonomous system need to communicate with each other, so they can tell each other what networks they are connected to, so others can figure out the best path to get to those networks. That is what was missing in the previous task.

This type of routing protocol is called IGP (Interior Gateway Protocol). There are several IGP protocols, including OSPF (a link state routing protocol) and RIP (a distance-vector routing protocol). BIRD supports both of them. In this task, we will only use the OSPF protocol.

Let us add the OSPF protocol to our BIRD configuration file. In the following, we create a "`protocol ospf`" block and specify a table called `t_ospf`. That means all the OSPF routing information will be stored in this table. Inside the "`protocol bgp ibgp`" block, we tell BIRD to use the `t_ospf` table to resolve the nexthop for the route obtained from the internal BGP peers.

```
table t_ospf;  # Define a new table

protocol bgp ibgp {
    import all;
    export all;

    igp table t_ospf;

    local    10.2.0.254 as 2;
    neighbor 10.2.0.253 as 2;
}

protocol ospf {
    table t_ospf;  # Without this entry, the "master" table is used.

    import all;
    export all;

    area 0.0.0.0 {
        interface "eth0" {};
        interface "eth1" { stub; };
    };
}
```

Detailed configuration of OSPF can be quite complicated, and it is beyond the scope of this lab. An area with ID 0 is called a backbone area. All other areas need to be connected to the backbone area. In our configuration, we put all routes in the backbone area to make things simple. This particular BGP router (`AS2`'s router in `IX100`) is connected to two networks, one through `eth0` and the other through `eth1`.

Information from both networks will be included in the OSPF protocol, so others know how to reach these two networks. However, while `eth0` is connected to the internal network, `eth1` is the interface used by the router to connect to an outside network, the IXP's network. This network is provided by IXP for peering purposes. We do need to include this network in the OSPF protocol, so the internal router knows how to reach this network. However, we will not run OSPF protocol with anybody on this network, because they do not belong to `AS2`; they are outsiders. You do not want to leak the internal network information to the outside; nor do you want the outsider to manipulate your internal routes using OSPF. Therefore, you should not run the OSPF protocol with the routers on this external network. You only run OSPF with the internal routers. That is why we use `stub`, meaning the information from this network will be used in the OSPF protocol, but we will not run OSPF on this network.

Testing (Your Task). Make changes to the BIRD configuration files in both `AS2`'s BGP routers, and then run the simulator. You may need to wait a little bit, because OSPF takes some time to run. Check the routing table on `AS2`'s BGP routers, you should be able to see that machines from `IX100` can reach the networks in `IX101`. Conduct your experiments, provide your results, and explain your observations.

```
root@26aab6aab5d5:/# ip route
default via 10.2.0.1 dev eth0
10.2.0.0/24 dev eth0 proto kernel scope link src 10.2.0.254
10.100.0.0/24 dev eth1 proto kernel scope link src 10.100.0.2
10.150.0.0/24 via 10.100.0.150 dev eth1 proto bird
```

```
10.151.0.0/24 via 10.100.0.151 dev eth1 proto bird
10.152.0.0/24 via 10.2.0.253 dev eth0 proto bird
```

5.4 Task 3.5. Inspecting BGP

In this task, we will use tools to conduct further investigation on the BGP protocol. BIRD provides a command-line utility called `birdc`, which can be used to interact with BIRD, to examine the routes and sessions. It is quite easy to use. First, just get a shell on a router, and then type `birdc`. You will be in BIRD's interactive shell. A link to the `birdc` manual is provided on the lab's website.

In BIRD's interactive shell, pressing `?` on your keyboard anytime, you can get helps related to the current context. For example, if you type `s ?`, it will show you all the commands starting with the letter `s`. Without the leading letter, it will show you all the commands. In this task, we are only going to use the `show`, `disable` and `enable` commands.

The `show` command allows us to see detailed information of protocol and routes. The `enable` and `disable` commands can be used to disable/enable protocols. In the following example, we first list all the running protocols, and then disable the BGP session `bgp1`. You can see the differences in routing table before and after.

```
bird> show protocol
name     proto    table    state  since       info
kernel1  Kernel   master   up     14:22:49
device1  Device   master   up     14:22:49
direct1  Direct   master   up     14:22:49
bgp1     BGP      master   up     14:22:53    Established

bird> show route
10.2.0.0/24        via 10.101.0.2 on eth1 [bgp1 15:30:16] * (100) [AS2i]
10.150.0.0/24      via 10.101.0.2 on eth1 [bgp1 15:30:16] * (100) [AS150i]
10.151.0.0/24      via 10.101.0.2 on eth1 [bgp1 15:30:16] * (100) [AS151i]
10.152.0.0/24      dev eth0 [direct1 14:22:50] * (240)

bird> disable bgp1
bgp1: disabled
bird> show route
10.152.0.0/24      dev eth0 [direct1 14:22:50] * (240)
```

There are several useful options in the `show` commands. We summarize some of them in the following. For detailed instructions, please read the manual listed on the lab's website.

