# Programming Assignment 6: Software Defined Internet Exchange Points: Instructions

## Introduction

In this exercise, you will learn how to configure and program a Software Defined Internet Exchange Point (SDX). An Internet Exchange Point (IXP) is a location on the network where different independently operated networks (sometimes also called autonomous systems, or domains) exchange traffic with each other.

There is growing interest in applying Software Defined Networking (SDN) could make some aspects of wide-area network management easier by giving operators direct control over packet-processing rules that match on multiple header fields and perform a variety of actions. Internet exchange points (IXPs) are a compelling place to start, given their central role in interconnecting many networks and their growing importance in bringing popular content closer to end users. The recent paper "SDX: A Software Defined Internet Exchange" describes some of this promise in greater detail, as well as the implementation of an SDX that you will be using to complete this assignment.

In this exercise, we'll explore one implementation of a software defined Internet exchange point (SDX). This implementation provides new programming abstractions allowing participants to create and run new wide area traffic delivery applications. SDX platform provides a scalable runtime that both, behaves correctly when interacting with BGP, and ensures that the applications do not interfere with each other.

Installation

SDX is not packaged with the course VM because it was developed more recently than when we distributed the VM. It also has several dependencies: Quagga, Mininext (an extension of Mininet). To complete this assignment, you'll want to update your course VM using the instructions that we have provided here.

Step 1. Install Quagga

Type the following command to install Quagga, a software router that we will use for the assignment.

```
$ sudo apt-get install quagga
```

Step 2. Install Mininext

Make sure that Mininext's dependencies are installed.

```
$ sudo apt-get install help2man python-setuptools
```

Clone miniNExT and install it.

```
$ git clone https://github.com/USC-NSL/miniNExT.git
$ cd miniNExT
$ git checkout 1.4.0
$ sudo make install
```

Step 3. Change Pyretic Branch

We need to change Pyretic's branch to get it to work with the SDX, as a couple of changes have been made to the latest Pyretic that interfere with SDX.

```
$ cd ~/pyretic
$ git checkout 34ff85547ecd59e9275034d83b63ddd93fac11d1
```

Note: Once you have completed the assignment, you may want to change your version of Pyretic back to the latest version. To do so, you should type "git checkout master" once you have completed the assignment.

Step 4. Install SDX

Clone the SDX installation into your Pyretic installation. Make sure that you checkout the `coursera-assignment` branch of the repository to complete this assignment.

```
$ cd ~/pyretic/pyretic/
$ git clone https://github.com/sdn-ixp/sdx.git
$ cd sdx
$ git checkout coursera-assignment
```

After cloning SDX, verify the presence of the directory `~/pyretic/pyretic/sdx`. It should have a number of subdirectories, including an `examples` directory.

# Walkthrough

## Overview

This part of the exercise allows you to get comfortable using the SDX software. You are not required to submit anything. All of the examples in SDX are organized in the directory called `~/pyretic/pyretic/sdx/examples`. We'll focus on the example `app_specific_peering_inboundTE` for the walkthrough.

