Aim: Emulate and manage a Data Center via a Cloud Network Controller: create a multi-rooted tree-like (Clos) topology in Mininet to emulate a data center. Implement specific SDN applications on top of the network controller in order to orchestrate multiple network tenants within a data center environment, in the context of network virtualization and management.

Objectives:

1. To understand the concept of Software Defined Networking (SDN).
2. To emulate and manage a data center environment using a cloud network controller.
3. To learn how to create a multi-rooted tree-like (Clos) topology in Mininet and implement specific SDN applications to orchestrate multiple network tenants within a data center environment.

Requirements:

1. Computer with any Linux OS installed
2. Python Runtime Environment
3. Mininet Emulator
4. POX Controller

Theory:

Data Center:

- A data center is a large facility used to house computer systems and related equipment, such as storage devices and networking equipment.
- It is designed to provide a secure and controlled environment for these systems to operate in, as they are critical to the functioning of many businesses and organizations.
- Data centers are used to store, process, and manage large amounts of data and applications, such as websites, mobile apps, and databases.
- They are typically equipped with backup power supplies, cooling systems, and fire suppression systems to ensure the equipment is protected and can operate continuously.
- Data centers can range in size from small server rooms to large, multi-story facilities that can house thousands of servers.
- They can be operated by a single organization, or by multiple organizations who share the resources of the data center.
- Data centers can be located on-premises, meaning they are physically located at the same site as the organization they serve, or they can be located off-premises, in a separate location, such as a colocation facility or a cloud provider's data center.

- The location and design of a data center can impact its efficiency and environmental impact, so many data centers are designed to be energy-efficient and environmentally friendly.

Data Center (Clos) Topology:

- The Clos topology is a network topology that is commonly used in data centers to provide high levels of scalability and redundancy.
- It is based on a multi-rooted tree-like structure, with multiple layers of switches or routers, each of which is interconnected in a specific way to provide maximum performance and reliability.
- At the core of the Clos topology are three layers of switches or routers, known as the spine layer, the leaf layer, and the host layer.
- The spine layer consists of a set of high-capacity switches or routers that are interconnected in a full-mesh topology, providing high-speed interconnectivity between the leaf layer switches.
- The leaf layer consists of a set of switches or routers that are connected to each spine switch, forming a leaf-spine architecture. Each leaf switch is typically connected to a large number of host devices, such as servers or storage devices.
- The host layer consists of the individual devices themselves, which are connected to the leaf switches. In this way, the host layer can scale up or down as needed, while the spine and leaf layers remain relatively static.
- One of the key advantages of the Clos topology is its ability to provide high levels of redundancy and fault tolerance. By using a full-mesh spine layer, traffic can be easily rerouted around any failed switches or links, ensuring that the network remains operational even in the event of a failure.
- Another advantage of the Clos topology is its ability to scale horizontally, allowing additional leaf switches to be added as needed to support increasing numbers of devices.
- However, one potential disadvantage of the Clos topology is its complexity. Because of the multiple layers and interconnectivity between switches, it can be challenging to manage and troubleshoot in large-scale deployments.

