University of Crete
Computer Science Department
Lecturer: Prof. Dr. X. Dimitropoulos
TAs: M. Lakiotakis,
A. Hatzivasileiou, A. Shevtsov

Autumn Semester 2019
HY436 - Software Defined Networks
Tasks of **Assignment 2**
Assigned: 24 October 2019
Due: 13 November 2019, 23:59

Assignment 2

1 Introduction

In this assignment you will acquire a practical view of the concepts of network virtualization and cloud management in the context of data center environments. In particular, you will write SDN applications on top of an OpenFlow controller (POX) to implement routing, enforce firewall policies and provide support for Virtual Machine migration over a data center-like topology. This controller will essentially act as a network controller for a small cloud facility.

In general, network virtualization permits to share physical network resources so as to form diverse virtual networks (slices) on top of the infrastructure: each of these slices has e.g., its dedicated bandwidth allocation, broadcast domain, etc. This concept simplifies network management and increases the resource utilization of the physical infrastructure, allowing a cloud provider to amortize the costs of the underlying network. Mechanisms related to network virtualization are commonly used in data centers to orchestrate their multiple applications –running on servers– and tenants. SDN in general and OpenFlow in particular can be used to implement such mechanisms, as you have learned in the lecture.

2 Assignment: Emulate a Data Center and Manage it via a Cloud Network Controller

In this exercise, your first task is to create a multi-rooted tree-like (Clos) topology [1] in Mininet [2,3,4] to emulate a data center (see Figures 1 and 2). Your second task is to implement specific SDN applications on top of a network controller (POX [8,9]) in order to orchestrate multiple network tenants within a data center environment, in the context of network virtualization and management.

In Figure 1, we see a sample data center network design with a number of Virtual Machines (VMs) being hosted by distinct VM end-host servers. For simplicity, we assume that each server hosts exactly one VM instance (in practice they can of course support tens of VMs). In Figure 1, each set of VMs with the same color belongs to a separate data center tenant; tenants may represent different application environments (different colors respectively) that the VMs host. For example, one tenant may be an enterprise client running its big data analytics, while another tenant may be a research group running simulations for an upcoming publication.

In this assignment you will not need to virtualize your own hosts, since we make the convention that each individual physical host supports a single VM or virtual host; Mininet takes care of emulating these virtual hosts. It is assumed that each application is pre-installed in each host, thus you will not need to deal with the application part within a host. The goal is

to become familiar with how the virtualized data center network supports and orchestrates the applications running on top of its infrastructure.

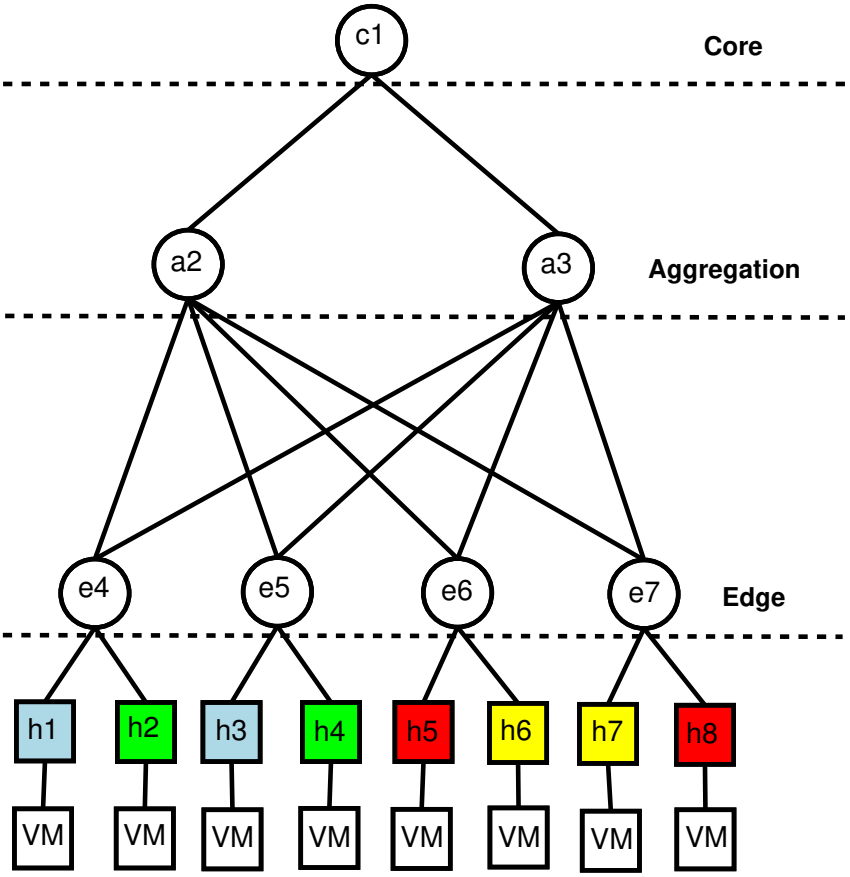

Figure 1: Sample layered data center topology

A properly designed data center network should efficiently route traffic across its rich interconnection fabric, satisfy isolation constraints, and provide VM migration capabilities for its tenants, among other requirements that you have learned about in the lecture. For example, routing strategies could exploit the rich topology of the data center, which contains multiple redundant paths between any pair of hosts, and choose the shortest path (in terms of number of hops) thus minimizing the allocated bandwidth on the network links.

