Sharif University of Technology
Department of Electrical Engineering

Software Defined Mobile Networks

SDMN Spring 2022

Assignment:

SDN-based Network Virtualization using ODL Controller

Before you start:

- Read the description of each problem carefully, a not insignificant amount of your solutions are found by searching, use the hints in the problems and the given links to make this process faster.
- In order to record your results from the terminal, you can use the Linux command `script` that records your commands and it's outputs in your terminal and saves them as a text file, you can use this instead of taking too many screenshots (although if you think that is easier, nothing is stopping you).
- Please follow the given format for your deliverables.
- A separate report file in .pdf format is not needed **as long as you are confident in documenting your code**, we do not expect a line by line description of your code, but we do expect to understand how to run and test your code from the given comments and docstrings, if you do not document your code, then please write a report in .pdf format.
- In case that the problem descriptions are not clear, or any other questions, feel free to ask in CW.
- The due date for this assignment is 31 Ordibehesht, 23:59.

Problem 1 (Mininet and OVS)

The goal of this problem is to get you familiar with the tools that you shall use to bring up the infrastructure for your virtual networks, you will need the following at you disposal:

- A Linux machine (or VM)
- A virtual switch

The first one is obvious enough, even though it is possible to implement the entirety of this assignment in a Windows machine or MAC OS, you will be making things unnecessarily difficult for your self, so use a Linux Machine, bear in mind also that you will need to use VMs anyway in the next problems.

I recommend that you use Ubuntu 18.04.

Part 1: Using Mininet

The Open vSwitch (OVS) is your fundamental tool in simulating virtual networks, a lot of details go into making this switch work but you will be only using it's higher level interfaces and the "Kernel Switch" which directly works with the Linux kernel to create forwarding tables and ensure QoS.

The Open vSwitch documentation can be your guide if you think that some of the things that you need are unknown to you.

It is possible to implement virtual networks with only Linux CLI and this switch alone, however, it can become tedious very fast, so we opt to use Mininet instead, which creates the networks and all the switches for you and gives you a CLI to interact with the network.

Mininet let's you create these virtual networks with simple Python code or shell commands, we will be using the Python API here, for reference you can see this tutorial page but the following script will give you a general idea of all you need for this assignment.

```
from mininet.net import Mininet
from mininet.node import RemoteController, OVSKernelSwitch
from mininet.cli import CLI
from mininet.log import setLogLevel
from mininet.link import TCLink

def topology():
    net = Mininet(
        controller=<controller>,# None if we don't want any controller
                                # RemoteController if you are using an
                                # external SDN controller (like ODL)
        switch=OVSKernelSwitch,
        link=TCLink
    )

    # Adding hosts

    h1 = net.addHost(
        name="your host name",
        ip="x.y.z.t/24",
        mac="00:00: ..."
    )

```

```
    # Adding switches

    s1 = net.addSwitch(
        name="your switch name"
    )

    # Adding controller (if any!)

    c1 = net.addController(
        name="your controller name"
        # Add an 'ip' and 'port' argument if you are
        # using a remote controller
    )

    # Adding links

    net.addLink(h1, s1)

    # Start controller on switches (if any!)

    s1.start([c1])      # More than one controller is possible!

    # Start CLI and build the network

    net.build()
    net.start()
    CLI(net)
    net.stop()

if __name__=='__main__':
    topology()
```

Quick reminder, Mininet runs with Python 2, if you compile this with Python 3 you will probably get an ImportError right away.

Part 1: Using the OVS switch

Using Mininet, implement the following simple LAN topology, use no default controller, you need to input flows into the switches, to this end you can use the ovs-ofctl command to add flows into the switches, you are using Openflow here, so you need to input the correct match fields and actions into the switches, in this simple topology, you just need to tell the switches to match on input and then output on the other port, check the OVS document or the ovs-ofctl man page to see some examples if you need it.

