SOFTWARE DEFINED WIDE AREA NETWORKS

ZAHRA KHALID

ZEYNEP TEMIZEL

CONVERGED OPTIMIZED NETWORKS 15 HP

HALMSTAD UNIVERSITY

SUPERVISOR: OVE ANDERSSON

1.ABSTRACT

Wide Area Networks provide the connectivity between the users and their applications. The traditional way in which WANs are implemented is that branches are connected to the data centers via a single router. With the introduction of cloud applications, the traffic patterns evolved to add delays in the performance of applications due to backhauling and a lot of bandwidth consumed. Moreover, implementing hardware makes it less time and cost-efficient and increasing the complexity. The problems faced by the users in the traditional WANs have driven a requirement for more intelligently steering traffic across the WAN. A Software-defined WAN not only routes the traffic more intelligently but also provides high performance, reliability, and security. In addition to this software-based WAN are cost and time-efficient.

This project demonstrated how efficient software-defined wide area networks compared to traditional WAN by implementing SD-WAN topology in Cisco DevNET's virtual environment (Multi-IOS test Network) using ansible and a python template jinja2. Ansible is a configuration management tool used for deploying topologies that utilized in this project to deploy a topology in a virtual environment.

2. TABLE OF CONTENTS

3. List of Figures

1. INTRODUCTION

3.1 PROBLEM BACKGROUND AND HOW TO SOLVE IT

In today's world, tirelessly evolving technologies and growing enterprises have made data trafficking cumbersome. Establishing large networks means deploying, configuring, and maintaining a more significant number of hardware devices, ultimately consuming a lot of time and money. Adding a branch to an already implemented network is also a complex task in conjunction with all these drawbacks. All these difficulties have led to a roadmap to software-defined networks, which offers a more secure, easy to manage, time and cost efficient way to build a network. With the advancement in communication technologies, social media platforms, software as a services (SaaS) cloud technology, the need to provide connectivity to users with their applications from anywhere globally with any device is a major requirement.

In traditional networks, every router has a control plane that commands the data plane how to behave. This control plane is the software part of the device configured via CLI. On the other hand, the data plane is the hardware part where the data come from other devices routing to one another by the current configuration. These two different entities are placed in the same place in a router, acting as one entity.

The concept of the SDN is data plane and control plane are separated. The software part is pulled to a central point to see and manage the whole network, evaluate it and decide on it. Now the control plane is in a central location, and it can see the whole network; It has a command on the neighbor and routing tables of all devices on the network. Decision-making becomes much more innovative. On the other hand, SDN is abstracting the infrastructure of the network from the applications. In this way, SDN makes it possible for data planes to be programmable.

With a centered, executive control plane, Developers are free from the dependency of the hardware and head to the software to change the whole network at once.

2. MAIN

4.1 A SIMPLE LOOK TO MPLS VS TO SD-WAN

Traditional solutions on WAN connections allow developers to interconnect main offices with branches and share resources. Access to centrally located data services or applications can be shared. Traditionally dedicated circuits are used to achieve this connectivity things such as a frame relay or MPLS, which uses tags assigned to data packets, and the packet transfer process is done only depending on the contents of these tags, regardless of the contents of the packet. Although these provide reliability and security for the connection, modern networks require some rethinking of this cloud usage, so new ways to simplify the management of wide-area networks and include cloud resources in the network are needed.

Software-defined networks' goal is to control and manage the interactions between branch locations and central resources. One another significant advantages are that there is no longer a need for backhauling traffic. In **Figure 1**:

Figure 1

In **Figure 1**, there is a branch location connected over an MPLS circuit, connected back to a data center. At the data center, there are some advanced security and inspection happening, so that is why all the traffic from the branch should be tunnel back to the data center for inspection [1]. In this traditional LAN setup, all the traffic from the branch backhauled to the data center for those security services that include traffic destined to the cloud or the public Internet and traffic destined locally within the organization. This situation can cause many performance issues, delays and depend on the circuit speed; it can cause some bandwidth issues.