According to the isolation requirement, VMs belonging to the same tenant –offering the same service– should be able to communicate only with each other. Inter-VM communication between different tenants should be blocked. This is required for security reasons, as tenants generally require to operate within their private broadcast domains. Regarding the migration part, let's assume that a tenant's host is planned to shut down for maintenance purposes. In this case, the hosted VM has to move into a different host, i.e., migrate, so as to be able to continue offering the service. Such VM transfers have to be done with minimal reconfiguration and service disruption within the network, assuming that the selected destination hosts have the available resources to host the migrated VMs. For example, assume that the green host h2 in Figure 1 is planned to go down. Red hosts h5 and h8 are both available (we assume that they do not belong to any tenant). Therefore, a possible migration choice is to move the VM from host h2 to host h5. For the exercise you do not need to take care of saving, migrating and restoring the VM state from the end-host's perspective; instead, you only need to take into account how the network treats the migration of traffic destined from the old VM to the new VM, since the

migration should be transparent on the IP layer.

In the following sections we describe the steps towards satisfying the aforementioned requirements in your network topology, using the network controller, so please follow each step carefully. Part A will take you through the processes of building the network topology, performing shortest path routing and implementing and configuring a firewall application. Part B will guide you during the implementation of a VM host migration mechanism.

Part A

2.1 Prerequisites

In this exercise, we use the POX OpenFlow controller [9] to implement our SDN network applications. The POX carp branch [8] is used as in Exercise 1. We use the Discovery component of POX [6, 7] for dynamic topology discovery, and the Python package NetworkX [5] for graph-related operations, such as finding suitable paths along which to route traffic within our network.

2.2 Understanding the code (Step 0)

As an initial step, please go through the scripts "clos_topo.py" and "CloudNetController.py", which are bundled with the main exercise assignment. The first script will be used to build the emulated network topology on Mininet. The second script contains the code relating to the SDN applications which will run over the controller, so as to orchestrate the data center network. The parts that you need to fill in are marked with comments such as:
"#WRITE YOUR CODE HERE". In the following, you will go over the steps needed to complete and test the respective parts.

2.3 Network Topology (Step 1)

Data centers are usually structured in a tree-like fashion consisting of three layers of switches. Starting from the root, we have the core, aggregation and edge/access layers. The end hosts are connected to the edge switches. In this assignment, the first step is to create a clos topology [1]. The number of core switches (tree roots) and the fanout (number of child switches per parent switch) should be treated as configurable parameters. In a basic tree topology, each switch (except for the core switches) has a single parent switch. However, in the clos-like topology that we want to emulate, each switch of both the aggregation and edge layers is connected to all the switches of the previous upper layer. An example of a clos-like topology with 2 core switches and fanout equal to 2 is illustrated in Figure 2. Examples of creating custom topologies using Mininet can be found at [2, 4], while the Mininet Python API Reference Manual can be found at [3].

2.4 Shortest Path Routing (Step 2)

Your second task is to build a network controller application that forwards traffic along *Shortest Paths* in the topology discussed above. In this topology, multiple shortest paths of equal length (in terms of hop count) may exist. Your application should calculate and store all the shortest paths of equal length between any two end-points; afterwards, it should randomly choose one of these paths along which to forward the traffic. Random choice of paths is one way to take advantage of the offered fabric and randomly distribute the load, assuming a large-scale data-center environment with redundant paths, used by several tenants. Note that an extra requirement is the differentiation between the different kinds of traffic flowing on those paths: e.g., UDP/other flows are required to follow different paths than TCP flows for any kind of host-to-host communication taking place over a switched traffic path.

