CS498 Systems and Networking Lab **Spring 2012**

Lab 1: Build your own ISP

Instructor: Matthew Caesar *Due:*

In this assignment you will learn how to manage routers within an ISP network. Here, we will be working with the Quagga open-source software router, which is set up and managed in a similar way to commercial routers. As part of this assignment, we will set up a small network of Quagga routers running the BGP and OSPF routing protocols, modify their configuration files to instrument routing policies, and send some test packets to verify correct setup (and to troubleshoot if necessary).

1.1 FAQ

1. *Which OS should I use on emulab?* Just leave that field empty, Emulab will select the right file.
2. *What if I run out of machines or get a ”no available resources” error?* You may need to change the machine type in the tcl file. For example, change ”pc3000” to ”pc2400w” to use the 2400 MHz machines instead of the 3000 MHz machines if they ran out of the 3000 MHz machines.
3. *How do I install quagga?* You can get it from the web site, or you can use sudo apt-get install quagga.
4. *What is zebra?* Zebra is a process used to ”coordinate” the other processes that make up quagga. Quagga is made up of multiple processes, one for each routing protocol. Zebra is an additional process that arbitrates between the decisions of these multiple processes to construct a single RIB, which is sends to the kernel.

1.2 Initial setup

Follow the steps below:

1. We will be running our experiments on Emulab. Create an account by going to `https://www.emulab.net/joinproject.php3`, and joining the project ”uiucnet”. Note this it may take several days to activate your account. Emulab is a shared resource – please minimize use of machines to ensure other users can run their jobs!
2. Download the latest stable release (0.99.20) of the Quagga software router from `http://www.quagga.net/download.php` .
3. Use Emulab to start an experiment. The Emulab web form will ask you for an NS file to set up your network topology. Here is a sample: `http://www.cs.uiuc.edu/homes/caesar/courses/CS498.S12/samplens.tcl` . This sample file creates a small two-node topology connected by a single link. You'll need to edit it to create the larger topologies used below.
4. Untar and install Quagga on two of the Emulab machines.
5. If you are on Ubuntu, you can run sudo apt-get install quagga
6. If you are on Fedora, you can run sudo yum install quagga

Here is the Emulab ”Getting Started” Tutorial, `https://users.emulab.net/trac/emulab/wiki`. You can check the availability of emulab nodes from `https://www.emulab.net/nodecontrol_list.php3`.

1.3 Configure OSPF

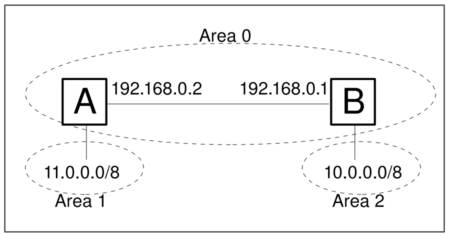

Figure 1.1: Topology after configuring OSPF in step 1.3.

Before we start Quagga, we need to tell it information about the topology (e.g. the IP addresses of other routers it will peer with) and policy (e.g. which IP prefixes it should originate and export). This is done by editing a *configuration file*. Quagga’s configuration file language is very similar to the one used in Cisco’s routers. Here, we will set up two Quagga routers as shown in Figure 1.1.

We will start by configuring an OSPF session between routers A and B. OSPF is a link-state routing protocol used for intra-domain routing (routing within a single network). Configuration of this OSPF session informs A and B that the link between them may be used for intra-domain routing. The OSPF protocol is responsible for sending probes (hello packets) over the link from A to B to detect failures of that link, using the link in shortest-path computations. Here we’re using a small network with only two routers, but if there were other routers within the network, OSPF would also inform the other routers about the existence of that link, so other routers could use it in their shortest path computation.