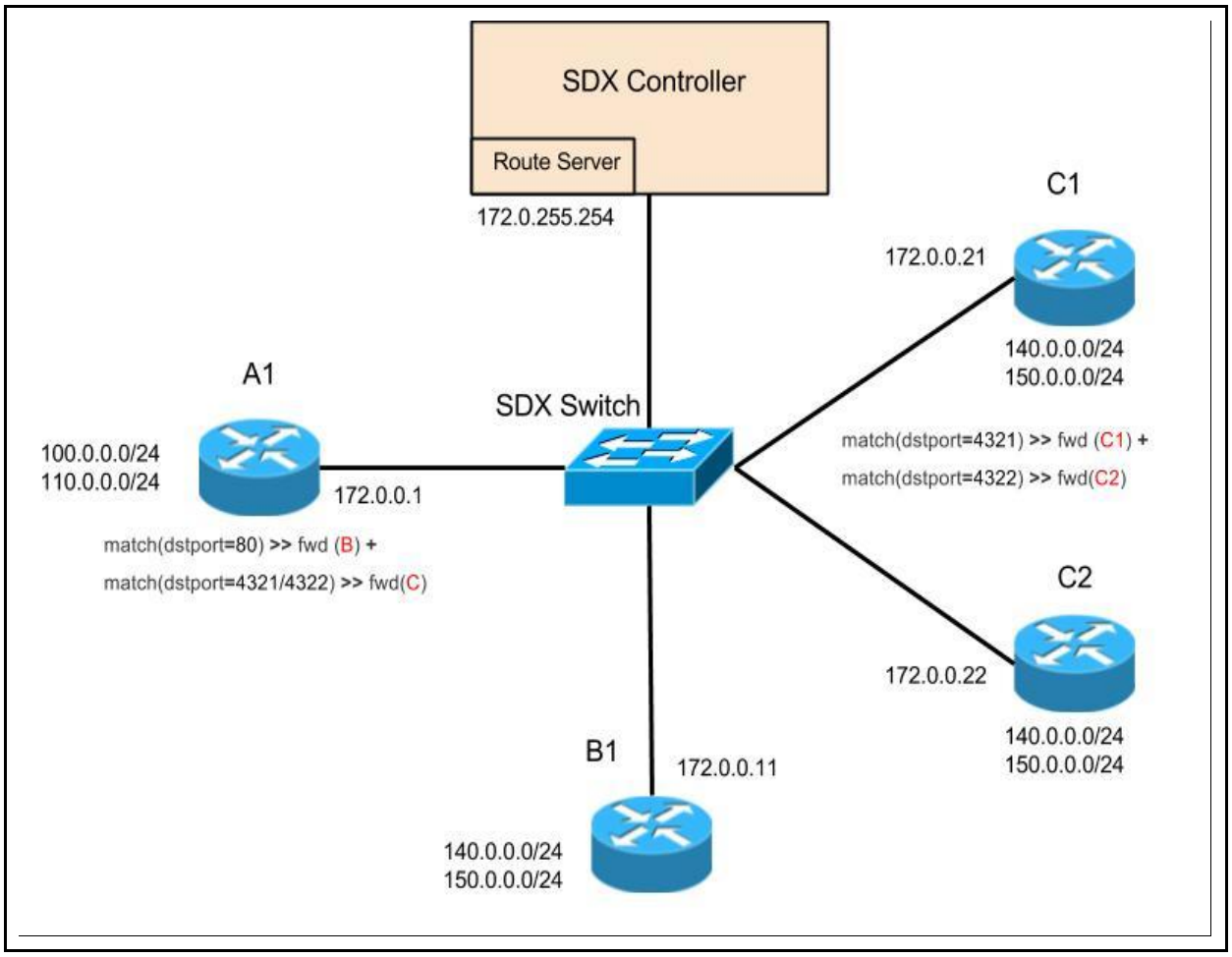


Figure 1. Example SDX topology.

172.* addresses refer to the IP addresses of the connected router interfaces. /24 IP prefixes are the routes that each router advertises. ASes A and C each have Pyretic policies, as shown. The setup consists of three participants, each representing a participating AS: A, B, and C. Participants A and B each have a single router. Participant C has two routers connected to the IXP. These routers are running the `zebra` and `bgpd` daemons, part of the Quagga routing engine. We are also using the `MiniNext` emulation tool to create this topology. MiniNext is a variant of Mininet that gives each node in the topology its own filesystem, thus allowing each node to run its own version of the routing software. The following sites have more information about MiniNext and Quagga.

## Understanding the SDX Setup

The example setup has two directories: `controller` and `mininet`. The `controller` directory has configuration files pertaining to SDX controller; the `mininet` directory has information pertaining to topology configuration, as well as information for configuring communication with the route server.

