Volume 62, Number 2, 2022

ACTA TECHNICA NAPOCENSIS
Electronics and Telecommunications

LEVERAGING SDN/SD-WAN IN VIRTUAL AND IOT DEPLOYMENTS THROUGH AUTOMATION

Robert BOTEZ, Gabriel LAZAR, Iustin-Alexandru IVANCIU, Virgil DOBROTA
Technical University of Cluj-Napoca, Communications Department, Cluj-Napoca, Romania
Robert.Botez@com.utcluj.ro; Gabriel.Lazar@com.utcluj.ro; Iustin.Ivanciu@com.utcluj.ro; Virgil.Dobrota@com.utcluj.ro

Abstract: This paper proposes a solution for the automation of Software-Defined Networking (SDN)/ Software-Defined Wide Area Network (SD-WAN) testbeds using Infrastructure-as-Code and Configuration Management techniques. Through automation, our solution also aims to reduce the difficulties raised by the implementation of the testbeds in various topologies with several hardware architectures. A virtualized scenario including hosts, switches and an SDN Controller was leveraged by Vagrant and configured using Ansible. We devised a second topology for edge deployments, incorporating multiple Raspberry Pi boards, to validate the interoperability of the solution. The experimental results showed that the highest value of the configuration time represents no more than half of the time needed in case of a manual approach.

Keywords: *Ansible, IoT, ONOS, SDN, SD-WAN, Vagrant.*

I. INTRODUCTION

Even though Software Defined Networking (SDN) was introduced in the early days of the last decade, the need for this technology has increased especially with the advent of 5G networks. As suggested in [1], the emergence of the 5th generation of mobile networks and beyond increased the demand for SDN, as this technology provides a wealth of opportunities to meet the requirements needed for deployments. Also, it is predicted that the SDN market size will reach almost 73 million dollars by 2027, compared to about 10 million dollars in 2019. The compound annual growth rate (CAGR) is expected to increase to 28.2% from 2020 to 2027. While the accelerated rate of adoption for SDN nowadays can be explained by the demands of deploying complex networks required in 5G or B5G, the use cases already existed even for older generation networks. Such a use case is the network slicing mechanism, which although mandatory for 5G, can be also enabled for 4G networks. The feasibility of introducing such a mechanism in previous networks using SDN was demonstrated in [2].

Before deploying a novel algorithm or a new feature in a network controlled by SDN, that feature should be extensively tested and validated in order to avoid, for example, connectivity issues or other unwanted behavior. For this, many network administrators and developers use different kinds of testbeds in order to implement and validate their applications before sending them in production environments. While many prefer network simulation tools such as Mininet [3] for its accessibility and ease of use, it can provide inaccurate network metrics which can lead to erroneous results. Thus, real testbeds or network emulators are needed in order to have a true validation of a certain SDN application. These observations are reinforced by the study conducted in [4], where the authors evaluate the usefulness of Mininet in a proper SDN virtualized testbed. They find that while Mininet can provide easy and fast deployment in the case of a low-complexity test environment, in the case of complex networks the configuration becomes even more complicated than if hardware or other virtualization solutions were used for the testbed. Also, in addition to the possibility of providing incorrect data, the authors suggest that Mininet also suffers from the network security point of view. Another paper [5] also points out that even though Mininet is consistent for many scenarios, some parameters are not consistent, such as the maximum Round Trip Time (RTT). Nonetheless, tools like Mininet remain a viable solution to test the early phase of an application before launching it in real environments. Having regard to the foregoing, other testbeds have been proposed to accurately reflect real-world scenarios, one of them being [6]. Here, the authors proposed a physical testbed, named SDN On-The-Go (OTG), consisting of four dedicated ZodiacFX SDN switches deployed on Raspberry Pi 3 hosts and a dedicated Kangaroo+ controller. Even though this solution can be used for real scenarios, the equipment involved costs approximately $1000 USD.

Our approach focuses on delivering an accurate environment for SDN and SD-WAN experiments, but without the need for dedicated equipment, so the cost of implementation is practically inexistent. We propose a fully automated virtual testbed, using the VirtualBox [8] hypervisor, and Vagrant [9] and Ansible [10] as automation tools. Using only open-source tools, the testbed can be run on any existing device which has VirtualBox installed. The logical topology of the testbed consists of an ONOS [11] SDN controller, two OpenFlow based virtual switches (we opted for Open vSwitch [12]) and two Linux virtual machines as hosts. We also validated our solution on another test bench composed of Raspberry Pi boards, which has another hardware architecture, to show its degree of integrability.