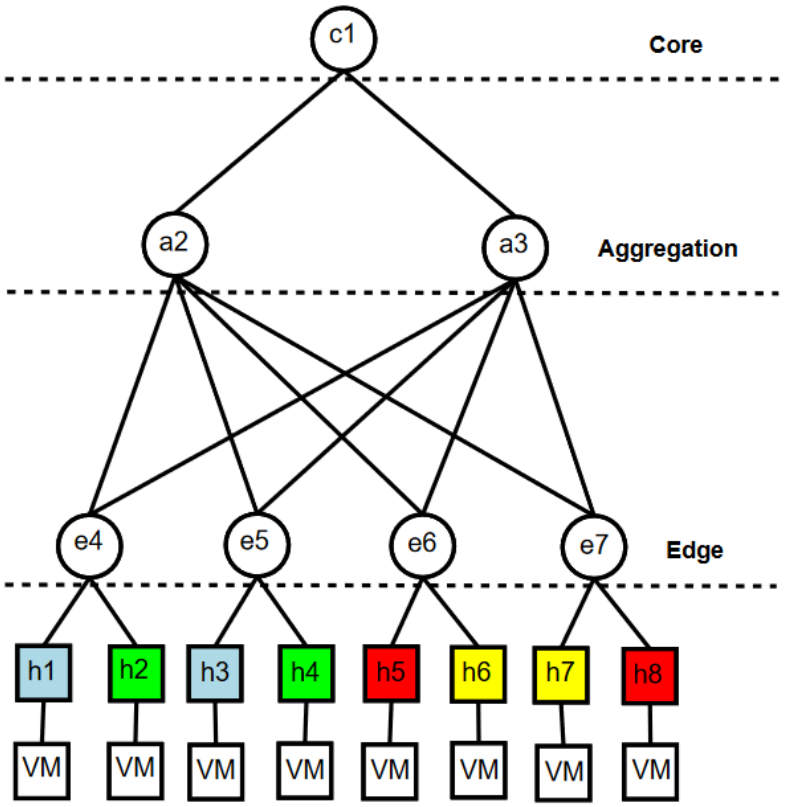

Figure 1: Sample layered data center topology

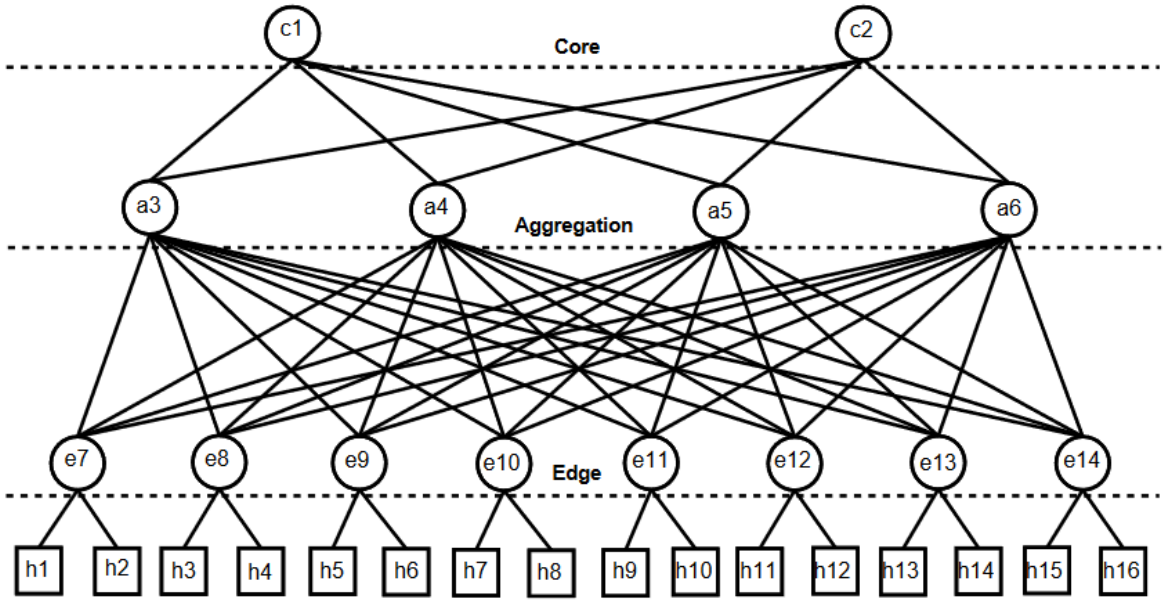

Figure 2: Clos topology with 2 core switches and fanout=2

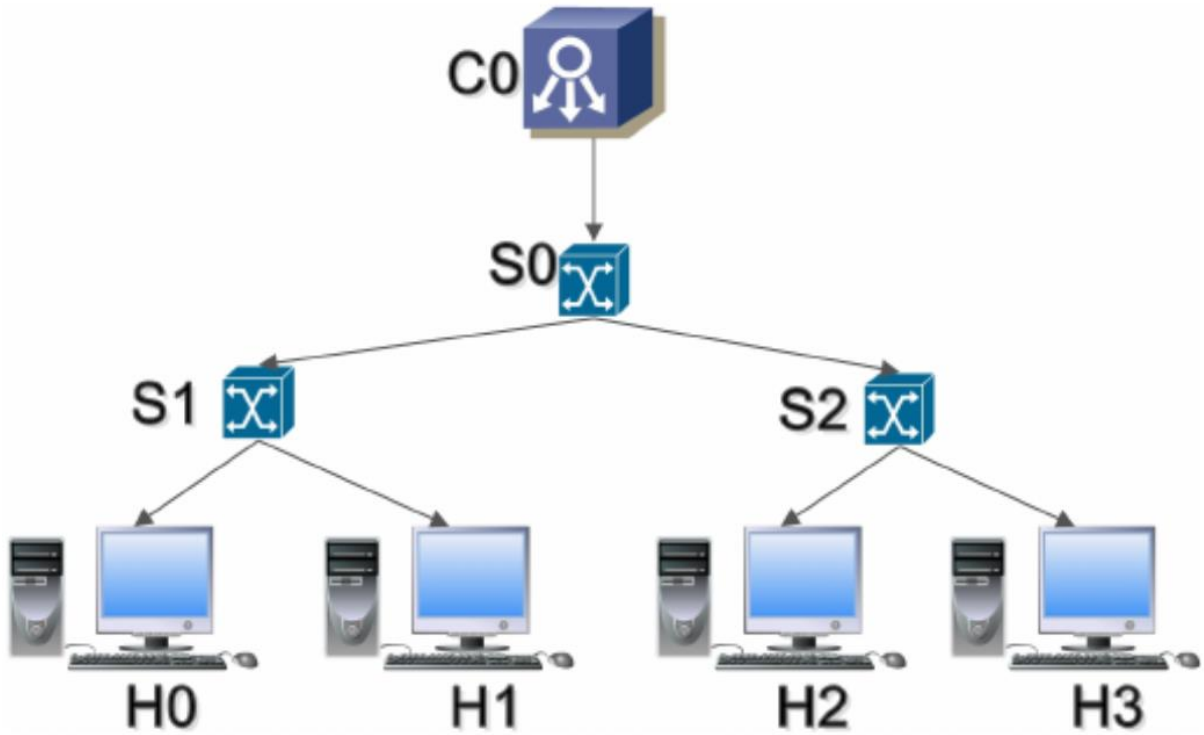

Figure 3: Mininet topology used for tree-type data center architecture

Procedure:

- **Step 0: Install Mininet and POX**
  - Mininet is a network emulator that allows you to create and run virtual networks on a single machine. POX is a network controller that allows you to program and control the behavior of the network using Python scripts.
  - You can install these tools by following the instructions on their websites: http://mininet.org/download/ and https://github.com/noxrepo/pox
  - Alternatively, you can use a virtual machine image provided by the course that already has these tools installed. You can download it from here: https://drive.google.com/file/d/1y9v9Z0n1yQl7y8J6wYw6j0p6bY4D7fL2/view
  - To run the virtual machine, you need to install VirtualBox: https://www.virtualbox.org/wiki/Downloads
  - To access the virtual machine, you need to use SSH: https://www.ssh.com/ssh/putty/download
- **Step 1: Run the script clos topo.py to create the data center network topology in Mininet**

- This script is provided by the course and it defines the structure and parameters of the network. The network consists of four layers: core, aggregation, edge, and host. Each layer has a number of switches and links that connect them. The switches are OpenFlow-enabled and can be controlled by POX.
- To run the script, you need to open a terminal and type: `sudo python clos_topo.py`
- This will create the network topology and start Mininet. You will see a prompt like this: `mininet>`
- You can use various commands to interact with the network, such as `pingall`, `dump`, `nodes`, etc. You can see a list of commands by typing `help`.