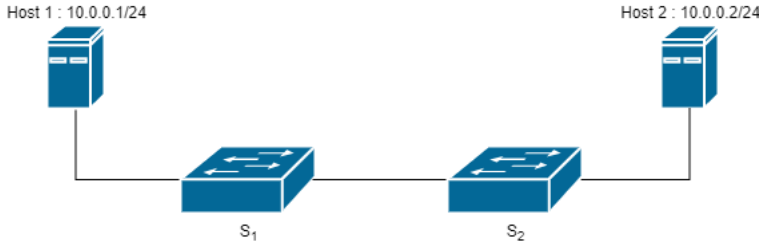

Figure 1: Simple LAN topology

Deliverable:

- A Python script `create_net.py` that creates the networks and the hosts.
- A shell script or another Python script `push_flows` that pushes all needed flows into the switches (Do not append these files to the previous file, keep them separated.)
- A screenshot or text script showing that the hosts can ping each other.

Part 2: Using Mininet and Implementing a Router

Your task is to create this network using the Mininet Python API, no controller is needed yet, since hosts are in different subnets, what you did in the previous part, yields no connectivity here, you need to input a correct flow for the router, which is itself an OVS switch.

The switches are obvious enough, the router however has to read the layer 3 header and decrement TTL, like the previous part, you need to use `ovs-ofctl` for this part

Deliverables:

- A python script `create_net.py` that creates the network and starts the Mininet CLI.
- A shell script or another Python script `push_flows` that pushes all needed flows into the switches and the router (Do not append these files to the previous file, keep them separated.)
- A screenshot or a text script showing that the hosts can ping each other.

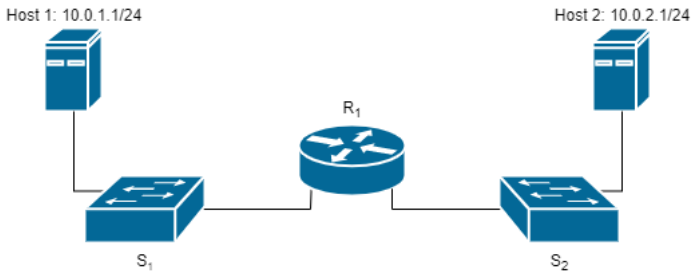

Figure 2: Topology with different subnets

Part 3: Implementing a MPLS network with OVS and OpenFlow

Multiprotocol Label Switching (MPLS) is a mechanism in high-performance telecommunications networks that directs data from one network node to the next based on short path labels rather than long network addresses, avoiding complex lookups in a routing table. The labels identify virtual links (paths) between distant nodes rather than endpoints. MPLS can encapsulate packets of various network protocols.MPLS works by prefixing packets with an MPLS header, containing one or more labels. This is called a label stack.
Note: These MPLS-labeled packets are switched after a label lookup/switch instead of a lookup into the IP table.

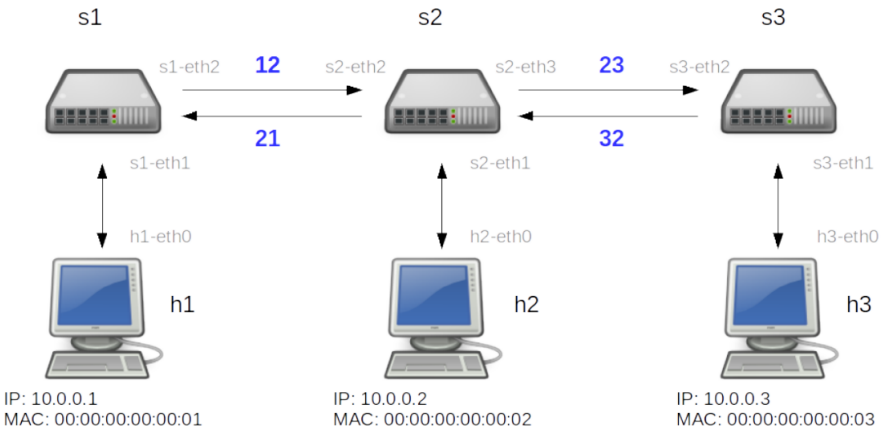

Figure 3: Simple MPLS Nework

Note: You should handle every host's(h1,h2,h3) packets.
The numbers in blue correspond to the label that the packets carry with them in each link.
As you can see, when packets travel from h1 to h3:

- The switch s1 pushes the label 12 and forwards the packet to switch s2.
- The switch s2 swaps the label 12 for the label 23 and forwards the packet to switch s3
- The switch s3 pops the label 23 and forwards the packet to host h3.