The motivation for this paper was to implement a testbed in which we could conduct accurate SDN and SD-WAN experiments in our research laboratory. Taking into account the difficulties we encountered during the implementation, we considered that the development of a general solution

Manuscript received: October 14, 2022; revised November 30, 2022

would bring benefit for other researchers looking to implement such a testbed. It is worth mentioning that even if we used VirtualBox, the solution can be extended to other virtualization solutions by changing the provider in the Vagrant configuration file, along with other specific configurations for the chosen provider. Finally, we created our topology only to prove the feasibility of our solution. Thus, changes should be made to the configuration files in order to change the topology. However, any type of topology can be automatically implemented with the right configurations using our solution, thus reducing extensive repetitive workloads and deployment times required for manual configurations.

The remainder of this paper is organized as follows. Section 2 presents the related work and a brief overview of SDN technology. The design and architecture of the proposed solution are discussed in Section 3, followed by experimental results and validation in Section 4. The paper ends with conclusions and future works.

II. RELATED WORK

In [13], the authors use an automated approach for an SDN-based testbed in order to evaluate the performance evaluation of multipath TCP (MPTCP) protocol. The testbed, consisting in network switches and hosts, was simulated and deployed in an automated manner over Mininet using a python application named MPTCP Test Manager. The authors state that through automation they could perform hundreds of experiments consecutively. However, they conclude that there are many factors that can affect the performance of an MPTCP connection; thus, the results obtained through a simulator should be validated in a real SDN network.

Another solution that sets out to implement an academic network testbed using SDN is presented in [14]. The authors tried to test the feasibility of using SDN rather than physical network resources in academia. Also, the challenges and obstacles encountered during the SDN adoption are presented. However, the networking component is emulated using Mininet. The feasibility of a complete migration from legacy to SDN-based networks should involve real equipment, for the reasons given in [4] and [5].

None of the solutions for deploying an SDN-based testbed discussed earlier include the CM of the SDN controller (SDNC). It is important to note that a fully automated deployment should include all the elements from resource provisioning (in virtual environments) to CM of all the components of the testbed: hosts, virtual or physical switches and SDN controller. Today, there are many types of SDN controllers (SDNCs) as well as studies on them. A very wide analysis was performed in [15] comparing twelve different SDN controllers. These were compared considering features like: programming language, quality of documentation, modularity, southbound and northbound interfaces, OpenFlow, OpenStack, platform and multithreading support and application domain. Also, different tests were done to evaluate the performances of these controllers in terms of throughput and latency. It was observed that the SDN controllers written in C/C++ had the highest performance in terms of throughput and Maestro had the best latency record. It was also observed that the controllers written in C and Java had the highest performance when varying the number of threads. However, the authors concluded that all the factors must be evaluated in order to do a fair comparison: ONOS and OpenDaylight present a very high modularity and are the most widely used controllers being integrated in many scenarios, whilst others, such as RYU, are better suited for research purposes.

Another study on the comparison between the most used SDN controllers was fulfilled in [16]. In this paper, the performance of four different SDNCs was evaluated: Ryu, Floodlight, ONOS and OpenDaylight. The authors extended the previous study [15] by adding an evaluation regarding the impact of the network load on these controllers. The comparison between the features of the controllers is the same as before. They concluded that in terms of integration with different vendors OpenDaylight is the best solution, but in terms of throughput ONOS had the best results.

III. OVERVIEW OF SDN

Software-Defined Networking [19] refers, in general, to the separation of the data plane from the control plane; this feature is mandatory for managing and configuring networks using software. In a legacy network, all the devices within that network had to be manually configured and managed, with the data and control planes being present on every of these devices.

The SDN concept is about moving the control plane from all the devices of a network, such as the one previously described, in order to have a centralized control plane for the whole network. This will lead to an adaptable network which will be sensitive to any change, such as a link failure or congestion, and will be capable of redirecting the network flow by changing the route. Given the above, SDN will deliver a more efficient use of the network capacity and will minimize the costs by virtualizing the actual network devices.