To do this, we will run Quagga’s ospf daemon. However, first, we need to start up Quagga’s *zebra* daemon. The zebra daemon is responsible for receiving routes from the various Quagga routing processes (e.g., ripd, bgpd, ospfd) and publishing them to the linux kernel. First, configure zebra by creating a zebra.conf file contining information about the interfaces attached to the router, and the IP address ranges associated with each interface:

```
!
! Zebra configuration saved from vty
! 2008/03/17 23:24:27
!
hostname zebrad
password zebra
!
interface eth0
ip address 192.168.31.0/24
!
interface eth1
ip address 192.168.35.0/24
!
interface lo
ip forwarding
!
log stdout
line vty
!
```

Since the IP addresses and interfaces of PCs in your setup may differ, you will likely have to edit these files to take that into account. You can use the *ifconfig* command to determine the IP address associated with a particular interface, and then assign the interface the /24 IP subnet containing the interface's IP address. Start up the zebra daemons on each machine, using the command *sudo ./zebra -f configfile -d*.

Next we will set up ospfd to build an OSPF session between the routers. To simplify this step, we will give you the configuration files for A and B. However, since the IP addresses and interfaces of PCs in your setup may differ, you will likely have to edit these files to take that into account. You can use the *ifconfig* command to determine the IP address associated with a particular interface.

Configuration file for router A:

```
! -*- ospf -*-
! OSPF Config file for router A (ospfA.conf)
hostname ospfd
password zebra
!
interface eth0
!
router ospf
network 10.0.0.0/7 area 0
network 11.0.0.0/8 area 1
```

Explanation: First, we need to tell Quagga which interfaces on the local PC may be used by the routing protocol. This is done with the *interface* command above. Here, we tell the router that interface with IP address 192.168.0.2 should be called eth0 and is made visible to routing protocols running at this router. OSPF will then automatically discover routers on the other side of the link. Next, the *router* command tells Quagga to create a new OSPF router instance. OSPF creates logical groups of routers called *areas* to improve scalability. Routing updates are constrained within OSPF areas. Here, we only create a single network-wide area called *area 0* (the top-level area in OSPF is always numbered zero). We tell Quagga this area contains prefixes 10.0.0.0/8 and 11.0.0.0/8, by saying that it contains the network with super-prefix 10.0.0.0/7. Finally, we tell Quagga that routes to 11.0.0.0/8 can be reached at the local router within area 1 with the command *network 11.0.0.0/8 area 1*.

(a) The configuration file for router A is given above. Write the configuration file for router B. Your file will look very similar to the configuration file for router A provided above. Your configuration file should make B part of area 2 containing prefix 10.0.0.0/8. Copy and paste the configuration file you write into your report.

(b) Save the first file as ospfA.conf in the quagga-0.98.6/ospfd directory on the machine you are using to emulate router A. Do the same for the configuration file you wrote (ospfB.conf) on the machine you are using for router B. Then, start up the ospf daemons on each machine, using the command *sudo ./ospfd -f ospfA.conf -d* on router A and *./ospfd -f ospfB.conf -d* on router B.

(c) Verify your setup is correct. Do this by logging in to the vty (router terminal) by doing *telnet localhost 2604*, and executing the commands *show ip ospf neighbor* and *show ip ospf route*. You should see output indicating that A is peered with B, and vice versa. If they are not, you have made an error in the above steps. Copy and paste the output you see from these commands into your writeup.

1.4 Configure iBGP

After completing the previous step, router A can discover router B's presence, forward data packets to hosts connected to B, and vice versa. In this step, we will also allow A and B to exchange externally-learned routes. In particular, router A and B will function as border routers connected to other ISP networks. Routes received from other networks need to be propagated internally, to ensure all internal routers know how to reach externally-learned prefixes. For example, if router A learns that it can reach $12.0.0.0/8$ from router C, it needs to inform B of that fact, so B knows it can reach $12.0.0.0/8$ by routing through A.