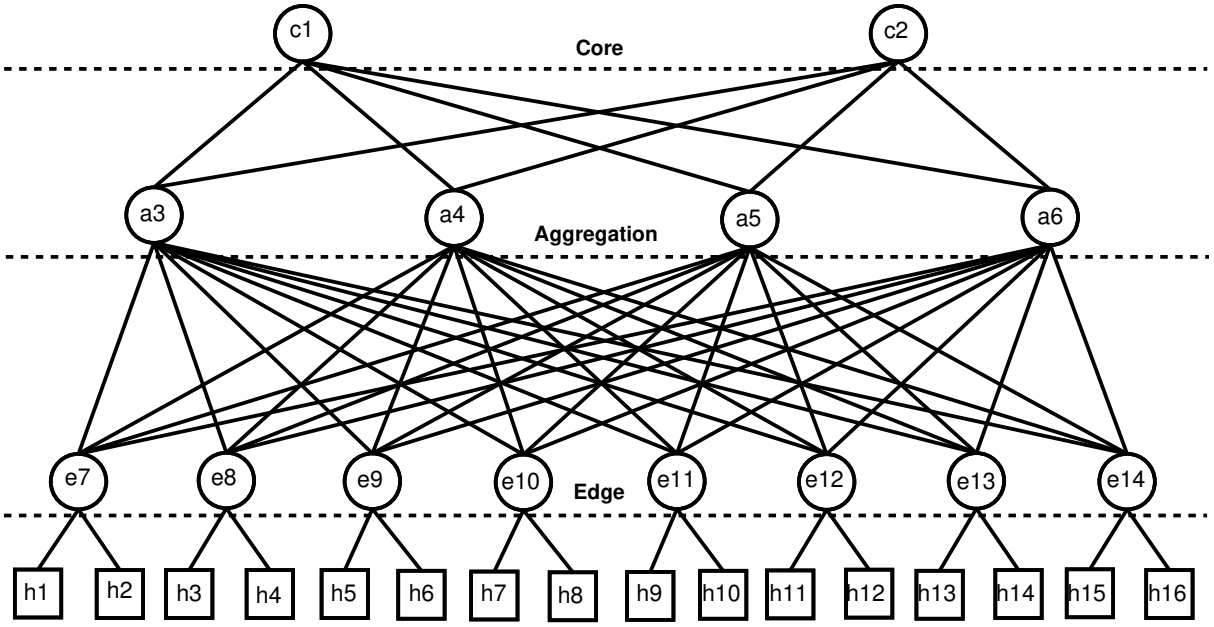

Figure 2: Clos topology with 2 core switches and fanout=2

For this purpose, you will use the NetworkX Python package [5]. Using the structure that holds information about the node (switch) adjacencies in the topology (provided in the code) and exploiting NetworkX capabilities, you should first generate the topology graph. Then, for a given source, you should retrieve the list of all equal-length shortest paths for each destination switch in the graph (HINT: *all_shortest_paths(G, source, target)* nx function) and store this information properly in the source *SwitchWithPaths* instance (kept on the controller's side), taking into account the transport protocol (UDP/TCP differentiation). The traffic differentiation that will be applied is depicted in Figure 3. The latter requirement is already taken care of using the method *getPathsperProto* of *SwitchWithPaths*. The function that you need to complete is the *ShortestPaths* one, which simply calculates all shortest paths from any source switch to any destination switch.

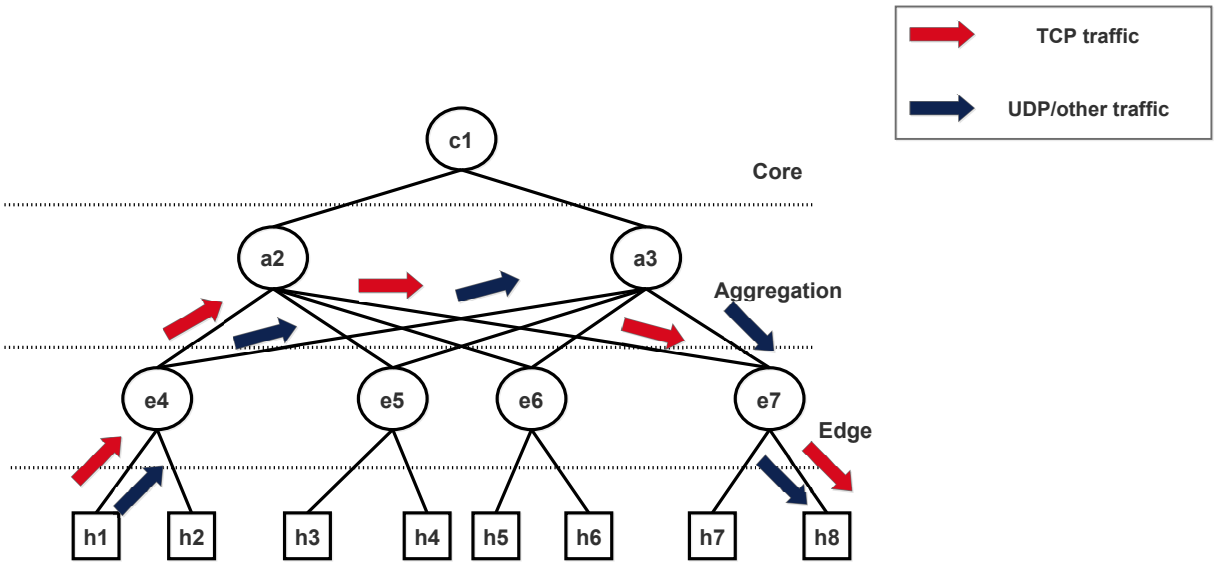

Figure 3: Traffic differentiation example

Note that the information about the node adjacencies is dynamically gathered using POX's Discovery component, which handles Link State events. This component uses the controller