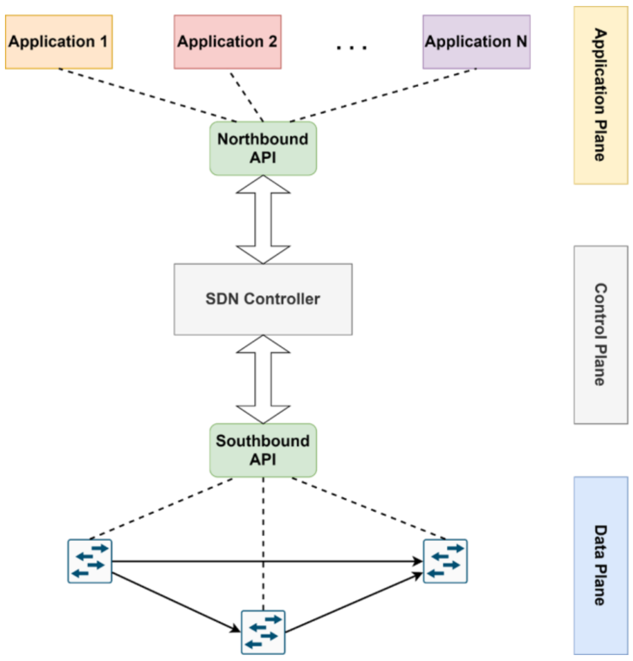


Figure 1. Software-Defined Networking Architecture

The SDN architecture is composed of three different layers (Figure 1) which communicate through APIs: the southbound API, which connects the Control and the Data planes, and the northbound API that connects the Application with the Control plane.

The Application plane consists of the applications and services implemented in order to assist the SDN controller in applying suitable policies for the network flows. Some of the common applications are monitoring modules that gather network metrics from the topology. These metrics are used to redirect the traffic in case of congestion.

The Control Plane represents the brain of the entire network; here resides the SDN controller which has a central vision of the network. The SDN controller takes instructions from the applications and applies them to the network components. It also sends network information back to the applications for analyzing the network state.

The Data Plane is formed by the network components. Different from a legacy approach, these components must be instructed by the controller to know how to forward the network traffic. The components can be switches, virtual switches, routers or other virtual or physical components.

The southbound API is used by the controller to communicate with the devices from the forwarding plane, so it could dynamically adjust the routes based on policies defined in the Application plane. One of the most used protocols for the southbound API is OpenFlow [22]. The OpenFlow protocol uses flows to define the path for a packet. These flows are stored in a flow table which is present on every OpenFlow switch and, based on the table flow, the switch will forward the traffic. Usually, when no flow matches a given packet, the switch will query the controller for the action it needs to take. Depending on the answer received, a new flow rule will be injected to forward the packet, or the packet will be dropped. The working principle of OpenFlow is illustrated in Figure 2. Nowadays other protocols can be also used for the southbound interface such as NETCONF, OSPF, MPLS or BGP [20].

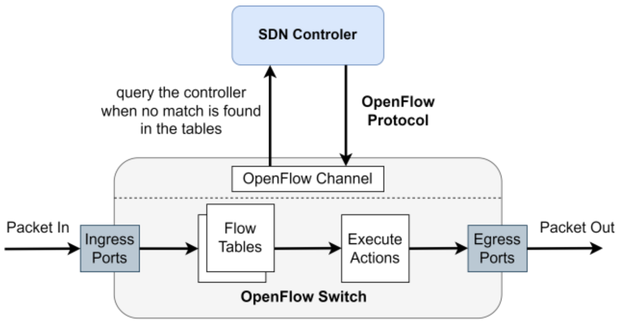


Figure 2. Working principle of OpenFlow

The northbound API is implemented as RESTful APIs and is used by the SDN controller to communicate with the Application plane. It is used by some applications and services which assist the SDN controller, but it can be also used for the integration with other tools and platforms such as OpenStack, Ansible, Chef or SaltStack [21].

III. ARCHITECTURE AND DESIGN

This paper presents a fully automated testbed for SDN/SD-WAN deployments using automation techniques, Infrastructure-as-Code (IaC) and CM tools such as Vagrant and Ansible. The proposed architecture is illustrated in Figure 3.