- **Step 2: Run the script CloudNetController.py to start the POX controller that will manage the network**
  - This script is provided by the course and it contains the code relating to the SDN applications that you will implement. The script uses POX's API to communicate with the switches and install flow rules on them. The script also defines some helper functions and classes that you will use in your tasks.
  - To run the script, you need to open another terminal and type: `./pox.py log.level --DEBUG CloudNetController`
  - This will start POX and load the script. You will see some messages like this: `INFO:core:POX 0.5.0 (carp) is up.`
  - The controller will connect to the switches and send them some initial flow rules. You will see some messages like this: `DEBUG:CloudNetController:Installing initial flow rules for switch s1`
- **Step 3: Implement basic routing**
  - To do this, we need to write a function called `task1_shortest_path_routing` that takes two parameters: `src` and `dst`, which are the source and destination host names, respectively.
  - The function should return a list of switch names that form the shortest path between the source and destination hosts. For example, if the shortest path from h1 to h4 is s1-s2-s4, the function should return ['s1', 's2', 's4'].
  - To find the shortest path, you can use the `get_shortest_path` function that is provided by the script. This function takes two parameters: `src` and `dst`, which

are the source and destination switch names, respectively. It returns a list of switch names that form the shortest path between them. For example, if the shortest path from s1 to s4 is s1-s2-s4, the function returns [‘s1’, ‘s2’, ‘s4’].

- To use the `get_shortest_path` function, you need to map the host names to switch names. You can use the `host_to_switch` dictionary that is provided by the script. This dictionary maps each host name to its connected switch name. For example, host_to_switch[‘h1’] returns ‘s1’.
- To write the function, you can follow these steps:
  - Initialize an empty list called `path`.
  - Get the switch name of the source host by using the `host_to_switch` dictionary and store it in a variable called `src_switch`.
  - Get the switch name of the destination host by using the `host_to_switch` dictionary and store it in a variable called `dst_switch`.
  - Call the `get_shortest_path` function with `src_switch` and `dst_switch` as parameters and store the result in a variable called `switch_path`.
  - Append `src_switch` to `path`.
  - Loop through `switch_path` from index 1 to the end and append each element to `path`.
  - Append `dst_switch` to `path`.
  - Return `path`.

- **Step 4: Implement firewall policies to block traffic between certain hosts or applications**
  - In this task, you need to write a function called `task2_firewall_policies` that takes three parameters: `src`, `dst`, and `app`, which are the source host name, destination host name, and application name, respectively.
  - The function should return a boolean value that indicates whether the traffic should be blocked or not. For example, if there is a firewall policy that blocks traffic from h1 to h2 for app1, the function should return True when called with src=‘h1’, dst=‘h2’, and app=‘app1’.
  - To implement the firewall policies, you can use the `firewall_`policies list that is provided by the script. This list contains tuples of three elements: source host name, destination host name, and application name. Each tuple represents a firewall policy that blocks traffic between those hosts for that application.

For example, ('h1', 'h2', 'app1') means that traffic from h1 to h2 for app1 is blocked.

- To write the function, you can follow these steps:
  - Initialize a variable called `blocked` with False.
  - Loop through the `firewall_policies` list and check if there is a tuple that matches the parameters.
  - If there is a match, set `blocked` to True and break out of the loop.
  - Return `blocked`.

- **Step 5: Implement virtual network slicing to isolate different tenants or applications**
  - In this task, you need to write a function called `task3_virtual_network_slicing` that takes three parameters: src, dst, and app, which are the source host name, destination host name, and application name, respectively.
  - The function should return a boolean value that indicates whether the traffic should be allowed or not. For example, if there is a virtual network slice that isolates tenant1 from tenant2, the function should return False when called with src='h1', dst='h3', and app='app1', where h1 belongs to tenant1 and h3 belongs to tenant2.
  - To implement the virtual network slicing, you can use the `virtual_network_slices` dictionary that is provided.

