Name 1: Tristan Overney
Name 2: Yoann Ponti

TCP/IP NETWORKING

LAB EXERCISES (TP) 4
DYNAMIC ROUTING AND SDN BASICS

November 10, 2016

Abstract

In this TP you will learn how to configure "distance-vector" routing protocols that typically run inside the network of an autonomous system (AS). The protocols set up automatically network routes that ensure shortest path between two points in the network with respect to a predefined metric (such as number of hops). You will also learn about software-defined networking (SDN) and study how it can be used to create forwarding rules on switches.

1 LAB ORGANIZATION AND INSTRUCTIONS

1.1 LAB ORGANIZATION

In this lab, you will exercise your knowledge about distance-vector routing. During the lab you will configure a fully functional network using Cisco-like routers. You will learn how to configure a network of routers and through some interesting scenarios, see how the routing works in the internet where the network is constantly changing. We complete the lab with an introductory exercise to software defined networking (SDN), where you will learn how does flow-based forwarding take place.

1.2 LAB REPORT

Type your answers in this document. We recommend you to use the latest or an updated version of Adobe Reader to open this PDF, as other readers (such as SumatraPDF, but also older versions of Adobe!) don't support saving HTML forms. That will be your Lab report (one per group). When you finish, save the report and upload it on moodle. Don't forget to write your names on the first page of the report.

The deadline is November 23 at 23:55.

2 SETUP AND ENVIRONMENT

```
BOFH excuse #427:
network down, IP packets
delivered via UPS
```

BOFH

The Internet core is run by powerful routers that can handle large amounts of traffic built by companies such as Cisco, Huawei or Juniper. Even in a large company, or on large university campuses (such as EPFL), these routers are present. They use proprietary operating systems (such as Cisco IOS, or JunOS) and can be accessed via control terminals tailored for network configuration, with commands that are quite different from those you may encounter in a UNIX/Linux console. In the following labs we will give a flavor of router configuration.

Ideally, we would like to run IOS in a virtual environment. While this is technically possible, it is not very legal, as Cisco or Juniper do not allow their OS to be run on a device other than their routers.

Instead we will use Quagga, which is a free routing software suite running on Linux. Quagga provides a control terminal that accepts similar commands to the ones in Cisco's IOS. For our convenience, Quagga integrates with Mininet via MiniNExT, which is an extension that enables PID namespaces, necessary to run Quagga independently on each virtual host. MiniNExT is already installed in the TCPIP-VM that you have used in Labs 1 and 2. For convention, in this lab, we will use the term "virtual router" to refer to each "virtual host" created by Mininet or MiniNExT.

2.1 WHAT IS QUAGGA AND HOW DOES IT WORKS?

Quagga is a routing software suite that provides implementations of several routing protocols (namely OSPF, RIP and BGP-4) for Unix platforms. It consists of a handful of processes that can be run in the background as daemons. Four Quagga processes (daemons) are important for executing this lab:

- *zebra*: is used to manage the network interfaces of a machine (in our case each virtual router). It allows you to configure them (using IPv4 and/or IPv6 addresses), to monitor their states, and it provides a more detailed view of the routing tables than the `route -n` command. In a way, *zebra* is a replacement for the Linux networking commands used during the first three labs (*i.e.,* `ifconfig`, `route`, etc.).
- *ripd*: handles RIP version 2 implementation.
- *ripngd*: handles RIP routing for IPv6 (*RIP next generation*).
- *quagga*: The main service, which is used to call the three daemons above.

As we'll see in Section 2.2, MiniNExT creates separate PID namespaces for each virtual router. This means that each virtual router inside Mininet's virtual environment will have its own `quagga` service running independently.

2.1.1 STARTING A QUAGGA PROCESS

Each daemon has its own `<daemon_name>.conf` file that needs to exist, even if it is empty, in order for Quagga to work properly. In the TCPIP-VM, Quagga is configured as a Linux service. To start the service, we need to configure two files: `daemons` and `debian.conf`. In the `daemons` file you should see something like this:

```
zebra=yes
bgpd=no
ospfd=no
ospf6d=no
ripd=yes
ripngd=no
isisd=no
```

where you need to specify which processes would you like Quagga to enable. Note that `zebra` process should always be enabled.

In the `debian.conf` file, you should see something like:

```
vtysh_enable=no
zebra_options=" --daemon -A 127.0.0.1 -u quagga -g quagga"
bgpd_options=" --daemon -A 127.0.0.1 -u quagga -g quagga"
ospfd_options=" --daemon -A 127.0.0.1 -u quagga -g quagga"
ospf6d_options="--daemon -A ::1 -u quagga -g quagga"
ripd_options=" --daemon -A 127.0.0.1 -u quagga -g quagga"
ripngd_options="--daemon -A ::1 -u quagga -g quagga"
isisd_options=" --daemon -A 127.0.0.1 -u quagga -g quagga"
```

where:

- `--daemon`: Makes the process run in the background
- `-A`: Specifies through which IP address the configuration mode can be accessed. By default only from localhost.
- `-u`: Username to start the process. Default is quagga
- `-g`: Specify the privilege group for the user. Default is quagga

Once each `conf` file is properly configured, we need to start the Quagga service like any other Linux service:

```
/etc/init.d/quagga start
```

There are two ways for configuring the Quagga processes: with the `<daemon_name>.conf` file or using "on the fly" configuration. In this lab, we insist on configuring the Quagga processes using the first method as it can be better adopted to Mininet's scripting nature. When "on the fly" (while running) configuration is used, for each process you need to enter the corresponding configuration mode using the command:

```
telnet localhost <process>
```

or for IPv6 routing protocols:

```
telnet ::1 <process>
```

If you want to go back from the process configuration-mode you should type

```
exit
```

As modifying the running configuration without stopping the processes will be desirable at some stages of this lab, in Section 3.1.3 we explain how the configuration of a virtual router can be modified "on the fly".

2.1.2 KILLING A QUAGGA PROCESS

Quagga processes can be killed using two methods:

1. **Stop quagga service**: Using the traditional command to stop a process in Linux:

   ```
   /etc/init.d/quagga stop
   ```

2. **Killing individual processes**: Sometimes you want to simulate a crash on a router, and you require to kill a specific process. To kill the individual Quagga processes that are running on a virtual router, type the following command in the terminal window:

   ```
   killall <process name 1> <process name 2> ...  <process name N>
   ```

   where `<process name 1> <process name 2> ... <process name N>` are the Quagga processes that are running on the virtual router (*e.g., zebra*, *ripd*, *ripngd*, etc). Note that if you want to start the individual process again, you will need to restart the `quagga` service, as this is the only option for starting the Quagga processes.

You can check whether a process is running on a virtual router, by observing the list of running processes. To do this, use the following command:

```
ps -A
```

2.2 INTEGRATING QUAGGA IN MININET USING MININEXT

MiniNExT (Mininet Extended) is an extension layer to Mininet that provides PID (process) namespaces, mount (filesystem) namespaces, log and runtime isolation; it is designed to build more complex networks than single Mininet. MiniNExT has a python-based building block specific for Quagga, which will be used in this lab to integrate Quagga into Mininet's virtual environment.

For this lab, we suggest to separate Python scripts in two: as the topology will not change during the lab, you can use one python script to create a topology class: build quagga-enabled routers, add hosts, switches and links. With the other script you can run the virtual environment: call the topology, add customized commands and start the command line and terminal window for each router. For your convenience, we will provide both scripts and they can be found in Moodle.

2.2.1 THE TOPOLOGY CLASS

The script is shown in Script 1. Take note of the following remarks regarding this script:

- We need to import the Topo class from MiniNExT and not Mininet. MiniNExT's extension includes a function for adding a node service, which is essential for Quagga (check line 13).

- The script assumes that you have a `configs/` folder in the same path where the script is located, and within the config folder we need one folder for each router, named exactly as the router (*e.g.,* $r1$, $r2$, etc.). Lines 48 and $76 - 77$ can be modified to change this behavior. and within.
- The quagga process can be configured to automatically start or stop. This is controlled in line 42.
- We used a *namedtuple* to manage all the attributes that we want to include when adding virtual routers to the network. If you would like to modify this and add an extra attribute (e.g., a loopback address for each router), you can modify the variable in line 22, and then the section beginning line 57.
- When adding a virtual router to the network, we need to enable process, log and run isolation (check line 73).
- We specify which interface we want to connect the virtual router to, otherwise Mininet will allocate the first available interface and the topology may not look as the one in the lab's figure (starting line 89).