This environment will be set up using only mininet and Open vSwitch. Please note that the label stack of the packets **is just one label deep**, this is because currently Open vSwitch only supports one label. **The whole experiment will be performed in userspace**, as the OVS kernel module does not support MPLS yet. For this reason you should use the option "datapath=user".
You should make a more general rule so both IPV4 and ARP packets are sent through MPLS. This should reduce the number of flows that is written in every switch.
Deliverables:

- A bash script `create_net.py` that creates the network and starts the Mininet CLI.
- A shell script or another Python script `push_flows` that pushes all needed flows into the switches (Do not append these files to the previous file, keep them separated.)
- Both screenshot and a shell script showing that the hosts(h1,h2,h3) can ping each other.
- Provide enough screenshots of your wireshark, showing correct label switchings and properly working of the implemented mpls protocol.

Problem 2 (Opendaylight)

Now, we wish to add a new abstraction layer to this network, in the form of an SDN controller.

You will be using Opendaylight Nitrogen here, though it is old, it provides the GUI features to let you get familiar with it, Do not use any other controller, use the given link.

The SDN controller has many features, it is essentially a small, fast and robust OS that is designed to interface with SDN enabled networking hardware. here you shall be using the Openflow plugin to push flows into the switches, first however, we are going to get a bit familiar with it, the next problem will leverage ODL in a more complicated problem.

Part 1: ODL intro

I highly recommend to read/see the documentation that shall be provide to you before proceeding to implement this problem, this is supposed to be an introduction to how ODL works without you necessarily knowing how it's built, thus knowledge of things like YANG models or what not aren't needed.

You will work with the ODL GUI here to get yourself familiar with it, to this end, install the following features on ODL:

- odl-dluxapps-applications
- odl-restconf-all
- odl-l2switch-all
- odl-openflowplugin-flow-services-rest

To summarize what they do, DLUX is the GUI application of ODL, that let's you see what is running on the controller with a browser. The RESTCONF protocol is the main Northbound protocol that you use to interface with the controller and tell it what to do (there is some NETCONF also here and there ... but it's not important). The L2 switch application as the name suggests, is an application that creates L2 learning switches in a LAN, you can provide connectivity in complex LANs without any need to get your hands dirty. The Openflow plugin should be obvious as you have seen in the course.

Do not install features that you don't need! it puts too much load on your machine if you unleash all the applications at once!

Your task is to create a tree topology like the following, let ODL route the topology by itself, provide screenshots of the GUI showing your topology recognized by ODL.

Also, try to use the YANG UI application in the GUI, see what the controller is actually using to work with your switches, the `opendaylight-inventory` API is what you should checkout, remember that ODL calls each entity a `node` and default name of each node starts with `openflow:` followed by a number.

Note: Take care to use Openflow 1.3 in the switches (ODL Nitrogen works best with this version, you can force this by passing a `protocols="OpenFlow13"` in you OVS switch constructor), bear in mind to use mininet's already existing tools to your advantage, if difficulties arise, refer to Opendaylight Openflow plugin user guide.