On the other hand, SD-WAN in **Figure 2**,

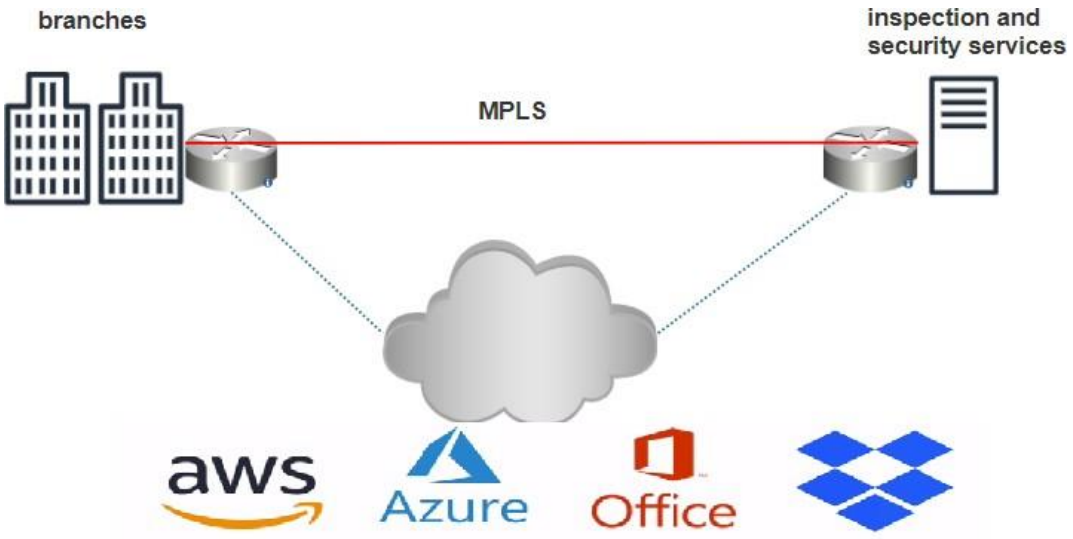

Figure 2

can interact with all kinds of cloud applications. Cloud-based technologies are more prominent nowadays. For example, how cheap and easy AWS storages now and how office 365 has turned to a cloud platform for enterprise email access. SD wan can easily interact with applications like AWS Dropbox, office 365, and many more [2]. This means that hosting applications in either public or private clouds will allow direct traffic between the cloud application and the branch location rather than backhauling all the traffic through a central data center. SD-WAN can intelligently control traffic flow to optimize traffic flow and reduce unnecessary bandwidth in networks. This situation does not have any security issues because SD-WAN provides end-to-end traffic encryption and inspection, an anti-malware system, and botnet intervention. [3]

SDN took the abstract logical plane concepts into actual devices. (**Figure 3**)

In the data plane, there are edge routers called vEdge. It is the software or hardware component that sits at the sites. It forms control plane connections with vSmart controllers and not between each other. Vsmart is the brain of the system. It stands in the control plane as a hub device and connects with vEdges. Its controllers advertise routing, security, and data plane policies [4]. vSmart is the device that establishes the control plane component of the architecture. For the management plane, vManage is the user interface of the system where users can control. vBond orchestrate connectivity in the whole system. For example, it tells vEdges how and where to connect the organization. [4]