### Configuring Topology and Routing Information in Mininext

We need to set up routers that run a routing engine to exchange BGP routes at the exchange point. Running a routing engine requires a particular filesystem that is not supported for the `Mininet` nodes. `MiniNext` enables a filesystem for its nodes enabling emulation of legacy switches, routers, etc.
The `mininet` directory has a Mininext script (`sdx_mininext.py`) and a directory containing the configuration of the Quagga router.
The following excerpt from `sdx_mininext.py` shows how each host is set up:

```
"Set Quagga service configuration for this node"
quaggaSvcConfig = \
{ 'quaggaConfigPath' : scriptdir + '/quaggacfgs/' + host.name }

"Add services to the list for handling by service helper"
services = {}
services[quaggaSvc] = quaggaSvcConfig

"Create an instance of a host, called a quaggaContainer"
quaggaContainer = self.addHost( name=host.name, ip=host.ip, mac=host.mac,
                                privateLogDir=True,
                                privateRunDir=True,
                                 inMountNamespace=True,
                                 inPIDNamespace=True)
self.addNodeService(node=host.name, service=quaggaSvc,
                                 nodeConfig=quaggaSvcConfig)


"Attach the quaggaContainer to the IXP Fabric Switch"
self.addLink( quaggaContainer, ixpfabric , port2=host.port)
```

Figure 2. Configuring a Quagga router from Mininet. (sdx_mininext.py)
The first part of the configuration simply tells Mininet where the configuration for Quagga is and adds a service to the service helper. The second part of the configuration creates an instance

of the host and then adds the quagga service to the host. Finally, we use the `addLink` function to add a link from the host running Quagga to the part of the topology running the SDX. Next, in a different part of `sdx_mininext.py`, we configure the SDX interfaces with `ifconfig`. Because we now need to do IP routing, we need to explicitly set the host and subnet information for each of those hosts.

```
print "Configuring participating ASs\n\n"
for host in hosts:
   if host.name=='a1':
   host.cmdPrint('sudo ifconfig lo:1 100.0.0.1 netmask 255.255.255.0 up')
       ...
   if host.name=='b1':
       host.cmdPrint('sudo ifconfig lo:140 140.0.0.1 netmask 255.255.255.0 up')
       ...
   ...
```

Figure 3. Configuring the host interfaces for the SDX setup from witin mininet. (sdx_mininext.py)
The SDX route server is based on ExaBGP and runs in the root namespace. We need to create an interface in the root namespace itself and connect it with the SDX switch.

```
" Add root node for ExaBGP. ExaBGP acts as route server for SDX. "
root = self.addHost('exabgp', ip = '172.0.255.254/16', inNamespace = False)
self.addLink(root, ixpfabric, port2 = 5)
```

**Figure 4. Adding the route server host, which we have named "exabgp", after the software on which the route server is based.** (sdx_mininext.py)
The `mininet` directory also has a `Quagga` config directory. We define BGP configuration for each of the participating `Quagga` router here. BGP configuration for participant A's router (`~/pyretic/pyretic/sdx/examples/app_specific_peering_inboundTE/mininet/quaggacfgs/a1/bgpd.conf`) looks like this:

```
router bgp 100
bgp router-id 172.0.0.1
neighbor 172.0.255.254 remote-as 65000
network 100.0.0.0/24
network 110.0.0.0/24
redistribute static
```

Figure 5. BGP routing configuration for participant A. (bgpd.conf)
This configuration indicates that this router has router-id `172.0.0.1`, A's AS number is `100` The `neighbor` configuration command tells the node to look for a remote BGP session (SDX's routeserver ) whose IP address is `172.0.255.254` and whose remote AS is `65000`. The network lines advertise the respective prefixes.

## Configuring the Controller Policy Using Pyretic

The control plane configuration involves defining participant's policies, which entails (1) configuring `bgp.conf` for SDX's route server and (2) configuring 'sdx_global.cfg' to provide each participant's information to the SDX controller.
The SDX presents a virtual SDX switch abstraction to each participant. Each participant writes policies for its virtual switch without bothering about other participant's policies. This limited view of the network provides desired isolation by ensuring that the participants are not allowed to write rules for other network's traffic. For more details about this abstraction, you can refer to the SIGCOMM paper about SDX.

In this example, participant A has outbound policies defined in `~/pyretic/pyretic/sdx/examples/app_specific_peering_inboundTE/controller/participant_policies/participant_A.py`, which are written as a Pyretic program:

```
prefixes_announced=bgp_get_announced_routes(sdx,'A')
final_policy= (
        (match(dstport=80) >> sdx.fwd(participant.peers['B']))+
                (match(dstport=4321) >> sdx.fwd(participant.peers['C']))+
                (match(dstport=4322) >> sdx.fwd(participant.peers['C']))+
                (match_prefixes_set(set(prefixes_announced))
>> sdx.fwd(participant.phys_ports[0]))
       )
```

Figure 6. Participant A's outbound policies. (participant_A.py)

Each participant's policies written in Pyretic. The policy shown above reflects AS A's policy for perform application-specific peering—it forwards web (i.e., port 80) traffic to peer B, and port 4321–4322 traffic to peer C. Participant C has inbound policies as defined in `~/pyretic/pyretic/sdx/examples/app_specific_peering_inboundTE/controller/participant_policies/participant_C.py`:

```
prefixes_announced=bgp_get_announced_routes(sdx,'C')
final_policy= (
        (match(dstport=4321) >> sdx.fwd(participant.phys_ports[0]))+
                (match(dstport=4322) >> sdx.fwd(participant.phys_ports[1]))
       )
```

Figure 7. Participant C's inbound policies. (participant_C.py)

Participant C has two input ports at the SDX. It writes policies for inbound traffic engineering--forwarding its port 4321 traffic to its port[0][a][b][c], and port 4322 traffic to peer port[1]. Participant B does not specify any specific policy, so its forwarding proceeds according to default BGP forwarding.

## Running the SDX Setup

Below, we explain how to perform each step manually, and we also tell you how to use the setup script to run each step. We have provided a script, `sdx-setup.sh`, for your convenience in running each part of the experiment. This script is located in the directory `~/pyretic/pyretic/sdx/scripts` in case you wish to use it. All of the commands below should be run as a non-root user (except, of course, the commands preceded by `sudo`, which will execute as root).

Step 1. Copy the controller configuration into the Pyretic path.[d][e][f][g]

```
$ cp -r
~/pyretic/pyretic/sdx/examples/app_specific_peering_inboundTE/controller/sdx_config/sdx_* ~/pyretic/pyretic/sdx
```

This command copies two files: `sdx_policies.cfg` (which points to the respective pyretic policies for each participant), and `sdx_global.cfg` (which defines the network topology for the setup).

```
$ cp -r
~/pyretic/pyretic/sdx/examples/app_specific_peering_inboundTE/controller/sdx_config/bgp.conf ~/pyretic/pyretic/sdx/bgp/
```

This command copies the Quagga configuration that we will use for each of the SDX participants. (In a real deployment, these policies might reside in separate files, as each participant would likely not be able to see the other participants' policies/setups.)

You can also run these commands with the setup script we provided.

```
$ cd ~/pyretic/pyretic/sdx/scripts
$./sdx-setup.sh init app_specific_peering_inboundTE
```

Step 2. Start the Pyretic SDX controller.

Run the Pyretic SDX controller. You may wish to do this in a second console.

```
$ ./pyretic.py pyretic.sdx.main
```

Alternatively,

```
$ ./sdx-setup.sh pyretic
```

Step 3. Launch the Mininet topology.

In a separate console, launch the Mininet topology using Mininext.

```
sudo
~/pyretic/pyretic/sdx/examples/app_specific_peering_inboundTE/mininet/
sdx_mininext.py
```

Alternatively,

```
$./sdx-setup.sh mininet app_specific_peering_inboundTE
```

This command launches emulation of three SDX participants connected to SDX. It also launches Quagga on each node, with each node running a Quagga process with the appropriate BGP configuration. Note that participant “C” has two ports connected at the SDX switch. The `sdx_mininext.py` script prints output of `ps aux` command for each host. Verify that Quagga routing daemons are running for each host in the setup. For host `a1` the output looks like:

```
*** a1 : ('ps aux',)
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME
COMMAND
root         1  0.0  0.2   5220  1272 pts/1    S+   13:50   0:00 bash
-ms mininet:a1
quagga      28  0.0  0.1   3556   628 ?        Ss   13:50   0:00
/usr/lib/quagga/zebra --daemon -A 127.0.0.1
quagga      32  0.0  0.2   4988  1480 ?        Ss   13:50   0:00
/usr/lib/quagga/bgpd --daemon -A 127.0.0.1
```

Step 4. Launch the SDX route server.

In a separate console, launch the SDX’s route server.

```
$ ~/pyretic/pyretic/sdx/exabgp/sbin/exabgp --env
~/pyretic/pyretic/sdx/exabgp/etc/exabgp/exabgp.env
~/pyretic/pyretic/sdx/bgp/bgp.conf
```

Alternatively,

```
$ ./sdx-setup.sh exabgp
```

Route server will connect with all the participant routers and establish BGP session. After some time you’ll see exchange of BGP routes between the participants and the route server. The console will output log messages indicating successful connection establishment and exchange of BGP routes.

```
Connected to peer neighbor 172.0.0.1 local-ip 172.0.255.254 local-as
65000 ...
...
Route added to neighbor 172.0.0.1 local-ip 172.0.255.254 local-as
65000 ...
```

Useful debugging step: Make sure the RIB has no old state.

During your debugging process as you complete the assignment, you may need to execute this command from time to time to make sure RIB state from older experiments is not interfering with your current experiment.

```
$ rm -rf ~/pyretic/pyretic/sdx/ribs/*
```

Alternatively,

```
$ cd ~/pyretic/pyretic/sdx/scripts
$ ./sdx-setup.sh clearrib
```

## Sanity Checks

You can now check to determine whether the participants received the routes from route server. For example, to see the routes on host a1, type the following:

```
mininext> a1 route -n
```

Verify that a1's routing table looks like this:

```
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric
Ref    Use Iface
140.0.0.0       172.0.1.2       255.255.255.0   UG    0      0
0 a1-eth0
150.0.0.0       172.0.1.2       255.255.255.0   UG    0      0
0 a1-eth0
172.0.0.0       0.0.0.0         255.255.0.0     U     0      0
0 a1-eth0
```

Specifiacally, you should see two entries in A's routing table for 140.0.0.0/8 and 150.0.0.0/8 whose next-hop IP address is 172.0.1.2.

A word about Virtual Next Hops:

As there are more than 500K IP prefixes in use, thus writing flow rules for each IP prefix will result in flow table explosion for the SDX's switch. Thus, to solve this scalability problem, SDX controller introduces the concept of Virtual Next Hops (VNHs). SDX platform assigns one (virtual) next hop for each set of IP prefixes with similar forwarding behavior. For example, in this example IP prefix pairs; (100.0.0.0/24, 110.0.0.0/24) & (140.0.0.0/24, 150.0.0.0/24) have similar forwarding behaviour. Thus the controller assigns one VNH for each pair. You can verify this behavior from the output messages from Pyretic's console:

```
Virtual Next Hop --> IP Prefix:  {'VNH1': set([u'110.0.0.0/24',
u'100.0.0.0/24']), 'VNH2': set([u'140.0.0.0/24', u'150.0.0.0/24'])}
Virtual Next Hop --> Next Hop IP Address (Virtual):  {'VNH1':
'172.0.1.1', 'VNH2': '172.0.1.2',...
```

This shows that the SDX controller assigns (100.0.0.0/24, 110.0.0.0/24) VNH1 (172.0.1.1) and (140.0.0.0/24, 150.0.0.0/24) VNH2 (172.0.1.2). Refer the SIGCOMM paper for more details on Virtual Next Hops.

## Testing SDX Policies

As a quick recap, A's app-specific policy is:

```
match(dstport = 80) >>  fwd(B) + match(dstport=4321/4322) >> fwd(C)
```

and C's inbound traffic engineering policy is:

```
match(dstport = 4321) >>  fwd(C1) + match(dstport=4322) >> fwd(C2)
```

Both B and C are advertising IP prefixes 140.0.0.0/24, 150.0.0.0/24 to A. SDX's route server decides best BGP path for these prefixes and advertises to A. In this example, routes advertised by B are preferred over C, as B's router-id is smaller than C's. We should expect that traffic from participant A for dstport 80 should go to b1, port 4321 to c1 and 4322 to c2. We can test this setup using iperf.

Starting the `iperf` servers:

```
mininext> b1 iperf -s -B 140.0.0.1 -p 80 &
mininext> c1 iperf -s -B 140.0.0.1 -p 4321 &
mininext> c2 iperf -s -B 140.0.0.1 -p 4322 &
```

Starting the `iperf` clients:

```
mininext> a1 iperf -c 140.0.0.1 -B 100.0.0.1 -p 80 -t 2
mininext> a1 iperf -c 140.0.0.1 -B 100.0.0.1 -p 4321 -t 2
mininext> a1 iperf -c 140.0.0.1 -B 100.0.0.1 -p 4322 -t 2
```

Successful `iperf` connections should look like this:

```
mininext> c2 iperf -s -B 140.0.0.1 -p 4322 &
mininext> a1 iperf -c 140.0.0.1 -B 100.0.0.1 -p 4322 -t 2
------------------------------------------------------------
Client connecting to 140.0.0.1, TCP port 4322
Binding to local address 100.0.0.1
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  3] local 100.0.0.1 port 4322 connected with 140.0.0.1 port 4322
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 2.9 sec   384 KBytes  1.09 Mbits/sec
```

In case the `iperf` connection is not successful, you should see the message, `connect failed: Connection refused.`

# Assignment

The setup for the assignment is similar to the previous example.

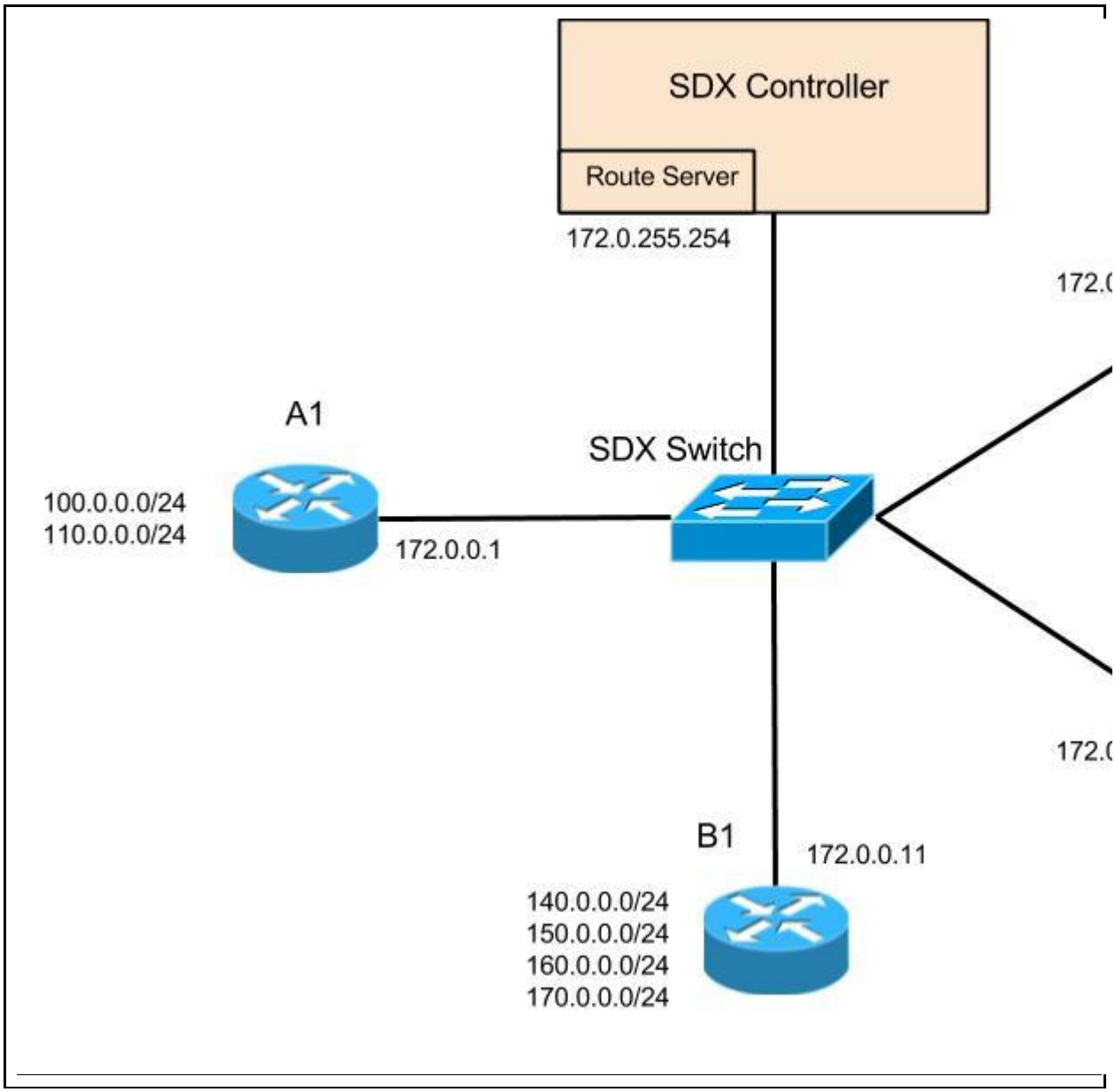


Figure 8. Topology that you will set up for the assignment.

172.* addresses refer to the IP addresses of the connected router interfaces. /24 IP prefixes are the routes that each router advertises.

In the figure, the IP addresses on each interface (`172.0.*.*`) refer to the interfaces on the local LAN that the routers (and the SDX controller/route server) use to communicate with one another. The /24 IP prefixes shown by each router in the figure indicate the IP prefixes that each router should be announcing to the neighboring ASes using BGP (i.e., using a BGP `network` statement, as we showed above in the example `bgpd.conf`).

You will need to modify files in the example `app_specific_peering_inboundTE` so that the behavior of the topology and forwarding is as we have have shown in the figure.

As with the walkthrough, the assignment has two parts.

## Part 1: Topology and route server configuration

First, you will configure the topology as shown in the figure. You will need two files:

- `sdx_mininext.py`: You will use this file to configure the SDX topology, as we have shown above. Similar to the walkthrough example, make sure that each router has a loopback address for each advertised route. For example, if the node `c1` advertises `140.0.0.0/24` then add the loopback interface `140.0.0.1` for `c1`.
- `bgpd.conf`: You will use this file to set up the BGP sessions for each of the participants and change the IP prefixes that each participant advertises. For example if node `c1` advertises `140.0.0.0/24`, then make sure that network `100.0.0.0/24` is added in `c1's bgpd.conf` file.

## Testing the topology and route server configuration

Follow the steps specified in the walkthrough example to run the setup and test your new topology and route setup.
The routing table for participant A (node a1) should look like this:

```
mininext> a1 route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric
Ref    Use Iface
140.0.0.0       172.0.1.3       255.255.255.0   UG    0      0
0 a1-eth0
150.0.0.0       172.0.1.3       255.255.255.0   UG    0      0
0 a1-eth0
160.0.0.0       172.0.1.2       255.255.255.0   UG    0      0
0 a1-eth0
170.0.0.0       172.0.1.2       255.255.255.0   UG    0      0
0 a1-eth0
172.0.0.0       0.0.0.0         255.255.0.0     U     0      0
0 a1-eth0
180.0.0.0       172.0.1.4       255.255.255.0   UG    0      0
0 a1-eth0
190.0.0.0       172.0.1.4       255.255.255.0   UG    0      0
0 a1-eth0
```

## Part 2: Policy configuration

In the second part, you will define policies for each participant. These policies should satisfy the following goals:

- Participant A should forward HTTP traffic to peer B, HTTPS and port 8080 traffic to C.
- Participant C should forward HTTPS traffic to c1, HTTP traffic to c2 and drop any traffic for port 8080.[h]

You will need to modify `participant_A.py` and `participant_C.py` from the walkthrough to implement these policies.

## Testing policy configuration

SDX's route server will select B's routes for the prefixes `140.0.0.0/24, 150.0.0.0/24, 160.0.0.0/24 & 180.0.0.0/24;` C's routes for the prefixes `180.0.0.0/24 & 190.0.0.0/24; and` A's routes for the prefixes `100.0.0.0/24 & 110.0.0.0/24.` Even though A's policy is to forward port 80 traffic to B, the SDX controller will forward port 80 traffic with dstip = 180.0.0.1 to C. Since C's inbound TE policy forwards the HTTP traffic to c2, thus this traffic should be received at c2. Similarly HTTPS traffic from A should be received at c1. We should also expect packet drops for port 8080 traffic forwarded to C.
Similar to the walkthrough example, you can use iperf to test the policy configuration. You can verify that port 80 traffic for routes advertised by B will be received by node b1.

```
mininext> b1 iperf -s -B 140.0.0.1 -p 80 &
mininext> a1 iperf -c 140.0.0.1 -B 100.0.0.1 -p 80 -t 2
------------------------------------------------------------
Client connecting to 140.0.0.1, TCP port 80
Binding to local address 100.0.0.1
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  3] local 100.0.0.1 port 80 connected with 140.0.0.1 port 80
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 3.0 sec   384 KBytes  1.06 Mbits/sec
```

You can verify that port 80 traffic from A for routes advertised only by C will be forwarded to node c1.

```
mininext> c2 iperf -s -B 180.0.0.1 -p 80 &
mininext> a1 iperf -c 180.0.0.1 -B 100.0.0.2 -p 80 -t 2
------------------------------------------------------------
Client connecting to 180.0.0.1, TCP port 80
Binding to local address 100.0.0.2
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  3] local 100.0.0.2 port 80 connected with 180.0.0.1 port 80
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 3.0 sec   384 KBytes  1.04 Mbits/sec
```

Finally, you can also verify that the port 8080 traffic forwarded to C will be dropped.

```
mininext> c1 iperf -s -B 180.0.0.1 -p 8080 &
mininext> a1 iperf -c 180.0.0.1 -B 100.0.0.1 -p 8080 -t 2
<Nothing happens, use ctrl+c to end this test>
```

In this case you should see iperf client's requests from A will not be received by the server running on c1.

# Submitting the Assignment

Copy the `submit.py` file that we have provided to the `~/pyretic/pyretic/sdx/examples/app_specific_peering_inboundTE/mininet/` directory.[1][2]
On one console launch the SDX controller:

```
$ ./sdx-setup.sh init app_specific_peering_inboundTE
$ ./sdx-setup.sh clearrib
$ ./sdx-setup.sh pyretic
```

On another console, start exaBGP:

```
$ ./sdx-setup.sh exabgp
```

### Method 1

To submit your code, run the submit.py script, now launch the submit.py script from the `~/pyretic/pyretic/sdx/examples/app_specific_peering_inboundTE/mininet/` directory.

```
$ sudo submit.py
```

Your mininet VM should have Internet access by default, but still verify that it has internet connectivity (i.e., eth0 set up as NAT). Otherwise, submit.py will not be able to post your code and output to our coursera servers.

The submission script will ask for your login and password. This password is not the general account password, but an assignment-specific password that is uniquely generated for each student. You can get this from the assignments listing page.

Once finished, it will prompt the results on the terminal (either passed or failed).

### Method 2: (alternative for those not able to submit using the above script)

Download and run the following commands:

```
$ cd
~/pyretic/pyretic/sdx/examples/app_specific_peering_inboundTE/mininet
$ wget
https://d396qusza40orc.cloudfront.net/sdn/srcs/Module%207%20Assignment
/m7-output.py
$ sudo python m7-output.py
```

This will create an output.log file in the same folder. Upload this log file on coursera using the submit button for Module 7 under the Week 7 section on the Programming Assignment page. Once uploaded, it will prompt the results on the new page (either passed or failed).

Note, if during the execution submit.py or m7-output.py scripts crash for some reason or you terminate it using CTRL+C, make sure to clean mininet environment using:

```
$ sudo mn -c
```

Also, if it still complains about the controller running. Execute the following command to kill it:

```
$ sudo fuser -k 6633/tcp
```