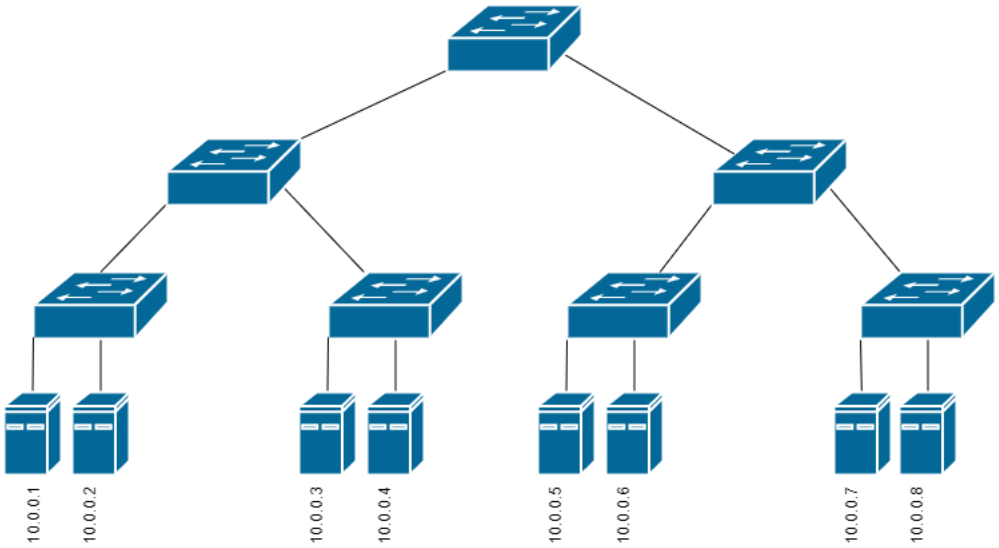

Figure 4: A tree LAN topology of 3 levels.

If you want to, pass simple MAC addresses when creating the topology or force Mininet to use simple MAC addresses to make things less hectic.

Deliverables:

- A shell or python script `create_net` that brings up the topology and connects it to ODL, you can use ODL's log in Karaf to check if all your switches connected successfully.
- Screenshot showing the recognized topology, try to use the YANG UI for the sake of familiarity but no screenshot is needed.
- Screenshot or text script showing the flows in the root switch, i.e. the switch in the top level, describe what these flows are doing and what the match fields are.

Part 2: Using RESTCONF

For this part, **make sure the L2 switch application is turned off** or else it will ruin your configurations.

In the last part, L2 switch did all the routing for you, however, when connecting hosts in different subnets, ODL does not have a well defined application (except maybe BGP and PCEP which are not our concern) to do this, thus external routing applications are used to do this, either integrated directly into ODL (which needs lots of Java code) or programs that interface with the NB API to tell the controller what to do, probably using RESTCONF.

You can imagine RESTCONF as just a service point, with many different URLs that serve GET, POST, etc. requests to your routing program, the exact URL for each endpoint can be found by using the YANG UI as you seen in the previous part, or the Openflow user guide which we linked previously (check that out by the way, it's the closest thing you have to a literal "Cheat Sheet").

Your task is to route the topology in Part 2 of Question 1 (the one with a router and 2 switches) but using Opendaylight this time, ALL switches must be under the control of Opendaylight and make sure to delete the flows on each switch using the tools of OVS if needed (although ODL does this for you no matter what).

You need to create XML payloads (or JSON, doesn't matter) that describe these flows, and then send them to the appropriate RESTCONF endpoint, you can use YANG UI or API testers like Postman for debugging, use Python for your program and save your flows in a separate folder, how you create the XML trees is completely under your control, you can format strings or build the tree from scratch.

If you think your flows are not working, check three things:

- Exploit Wireshark to check where your pings are failing.
- Check the ODL log (use `log:tail` in the Karaf CLI) to check what the controller is doing with your requests.
- Sometimes your flows fail silently, and the controller will not push them, use OVS tools to check if your flow is actually in the switch.

Deliverables:

- A Python script `create_flows.py` that creates flows and saves the XML tree of each flow in a separate folder with appropriate naming.
- A Python script `push_flows.py` that sends the flows created by the last script to the controller.
- A screenshot or text script showing that the hosts are able to ping each other.
- A folder that contains all the flows that you created and sent to the controller.

Problem 3 (Minimum weight routing)

In this problem, we want you to implement a minimum cost path routing algorithm on a network.

You are provided an adjacency matrix $A \in {\mathcal{R}_+}^{n \times n}$ that holds link weights between n switches in the network, the matrix has the following properties:

- $A_{ij} = 0$ iff no link exists in either direction between nodes i and j.
- $A_{ij} > 0$ if a one-way link exists that connects node i to node j.