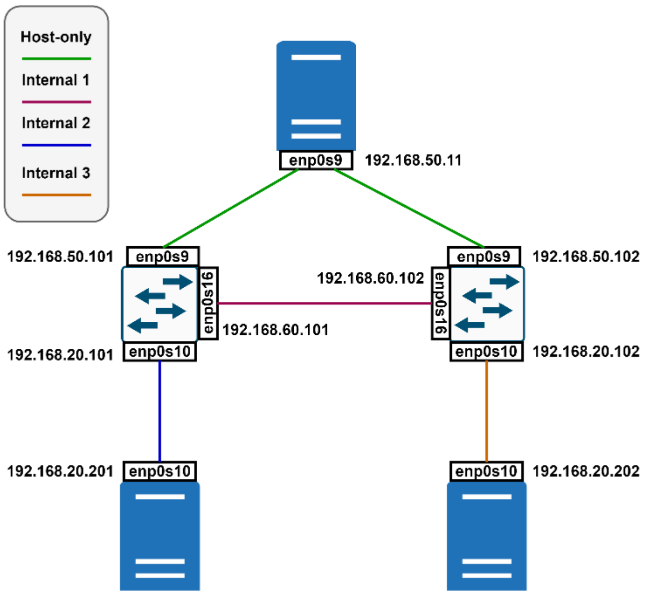


Figure 3. Network topology diagram

To implement the previous testbed, we used a management server having the requirements presented in Table I.

TABLE I. HARDWARE SPECIFICATIONS OF THE HOST

Operating System	Windows 10
CPU	Intel Core i7 7700 @ 3.60 GHz
RAM	32 GB DDR4
Storage	512 GB SSD
Network Adapter	Gigabit Ethernet

The flow of the process contains an Ansible playbook with 4 plays and will be described next. The first play includes the provisioning phase, but if the deployment is to be done on existing resources, this can be skipped. Otherwise, resources should be provisioned prior to implementation. In this step, a Vagrant file is copied to the target host. Based on the previous file, Vagrant will instruct the provider VirtualBox how to configure the VMs and also how to start them. We used five virtual machines configured as follows: one SDNC, two OpenFlow switches, and two hosts. Each virtual switch should have three network interfaces: one for communication with the controller, one to communicate with the other switch and one for communication with the hosts. The last two interfaces were added as ports to a bridge created by Open vSwitch. Another important thing to be mentioned here is that these interfaces which define the data plane should be set in promiscuous mode. Otherwise, the LLDP and ARP frames used for link and host discovery will not be forwarded and thus will not be visible to SDNC. The last steps imply the configuration of the virtual machines based on their Ansible roles: SDNC, vSwitch or host. In this step multiple applications needed for tasks such as host or link discovery, OpenFlow support or reactive forwarding were activated. Additionally, we created another playbook which is responsible for deleting the deployment, as well as the underlying virtual machines. The entire flow of leveraging the previous testbed is illustrated in Figure 4.