Script 1: Lab topology using MiniNExT

```
1  """
2  Topology for lab4, see fig.1 of PDF
3  """
4
5  # Modules needed to get the absolute path to this file for quagga configuration
6  import inspect
7  import os
8
9  # You can show useful information during script execution
10 from mininet.log import info
11
12 # Class in mininext which includes PID namespaces, log and run isolation.
13 from mininext.topo import Topo
14
15 # Class in mininext to setup the quagga service on router nodes
16 from mininext.services.quagga import QuaggaService
17
18 # A container in python
19 from collections import namedtuple
20
21 # Variable initialization
22 NetworkHosts = namedtuple("NetworkHosts", "name IP DG")
23 net = None
24
25
26 class Lab4Topo(Topo):
27
28     "Creates Lab4 Topology"
29
30     def __init__(self):
31         """Initialize a Quagga topology with 2 hosts, 5 routers, 7 switches, quagga
       service, in a topology according to
32   Fig 1. of Lab4"""
33         Topo.__init__(self)
34
35         info( '*** Creating Quagga Routers\n' )
36         # Absolute path to this python script
37         selfPath = os.path.dirname(os.path.abspath(inspect.getfile(inspect.
       currentframe())))
38
39         # Initialize the Quagga Service
40         # autoStart=True (default) --> starts automatically quagga on the host
41         # autoStop=True (default) --> stops automatically quagga (we don't want this)
42         quaggaSvc = QuaggaService(autoStop=False)
```

```
43
44          # Configuration file path for quagga routers
45          # We require a "config" folder in the same path of the lab4_topo file, and
    within
46          # the config folder, we require one folder for each host, named as the host
    itself.
47          # with the corresponding daemons, zebra.conf, ripd.conf and ripngd.conf files
48          quaggaBaseConfigPath = selfPath + '/configs/'
49
50          # Initializing local variables
51          netHosts = []
52          NodeList = []
53
54          # List of all hosts in the network.
55          # Note that each node requires at least one IP address to avoid
56          # Mininet's automatic IP address assignment
57          netHosts.append(NetworkHosts(name='h1', IP='10.10.11.10/24', DG='via
    10.10.11.1'))
58          netHosts.append(NetworkHosts(name='h2', IP='10.10.12.20/24', DG='via
    10.10.12.2'))
59          netHosts.append(NetworkHosts(name='r1', IP='10.10.11.1/24', DG=''))
60          netHosts.append(NetworkHosts(name='r2', IP='10.10.12.2/24', DG=''))
61          netHosts.append(NetworkHosts(name='r3', IP='10.10.23.3/24', DG=''))
62          netHosts.append(NetworkHosts(name='r4', IP='10.10.14.4/24', DG=''))
63          netHosts.append(NetworkHosts(name='r5', IP='10.10.25.5/24', DG=''))
64
65          for host in netHosts:
66            # We create a list of node names
67            NodeList.append(host.name)
68            if host.name in ['h1','h2']:
69                # We configure PCs with default gateway and without quagga service
70              AddPCHost = self.addHost(name=host.name, ip=host.IP, defaultRoute=host.DG,
     hostname=host.name,  privateLogDir=True, privateRunDir=True, inMountNamespace=True
    , inPIDNamespace=True, inUTSNamespace=True)
71          else :
72              # We configure routers with quagga service without default gateway
73              AddQuaggaHost = self.addHost(name=host.name, ip=host.IP, hostname=host.
    name,  privateLogDir=True, privateRunDir=True, inMountNamespace=True,
    inPIDNamespace=True, inUTSNamespace=True)
74              # We setup Quagga service and path to config files
75              # Note that we require one folder for each host, named as the host itself
76              quaggaSvcConfig = {'quaggaConfigPath': quaggaBaseConfigPath + host.name}
77              self.addNodeService(node=host.name, service=quaggaSvc, nodeConfig=
    quaggaSvcConfig)
78
79          # Adding switches to the network, we specify OpenFlow v1.3 for better IPv6
    multicast support
80          SW1 = self.addSwitch('SW1', protocols='OpenFlow13')
81          SW2 = self.addSwitch('SW2', protocols='OpenFlow13')
82          SW3 = self.addSwitch('SW3', protocols='OpenFlow13')
83          SW4 = self.addSwitch('SW4', protocols='OpenFlow13')
84          SW5 = self.addSwitch('SW5', protocols='OpenFlow13')
85          SW6 = self.addSwitch('SW6', protocols='OpenFlow13')
86          SW7 = self.addSwitch('SW7', protocols='OpenFlow13')
87          # We add links between switches and routers according to Fig.1 of Lab 4
88          info( '*** Creating links\n' )
89          self.addLink( SW1, NodeList[0], intfName2='h1-eth1' )
90          self.addLink( SW1, NodeList[2], intfName2='r1-eth3' )
91          self.addLink( SW2, NodeList[1], intfName2='h2-eth1' )
92        self.addLink( SW2, NodeList[2], intfName2='r1-eth1' )
93        self.addLink( SW2, NodeList[3], intfName2='r2-eth1' )
94        self.addLink( SW3, NodeList[3], intfName2='r2-eth2' )
```

```
95        self.addLink( SW3, NodeList[4], intfName2='r3-eth1' )
96        self.addLink( SW4, NodeList[2], intfName2='r1-eth2' )
97        self.addLink( SW4, NodeList[5], intfName2='r4-eth1' )
98        self.addLink( SW5, NodeList[3], intfName2='r2-eth3' )
99        self.addLink( SW5, NodeList[6], intfName2='r5-eth1' )
100       self.addLink( SW7, NodeList[4], intfName2='r3-eth2' )
101       self.addLink( SW7, NodeList[6], intfName2='r5-eth2' )
102       self.addLink( SW6, NodeList[5], intfName2='r4-eth2' )
103       self.addLink( SW6, NodeList[6], intfName2='r5-eth3' )
```

2.2.2 THE SIMULATION SCRIPT

We can see the example in Script 2. The remarks for this script are the following:

- MiniNExT documentation suggests to patch the `isShellBuiltin` module as it might cause problems when executing bash code from MiniNExT terminal. This code is presented in lines $17-20$.
- As opposed to a common Mininet script, in MiniNExT we use the MiniNExT() constructor to build the network. Check lines $36, 55$.
- Although you can execute any command from the prompt (`mininext>`), it is useful to have separate terminal windows for each router. The code is presented in lines $76-77$.
- The `stopNetwork()` function cleans properly all log files when the script is stopped. To change that behavior (*i.e.,* to keep the logs after you exit the script), you have to comment lines $92-97$.