Links can be one-way, thus the matrix can be asymmetric, links with lower weights are preferred to links with a higher weights.

Given this matrix, one can solve the problem of finding the path with minimum weight between two nodes on the graph, without loss of generality we can assume this path is to connect nodes 1 and n together, thus we have two hosts connected to switches 1 and n that we wish to connect.

Create a network corresponding to a given adjacency matrix and use a suitable minimum cost path algorithm to find the route that connects nodes 1 and n, you can use Dijkstra's algorithm for this, and there is no need to implement it yourself, use any available implementation (like the one in NetworkX).

Here is a sample input and output for case $n = 4$.

$$A = \begin{pmatrix} 0 & 2 & 3 & 4 \\ 2 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

Outputs:

$$\text{Path}_{1 \to n} : s1 \to s2 \to s4$$
$$\text{Path}_{n \to 1} : s4 \to s1$$

You need to convert this path into a series of flows that can be pushed into the switches, once again remember to save the flows in a separate folder, make use of the implementation of the previous problem to your advantage.

Hosts are assigned the IP addresses same as the one used in Part 2 of Question 1, so they are not in the same subnet.

Deliverables:

- A Python script `create_net.py` that accepts the matrix and creates the network, connecting it to the controller.
- A Python script `spf.py` that given the matrix, finds the minimum weight path between nodes 1 and n and outputs the paths.
- A Python script `create_flows.py` that creates the flows from the paths.
- A Python script `send_flows.py` that sends the created flows to the controller.
- A folder containing the flows that you have created and sent to the controller and is created and maintained by your code.

Problem 4 (Dynamic routing)

This is just an extension of the previous problem, instead of giving you an static adjacency matrix,you will use the matrix to create the network, but you have to keep an eye on the network in case any link fails.

To this end, add a script to the previous problem that monitors the network topology, by querying the topology from ODL, checking the links and making sure nothing bad has happened, this can be done for example, every 30 seconds or so (this is up to you, not important).

The link weights are assumed to be static, keep the link weights just like the one that the initial adjacency matrix provides it to you, it is just the fact that the links can break that is the new complication since the last problem.

For testing, you can shut off a link from the Mininet command line like the following:

```
mininet> link <endpoint 1> <endpoint 2> down
```

and see how your program behaves, it will be nice that your program keeps a log of such events and is not completely silent.

Deliverables:

- A Python script `create_net.py` that accepts the matrix and creates the network, connecting it to the controller.
- A Python script `spf.py` that given the matrix, finds the minimum weight path between nodes 1 and n and outputs the paths.
- A Python script `create_flows.py` that creates the flows from the paths.
- A Python script `send_flows.py` that sends the created flows to the controller.
- A Python script `watcher.py` that queries the topology from ODL and calls the other scripts if necessary, in case it founds out that a link has died.
- A folder containing the flows that you have created and sent to the controller and is created and maintained by your code.

Note: When testing, we keep your networks small, we also do not damage the network to a point that no connection is possible at all, also do not flood the controller with unnecessary flows, pushing a flow into a network is a surprisingly lengthy and hard process and should be done ONLY when needed.

Problem 5 (VXLAN)[Optional]

If you ever had an experience working with large cloud computing infrastructures, then you probably at least have heard of VXLAN. informally, VXLAN is a tunneling protocol that can encapsulate IP headers and route them like other packets, but after the packet is parsed it can be treated differently, it is the main tool in deploying vast overlay networks in a cloud, when modification of the underlay network is hard or not desirable.

The main purpose of VXLAN was to solve the scaling challenges of VLAN, however the operation is also a bit different, which we will not dive into that ...

What we will dive into however, is how to use it in SDN, to this end a simple understanding of VXLAN is needed, check this for a simple introduction and this for a bit more detail.

Part 1 (Using OVS)

You are tasked to first create the topology below and assign the given tunnel ids to each data path, please take care to use the EXACT same names as the ones here.