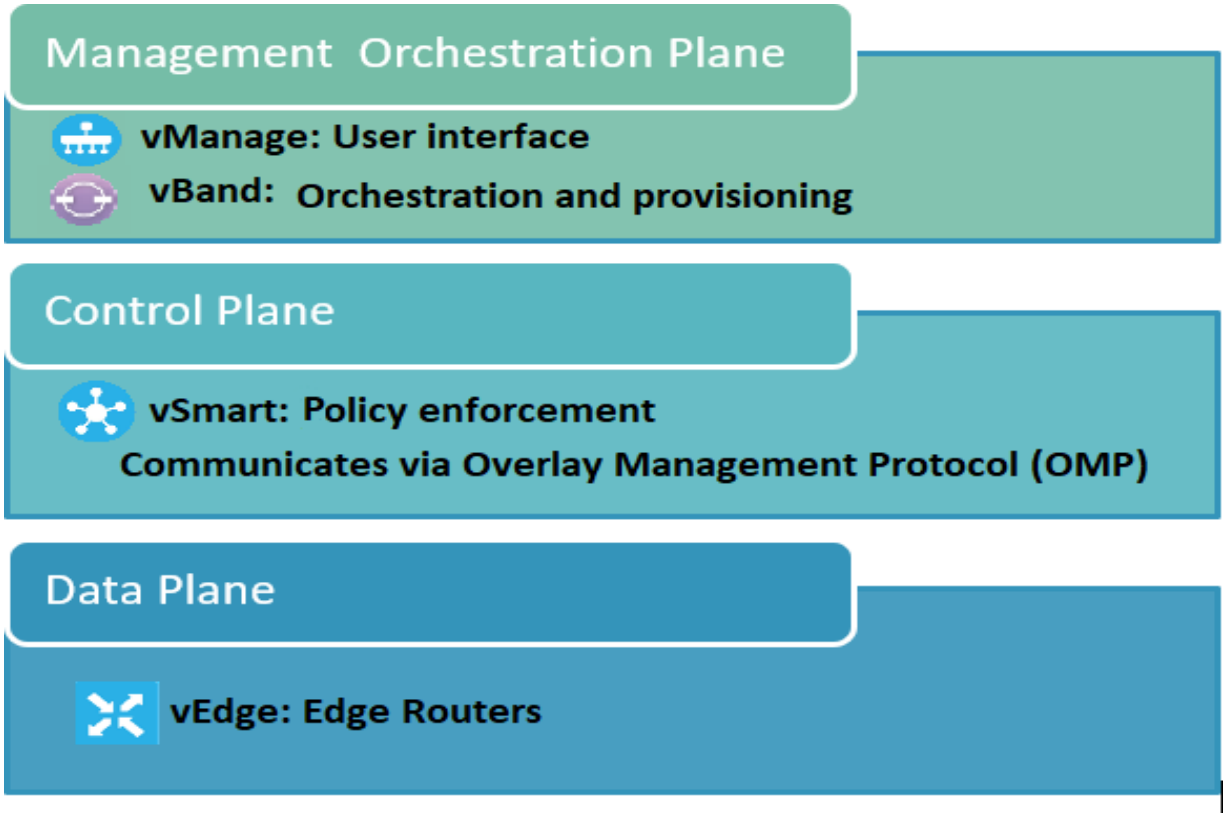

Figure 3

3. THE ACTION

Ansible and python template Jinja2 used for this part of the project.

Ansible is an automation system required in the orchestration plane that provides management ofin-house data centers, cloud structure, many servers in physical or virtual environments without the necessity for agent installation [5]. For example, application distribution, software provisioning, configuration management. Ansible uses the playbook to execute commands and jinja2 templates. Those playbooks are files that contain automatizations in YAML format. Jinja2 is a template developed for Ansible.

In this part of the report, SD-WAN solution is applied with Ansible using a sandbox Multi -IOS test Network and a VIRL environment in Cisco DevNet.

The objective of this act is to start a VIRL simulation and use Ansible to dynamically build a Virtualtopology from inventory data.

The basic process for automating VIRL topologies with Ansible is to:

- Create the required inventory data for each node in the simulation (stored in **virl.yml**)
- Run the **build.yml** playbook

5.1 STEP ONE

Cisco CPN has been used to make a connection with Cisco sandbox and VIRL in order to access SSH and developer mode.
The code below starts the Viptela SD-WAN simulation.