to inject LLDP messages on the switches; these messages are then propagated to their neighbours. The component then infers link state information based on LLDP *PacketIns* stemming from these neighbour switches. You will not need to implement any discovery functionality, but will directly use the switch adjacency structure populated by this component. At this point, it is worth mentioning that the shortest path calculation is performed after each link discovery event in the provided code. This implies that the graph might not be totally connected until all network links have been discovered. The discovery process is dynamic; in our case it can take several seconds to discover the full graph and be able to have stable routing. Therefore, your code should handle the case when the NetworkX package is unable to calculate a path for a given source and destination because of a disconnected network/graph (HINT: *NetworkXNoPath exception*).

After the shortest path calculation has been successfully completed on the discovered graph, your task is to forward traffic along a randomly selected equal-length shortest path. When new IP flows (with different source-destination tuples) are triggered, *PacketIn* events are invoked. As a result, the controller has to choose at random one of the available shortest paths between the source and the destination host (depending on the protocol used, different pools of available paths are used) and then install the corresponding flow rules to all the switches along the selected path. The installation of flow rules is performed in an inverse fashion, i.e., starting from the destination switch up to the source switch. This is useful for limiting the occurrence of *PacketIns* on switches that are close to the destination while a *FlowMod* is on the fly between the controller and these switches. Note that the nodes contained in the list of the selected path are stored in the direct order, i.e., from the source to the destination switch. The function that you need to complete for this path installation for 3-tuple (source IP, destination IP and transport protocol) IP flows is called *install_end_to_end_IP_path*.

Remember that you need to match the IP protocol before matching on the src and dst IP addresses in OpenFlow. You can set dynamic flow expiration to happen after a 10 sec idle timeout (i.e., period of inactivity), while default hard timeouts do not change (infinite duration). Also, note that a custom function from the existing code can be used for the per-switch flow rule installation, wrapped around *flow-mod* messages: this is the *install_output_flow_rule* function. We make clear that the packets that invoked the relevant *PacketIn* events are not handled by these functions (you can see that no *buffer_id* option is used). This is because we use two low priority rules from the *_handle_ConnectionUp* function to forward **entire** packets to the controller, if no other higher priority rules can handle them. We did that to deal with some timing issues related to the communication between the controller and the switch, in terms of buffered packets. This means that you should also handle the current full packet that you have on the controller's side always with *PacketOuts*, while all the consecutive packets are handled with flow rules. Therefore, a *PacketOut* should also happen on the source switch (using another custom function for this purpose, the *send_packet*), to propagate the packet along the path. To retrieve information about the ports on which a flow rule should be applied along the selected path, we provide a structure in the given code, i.e., the *sw_sw_ports*. This structure is updated upon link discovery events and holds information about the network links. A network link is identified when two switches are connected. This pair is mapped to the switch ports that are actually connected with the link. To express the direction of the link, the corresponding port of the first switch in the pair is stored as value in the 2-level dict structure e.g., (switch1, switch2) $\rightarrow$ switch1_port, (switch2, switch1) $\rightarrow$ switch2_port.

The OpenFlow protocol does not provide any guarantees regarding the time when a flow rule is installed on a switch. The controller orders a switch to install a flow rule, but the installation is not performed immediately. Therefore, a number of *PacketIn* events might be triggered on the same switch until the flow rule has finally been installed. To solve this problem, on the

end to end path installation function (*install_end_to_end_IP_path*), we initially check whether the switch that triggered the *PacketIn* is the destination switch or not. If this is the case, we install a corresponding flow rule and send the packet (*PacketOut*). If not, we randomly select a shortest path from the switch that triggered the *PacketIn* until the destination switch as it was previously described in detail, and send the packet on its way (all consecutive packets will be handled at line-rate upon flow rule installation).

To make the routing application run properly, you should also fill in the missing code in the *SwitchWithPaths* class, i.e., *flood_on_switch_edge* and *send_arp_reply* functions. In the initialization of the network, the controller does not have any information about the connected hosts (i.e., IP, MAC, port of connected switch). Therefore, any requests among hosts should be flooded throughout the network. To avoid routing loops (i.e., switches receive packets that were flooded previously on the same port(s)), we have developed a mechanism that identifies the ports on a switch that are connected to an adjacent switch. In our approach, the remaining ports on a switch are those connected to hosts. Thus, instead of flooding within the whole network, we should just flood the ports on each switch that are connected to hosts (HINT: *flood_on_all_switch_edges*). Regarding the *send_arp_reply* function, when the controller holds information about the connected network hosts, it knows where to forward the requests. In case of an ARP request, instead of flooding the network during the initialization phase, the controller replies directly to the corresponding host, acting as an ARP proxy. The controller finds the switch connected to the host which issued the request, crafts an ARP reply and sends it through that switch to the host, using the already learned host-facing port of the respective switch.