Script 2: Calling the virtual environment with MiniNExT

```
1  #!/usr/bin/python
2
3  """
4  This script creates the network environment for Lab4:
5  - 5 routers, 7 switches and 2 hosts interconnected according to Fig 1 of Lab4
6  - Quagga service enabled in all routers
7  - IPv4 and IPv6 addressing given via zebra
8  - XTerm window launched for all devices.
9  """
10 # Needed to patch Mininet's isShellBuiltin module
11 import sys
12
13 # Run commands when you exit the python script
14 import atexit
15
16 # patch isShellBuiltin (suggested by MiniNExT's authors)
17 import mininet.util
18 import mininext.util
19 mininet.util.isShellBuiltin = mininext.util.isShellBuiltin
20 sys.modules['mininet.util'] = mininet.util
21
22 # Loads the default controller for the switches
23 # We load the OVSSwitch to use openflow v1.3
24 from mininet.node import Controller, OVSSwitch
25
26 # Needed to set logging level and show useful information during script execution.
27 from mininet.log import setLogLevel, info
28
29 # To launch xterm for each node
30 from mininet.term import makeTerms
31
32 # Provides the mininext> prompt
```

```
from mininext.cli import CLI

# Primary constructor for the virtual environment.
from mininext.net import MiniNExT

# We import the topology class for Lab4
from lab4_topo import Lab4Topo

# Variable initialization
net = None
hosts = None


def run():
    " Creates the virtual environment, by starting the network and configuring debug
    information "
    info('** Creating an instance of Lab4 network topology\n')
    topo = Lab4Topo()

    info('** Starting the network\n')
    global net
    global hosts
    # We use mininext constructor with the instance of the network, the default
    controller and the openvswitch
    net = MiniNExT(topo, controller=Controller, switch=OVSSwitch)
    net.start()

    info('** Executing custom commands\n')
    ################################################
    # Space to add any customize command before prompting command line
    # We provide an example on how to assign IPv6 addresses to hosts h1 and h2 as they
    # are not configured through Quagga
    # If required, you can add any extra logic to it

    # We gather only the hosts created in the topology (no switches nor controller)
    hosts = [ net.getNodeByName( h ) for h in topo.hosts() ]
    info('** Adding IPv6 address to hosts\n')
    for host in hosts:
        if host.name is 'h1':
            host.cmd('ip -6 addr add 2001:1:0:11::10/64 dev h1-eth1')
            host.cmd('ip -6 route add default via 2001:1:0:11::1')
        elif host.name is 'h2':
            host.cmd('ip -6 addr add 2001:1:0:12::20/64 dev h2-eth1')
            host.cmd('ip -6 route add default via 2001:1:0:12::2')

    info('** Enabling xterm for all hosts\n')
    makeTerms( hosts, 'node' )


    ################################################
  # Enable the mininext> prompt
    info('** Running CLI\n')
    CLI(net)

# Cleanup function to be called when you quit the script
def stopNetwork():
    "stops the network, cleans logs"

    if net is not None:
        info('** Tearing down Quagga network\n')
        # For sanity, when leaving the python script, we clean the logs again
        info('** Deleting logs and closing terminals for hosts\n')
```

```
93          for host in hosts:
94              if host.name in ['h1','h2']:
95                  pass
96              else :
97                  host.cmd( "sh configs/%s/clean.sh" ) % host.name
98
99          # This command stops the simulation
100         net.stop()
101
102 if __name__ == '__main__':
103     # Execute the cleanup function
104     atexit.register(stopNetwork)
105     # Set the log level on terminal
106     setLogLevel('info')
107
108     # Execute the script
109     run()
```

As mentioned previously, both scripts are available in moodle: The topology script is `lab4_topo.py` and the running script is `lab4_network.py`.

We are ready to start using the scripts. Set the quagga service to NOT start automatically and start the simulation environment.

Q1/ From the MiniNExT prompt, copy the output of the *net* command and check that the network was built as it is shown in Figure 1. Can you ping between all virtual routers?. Explain why you can or cannot.

```
h1 h1-eth1:SW1-eth1
h2 h2-eth1:SW2-eth1
r1 r1-eth3:SW1-eth2 r1-eth1:SW2-eth2 r1-eth2:SW4-eth1
r2 r2-eth1:SW2-eth3 r2-eth2:SW3-eth1 r2-eth3:SW5-eth1
r3 r3-eth1:SW3-eth2 r3-eth2:SW7-eth1
r4 r4-eth1:SW4-eth2 r4-eth2:SW6-eth1
r5 r5-eth1:SW5-eth2 r5-eth2:SW7-eth2 r5-eth3:SW6-eth2
SW1 lo:  SW1-eth1:h1-eth1 SW1-eth2:r1-eth3
SW2 lo:  SW2-eth1:h2-eth1 SW2-eth2:r1-eth1 SW2-eth3:r2-eth1
SW3 lo:  SW3-eth1:r2-eth2 SW3-eth2:r3-eth1
```

[A1]

3 HOST NETWORK CONFIGURATION WITH QUAGGA

Each virtual router is, in essence, the same as the physical Linux machine thus they can be configured the same way. This means that you could reuse the same set of commands that you used for previous labs, to configure network interfaces, to monitor their states, to inspect the contents of the routing tables, etc.

However, instead of this set of Linux networking commands, in this lab you will be using the tool suite from *Quagga*. An advantage of Quagga is that its commands represent a subset of commands used to configure Cisco networking equipment. Thus, documentation can be found on the Quagga website (*www.nongnu.org/quagga/*), but good references can also be found on the Cisco website.

3.1 CONFIGURING NETWORK INTERFACES

It is time to put in practice some of the concepts introduced in Section 2.1. For this purpose, we will be using the scenario from Figure 1. In principle, you could create a topology without Quagga, launch a terminal window for each router and configure its network interfaces using the `ip addr` command (following the scheme shown in Figure 1). However, in this lab, we will be using the `zebra` daemon instead.

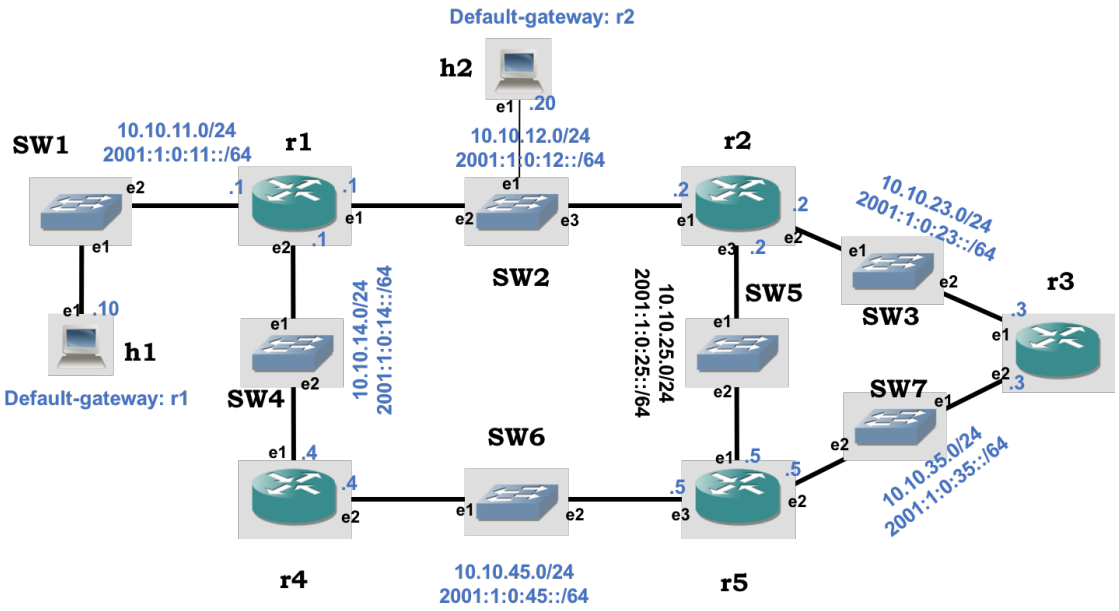

Figure 1: Lab 4 topology

Note: Never try to configure the network interfaces on a machine using both the `ip addr` command and *zebra* daemon, as interaction between the two is not always clear and the outcome is uncertain.

3.1.1 Configuring interfaces using configuration files

Before running your topology script, a configuration file must be created and edited at least for the `zebra` process. Script 3 is an example of such a file written for the router `r1`. This file can be found among setup files available on Moodle. In the terminal, enter the following command to open the configuration file.

Script 3: Example of a zebra.conf file

```
!
! Zebra configuration file for r1
!
hostname r1
password quagga
enable password quagga

log file /home/lca2/Desktop/shared/labs/lab4/configs/r1/logs/zebra.log
debug zebra packet

ipv6 forwarding

interface r1-eth1
no shutdown
ip address 10.10.12.1/24
ipv6 address 2001:1:0:12::1/64

interface r1-eth2
no shutdown
ip address 10.10.14.1/24
ipv6 address 2001:1:0:14::1/64

interface r1-eth3
no shutdown
```

```
ip address 10.10.11.1/24
ipv6 address 2001:1:0:11::1/64


line vty
```

```
leafpad /<path-to-r1's files>/zebra.conf
```

Edit the configuration file to look like Script 3 and save the changes. Let's examine the content of this configuration file:

- `!`: the lines starting with `!` are comments, they are ignored;
- `password`: this will be password used to log in to the zebra process to view the changes
- `enable password`: this will be password used to enable "on the fly" configuration of zebra process
- `log file`: Allows you to specify the file to which zebra related information is logged (adding, deleting routes in principle). **Make sure you write the full path to the file as this could prevent the Quagga service from starting**;
- `debug zebra packet`: more detailed debugging, *i.e.,* allows you to see when routes are added or deleted from the routing table;
- `ipv6 forwarding`: This instructs the zebra process to enable IPv6 routing on the virtual router.
- `interface <interface name>`: this command starts the interface configuration mode
- `ip address`: assigns an IPv4 address to the selected interface
- `ipv6 address`: assigns an IPv6 address to the selected interface
- `line vty`: enables *telnet* access to the process.