ansible-playbook build.yml

Figure 4

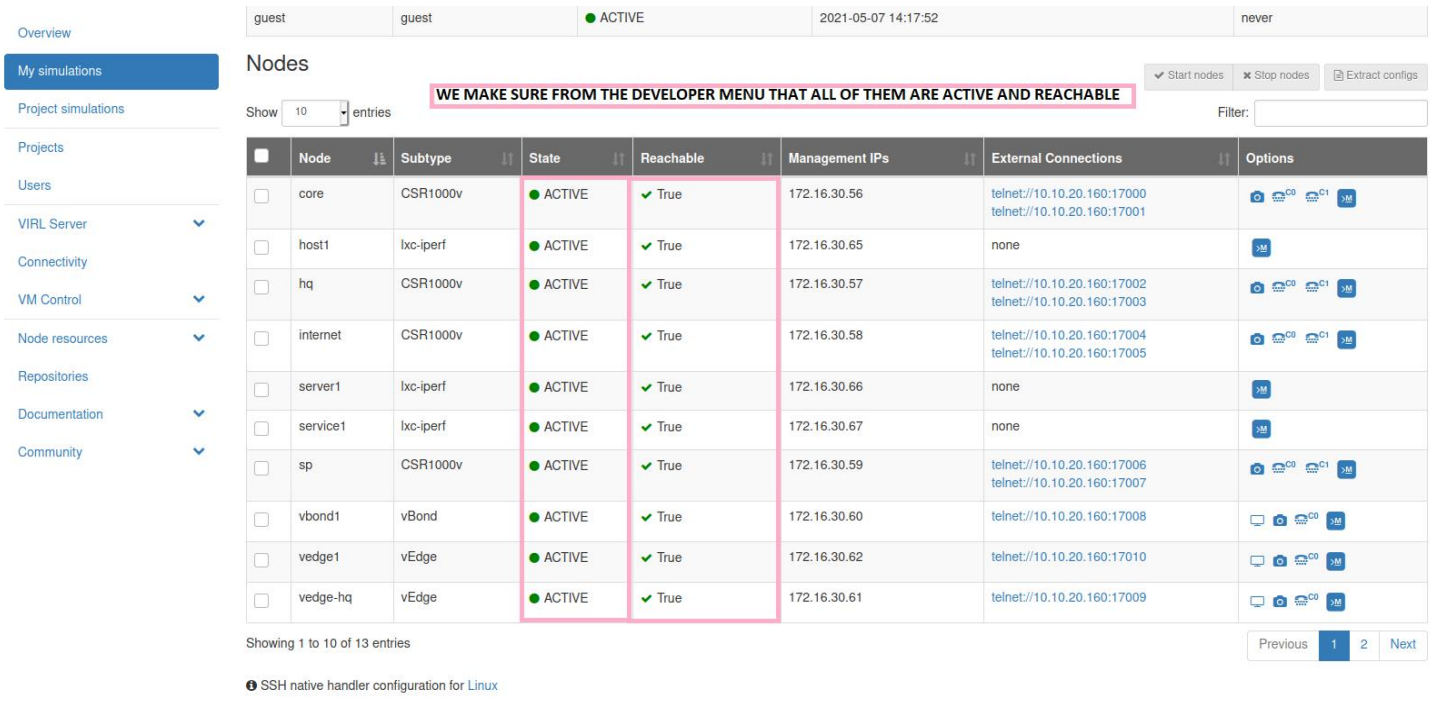

Figure 5

```
- name: Add host to topology
  hosts: all
  gather_facts: no
  tags:
    - group
  vars:
    virt_platform: none
  tasks:
    - block:
      - name: Check for VIRL information
        set_fact:
          virt_platform: virl

      - name: Generate Day0 config
        set_fact:
          day0_config: "{{ lookup('template', virl.config_template) }}"
        when: virl.config_template is defined
      when: virl is defined

    - name: Add to VIRL topology
      group_by:
        key: "virt_{{ virt_platform }}"
```

```
- name: Generate topology and start simulation
  hosts: localhost
  connection: local
  run_once: yes
  gather_facts: no
  vars:
    topo_file: topo.virl
    topo_name: "{{ topo_file.split('.')[0] }}"
    topo_id: "{{ lookup('password', '/dev/null length=4 chars=ascii_letters') }}"
  tasks:
    - name: Check for existing simulation
      stat:
        path: "{{ virl_sim_file }}"
      register: stat_result

    - block:
      - name: Generate the Topology
        set_fact:
          topo_data: "{{ lookup('template', 'virl/topology_v1.j2') }}"
          session: "{{ virl_tag }}_{{ topo_name }}_{{ topo_id }}"

      - name: Write debug topology file
        copy:
          content: "{{ topo_data }}"
          dest: topo.virl

      - name: Create simulation environment directory
```

Figure 6

This is the output of the playbook. In the first section of **Figure 6,** if the hosts in the inventory have VIRL data, the list of the hosts is added to it. After that, with the help of the jinja2 template specified in the virl.config, Day Zero configuration generated.