2.4.1 Sanity checks

Once you have finished your routing-related code, you should check that traffic between any two VMs is routed and forwarded properly, e.g., using pings. Note that the firewall and migration scenario which are your next tasks in this assignment are partly integrated in the provided code. To avoid any relevant errors, since the firewall and migration code are not yet functional at your current stage, run your code using the following command:

```
~/pox> ./pox.py openflow.discovery CloudNetController --firewall_capability=False
--migration_capability=False
```

Initiate the Mininet topology script, and wait a bit for the controller to connect to the switches. The script will do an initial *pingall* as a sanity check.

```
sudo python clos_topo.py -c 2 -f 2
```

In order to generate TCP or UDP traffic, use the appropriate scripts that are provided. They operate typical TCP or UDP communication following the client-server model. The script names denote their functionality. In order to use them, after creating the network topology in Mininet, you can have access to each emulated host by using the *xterm* command. Choose the pair of hosts that you want to participate in the communication and run the scripts. For example, in Figure 3, assume that Host 1 and Host 8 communicate over UDP. Host 1, whose IP is 10.0.0.1 is the transmitter; Host 8 is the receiver with IP 10.0.0.8. Initially, run the *udp_receiver* script on Host 8 by typing:

```
sudo python udp_receiver.py 10.0.0.8 49160 5
```

The first argument denotes the receiver's IP address, the second argument denotes the port number to which the socket is bound, and the third argument indicates the buffer size.
In Host 1 xterm, just run the *udp_sender* script by typing:

```
sudo python udp_sender.py 10.0.0.8 49160
```

The 2 arguments denote the IP address and port number to which the socket on the receiver side is bound. After this step, UDP communication between the 2 hosts can be established (as long as there is a valid path for the UDP traffic to follow!). Packets are sent in 2-second intervals.
For TCP communication, you similarly execute the appropriate *tcp_sender* and *tcp_receiver* scripts on the 2 hosts.
For example, for the same pair of hosts (Host 1 and Host 8):
In Host 8 terminal type:

```
sudo python tcp_receiver.py 10.0.0.8 49160 5
```

In Host 1 terminal type:

```
sudo python tcp_sender.py 10.0.0.8 49160
```

For the same pair of hosts, do the UDP and TCP paths differ? Please check this yourself by printing the selected paths by the controller.
For the ping connectivity test, just run *pingall* command in Mininet yourselves. Is traffic routed and forwarded as expected? Are all pings successful? Are there any unexpected packet losses?

2.5 Firewall Configuration / Isolation (Step 3)

Your next task is to enhance your controller application with isolation capabilities. The isolation functionality should filter packets based on an access list; packets with source and destination IP addresses belonging to hosts of the same tenant are only allowed to be exchanged. Therefore, you are actually asked to implement a firewall module enabled by a white-list of host IPs for each application. This firewall will operate in both Layers 2 and 3; firewall controls access for both ARP and IP packets. You should create a *firewall_policies.csv* input file which your program will read. The program will then handle ARP and IP *PacketIns* and install rules on the OpenFlow switches to drop packets whenever tenants from different services attempt to communicate. Given a topology with 2 core switches and fanout of 2, containing 16 end hosts, the format of the file should be:

1,10.0.0.1,10.0.0.3,10.0.0.5,10.0.0.7,10.0.0.9,10.0.0.11,10.0.0.13,10.0.0.15
2,10.0.0.2,10.0.0.4,10.0.0.6,10.0.0.8,10.0.0.10,10.0.0.12,10.0.0.14,10.0.0.16

Each row starts with a distinct tenant ID, followed by a list of IP addresses of hosts assigned to the same tenant. A sample file is available within the assignment code folder. For simplicity, we assume that all odd-numbered hosts (h1,h3,...,h15) belong to tenant 1 while all even-numbered hosts (h2,h4,...,h16) belong to tenant 2.

You should place the *firewall_policies.csv* file under ∼/pox/ext/ . The provided code reads and properly handles the information contained in the file; check the *read_firewall_policies* function to see how the policy information is stored, mapping IPs to tenants. Then, in presence of an ARP or IP packet in a *PacketIn* event, the controller should check the corresponding packet headers to find out whether the source and destination IP addresses of the packet belong to the same tenant (same row in the file) or not[1]. If this is not the case, the controller should install a flow rule in the switch which generated the *PacketIn* in order to drop packets belonging to the same micro-flow (HINT: *drop_packets* function). In a nut-shell, you implement a packet-level firewall by installing specific micro-flow rules matching exactly the packet headers with an empty action list, equivalent to a drop action. The current packet which generated the *PacketIn* can simply be ignored, if it is illegal. Packets that have passed the firewall check can be processed by later stages of the controller application (e.g., by the normal shortest path routing process).