Routes between ISPs are propagated using the Border Gateway Protocol (BGP). BGP is a “path-vector” protocol – it propagates advertisements containing a list of hops to a particular destination. BGP operates on prefixes – each advertisement contains a list of prefixes, and the AS-level path used to reach those prefixes. The AS-path is used to avoid routing loops and ”count-to-infinity” problems, by having each router check to see if its own AS number already appears in the path, and if so, dropping the advertisement. The AS-path may also be used in routing policies, e.g., ASes may add multiple copies of their AS number in routing advertisements to increase the path length on certain routes, thereby shifting traffic to alternate paths.

Here, we will configure a BGP session between routers A and B. When BGP is run internally within an AS, it is referred to as ”iBGP” (internal-BGP). Again, we will give you the configuration files to use:

Configuration file for router A:

```
! -*- bgp -*-
! Config file for router A (bgpdA.conf)
!
hostname bgpd
password zebra
!
! Let quagga know that this router is within AS number 8000
router bgp 8000
! The "router id" is used as the IP address of the router,
! and is also used to break ties (after all steps of the decision process)
 bgp router-id 192.168.0.2
 network 10.0.0.0/7
! tell quagga that it should connect to (peer with) a neighboring router
! with router-id (IP address) "192.168.0.2", which is also in AS 8000
 neighbor 192.168.0.1 remote-as 8000
 neighbor 192.168.0.1 update-source lo0
!
log stdout
```

Explanation: First, we need to tell Quagga to create a new bgp router instance. This is done with the *router bgp 8000* command. The ”8000” at the end tells the Quagga bgp daemon the name of the local AS number, so the router can append its AS number to advertisements, etc. Next, we tell Quagga the IP address that identifies this BGP router with the *bgp router-id* command. Finally, we tell Quagga to create an iBGP session to router B, which has loopback address 192.168.0.1. Since B is within the same AS as A, we want to tell BGP to forward the traffic via OSPF, rather than a BGP route. We do this by specifying the command ”update-source lo0”, which tells the router to contact 192.168.0.1 via the loopback address.

Similarly, here’s the configuration file for router B:

```
! -*- bgp -*-
! Config file for router B (bgpdB.conf)
!
hostname bgpd
password zebra
!
router bgp 8000
 bgp router-id 192.168.0.1
 network 10.0.0.0/7
 neighbor 192.168.0.2 remote-as 8000
 neighbor 192.168.0.2 update-source lo0
!
```

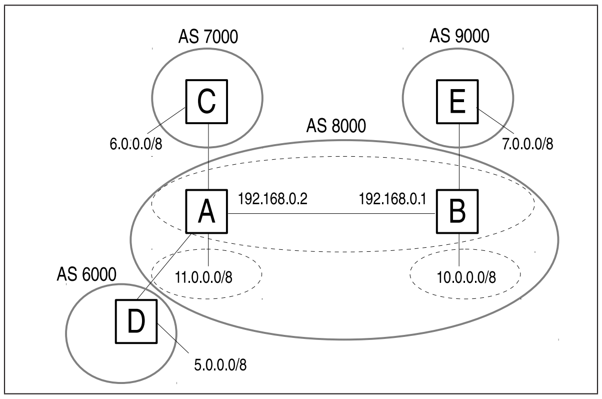

Figure 1.2: Topology after configuring iBGP and BGP sessions in steps 1.4 and 1.5.

```
log stdout
```

(a) Place these files in quagga-0.98.6/bgpd, and start up the two Quagga instances with the commands *sudo ./bgpd -f bgpdA.conf -d* on router A and *sudo ./bgpd -f bgpdB.conf -d* on router B.