Python Implementation of the above:

```
def task1_shortest_path_routing(src, dst):
  # Initialize an empty list called path
  path = []
  # Get the switch name of the source host by using the host_to_switch
dictionary
  src_switch = host_to_switch[src]
  dst_switch = host_to_switch[dst]
  switch_path = get_shortest_path(src_switch, dst_switch)
  # Append src_switch to path
  path.append(src_switch)
  # Loop through switch_path from index 1 to the end and append each
element to path
  for i in range(1, len(switch_path)):
    path.append(switch_path[i])
  # Append dst_switch to path
  path.append(dst_switch)
  # Return path
  return path
```

```
def task2_firewall_policies(src, dst, app):
  # Initialize a variable called blocked with False
  blocked = False
  # Loop through the firewall_policies list and check if there is a tuple that matches the parameters
  for policy in firewall_policies:
    # If there is a match, set blocked to True and break out of the loop
    if policy == (src, dst, app):
      blocked = True
      break
  # Return blocked
  return blocked
```

```
def task3_virtual_network_slicing(src, dst, app):
  # Get the tenant name of the source host by using the host_to_tenant dictionary
  src_tenant = host_to_tenant[src]
  # Get the tenant name of the destination host by using the host_to_tenant dictionary
  dst_tenant = host_to_tenant[dst]
  # Check if the source and destination hosts belong to the same tenant
  if src_tenant == dst_tenant:
    # If they do, return True to allow the traffic
    return True
  else:
    # If they don't, check if there is a virtual network slice that allows traffic between them for that application
    # Use the virtual_network_slices dictionary to get the list of applications that are allowed for each pair of tenants
    # Use the tuple (src_tenant, dst_tenant) as the key to access the dictionary
    allowed_apps = virtual_network_slices[(src_tenant, dst_tenant)]
    # Check if the application is in the list of allowed apps
    if app in allowed_apps:
      # If it is, return True to allow the traffic
      return True
    else:
      # If it isn't, return False to block the traffic
      return False
```

```
kris@Ubuntu-22:~$ sudo mn --controller remote --custom datacenter.py --topo datacenter
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6653
Connecting to remote controller at 127.0.0.1:6633
*** Adding hosts:
h1 h2 h3 h4 h5 h6 h7 h8
*** Adding switches:
a1 a2 a3 a4 c1 c2 e1 e2 e3 e4 e5 e6 e7 e8
*** Adding links:
(a1, e1) (a1, e2) (a2, e3) (a2, e4) (a3, e5) (a3, e6) (a4, e7) (a4, e8) (c1, a1) (c1, a2) (c1, a3) (c1, a4) (c2
, a1) (c2, a2) (c2, a3) (c2, a4) (e1, h1) (e2, h2) (e3, h3) (e4, h4) (e5, h5) (e6, h6) (e7, h7) (e8, h8)
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8
*** Starting controller
c0
*** Starting 14 switches
a1 a2 a3 a4 c1 c2 e1 e2 e3 e4 e5 e6 e7 e8 ...
*** Starting CLI:
mininet> h2 ping h1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
From 10.0.0.2 icmp_seq=1 Destination Host Unreachable
From 10.0.0.2 icmp_seq=2 Destination Host Unreachable
From 10.0.0.2 icmp_seq=3 Destination Host Unreachable
64 bytes from 10.0.0.1: icmp_seq=4 ttl=64 time=2437 ms
64 bytes from 10.0.0.1: icmp_seq=5 ttl=64 time=1436 ms
64 bytes from 10.0.0.1: icmp_seq=6 ttl=64 time=428 ms
64 bytes from 10.0.0.1: icmp_seq=7 ttl=64 time=0.035 ms
64 bytes from 10.0.0.1: icmp_seq=8 ttl=64 time=0.034 ms
^C
--- 10.0.0.1 ping statistics ---
19 packets transmitted, 5 received, +3 errors, 73.6842% packet loss, time 18370ms
rtt min/avg/max/mdev = 0.034/860.299/2437.445/947.184 ms, pipe 4
```

Figure 4: Expected Output

Conclusion: Thus, we learned how to emulate and manage a data center via a cloud network controller using Mininet and SDN applications. We created a multi-rooted tree-like (Clos) topology in Mininet to emulate a data center with multiple hosts and switches. We implemented specific SDN applications on top of the network controller in order to orchestrate multiple network tenants within a data center environment, in the context of network virtualization and management. We also experimented with different traffic patterns and routing algorithms to optimize the network performance and reliability.