[1]e.g., for ARP requests you should check whether an IP which belongs to tenant 1 is requesting the MAC of an IP which belongs to a different tenant; this communication is illegal and should be dropped

2.5.1 Sanity checks

Once you have finished your firewall code, you should check that traffic belonging to the same tenant is routed and forwarded properly, whereas any other inter-tenant traffic, ARP or IP, is blocked. Note that the migration scenario which is your next task in this assignment is partly integrated in the provided code. To avoid any relevant errors, since the migration code is not yet functional at your current stage,run your code using the following command:

```
~/pox> ./pox.py openflow.discovery CloudNetController --firewall_capability=True
--migration_capability=False
```

Initiate the Mininet topology script, and wait a bit for the controller to connect to the switches. The script will trigger an initial *pingall* as a sanity check.

```
sudo python clos_topo.py -c 2 -f 2
```

You should open three *xterms*, e.g. at hosts h1, h3 and h4. Then, from h1 ping h3 checking whether the traffic is forwarded as expected. Afterwards, from h1 ping h4 checking whether the traffic is blocked as expected. For larger scale testing, perform a *pingall* command in Mininet. All pings between odd-odd and even-even hosts should be successful, while all odd-even or even-odd pings should fail. Be careful with packet losses that are not related to the firewall functionality; such losses could indicate bugs in your controller code (e.g., wrong packet handling and timing issues with the shortest path routing process).

Part B

2.6 Host Migration (Step 4)

Under certain circumstances (e.g., maintenance, failure, resource allocation optimization etc.), running application VMs have to move from some hosts (servers) to other hosts within a data center network. This VM transfer is usually called VM migration. Since we assume that a single host (server) serves exactly one tenant VM, we call it host migration in our case. One crucial point in the migration process is that it needs to be transparent on the IP layer. This practically means that we have to avoid service interruption (e.g., TCP sessions should not break; IP addresses should be at least preserved). Figures 4 and 5 describe what is expected to happen when a migration event is triggered.
We assume that host 1 runs the same application and belongs to the same tenant (e.g., tenant A) with hosts 3, 5 and 7 (see Figure 4). We also assume that host 5 is initially inactive, while the other hosts (1,3,7) are actively running the application. Then, a migration event is triggered and we decide to migrate host 1 to host 5 (since the latter one is inactive) on the fly. Now host 1 becomes inactive/unavailable, while host 5 assumes its role. You don't need to take into account the details of the end host state migration in this exercise, but you need to take into account that the new host adopts the IP of the old one. According to Figure 5, you notice that host 5 starts running application A, while host 1 becomes unavailable. This happens in a transparent way: the service is interrupted for the least possible time and host 5 now serves the application pretending to be host 1. Similarly to the isolation part of the assignment, you should create a file called *migration_events.csv* as input, to describe this migration event. This file should look like the following:

180,10.0.0.1,10.0.0.5

The first column represents how much time (in secs) it takes to trigger the migration event, after the initial execution of the controller application script. This is used in order to simulate a planned host migration event (e.g., for maintenance reasons). The pair of IPs that follows

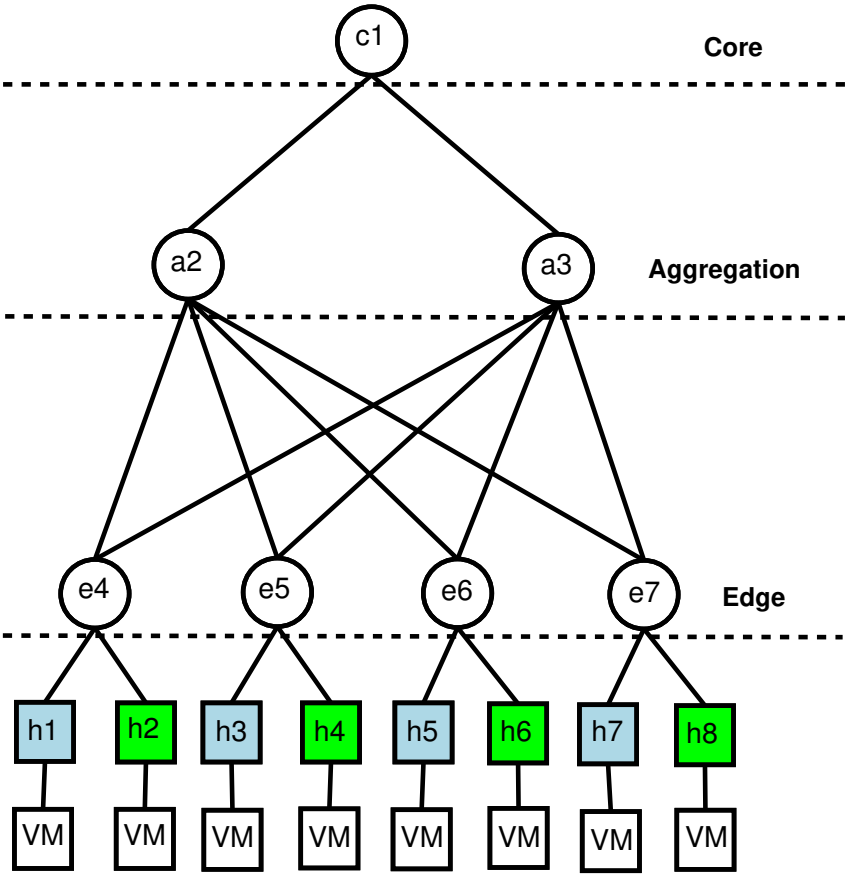

Figure 4: Data center topology with VMs before migration event. Hosts 1, 3 and 7 serve the application of tenant A while host 5, belonging to the same tenant, is inactive.