(b) Now, verify they are correctly peering. Quagga (like many commercial routers) supports a shell interface where you can log in and execute commands to check the current state of the router and its connections. You can access this shell by doing *telnet localhost 2605*, then typing *"show ip bgp"*. The output should look similar to this on router A:

```
    Network          Next Hop           Metric LocPrf Weight Path
  *>  11.0.0.0/8      10.0.0.2             0    100      0  100 ?
  *>a 10.0.0.0/8                           0    0        0  i
```

and similar to this on router B:

```
    Network          Next Hop           Metric LocPrf Weight Path
  *>  10.0.0.0/8      10.0.0.1             0    100      0  100 ?
  *>a 11.0.0.0/8                           0    0        0  i
```

1.5 Configure BGP

Now, extend the configuration to the topology shown in Figure 1.2. To do this, you'll need to start up 3 more instances of Quagga, each running on a different machine, to create a network composed of 5 software routers total. Have routers in AS 8000 advertise a static route to 10.0.0.0/7 (the aggregation of 10.0.0.0/8 and 11.0.0.0/8). Print your configuration files as well as the terminal output of "show ip bgp", and attach them to your writeup.

Hint: configuring external BGP sessions is a little different from iBGP sessions. You may want to do something along these lines, which combines the iBGP file from above with commands to create the eBGP sessions:

```
! -*- bgp -*-
! Config file for router A (bgpdA.conf)
!
hostname bgpd
password zebra
```

```
!
router bgp 8000
 bgp router-id 10.0.0.1
 neighbor 192.168.0.1 remote-as 8000
 neighbor 192.168.0.1 update-source lo0
 neighbor 192.168.0.1 ebgp-multihop
 neighbor 192.168.0.1 next-hop-self
 network 10.0.0.0/7
 neighbor 10.0.0.6 remote-as 7000
 neighbor 10.0.0.6 ebgp-multihop
 neighbor 10.0.0.6 next-hop-self
!
log stdout
```

Explanation: The next-hop-self command ensures that router A's IP address is advertised as the next-hop used to reach routes it advertises. The ebgp-multihop command allows connections to peers that are not directly connected. While this isn't strictly necessary in this example, it may be used in cases where the border router's peer is multiple router-level hops away.

1.6 Policy configuration:

Next, we will configure policies to limit route export. We will consider routers C and E to be *providers*, and router D to be a *customer* of the ISP containing A and B, as shown in

(a) If router C advertises a route, which routers should receive that route? What about if router E advertises a route? What about if router D advertises a route?

(b) Modify your configuration (hint: you may want to do this by tagging route advertisements with a community attribute indicating what kind of peer they are received from, and then also add an export filtering rule for providers that filters routes from customers. An example of this is shown below.) Print your configuration files as well as the terminal output of ”show ip bgp”, and attach them to your writeup.

You can tag route advertisements with community attributes as follows:

```
ip community-list 1 permit 0:1000
route-map IMPORT-CUST permit 10
  set community 0:1000
!
neighbor <customer-neighbor-IP-address> route-map IMPORT-CUST in
```

Explanation: A ”route-map” is a construct in router configuration languages used to express policies that should be applied to groups of routes. A route map consists of a name (*IMPORT-CUST*, a set of rules denoting which a match to this rule should be triggered (with the ”match” command, here we leave out the match command to match all routes), and the commands to be applied to updates triggering a match (*set community 0:1000*, which adds the community attribute ”0:1000” to the routing update). The ”neighbor” command at the end applies the route-map to all updates received from *customer-neighbor-IP-address*. The ”in” statement at the end of that line means the rule is applied to inbound updates, as opposed to updates this router advertises to customer-neighbor-IP-address.

You can prevent route advertisements with a given community attribute from being advertised to a peer (export filtering) by as follows:

```
route-map EXPORT-CUST permit 10
  match community 1
```

```
!
neighbor <prov/peer-IP-address> route-map EXPORT-CUST out
```

Explanation: Here, we instruct Quagga to apply the route-map *EXPORT-CUST* to updates received from the neighbor with loopback IP *prov/peer-IP-address*. We use the match command to only select routes advertised by customers for export on the peering session.

(a) Attach all the configuration files you wrote to your report. (b) Attach the output of the command ”show ip bgp community” with all the user-defined communities from router A and B