- `show route all`: print out detailed information about routes.
- `show protocol all bgp1`: print out detailed information about a BGP session (`bgp1`).

Sniffing BGP packet. To observe how the BGP protocol works, we can use Wireshark to capture all the protocol packets. In our simulator, we have created many networks. Our hosting VM is actually attached to all of these networks; that makes our lives much easy. The IP address assigned to our VM is `.1`. For example, on the `10.2.0.0/24` network created in the simulator, the IP address of our VM is `10.2.0.1`. Therefore, if we want to capture the traffic on any particular network, we just need to select the corresponding interface.

When we select interfaces inside Wireshark, we will see that the interface name cannot tell which network it belongs to. We can run the "`docker network ls`" command to find the mapping, and then use the mapping to select the correct interface in Wireshark.

```
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
...
537a85e6dfb8        seed_as2_net        bridge              local
16ba349d78a7        seed_as150_net      bridge              local
4f4061c87fcd        seed_as151_net      bridge              local
6ccf69ad6c00        seed_as152_net      bridge              local
...
```

Your Tasks. Using `birdc` and Wireshark, please finish the following tasks.

- Run `"show route"` command on `AS152` router, and explain what you see.
- Disable an BGP session and then enable it. Capture all the BGP packets using Wireshark, take a closer look at those packets, and report your findings.

6 Task 4. Enlarging Our Internet Simulator

We have now learned how to build an Internet Simulator. Let us enlarge this simulator by adding more components to it. We will add an IX (`IX102`), a new transit AS (`AS3`), and two stub ASes (`AS160` and `AS161`). Their peering relationship is depicted in Figure 2.

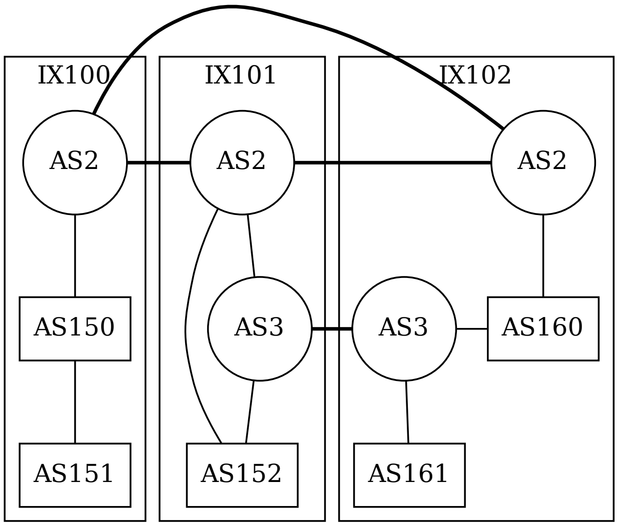

Figure 2: Internet Simulator for Task 4

The IP addresses for the BGP routers of the transit systems (`AS2` and `AS3`) are already provided in the following. They are based on our convention.

```
Router/Host              Interface 1     Interface 2
------------------------------------------------------
AS2’s Router @IX100      10.2.0.254      10.100.0.2
```

```
AS2's Router @IX101      10.2.0.253      10.101.0.2
AS2's Router @IX102      10.2.0.252      10.102.0.2
------------------------------------------------------
AS3's Router @IX101      10.3.0.254      10.101.0.3
AS3's Router @IX102      10.3.0.253      10.102.0.3
```

For the network of other autonomous systems, students should pick some real networks (e.g., a real company's or university's network). For example, `128.230.0.0/16` belongs to Syracuse University, and I can pick this network for `AS150`. Please be noted that students should pick their networks independently. The chance for two students to pick the identical set of 5 networks is very low. If that does happen, the instructor should pay a closer attention to the lab reports for potential plagiarism. Moreover, to make it hard for students to use the work from the previous semesters, the instructor should randomly choose a network ID for one of the autonomous systems, and change the choice every semester. Students should ask their professors for the choice.

Please build this simulator, run it, and provide the testing results to show that your Internet Simulator is functioning as expected. In the lab report, please show all the configuration files or contents that you add to the BIRD, Docker, and Docker Compose.

7 Submission

You need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. You also need to provide explanation to the observations that are interesting or surprising. Please also list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits.