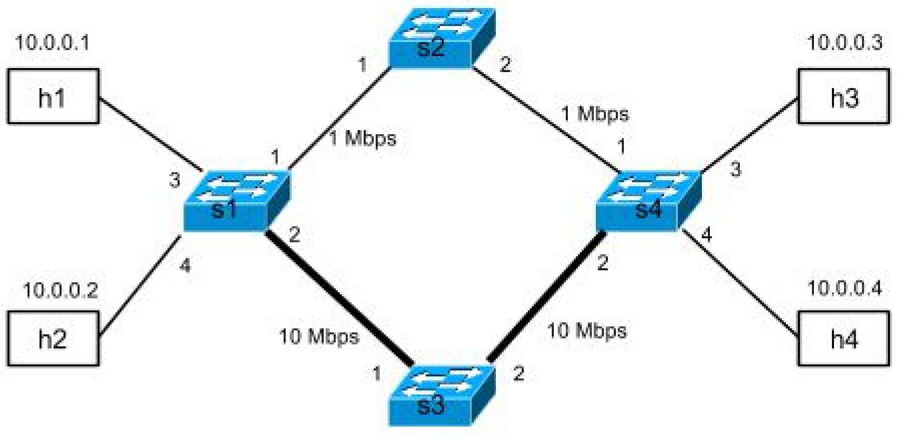
In this assignment, you will learn how to slice your OpenFlow network using the "Pox" OpenFlow controller. In the process, you will also learn more about the concept of flowspaces and how the centralized visibility and “layerless-ness” of OpenFlow enables flexible slicing.

The purpose of this exercise is to help motivate network virtualization and show you different ways in which network virtualization can be implemented. In the lessons, you learned about FlowVisor, a mechanism for allowing multiple controllers to operate on distinct portions of network flowspace. Flowvisor is implemented in Java, which would require you to install a completely new tool and operating environment. In lieu of doing so, we have created a simpler assignment that illustrates some of the concepts of slicing that Flowvisor uses. We encourage you to follow the Flowvisor Tutorial if you are interested in playing specifically with FlowVisor.

The Flowvisor tutorial teaches you how to create multiple network slices and control different slices with different controllers. Here, because we are going to implement everything directly in Pox, your task is simpler: You will create a network application that creates multiple Layer 2 network slices for different portions of the flowspace.

Read further for instructions on creating and submitting your code. Make sure that you follow each step carefully.

Setup



In this assignment we will slice a wide-area network (WAN). The WAN shown in the figure above connects two sites. For simplicity, we'll have each site represented by a single OpenFlow switch, s1 and s4, respectively. The sites, s1 and s4, have two paths between them:

- a low-bandwidth path via switch s2
- a high-bandwidth path via switch s3

Switch s1 has two hosts attached: h1 and h2, and s4 has two hosts attached: h3 and h4.

As a quick refresher, remember to always ensure that there is no other controller running in the background. Check if any controller is running in the background.

```
$ ps -A | grep controller
```

Kill the controller in case any of them is still running.

```
$ sudo killall controller
```

Restart Mininet to make sure that everything is clean and using the faster kernel switch.

```
$ sudo mn -c
```

We'll use a Mininet script, `mininetSlice.py`, to create a network with the topology provided above. The topology we have provided should correspond to the one shown in the figure above, although for the second part of the assignment we have used Pox's link-layer discovery protocol (LLDP) features to ensure that you can complete the assignment without ever explicitly referring to port numbers.

```
self.addLink('s1', 's2', port1 = 1, port2 = 1, **http_link_config)
```

Code Overview

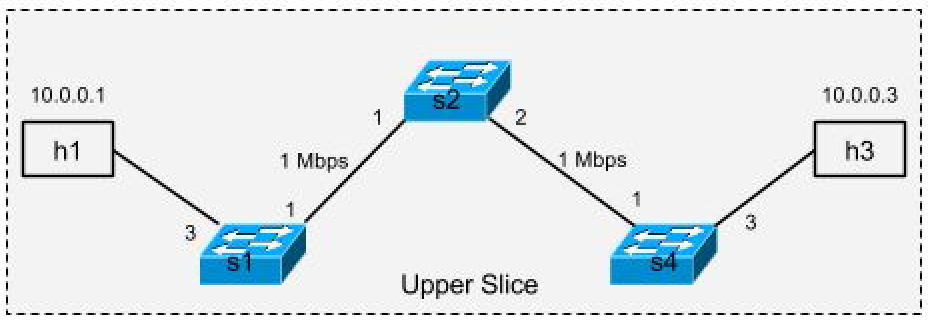
The assignment has two parts, (1) simple topology-based slicing and (2) advanced flowspace slicing. To get started, download the zip file for the assignment. It consists of four files:

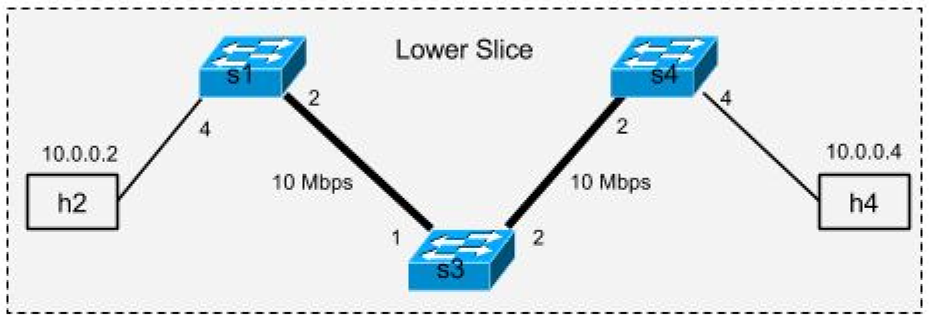
- `mininetSlice.py`: Mininet script to create the topology used for this assignment.
- `topologySlice.py`: A skeleton class where you will implement simple topology-based slicing logic.
- `videoSlice.py`: A skeleton class where you will implement advanced flowspace slicing logic.
- `submit.py`: The submission script to submit your assignment's output to the Coursera servers.

We'll now see how you will use these files to run the exercise and submit your assignment.

Part 1: Topology-based Slicing

The first part of the assignment should give you a basic understanding of how to "slice" a network using a SDN controller. The goal of this part is to divide the network into two separate slices: upper and lower, as shown below. Users in different slices should not be able to communicate with each other. In practice, a provider may want to subdivide the network in this fashion to support multi-tenancy (and, in fact, the first part of this assignment provides similar functionality as ordinary VLANs).





To implement this isolation, we need to block communication between hosts in different slices. You will implement this functionality by judiciously inserting drop rules at certain network switches. For example, host h1 should not be able to communicate to host h2. To implement this restriction, you should write OpenFlow rules that provide this isolation. (Unlike previous assignment, this topology has multiple switches; each switch has its own flow table; the controller uses each switch's datapath ID to write flow rules to the appropriate switch.)

Understanding the code

The `topologySlice.py` file provides skeleton for this implementation. As in the previous assignment, we have implemented a `launch()` function, which registers a new component. The`_handle_ConnectionUp()` callback is invoked whenever a switch connects to the controller.

We also launch the `discovery` and `spanning tree` modules, which compute a spanning tree on the topology, thus preventing any traffic that is flooded from looping (since the topology itself has a loop in it, computing a spanning tree is required). More information about these modules is available here: Spanning Tree, Discovery

Testing your code

Once you have implemented the logic for topology based-slicing, you can test it following these instructions:

Move your files to POX's directory, `~/pox/pox/misc`. This should prevent any potential `PYTHONPATH` issues you might encounter.

```
$ mv topologySlice.py ~/pox/pox/misc/
$ mv mininetSlice.py ~/pox/pox/misc/
```

Launch the POX controller.

```
~/pox/pox/misc$ pox.py log.level --DEBUG misc.topologySlice &
```

In a separate terminal, launch your Mininet script.

```
~/pox/pox/misc$ sudo python mininetSlice.py
```

Wait until the application indicates that the OpenFlow switch has connected and that all spanning tree computation has finished. You should see some messages such as `DEBUG:openflow.spanning_tree:Requested switch features for [00-00-00-00-00-03 4]` after a while, possibly followed by some flooding.