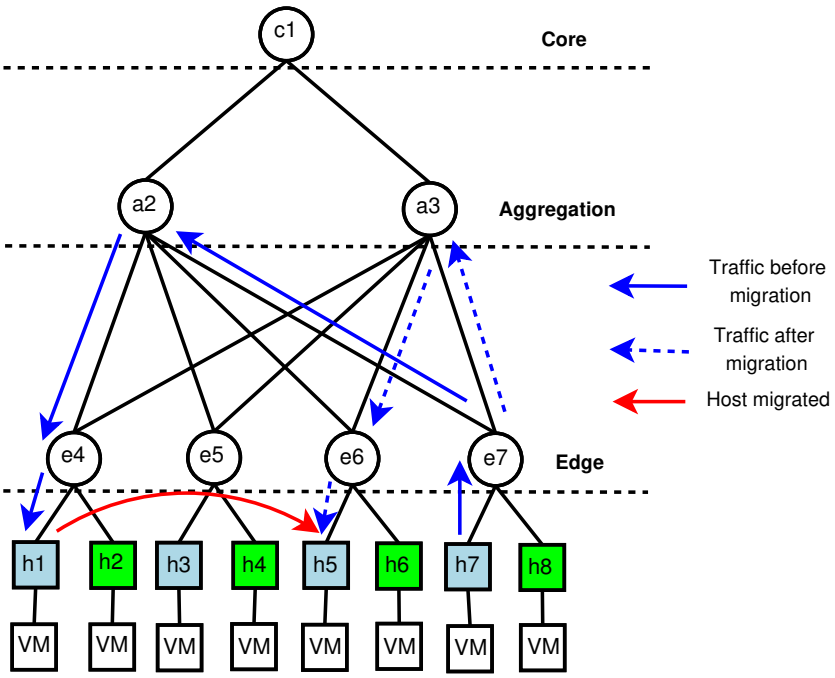

Figure 5: Data center topology with VMs after migration event. Host 1 has moved to host 5. IP traffic from other sources (e.g., host 7) destined to the IP address of host 1 should be transparently redirected to host 5.

mandates that the host with IP address equal to 10.0.0.1 (host 1) is migrated to the host with IP address 10.0.0.5 (host 5).

Now, host 5 is reserved to serve the application for host 1 while host 1 remains inactive. You should place the *migration_events.csv* file under ∼/pox/ext/ . The provided code reads and properly handles the information contained into the file. Check the *read_migration_events* function to see how the information is stored internally on the controller's side.

When the migration event is triggered, all the switch flow rules that relate to flows towards the old destination IP of host 1 (including the ones that construct the path to the to-be-migrated host) are deleted so as to "force" the migration (HINT: *handle_migration* function). This will result in *PacketIn* events directed to the controller, for any new IP flow destined to the old IP. Now the controller should install rules that redirect IP traffic that is destined to host 1 towards host 5. The host that now communicates with host 5 should still think that it communicates with host 1, since it addresses the same IP. Therefore, relevant forward (traffic from host X to host 1) and reverse (traffic from host 5 to host X) path installation and packet header rewriting are needed in order to enable the migration on the IP layer. The controller should act (HINT: *install_migrated_end_to_end_IP_path* function) in a similar way as in the context of the routing/forwarding task that was discussed in the beginning (HINT: *install_end_to_end_IP_path* function), including the transport protocol flow differentiation.

However, the controller should also instruct the switch to rewrite the packet headers to make the migration transparent to the rest of the network (remember the "Transparent Load Balancer" assignment). To enable this mechanism, the controller identifies the direction of the relevant packets (i.e, traffic stemming from host 5 or directed towards host 1), imposes the relevant header translations and then installs the corresponding randomly selected shortest paths. Note that the headers should only be rewritten at the switch that raised the corresponding *PacketIn*; the updated headers should then match the intermediary switches' flow tables. Therefore, relevant flow rules should be installed.

Be careful while rewriting the forward and reverse paths (HINT: you will need to use both the *install_forward_migration_rule* and *install_reverse_migration_rule* from the *SwitchWithPaths* instance for *FlowMods*, plus the *send_forward_migrated_packet* and *send_reverse_migrated_packet* for managing the *PacketIn* full packets via sending *PacketOuts*). Flow rules that match the new packet headers should be installed along the remaining path nodes. If the process is done correctly, IP traffic towards the old host should migrate transparently to the new host (addressed now to the new destination IP and MAC, after the rewrites), while traffic stemming from the new host should be masked as if it originated from the old host (using the old source IP and MAC, after the rewrites).