Don't forget to configure *zebra=yes* in the `daemons` file before starting the quagga service. To start the quagga service, just type from any router terminal:

```
/etc/init.d/quagga start
```

We already created all the configuration scripts for the routers $r1$, $r2$ and $r4$ and you can find them on Moodle too. The creation of the configuration scripts for the routers $r3$ and $r5$ is left to you.

You should now start configuring the interfaces on the routers (assignment of IP addresses). Thus, on all your `daemons.conf` file, make sure that only the zebra process is enabled, all other processes should be set to "no".

Note: Make sure that the permissions for the `zebra.log` files allow you to read-write them. To make a script read-write for anyone, you can type the following command in the terminal window (running with root privileges):

```
chmod 666 <script name>
```

3.1.2 QUAGGA MONITORING MODE

Start the `quagga` service in all routers (i.e, you can set Quagga to start automatically in the `lab4_topo.py` script). In order to monitor the activity of a running Quagga process, say `zebra`, you can enter the monitoring mode by connecting to it using

```
telnet localhost zebra
```

Let us see what information can be obtained now. To inspect the contents of the IPv4 and IPv6 routing tables type `show ip route` and `show ipv6 route` respectively. At all times, you can view the entire list of the commands at your disposal using `list`.

Q2/ What subnets can be found in the two routing tables on $r1$ at this point? Explain.

[A2]
10.10.11.0/24 is directly connected, r1-eth3
10.10.12.0/24 is directly connected, r1-eth1
10.10.14.0/24 is directly connected, r1-eth2
127.0.0.0/8 is directly connected, lo

::1/128 is directly connected, lo
2001:1:0:11::/64 is directly connected, r1-eth3
2001:1:0:12::/64 is directly connected, r1-eth1
2001:1:0:14::/64 is directly connected, r1-eth2
fe80::/64 is directly connected, r1-eth2

Next, check the status of the network interfaces on router $r3$ by typing `show interface`.

Q3/ How many interfaces with IP addresses are shown in the `show interface` command? Write those IPv4 and IPv6 addresses.

[A3]
There are three interfaces:

Interface lo is up, line protocol detection is disabled
  index 1 metric 1 mtu 65536
  flags: <UP,LOOPBACK,RUNNING>
  inet 127.0.0.1/8
  inet6 ::1/128
Interface r3-eth1 is up, line protocol detection is disabled
  index 241 metric 1 mtu 1500
  flags: <UP,BROADCAST,RUNNING,MULTICAST>

To view the current running configuration, you need to first enable the configuration mode using

```
enable
```

Finally, inspect the running configuration of the router r3 using the `show running-config` command.

3.1.3 "ON THE FLY" CONFIGURATION

Let us now consider how a running configuration can be modified "on the fly", without changing the configuration files. Remember that this is not the recommended approach. Here we will consider an example where "on the fly" configuration could be useful.

You might have noticed that the *zebra* configuration file for the router $r4$ has one line commented out, *i.e.,* the interface `r4-eth1` is not assigned an IPv4 address. You could of course stop the *quagga* service on this router, modify the line in the configuration file and restart the service. However, we will fix this problem

without stopping the service. For this purpose, we have to enter Quagga configuration mode on the router $r4$ and enter the *interface configuration mode*.

To enter the *interface configuration mode* follow the steps below:

- First, enter the zebra monitoring mode using command `telnet localhost zebra`.
- Next, enable the configuration mode by using `enable`.
- Next, move to the *configuration mode* by typing the command `configure terminal`. The cursor remains the same, but the name before the cursor changes to `r4(config)`.
- Finally, in the *configuration mode* type `interface r4-eth1`. The name before the cursor changes to `r4(config-if)`, which indicates that you entered the *interface configuration mode*.

Now, you can assign an IPv4 address to the interface `r4-eth1`, by using the same command that is used in the configuration file for this purpose. Simply type `ip address 10.10.14.4/24`. Exit the *interface configuration mode* and the *configuration mode*, by typing `exit` twice and verify that the modification took effect with the `show interface` command.

Before leaving this section, make sure to change the `zebra.conf` file of $r4$, to remove the comment on interface `r4-eth1` as "on the fly" configuration does not save changes to the configuration file.

3.2 CONFIGURING RIPV2

Similarly to the `zebra` process, the `ripd` process can be configured by editing a configuration file or "on the fly", as illustrated in Section 3.1.3. Note that the `zebra` process must be enabled with the `ripd` process in the `daemons` configuration file, as they work together.

Like the `zebra` configuration files, we are providing the "ready-to-use" `ripd.conf` configuration files. However, they are available only for the routers $r1$, $r2$ and $r4$. Similar to configuration for zebra, these files should be created in the same folder as the `zebra.conf`. We left the creation of the `ripd.conf` configuration files for the routers $r3$ and $r5$ to you.

Below, we explain the steps needed to configure RIPv2 on a router. These steps should allow you to understand the content of the `ripd.conf`configuration files on the routers $r1$, $r2$ and $r4$ and to create similar files for the routers $r3$ and $r5$.

The inevitable steps when configuring the RIP protocol on a router are:

- enabling RIP, and choosing its version;
- choosing which networks should be advertised via RIP;
- specifying the interfaces that take part in the exchange of routing information.

Some optional steps include debugging, logging, route filtering, etc. We give an example of `ripd.conf` configuration file in Script 4.

Script 4: Example of a ripd.conf file

```
hostname r1
password quagga
enable password quagga

log file /home/lca2/Desktop/shared/labs/lab4/configs/r1/logs/ripd.log
!
!log stdout
debug rip events
debug rip packet

```

```
router rip
version 2

redistribute connected

network 10.10.12.0/24
network 10.10.14.0/24

line vty
```

Let's have a closer look at the commands in the file above (you are already familiar with some of them):

- `log file`: Same as with `zebra`, it specifies the file to which the RIP related information is logged;
- `debug rip events`: less detailed debugging, *i.e.,* only the coarse events, such as sending and receiving RIP updates, can be seen;
- `debug rip packet`: more detailed debugging, *i.e.,* allows you to see the content of the sent and received RIP updates;
- `router rip`: enables the *ripd* process;
- `version`: allows you to specify the version of RIP the *ripd* process will run;
- `network`: You will learn about this command in Section 4.1
- `redistribute`: enables the announcement of prefixes learned from a given source. The syntax is:

```
redistribute protocol
```

where `protocol` denotes the source from which the prefixes have been learned. It can be:

(1) a routing protocol: such as `bgp` or `ospf` (*e.g.,* if you want to redistribute the routes from these protocols into RIP),
(2) `static`: if you want to redistribute the static routes
(3) `connected`: if you want to announce only the prefixes the router learned from the configuration of its own interfaces (*i.e.,* redistribution of the directly connected networks).

Using the example configuration file shown above and the address scheme in Figure 1, create the *ripd* configuration files for the routers $r3$ and $r5$ (configuration files for $r1$, $r2$ and $r4$ can be downloaded from Moodle). Make sure that both debugging modes are enabled for RIP (*i.e.,* `debug rip events` and `debug rip packet`). On the `daemons` scripts, make sure you have *zebra* and *ripd* processes enabled on all routers.

Make sure that you start the virtual routers in the following order: $r3$, $r4$, $r5$, $r1$, then wait around $10s$ before starting the `quagga` service on $r2$.

Q4/ Open wireshark on router $r1$. By looking at the exchange of packets, how is the routing information between routers exchanged (*i.e.,* by using broadcast, unicast or multicast)?. Open the */configs/logs/ripd.log* file on router $r1$ and compare to what you see in wireshark. Copy the line from the log file that confirms your observation in Wireshark..