Whether you use Mininet or just OVS to create your network is of no consequence to you (except in terms of how comfortable you are with it), however these two points are mandatory:

- You should not use a controller yet to push tunnel ids, use the tools provided by OVS to do this, it is possible that you need to search a bit, since learning a bit about VXLAN by itself is actually part of this problem and not only the implementation.
- You **MUST** use two different machines (VMs) here, we are trying to simulate a cloud (a pathetically small cloud, but a cloud nonetheless) which has two nodes in a LAN, you can assign ip addresses in range of 192.168.1.x to your VMs and let your host route them, if you are using Oracle VirtualBox you will be told how to do that in a separate guide.

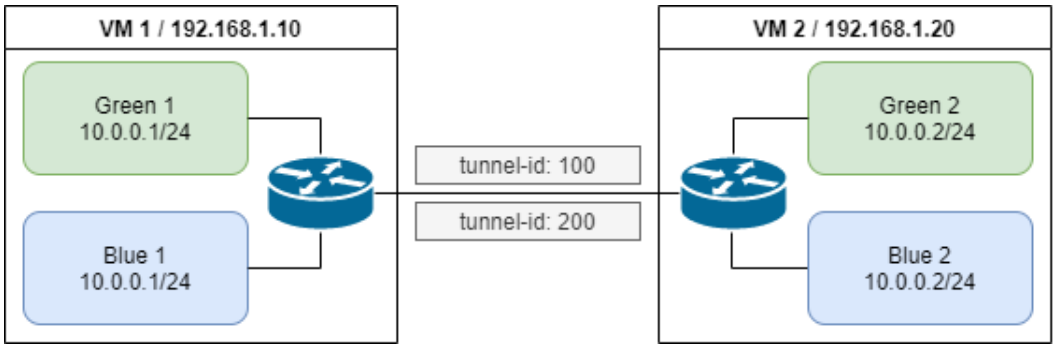

Figure 5: A simple 2-tenant topology.

Also, take note that at no point in the flows that you have used, the IP address of the VMs are to be used in any way, this is an overlay network and it has **no knowledge** about what is going on in it's underlay network, if it was to do that, it would defeat the whole purpose of this protocol.

If your network is working fine, it is expected that when a Blue host attempts to reach the IP address of it's destination (i.e. if they are assigned 10.0.0.1, they try to get to 10.0.0.2 in the other machine) it

Must ping it's Blue pair, not the Green one, even though they have literally the same IP address.

Deliverables:
Divide your files into two separate folders, called VM1 and VM2, each folder contains these files that correspond to that VM alone.

- A Python or shell script create_net that creates the network and then configures the VXLAN on each device needed.
- A shell script push_flows.sh that pushes the flows needed into the switches in the VM.
- A screenshot or text script showing that the corresponding hosts are pinging each other.

Part 2 (Using ODL)

Do what you did in the previous part using the ODL controller instead of OVS CLI tools on each VM, you can put the controller on any of the two VMs (ideally we would have want it to be a separate VM altogether but that is a bit of a nightmare for a single normal PC).

I suggest you read Problem 2 to the end before you attempt to tackle this problem, it will be MUCH easier then, all you need to do is to set the tunnel-id filed in your flows and push them into the switches, the rest are normal ARP and IP flows which you have already seen.

Deliverables:

Divide your files into two separate folders, called VM1 and VM2, each folder contains these files that correspond to that VM alone, one of the VMs also hosts a controller in your implementation, put the files needed into that folder (i have marked for you which one is for the controller).

- A Python or shell script create_net that creates the network and then configures the VXLAN on each device needed and connects it to the controller.
- A screenshot or text script showing that the corresponding hosts are pinging each other.
- **(For the controller)** A Python script create_flows.py that creates flows and saves the XML tree of each flow in a separate folder with appropriate naming.
- **(For the controller)** A Python script push_flows.py that pushes the flows that you have created in the previous part into the switches **IN BOTH** machines.
- **(For the controller)** A folder containing the flows that you have created and sent to the controller.