In the second on **Figure 6** section:

name: Generate topology and start simulation

is the main play. Once the hosts that contribute to the topology are set, this task will transform the VIRL data into a valid topology and launch the simulation. Generating the topology and storing its data happens in the Generate the Topology task.

The output of the code below will show the topology template.

cat templates/virl/topology_v1.j2

```
{#                                                                              #}
{# Globals                                                                      #}
{#                                                                              #}
{% set network_connections = {} %}
{% set global = {} %}
{% set _ = global.update({'node_count': 1}) %}
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<topology xmlns="http://www.cisco.com/VIRL" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" schemaVersion="0.95" xsi:schemaLoca
tion="http://www.cisco.com/VIRL https://raw.github.com/CiscoVIRL/schema/v0.95/virl.xsd">
    <extensions>
        <entry key="management_network" type="String">flat</entry>
{#        <entry key="management_lxc-iperf" type="Boolean">false</entry>#}
    </extensions>
{#                                                                              #}
{# Network Nodes                                                                #}
{#                                                                              #}
{% for node in groups.virt_virl %}
{# Add each host in the inventory to the topology if they have the 'virl'      #}
{# information defined                                                          #}
{%  if hostvars[node].virl is defined %}
{%   set node_number = global.node_count %}
    <node name="{{ node }}" type="{{ hostvars[node].virl.type|default('SIMPLE') }}" subtype="{{ hostvars[node].virl.subtype }}">
        <extensions>
            <entry key="ansible_group" type="String">virl_node</entry>
{%  if hostvars[node].day0_config is defined %}
{#                                                                              #}
{# Add day0 config                                                              #}
{#                                                                              #}
            <entry key="config" type="String">{{ hostvars[node].day0_config }}</entry>
{%  endif %}{# virl.config is defined  #}
        </extensions>
{%   for interface in hostvars[node].virl.interfaces|default([]) %}
{%    set network = network_connections[interface.network]|default([]) %}
{%    set network = network + [{'node': node_number, 'interface': loop.index}] %}
{%    set _ = network_connections.update({interface.network: network}) %}
        <interface id="{{ loop.index0 }}" name="{{ interface.name }}"/>
{%   endfor %}
    </node>
{%   set _ = global.update({'node_count': node_number + 1}) %}
{%  endif %}{# virl is defined #}
{% endfor %}
{#                                                                              #}
{# Networks                                                                     #}
{#                                                                              #}
{% for network, connections in network_connections.items() %}
{%  set node_count = global.node_count %}
    <node name="{{ network }}" type="SIMPLE" subtype="Unmanaged Switch">
{%  for connection in connections %}
        <interface id="{{ loop.index0 }}" name="link{{ loop.index0 }}"/>
{%  endfor %}
    </node>
{% endfor %}
{#                                                                              #}
{# Connections                                                                  #}
{#                                                                              #}
{% for network, connections in network_connections.items() %}
{%  set node_count = global.node_count %}
{%  set node_number = loop.index0 + node_count %}
{%  for connection in connections %}
    <connection dst="/virl:topology/virl:node[{{ node_number }}]/virl:interface[{{ loop.index }}]" src="/virl:topology/virl:node[{{ c
onnection.node }}]/virl:interface[{{ connection.interface }}]"/>
{%  endfor %}
{% endfor %}
```

Figure 7

Figure 7 is the template of the VIRL inventory data in XML. There are three sections that required for a valid VIRL topology: [6]

- **Network Nodes** - the nodes and their Day Zero configuration.
- **Networks** - the networks.
- **Connections** - the connections between nodes and networks.