It is using an ipv4 multicast address (224.0.0.9) with a RIPv2 payload. We can confirm this b looking at the logs and by knowing that 224.0.0.0/4 is reserved for multicast. By looking on wikipedia we can learn that 224.0.0.9 is the RIPv2 mutlicast group address for all RIPv2 aware routers on the network segment.

Example of line confiming the above:
2016/11/23 10:51:21 RIP: multicast join at r1-eth1
2016/11/23 10:51:21 RIP: multicast request on r1-eth1
2016/11/23 10:51:21 RIP: rip_send_packet 10.10.12.1 > 224.0.0.9 (r1-eth1)

[A4]

Q5/ Check the */configs/logs/ripd.log* file on router $r1$. What is the time (t_1) when $r1$ receives the first routing update from $r4$ and what are the networks that are advertised?

t1 = 10:51:21
2016/11/23 10:51:21 RIP: RECV packet from 10.10.14.4 port 520 on r1-eth2
2016/11/23 10:51:21 RIP: RECV RESPONSE version 2 packet size 84
2016/11/23 10:51:21 RIP: 10.10.23.0/24 -> 0.0.0.0 family 2 tag 0 metric 3
2016/11/23 10:51:21 RIP: 10.10.25.0/24 -> 0.0.0.0 family 2 tag 0 metric 2
2016/11/23 10:51:21 RIP: 10.10.35.0/24 -> 0.0.0.0 family 2 tag 0 metric 2
2016/11/23 10:51:21 RIP: 10.10.45.0/24 -> 0.0.0.0 family 2 tag 0 metric 1

We can see that the router r4 advertised four networks
[A5] - 10.10.23.0/24

Q6/ Check again the same file on router $r1$. What is the time (t_2) when $r1$ receives the first routing update from $r2$ and what are the networks that are advertised?

t2 = 10:52:00
2016/11/23 10:52:00 RIP: RECV packet from 10.10.12.2 port 520 on r1-eth1
2016/11/23 10:52:00 RIP: RECV RESPONSE version 2 packet size 84
2016/11/23 10:52:00 RIP: 10.10.23.0/24 -> 0.0.0.0 family 2 tag 0 metric 1
2016/11/23 10:52:00 RIP: 10.10.25.0/24 -> 0.0.0.0 family 2 tag 0 metric 1
2016/11/23 10:52:00 RIP: 10.10.35.0/24 -> 0.0.0.0 family 2 tag 0 metric 2
2016/11/23 10:52:00 RIP: 10.10.45.0/24 -> 0.0.0.0 family 2 tag 0 metric 2

We can see that the router r2 advertised networks 10.10.23.0/24, 10.10.25.0/24,
[A6] 10.10.35.0/24 and 10.10.45.0/24

Q7/ Now open the */configs/logs/zebra.log* file on router $r1$. Find entries that correspond to time t_1 ($\pm 1s$) and look for ZEBRA_IPV4_ROUTE_ADD messages? How many are there? Explain.

2016/11/23 10:51:21 ZEBRA: zebra message received [ZEBRA_IPV4_ROUTE_ADD] 19
2016/11/23 10:51:21 ZEBRA: zebra message received [ZEBRA_IPV4_ROUTE_ADD] 19
2016/11/23 10:51:21 ZEBRA: zebra message received [ZEBRA_IPV4_ROUTE_ADD] 19
2016/11/23 10:51:21 ZEBRA: zebra message received [ZEBRA_IPV4_ROUTE_ADD] 19

There is four. Indeed, r1 receives an update from r4 with network the four networks listed above in Q6. Given that r4 offers fewer hop than before (infinite), four entry trough r4 for are added.

[A7]

Q8/ Now check the */usr/local/quagga/zebra.log* file on router $r1$ at time t_2 ($\pm 1s$) and look for ZEBRA_IPV4_ROUTE_ADD messages? How many are there? Explain.

[A8]
2016/11/23 10:52:00 ZEBRA: zebra message received [ZEBRA_IPV4_ROUTE_DELETE] 19
2016/11/23 10:52:00 ZEBRA: zebra message received [ZEBRA_IPV4_ROUTE_ADD] 19
2016/11/23 10:52:00 ZEBRA: zebra message received [ZEBRA_IPV4_ROUTE_DELETE] 19
2016/11/23 10:52:00 ZEBRA: zebra message received [ZEBRA_IPV4_ROUTE_ADD] 19

Here two routes are added. The one for reaching 10.10.23.0/24 and 10.10.25.0/24. Indeed if we look at the Q5 and Q6 the metric decrease if the routes go through r2 and thus the routes are updated. The route for reaching 10.10.45.0/24 is not modified because going through r4 require fewer hop than going trough r2. And the route to reach 10.10.35.0/24 is not modified because the metric is the same trough r4 and r2.

3.2.1 MONITORING COMMANDS

We have seen before that we can use command `show ip route` to display the IPv4 routing table. However, at this point we should introduce yet another concept, i.e. RIP database (RIP Routing Information Base). RIP database is the place where networks known to RIP are stored. They are candidates for becoming part of the IPv4 routing table. We might have other routing protocols with alternative routes for the same networks. The decision which routing protocol to choose is made based on the routing protocol preference. In our case, we have only RIP running in our network. As a result all the entries from the RIP database will become part of the routing table.

To inspect the RIP database (that contains the routes known to RIP), you can enter to Quagga configuration mode. This is done in the same way as in the case of *zebra* process (*i.e.,* `telnet localhost ripd`). After that, you can display the content of the RIP database using the command:

```
show ip rip
```

Q9/ Write the content of the RIP database on the router $r5$. Can you infer the value of the *update timer* from the RIP database? If so, what is its value? Is it constant? In the case of the network prefixes that are not directly connected to the router (learned via RIP), what does the column *Time* represent?

[A9]
	Network	Next Hop	Metric	From	Tag	Time
R(n)	10.10.11.0/24	10.10.45.4	3	10.10.45.4	0	02:49
R(n)	10.10.12.0/24	10.10.25.2	2	10.10.25.2	0	02:43
R(n)	10.10.14.0/24	10.10.45.4	2	10.10.45.4	0	02:49
R(n)	10.10.23.0/24	10.10.35.3	2	10.10.35.3	0	02:41
C(i)	10.10.25.0/24	0.0.0.0	1	self	0	
C(i)	10.10.35.0/24	0.0.0.0	1	self	0	
C(i)	10.10.45.0/24	0.0.0.0	1	self	0	

1) Yes we can using the Time column
2) By repeating the command, we can see that the timer decrease from ~ 3:00 to ~2:30 and then start again at ~3:00. Thus we can infer that r5 receive an update approximatively every 30 secondes from it's neighbors .
3) No, it is not constant. It seems to be randomized.
4) The time column represents the current timeout time for the corresponding entry. By typing

3.3 CONFIGURING RIPNG

RIPng (RIP next generation), defined in RFC 2080, is an extension of RIPv2. It is an IPv6 reincarnation of the RIPv2 protocol. RIPng sends updates on UDP port 521 using the multicast group FF02::9.

As multiple routing protocols can run in parallel on the majority of routers, Quagga allows you to configure RIPng in parallel to RIPv2. Just like `zebra` and `ripd` processes, the `ripngd` process can be configured by editing a configuration file or via telnet. Again, the *zebra* process must be enabled along with the *ripngd* process in the `daemons` configuration file.

As in the case of `ripd`, we are providing the *ripngd* configuration files for the routers $r1$, $r2$ and $r4$. It is left to you to create similar files for the routers $r3$ and $r5$. Like in the case of RIPv2, you should advertise via RIPng all the directly connected IPv6 subnets.

The main configuration steps required to configure RIPng on a router do not differ from those required to configure RIP. In terms of commands used to configure *ripngd*, very few differ from those used to configure *ripd*. The few exceptions are listed below:

- `router ripng`: this command is used instead of `router rip` to declare the routing protocol
- `version`: this command does not exist in the case of RIPng
- `debug ripng events` and `debug ripng packet`: used instead of the `debug rip events` and `debug rip packet` commands

You should make sure that you log *ripng* events/packets into a separate file than the *ripd* events/packets.