Now, verify that the hosts in different slices are not able to communicate with each other.

```
mininet> pingall
```

You should see the following output:

```
h1 -> X h3 X
h2 -> X X h4
```

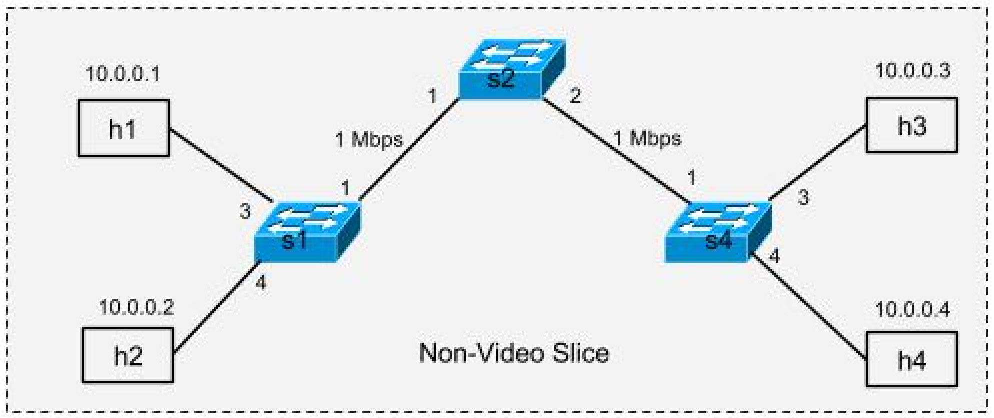
```
h3 -> h1 X X
h4 -> X h2 X
*** Results: 66% dropped (8/12 lost)
```

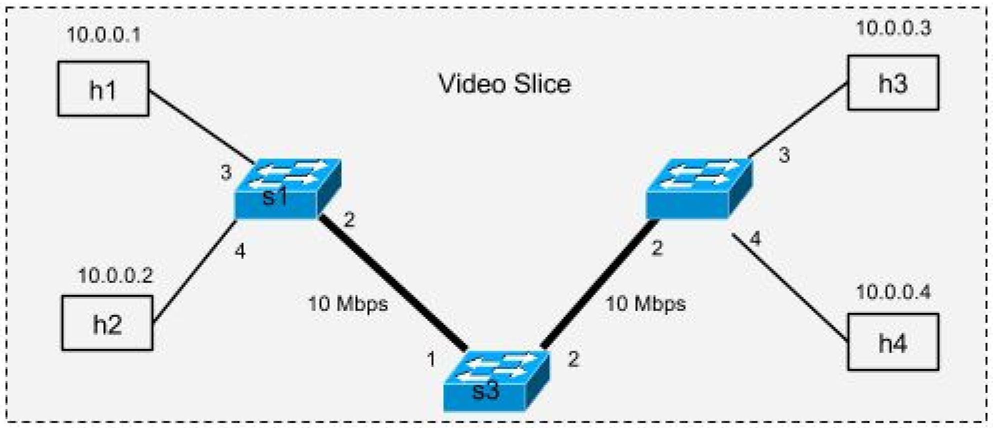
Part 2: Flowspace Slicing

In the previous part of the assignment, you learned how to slice a network based on the network's physical topology. It is also possible to slice the network in more interesting ways, such as based on the application that is sending the traffic. SDN networks in principle can be sliced on any attributes of flowspace.

Recall that the topology has two paths connecting two sites: a high-bandwidth path and a low-bandwidth path. Suppose that you want to prioritize video traffic in our network by sending all the video traffic over the high bandwidth path, and sending all the other traffic over the default low bandwidth path. File transfers won't be affected by the video traffic, and vice versa. In this part of the assignment, you'll use network slicing to implement this isolation.

Let's assume for simplicity that all video traffic goes on TCP port 80. In this assignment you are required to write the logic to create two slices, "video" and "non-video", as shown below.