Most of work done on the network nodes section and accomplish followings: [6]

- Iterate over all hosts with virl data defined.
- Set the VIRL node type (type of the host).
- If defined in the VIRL data, set the Day Zero config.
- Create a list of required networks.
- Create a list of required connections between nodes and networks.

The final topology createded by build.yml.

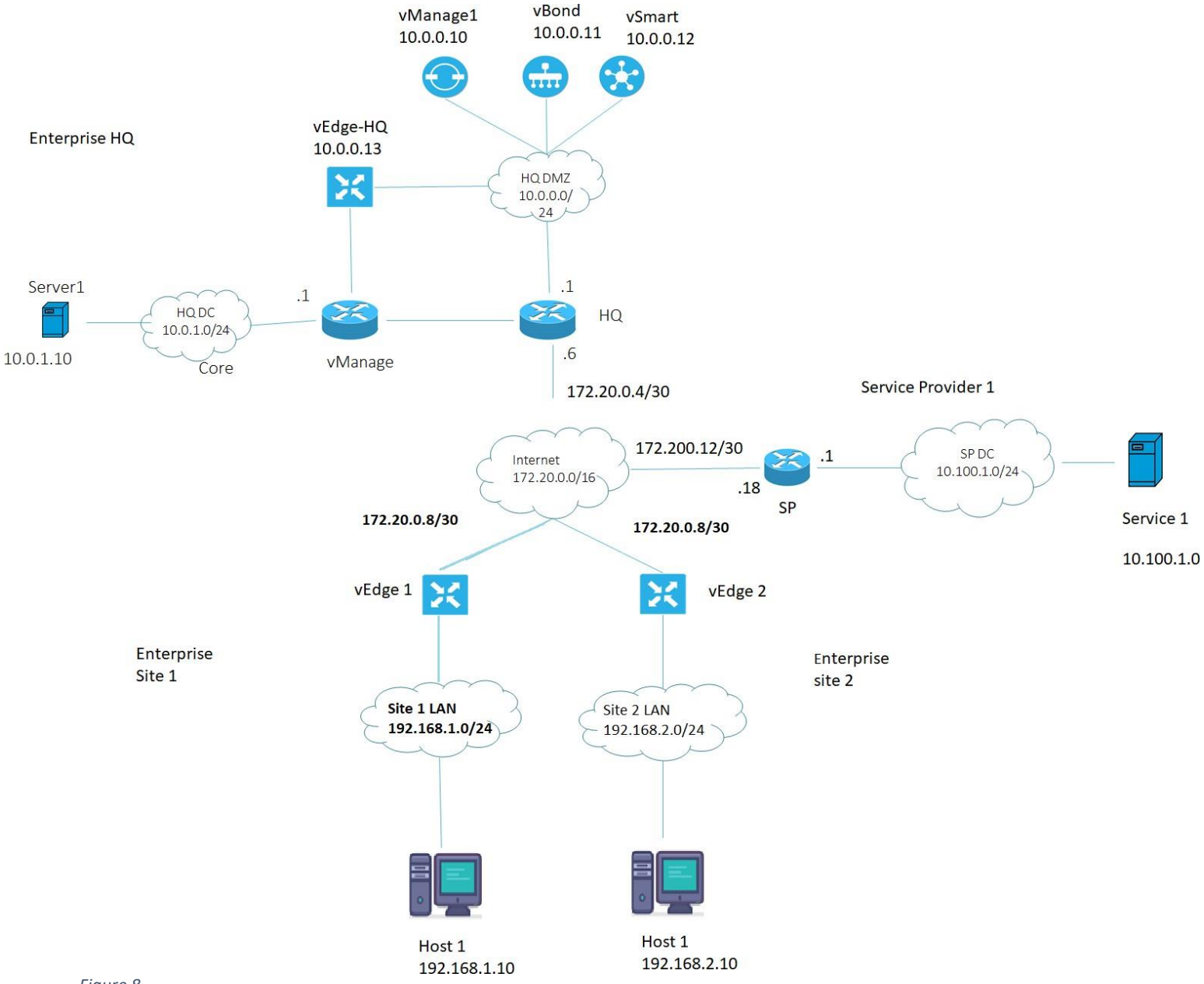

Figure 8

The topology is shown in above (**Figure 8**) has been implemented by executing the playbook.