3.3.1 MONITORING COMMANDS

To inspect the RIPng database (RIPng Routing Information Base), which contains the routes to IPv6 subnets known to RIPng, you should execute `telnet ::1 ripngd` command. To display the content of the RIPng database type:

```
show ipv6 ripng
```

Finally, to display the contents of the IPv6 routing table type the following in the terminal.

```
show ipv6 route
```

Q10/ Check the contents of the IPv6 routing table on router $r5$. Explain what does number between the brackets ([ ]) mean, that is present in all non-directly connected routes. Note that this number is also present in the IPv4 routing table.

[A10]
First we can see that the next hop is always a link local address (fe80::/64). Also the number in bracket indicates [administrative distance of the information source, metric for the route]

cf: cisco doc

R>* 2001:1:0:11::/64 [120/3] via fe80::c848:49ff:fe96:43d3, r5-eth1, 00:12:53
R>* 2001:1:0:12::/64 [120/2] via fe80::c848:49ff:fe96:43d3, r5-eth1, 00:12:53
R>* 2001:1:0:14::/64 [120/2] via fe80::dc40:2aff:fe32:51d0, r5-eth3, 00:12:53
R>* 2001:1:0:23::/64 [120/2] via fe80::c848:49ff:fe96:43d3, r5-eth1, 00:12:53
C>* 2001:1:0:25::/64 is directly connected, r5-eth1

4 RIP AND RIPNG PLAYGROUND

In this section you will be confronted with a number of situations that might arise in a RIPv2 or a RIPng network. Answering the questions will allow you to better understand RIPv2, RIPng and dynamic routing in general.

Notes: It might be a good idea to use two terminals, one for the Quagga process commands and one for Linux commands. To have a second `xterm` window for a particular virtual router, you can type: `xterm <hostname>` in the `mininext>` prompt.

4.1 UNDERSTANDING NEIGHBORS IN RIP

In this section we will learn about how do neighbors interact within RIPv2. Make sure RIPv2 and RIPng are running and configured properly before proceeding any further.

Q11/ If we added the `network 10.10.11.0/24` command to $r1$'s RIP configuration, would $h1$ be able to update its routing information with $r1$'s database?. Explain your answer

[A11] No, h1 is not a router. It does not have a routing table to update. It only know it's default gateway (r1).

Now, from $h1$, do a ping and a traceroute to the IPs on the interfaces `r2-eth1` and `r2-eth2` of $r2$. Take a note of the answers for a later comparison. From $r1$'s `ripd.conf` file, comment out (using "!" at the beginning of the line) the `network 10.10.12.0/24` command and restart the quagga service. Analyze the `show ip rip` command in $r2$, and focus on the routes advertised by $r1$. Open a Wireshark session in $r2$ and check for packets coming from $r1$.

Q12/ What's happening?

[A12]
R2:
R>* 10.10.11.0/24 [120/4] via 10.10.25.5, r2-eth3, 00:56:02
C>* 10.10.12.0/24 is directly connected, r2-eth1
R>* 10.10.14.0/24 [120/3] via 10.10.25.5, r2-eth3, 00:56:02
C>* 10.10.23.0/24 is directly connected, r2-eth2
C>* 10.10.25.0/24 is directly connected, r2-eth3
R>* 10.10.35.0/24 [120/2] via 10.10.23.3, r2-eth2, 01:07:09
R>* 10.10.45.0/24 [120/2] via 10.10.25.5, r2-eth3, 01:07:09
C>* 127.0.0.0/8 is directly connected, lo

Wait 4 mins, and one more time do a ping and traceroute to the IPs on the interfaces `r2-eth1` and `r2-eth2` of $r2$, sourcing from $h1$.

Q13/ Describe the difference (if any) between the first and the second try. Explain your results.

Before:
traceroute to 10.10.12.2 (10.10.12.2), 64 hops max
1 10.10.11.1 s 0.006ms 0.005ms 5.577ms
2 10.10.12.2 1.192ms 2.346ms 7.781ms

traceroute to 10.10.23.2 (10.10.23.2), 64 hops max r2-eth1
1 10.10.11.1 0.005ms 0.003ms 0.002ms
2 10.10.23.2 0.003ms 5.020ms 2.934ms

[A13] After:

Q14/ Conclude how does the RIP protocol sees the connection between $r1$ and $r2$

The RIP protocol is note "real time" indeed when r1 stops sending route information to r2. r2 has to wait for a timeout to recompute the right distance to 1 and update the route to r1 through r4. The connection is thus seen unilateraly by r1 and r2.

[A14]

Before going to the next subsection, uncomment the `network` command in the `ripd.conf` of $r1$ and restart the `quagga` service.

4.2 BROKEN LINK

Do the following section of actions. Stop and start the `quagga` service in the following order: $r1$, $r2$ and $r4$. It is assumed that all processes are already running on $r3$ and $r5$.

Display the content of the routing table and the RIP database of $r1$ (`show ip route` and `show ip rip` commands). You can write it down for later comparison.

To simulate broken link between $r1$ and $r2$, shutdown the *r2-eth1* interface on $r2$. To do this, type on $r2$ the following commands:

```
telnet localhost zebra
enable
configure terminal
interface r2-eth1
shutdown
```

Note that it is convenient to use the “on the fly configuration” here.

Q15/ Observe the changes in the routing table and the RIP database of $r1$. Explain what happens.

Before:
C>* 10.10.11.0/24 is directly connected, r1-eth3
C>* 10.10.12.0/24 is directly connected, r1-eth1
C>* 10.10.14.0/24 is directly connected, r1-eth2
R>* 10.10.23.0/24 [120/2] via 10.10.12.2, r1-eth1, 00:00:34
R>* 10.10.25.0/24 [120/2] via 10.10.12.2, r1-eth1, 00:00:34
R>* 10.10.35.0/24 [120/3] via 10.10.12.2, r1-eth1, 00:00:33
R>* 10.10.45.0/24 [120/2] via 10.10.14.4, r1-eth2, 00:00:32
C(r) 10.10.11.0/24 0.0.0.0 1 self (connected:1) 0
[A15] C(i) 10.10.12.0/24 0.0.0.0 1 self 0

After 4-5 minutes, bring the *r2-eth1* interface on $r2$ up using the command:

```
no shutdown
```

Q16/ Once the *r2-eth1* interface on $r2$ is up, wait for the things to converge and then check the content of $r1$'s routing table. Is it the same as in the beginning (before we brought down the *r2-eth1* interface on $r2$)?. Explain.

No, it is slightly different.

Before:
C(r) 10.10.11.0/24 0.0.0.0 1 self (connected:1) 0
C(i) 10.10.12.0/24 0.0.0.0 1 self 0
C(i) 10.10.14.0/24 0.0.0.0 1 self 0
R(n) 10.10.23.0/24 10.10.12.2 2 10.10.12.2 0 02:42
R(n) 10.10.25.0/24 10.10.12.2 2 10.10.12.2 0 02:42
R(n) 10.10.35.0/24 10.10.12.2 3 10.10.12.2 0 02:42
[A16] R(n) 10.10.45.0/24 10.10.14.4 2 10.10.14.4 0 02:44

4.3 MODIFIED LINK METRIC

Before starting this part, put the interface *r2-eth1* on $r2$ up again and wait for the routing tables to converge. RIP and RIPng are distance vector protocols. By default, the directly connected subnets are treated as having the distance equal to 1. Each additional "hop" (router) adds 1 to this distance metric. Nevertheless, RIP and RIPng offer you the possibility to modify the metric of an arbitrary link, changing the desirability of the routes that contain this link.

Let's assume we want to make $r2$ prefer the route to `2001:1:0:14::/64` via $r5$ and $r4$, instead of the route via $r1$. To fulfill the task we have two approaches:

- From $r1$, we can announce to $r2$ the route for `2001:1:0:14::/64` with an increased metric. To do this, we use a `route-map` in $r1$. The `route-map` will look at routes learned from interface `r1-eth2`, and set arbitrarily the metric to any value and then $r1$ will send these routes with the updated metric to its neighbors. The commands needed to configure a `route-map` (assuming we set the metric to 5) are:

```
route-map incLinkMetric permit 10
match interface r1-eth2
set metric 5
```

Then, in order to apply the `incLinkMetric` to the RIP instance of $r1$, we need to modify the `redistribute connected` command with

```
redistribute connected route-map incLinkMetric
```

.