Understanding your code

In `videoSlice.py`, you have a class called (`VideoSlice`). We have provided you with some of the logic to implement slicing. You should fill in the `portmap` data structure and the missing parts of the `forward` function. We have included a line of the portmap data structure as a hint: the data structure should map a switch and a portion of flowspace to the dpid of the next switch. You will need to figure out how to implement a wildcard, in addition to explicit flowspace directives for port 80.

```
        '''
        The structure of self.portmap is a four-tuple key and a string value.
        The type is:
        (dpid string, src MAC addr, dst MAC addr, port (int)) -> dpid of next switch
        '''
         self.portmap = {
```

```
                        ('00-00-00-00-00-01', EthAddr('00:00:00:00:00:01'),
                         EthAddr('00:00:00:00:00:03'), 80): '00-00-00-00-00-03',
                ...
```

The discovery module that we discussed above also passes information to the Pox controller about the topology, such as the switches that it knows about, and the ports of each switch that are connected to one another. This discovery protocol, link-layer discovery protocol (LLDP), is useful for part 2 of the assignment, but we've included the code to do that discovery for you. You can look at the skeleton code for an example of how LLDP is used (see the `handleLinkEvent` function). The `l2_multi.py` module in the Pox distribution, which performs Layer 2 learning for multiple switches in a single topology, also makes use of LLDP.

Another possible hint is that because the controller runs a spanning tree protocol (you should familiarize yourself with what this does and why we have included it, if you haven't already), simply flooding from certain switches in the network as default behavior may not work. (In other words, if you find packets not getting through, consider being explicit in your portmap structure.)

Testing your code

Once you are done updating the video slice component, you can test it as follows:

As above, make sure that your `videoSlice.py` and `mininetSlice.py` are in same directory (i.e.`~/pox/pox/misc`).

Launch the POX controller

```
~/pox/pox/misc$ pox.py log.level --DEBUG misc.videoSlice
```

In a separate terminal, launch your Mininet script.

```
~/pox/pox/misc$ sudo python mininetSlice.py
```

Wait until the application indicates that the OpenFlow switch has connected.

Now verify that the all the hosts are able to communicate with each other. This is a simple sanity check to make sure that your logic is not affecting connectivity in general.

```
mininet> pingall
```

You should see this output:

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4
h2 -> h1 h3 h4
h3 -> h1 h2 h4
h4 -> h1 h2 h3
*** Results: 0% dropped (12/12 received)
```

You can now test that your slices work properly be ensuring that video (in this case, port 80) traffic traverses the 10 Mbps link and non-port 80 traffic traverses the 1 Mbps link. For example, you can test the two paths between h2 and h3 as below. (Your code should work for *all* pairwise paths.)

```
mininet > h3 iperf -s -p 80 &
mininet > h3 iperf -s -p 22 &
mininet> h2 iperf -c h3 -p 80 -t 2 -i 1
------------------------------------------------------------
Client connecting to 10.0.0.3, TCP port 80
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  3] local 10.0.0.2 port 52154 connected with 10.0.0.3 port 80
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 2.1 sec  2.50 MBytes  9.77 Mbits/sec
mininet> h2 iperf -c h3 -p 22 -t 2 -i 1
------------------------------------------------------------
```

```
Client connecting to 10.0.0.3, TCP port 22
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  3] local 10.0.0.2 port 53695 connected with 10.0.0.3 port 22

[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 4.0 sec   512 KBytes  1.05 Mbits/sec
```

Note that you can observe about 10 Mbps throughput for port 80 traffic and about 1 Mbps throughput for port 22 traffic (or any port that is not port 80).

Submitting your code

Once you are done testing both the topology and video slice component, you can test it as follows:

Copy the provided `submit.py` to `~/pox/pox/misc.`

As above, make sure that your `topologySlice.py, videoSlice.py` and `mininetSlice.py` are in same directory (i.e.`~/pox/pox/misc`).

To submit your code, run the submit.py script:

```
~/pox/pox/misc$ sudo python submit.py
```

Your Mininet installation 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 the submission script completes finished, it will provide feedback on the terminal as to whether you have passed or failed.

Note, if during the execution `submit.py` script crashes for some reason, or if 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
```

Part of this assignment is adapted from the Flowvisor Tutorial. If you are feeling brave, you may want to work your way through that tutorial to learn more; we chose to adapt the assignment for Pox to allow you to work in the Python environment that you're already familiar with, rather than having to install (and re-learn) the Flowvisor's Java-based environment.