2.6.1 Sanity checks

Once you have finished your solution, you should check that the transparent migration properly works. Run your code using the following command:

```
~/pox> ./pox.py openflow.discovery CloudNetController --firewall_capability=False
--migration_capability=True
```

This ensures that your code for this task will run irrespectively of whether you have completed your previous task or not. In order to test a unified solution for both the isolation and migration tasks, you could simply run your network application using the following command:

```
~/pox> ./pox.py openflow.discovery CloudNetController --firewall_capability=True
--migration_capability=True
```

Initiate the Mininet topology script, and wait a bit for the controller to connect to the switches. The script will do an initial *pingall* as a sanity check.

```
sudo python clos_topo.py -c 2 -f 2
```

You should open one *xterm* at either host h3 or h7, continuously pinging host h1 and wait until the migration event is triggered (after about 180 secs from controller startup). Remember that you can tune the waiting value in the file; of course please make sure that the controller has enough time in the beginning to detect the network and achieve stable routing, before the migration happens. You might use Wireshark or Tcpdump to verify the accuracy of your solution, e.g., does the service (ping in our case) still run without interruptions? Is traffic redirected as expected? For example, pings should not break during the testing process, while no packet loss should be observed. Does UDP redirection also function properly? Are TCP sessions re-established? If a TCP session breaks during the move, what is the underlying reason: an issue with the network or the host state? Check with the scripts used for the sanity checks of step 2.

3 Final Notes

3.1 How to submit your code

Please submit your code using the TURNIN submission program [10]:

- Log-in to one of the CS department's systems.
- Create a folder named ask2.
- ask2 folder should contain the following 2 scripts:
  1. clos_topo.py
  2. CloudNetController.py
- Use cd to make the directory one level above ask2 your current working directory.
- Issue the following command:

  ```
  turnin assignment2@hy436 ask2
  ```

Regarding the code, comments are welcome to aid the examiner understand your code. The deadline for submission is: **November 13, 23:59**.
This script should work exactly as expected to get full grade. Points will be subtracted for missing or incorrect functionality. The penalty for late submission is 10% per day. The submitted code will be tested for plagiarism using plagiarism-detection software. Any attempt to plagiarize will be accordingly punished with 0 grade. The exercise weight is 25% of the overall lab grade.

3.2 Debriefing

All the students who have submitted their code are requested to attend the debriefing session, where a sample solution will be presented by the assistants and a discussion of the exercise will follow. The date of the debriefing session will be announced via email in Moodle forum.

3.3 Oral Exam

All the students who have submitted their code are requested to attend the oral exam, in order to present their solutions to the teaching assistants (bring your laptops please). A timeslot during the oral exam session will be assigned to each student using Doodle.
Attention:

- **Each student will only be examined during the timeslot assigned.**
- **During this session both the Assignments 1 and 2 will be examined.**
- **Both the timely submission and the oral exam session will contribute to the grading of the assignment.**

3.4 Asking for help

For any issues you may encounter regarding the exercise, please ask the teaching assistants during the exercise sessions (14:00-16:00) at A.125 on 24/10, 31/10 and 07/11, or post your questions in Moodle forum.
Before contact, please make sure that you have formulated your question clearly and that you have already studied the [1, 2, 3, 4, 5, 9] thoroughly. Also, please check the code and the comments that have been already provided within the code by the assistants.
Good luck!

4 Acknowledgments

This exercise was created based on cooperation among Dimitrios Gkounis and George Nomikos from FORTH Greece, and Vasileios Kotronis from ETH Zurich (now at FORTH-ICS).

References

[1] Clos Topology. `http://en.wikipedia.org/wiki/Clos_network`.

[2] Mininet Introduction: Creating Topologies. `https://github.com/mininet/mininet/wiki/Introduction-to-Mininet#creating`.

[3] Mininet Python API Reference Manual. `http://mininet.org/api/`.

[4] Mininet Walkthrough. `http://mininet.org/walkthrough/`.

[5] NetworkX. `https://networkx.github.io/`.

[6] POX Discovery Component Overview. `https://openflow.stanford.edu/display/ONL/POX+Wiki#POXWiki-openflow.discovery`.

[7] POX Discovery Component Source Code. `https://github.com/noxrepo/pox/blob/carp/pox/openflow/discovery.py`.

[8] POX Repository: Carp Branch. `https://github.com/noxrepo/pox/tree/carp/`.

[9] POX Wiki. `https://openflow.stanford.edu/display/ONL/POX+Wiki`.

[10] Turnin User Guide. `http://www.csd.uoc.gr/services/useful-info/use-the-turnin.html`.