- From $r2$, we can directly inject an offset in $r2$'s computation of the cost to reach to `2001:1:0:14::/64` via $r1$. To do this, we use the `offset-list` command in $r2$. In this case, $r2$ will receive the updates from $r1$, and it will add the desired offset in order to calculate the shortest-path tree. The commands needed (assuming we add an offset of 4 to any route learned from `r2-eth1` interface) are:

```
offset-list addExtraMetric in 4 r2-eth1
ipv6 access-list addExtraMetric permit any
```

Note that *offset-list* and *access-list* commands have to come before the *route-map* in the *ripngd* configuration file in order to avoid runtime errors.

Apply the first approach to $r1$'s config and restart its `quagga` service. Verify whether the desired routing from $r2$ to *2001:1:0:14::/64* is indeed put in place.

[A17] **17/** In the RIPng log file on $r2$ find the lines that correspond to the packets received after t_0 that contain information about network 2001:1:0:14:: . Are there some events in zebra log file that correspond to the times of reception of identified packets? How many changes of the $r2$ routing table occurred and why?.

```
ripndg.log
2016/11/20 04:47:45 RIPNG: RIPng packet received from fe80::ac14:faff:fe55:4f3b port 521
on r2-eth1
2016/11/20 04:47:45 RIPNG: RECV response version 1 packet size 44
2016/11/20 04:47:45 RIPNG:   2001:1:0:14::/64 metric 5 tag 0

zebra.log
2016/11/20 04:47:45 ZEBRA: zebra message received [ZEBRA_IPV6_ROUTE_DELETE] 37
2016/11/20 04:47:45 ZEBRA: zebra message received [ZEBRA_IPV6_ROUTE_ADD] 41

Before
  Network      Next Hop                  Via     Metric Tag Time
R(n) 2001:1:0:11::/64
               fe80::ac14:faff:fe55:4f3b   r2-eth1    2    0  02:52
```

If all the changes are done correctly a packet from $r2$ to 2001:1:0:14::4 (r4-eth1 interface of $r4$) should go through $r5$ (not $r1$). Now let's imagine that we want to enforce that packets in the opposite direction take the alternative route. Concretely, the packet sent from $r4$ to `2001:1:0:25::2` (r2-eth2 interface of $r2$) should go through $r1$ (not $r5$).

Q18/ Propose the two configurations for the two approaches, and specify on which router you would apply each approach

[A18]

ON R5:
redistribute connected route-map
incLinkMetric
route-map incLinkMetric permit 10
match interface r5-eth3
set metric 5

The second approach is to change r4:
offset-list addExtraMetric in 2 r4-eth2
ipv6 access-list addExtraMetric permit any

The other option would be to directly apply the configuration in $r4$, and add an offset to the metric of something equal or higher than 2 to all the routes coming from interface `r4-eth2` :

```
offset-list addExtraMetric in 2 r4-eth2
ipv6 access-list addExtraMetric permit any
```

4.4 PROCESS CRASH

At this point it is assumed that *zebra*, *ripd* and *ripngd* processes are running on all five routers and that the network is configured according to the scheme shown in Figure 1. Revert all the changes you did, i.e. all links should have distance metric of 1 again!

Now, kill only the *ripd* processes on routers $r2$ and $r5$. All other processes (*i.e., zebra* and *ripngd*) running on these two routers, as well as the *zebra*, *ripd* and *ripngd* processes on the remaining three routers, should continue running.

Observe the changes in the IPv4 routing table and the RIP database at routers $r1$ and $r4$, over the period of a few minutes. The right way to do this is to refresh (display) the content of both tables every $10 - 20$ seconds. Focus on the entries that contain routes to the prefixes that are directly connected to $r3$ and pay special attention to the `Time` column in the RIP database.

Q19/ After how much time do the prefixes for the subnets that are directly connected to $r3$ disappear from the routing table of $r1$ and $r4$? How do you explain this?

[A19]

After the expiration of a timer (3:00 minutes). The timer "time" can be seen in the column " Time".

Q20/ After how much time do the prefixes for the subnets that are directly connected to $r3$ disappear from

the RIP database of $r1$ and $r4$? Explain.

After the expiration of a timer (3:00 minutes) plus 2:00 minutes. This correspond to the hold of time found in the routing table.

[A20]

Q21/ Can you ping the IPv4 addresses of the $r3$'s ethernet interfaces from the router $r1$? How about the IPv6 addresses assigned to the same interfaces (use *ping ipv6* from Quagga configuration mode or *ping ipv6* if you are not in Quagga configuration mode)? Explain.

You can no long ping the interfaces of R5 via IPv4. Indeed, the r1 no longers now how to reach the specific subnet. On the other hand the interfaces of R5 are still reachable through IPv6. This comes from the fact that RIPv2 and RIPng are independent process and thus losing information in on routing table do not influence the other.

[A21]

5 SOFTWARE DEFINED NETWORKING

Software defined networking (SDN) is an approach to TCP/IP networking that allow network administrators to manage services through flexible interfaces provided by a control-layer. A typical SDN network consists of routers or switches interconnected, each of them with a "listener" daemon running in which they receive forwarding-decision rules, called flows, from one (or many) controller(s) using, for example, the OpenFlow protocol. The network administrators make changes to the controller and these changes are disseminated through the control-layer (typically another IP-based connection) so that appropriate routing/switching decisions can be applied. Note that in this section, we will work only with Mininet (and not MiniNExT) as we will apply flow-policy in switches (not in routers). SDN routing policies are only available with proprietary hardware and controller (Cisco, Huawei, Juniper, etc.)

The Mininet virtual environment used in this lab is compatible with SDN. In fact, the openVSwitch (OVS), which is the switch we have been used so far through the whole lab, is a software listener switch that is controlled using the OpenFlow protocol. The central controller that comes with Mininet has no special features or policies, which make the openVSwitches used in the virtual environment to behave as a basic L2-switch.

In order to explore SDN, we need a new controller. We will use the POX controller [1], which is an open-source controller written in Python and is already installed in the virtual machine. The configuration changes for using an external controller are minimum. Basically we need to call the `RemoteController` class by importing it from `mininet.node`, and then when we call the constructor `Mininet`, and we pass the attribute `controller=RemoteController`. With this configuration, the switches will look on the VM's loopback address (default port 6633) for the controller. To launch the POX controller, you need to type in a new shell a command like:

```
sudo  /pox/pox.py forwarding.l2_learning
```

The above command runs the POX controller with the `l2_learning` component. Components are "functionality add-ons" that are loaded to the POX controller, as the controller itself doesn't do much. In partic-

[1] https://github.com/noxrepo/pox

ular, the `l2_learning` component which comes as default component in Mininet's controller implementation, makes the OpenFlow switches to act as a "type" of layer-2 learning switch. The component learns MAC addresses, and the flows it installs are exact-matches on as many fields as possible. Therefore, for different TCP/UDP connections between two hosts, you will have different flows being installed.

You can specify multiple components for a controller instance. The POX controller comes with a handful of components that are located in the `~/pox/pox` folder of the VM. We have created a custom component (`lab4_component.py`), which is located in the `~/pox/ext` folder and will be used later in this section.

The topology for this section is composed of 5 switches and is depicted in Figure 2. The script `lab4_sdn.py` (available in Moodle) has been created for you to match the topology described. Note that in this section we use Mininet instead of MiniNExT as we don't have a requirement for PID namespaces.