5.2 STEP TWO

The Jinja2 templates used by the build.yml playbook to dynamically generate a VIRL topologyfile based on Ansible inventory data. As a result, the Ansible inventory data becomes the foundation for the Infrastructure-as-code; if the inventory data changes, so do the resulting VIRL topology file. The relationship between Ansible inventory data and the VIRL topologyis investigated in this step.

Every host in the simulation should have a virl.yml data file. In **Figure 9** data for vmanage and HQrouter hosts are shown.

```
(venv) [developer@devbox sd-wan-ansible-pipeline-code]$ cat inventory/host_vars/vmanage1/virl.yml
virl:
  subtype: vManage
  interfaces:
    - network: hq-dmz
      name: eth1
(venv) [developer@devbox sd-wan-ansible-pipeline-code]$ cat inventory/host_vars/hq/virl.yml
virl:
  subtype: CSR1000v
  interfaces:
    - network: hq-wan
      name: GigabitEthernet2
    - network: hq-dmz
      name: GigabitEthernet3
    - network: dc-core
      name: GigabitEthernet4
```

Figure 9

In **Figure 9**, first section is for vManage node. Eth1 interface connected to the network hq-dmz.
Properties down below are shown in the first section: [7]

- *VIRL subtype*
- *interfaces/networks this node will have*
- Day *Zero template to use, if any*

The second section is about HQRouter node. It connected to the hq-dmz in the GigabitEthernet3 interface. This is how the nodes are connected. In this case, v-manage directly connected to the HQ router.

With Ansible inventory data, any arbitrary VIRL topology can be created by combining the subtype, the interfaces, and the config template for each node. [7]

4. CONCLUSION

In practical work, configuration of a topology which consists of a branch of 10 devices, using playbook in ansible. This amount of configuration could have taken a lot of time without software Defined Network as each individual device has to be configured one by one. This is time, money, and prestige consuming. Also, new services sometimes took months to be available for all branches and users. However, with SD-WAN technologies, it is fast as a blink. Network branches can be added with zero-touch deployment. There is rarely any need to configure devices manually in the branch location. Centralized control makes the whole network more secure and easier to manage.

References

[1] N. Shah, "SD-WAN vs. MPLS: Why SD-WAN is a Better Choice in 2020," Fortinet, 9 September 2019. [Online]. Available: https://www.fortinet.com/blog/business-and-technology/advantage-of-sdwan-over-mpls. [Accessed 3 June 2021].

[2] M. Moravčík, P. Segeč and O. Yeremenko, "SD-WAN - architecture, functions and benefits," in *18th International Conference on Emerging eLearning Technologies and Applications (ICETA)*, 2020.

[3] R. Mehra, R. Ghai and B. Casemore, "SD-WAN: Security, Application Experience and," CISCO, 2019.

[4] C. Marshall, "Cisco SD-WAN Series Part 1 – Architecture Component," Looking Point, 7 June 2020. [Online]. Available: https://www.lookingpoint.com/blog/cisco-sdwan-architecture-components. [Accessed 3 June 2021].

[5] RedHat, "Ansible Documentation," RedHat, 19 May 2021. [Online]. Available: https://docs.ansible.com/ansible/latest/index.html?extIdCarryOver=true&sc_cid=701f2000001OH7EAAW. [Accessed 3 June 2021].

[6] CISCO, "Cisco Devnet Learning Labs step 4," CISCO, [Online]. Available: https://developer.cisco.com/learning/tracks/sd-wan_programmability/sd-wan-ansible-pipeline/sdwan-automation-virl-topologies/step/4. [Accessed 2021 June 3].

[7] CISCO, "Cisco Devnet Learning Labs step 1," CISCO, [Online]. Available: https://developer.cisco.com/learning/tracks/sd-wan_programmability/sd-wan-ansible-pipeline/sdwan-automation-virl-topologies/step/5. [Accessed June 3 2021].