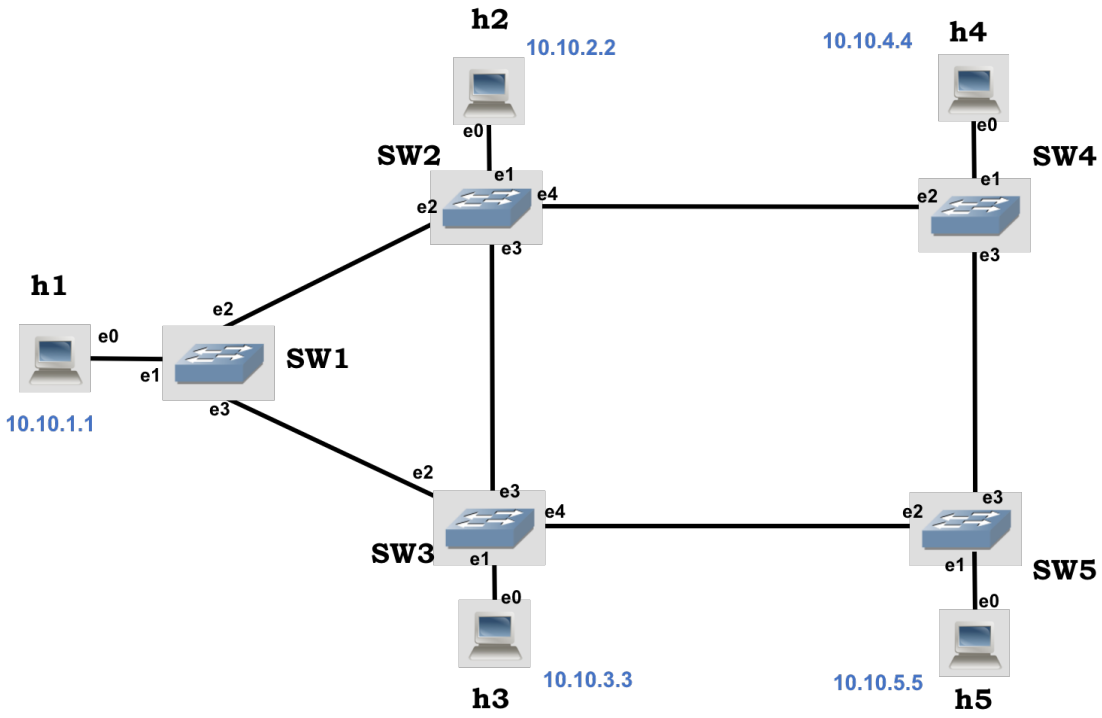

Figure 2: Lab 4 SDN topology

Q22/ How many subnets are there in Figure 2?. What would be the correct network mask for each IP address?

[A22] There is only one subnet. Indeed the intermediary systems are switched and thus works at the MAC layer.

The network mask for all IP is /21 or 255.255.248.0ns=output

5.1 Controlling the OpenFlow Switch in Standalone Mode

The OVS switch comes with the `ovs-ofctl` utility, which uses the terminal prompt to send basic OpenFlow messages, for basic configuration and retrieving important information such as installed flows, port information, dump statistics, etc. For configuration, a non-exhaustive list includes manually adding/deleting

flows, configuring flags of a particular interface, etc. To display a complete list of the commands (with a brief description of what it does), type the `ovs-ofctl -h` command from a terminal prompt. Note that Mininet also comes with a similar utility called `dpctl`, which has a short version of the `ovs-ofctl` utility. Unlike the former, `dpctl` is available through the `mininet>` prompt.

Let's analyze the behavior of the OVS switch in the absence of the controller. Run the `lab4_sdn.py` script. Now, from $h1$ try to make three pings to $h2$:

```
mininet>h1 ping -c 3 h2
```

As you probably thought, the pings were unsuccessful as the switches in the network don't know what to do with the incoming packet. Now, let's use the `ovs-ofctl` utility to help $h1$ and $h2$ to communicate. From a new terminal prompt (from Linux, not Mininet), type the following commands:

```
sudo ovs-ofctl add-flow SW1 in_port=1,actions=output:2
sudo ovs-ofctl add-flow SW2 in_port=2,actions=output:1
```

From the commands above:

- `add-flow`: refers to the command we are sending to the OVS switch, other options are *del-flow, dump-flows, dump-port, mod-port, etc.*
- `<switch>`: refers to the OVS switch we would like to send the message to. By default it resolves the names of the switches that are in the same machine, for others we need to specify IP address and TCP port.
- `in_port=1`: we tell the OVS switch on which port it should apply the flow to, in the inward direction.
- `actions=output:2`: we tell the OVS switch, what are we doing with packets matching the flow statement. In this case, we just send it out through a particular interface. Other options include flooding to all ports, setting a new *next-hop*, changing a particular field in the IP packet (priority, TTL, flags), etc.

Q23/ Apply the `ovs-ofctl` commands and test the ping command again. Does it work?. Explain what is happening and which commands (if any) would help you fix the problem. **Hint**: use wireshark to check for packet arrival

[A23] No, it still does not work. By looking at the Wireshark capture, we can see that ARP packet are sent but no reply is received. Indeed the return route is not configured.

sudo ovs-ofctl add-flow SW1 in_port=2,actions=output:1
sudo ovs-ofctl add-flow SW2 in_port=1,actions=output:2

5.2 CONTROLLING THE OPENFLOW SWITCH USING POX

Now it is time use a remote controller. Before continuing, make sure you erase all flows from $SW1$ and $SW2$ by issuing the command `ovs-ofctl del-flows <switch>` from a Linux's terminal window (as root).

Stop the simulation in Mininet and start the POX controller using a separate Linux's terminal window:

```
sudo python /home/lca2/pox/pox.py forwarding.l2_learning
```

For all cases, start the POX controller first, and then start Mininet. In this way, you can see the progress of all logs from the switches in the POX controller console, which will be helpful to answer many questions in this section.

With the controller started, the OVS switches should now have automatic flows installed for every packet and all hosts should be reachable. From the `mininet>` prompt do a `pingall` command.

Q24/ What is happening?. What error messages are shown in the POX controller?. Why do you think ping was unsuccessful?.

[A24] It not possible to ping at all. The POX controller throws a OpenFlow Error.

ERROR:openflow.of_01:[1c-a2-1a-b4-05-05|1285 6] OpenFlow Error:
[1c-a2-1a-b4-05-05|1285 6] Error: header:
[1c-a2-1a-b4-05-05|1285 6] Error: version: 1

Now, stop the POX controller by using `Ctrl + C` in the Linux terminal window. Next, let's explore a different complement. Before restarting the POX controller, add the `delay=2` attribute to the `net.start()` function in the SDN Mininet script (line 54). It should look something like this:

```
net.start(delay=2)
```

Now, start the POX controller using the `lab4_component.py` script:

```
sudo python /home/lca2/pox/pox.py lab4_component
```

Q25/ Wait around one minute and do again a `pingall` from Mininet. Keep a look at the logs from the controller. Does it work now?. Use the `ovs-ofctl show <switch>` command (from Linux's terminal) and try to discover how the POX component solves the previous problem (what changes are made by this script). Write the path that flows take to reach $h3$ from $h2$.

[A25] Yes, now it works!

From the ovs-ofctl command and the POX console output, we can see that the new script create a spanning tree and add the flows.

Keep a continuous ping between $h1$ and $h2$, and use the command `link SW1 SW2 down` from the `mininet>` prompt to bring down the link between SW1 and SW2.

Q26/ Enumerate all changes (if any) that happened to the network and identify all possible timeouts. Hint: use the logs from POX controller, Wireshark and the commands learned from last question.

We can see two timeouts. At the same time the ping stops working (host unreachable). The timeout corresponds to the time the timer of SW1 and SW2 related to the link SW1-SW2 reach zero. SW1 and SW2 determine that the link is down. After a few secondes the ping start again. Indeed a new flow going through SW3 is established.

[A26]

To return the network to normal operation, type `quit` from the `mininet>` prompt and relaunch the SDN python script.

Now, open the `lab4_component.py` script, located in the `~/pox/ext` folder (remember to use sudo). Uncomment the last two lines of the script to enable the new component `bypass` to be used in the POX Controller.

Launch again the simulation environment by starting first the POX Controller and then Mininet.

Q27/ Do a `pingall` command and report the differences (if any) as compared to your finding in **Q.25**. Bring down the link between $h2$ and $h3$ and report all differences (if any) as compared to your findings in **Q.26**

At first, the pingall command doesn't suffer any packet loss (just as Q25).
After bringing the link down, the pingall command now has a 10% packet loss, this means that the POX controller received a timeout but is not able to recover a new spanning tree i.e. the routes are not updated anymore. (which wasn't the case earlier)

[A27]