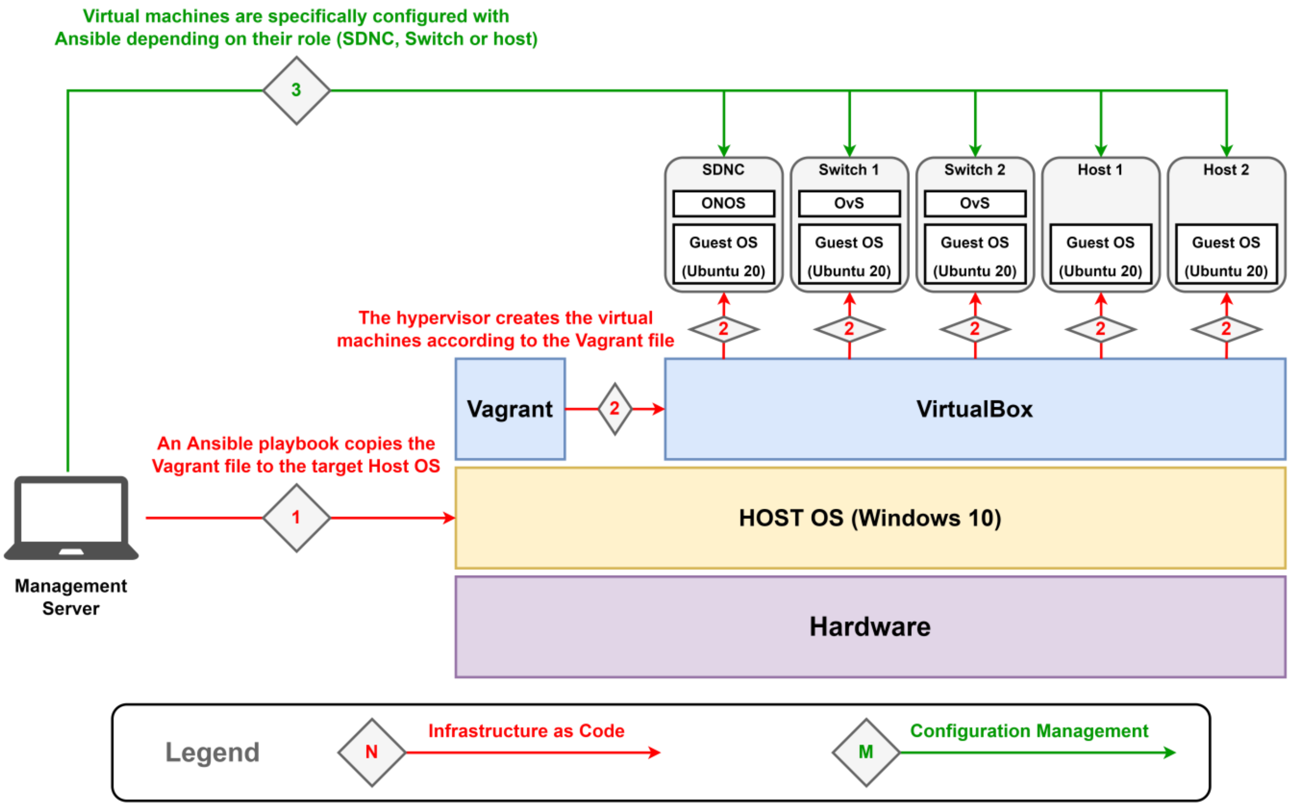


Figure 4. The flow of the process

The above configurations can also be used for physical instances running Ubuntu operating systems newer than version 18 or based on Debian. This is possible because the package manager is the same, apt, and the installed software packages can be found on all these distributions. However, the previous task was only a use case, but we aimed for a general solution for automating an SDN testbed. As a result, we developed an algorithm suited with both physical and virtual topologies. The algorithm is illustrated next.

Algorithm 1 SDN Testbed Deployment Algorithm

```
   Initialize the configurations and the number of:
       SDN Controllers -> SDNC,
       Switches -> S and
       Hosts -> H
   if infrastructure does not exist then
       for index =1, SDNC + S + H do
           Provision the virtual resources according to its configuration
       end for
   end if
   for index =1, SDNC do
       Deploy and configure the SDN Controllers
   end for
   for index =1, S do
       Deploy and configure the OpenFlow switches
       Connect the switches to the SDNC
   end for
   for index =1, H do
   Deploy the hosts
   end for
```

The quantity of resources must be stated as part of the input data for each entity, be it an SDNC, OpenFlow switch, or host. In the event that the infrastructure on which the testbed is to be built does not already exist, it is necessary to create it in advance. In this particular instance, it is given as data and input in addition to various settings for the components that have already been discussed, such as the number of CPU cores, RAM memory, IP addresses, and operating systems.

In such a case, each entity must be configured individually. The configuration stages are determined by the SDNC solution that is selected. In this particular piece of work, ONOS was used, and the procedures that needed to be carried out were outlined in advance. On the other hand, in our future work, we will include a flag that allows the user to choose the controller solution from a list that has been preset. In this instance, we will make use of many playbooks, the specifics of which will be determined by the controller version that is selected.

To illustrate the algorithm's versatility, we intended to evaluate it on a separate testbed with a different hardware architecture. For the second experiment, we chose a different topology focused on edge deployments. This topology consists of one Raspberry Pi 4 acting as an OvS, two Raspberry Pi Zero 2 W acting as hosts and one VM running Ubuntu 20 deployed in a private cloud orchestrated by OpenStack as SDNC. In this case, as stated above, we skipped the provisioning tasks, since this time the deployment is based on physical hardware. Only the virtual machine running the ONOS controller hosted in OpenStack was provisioned. The testbed together with its logical architecture are illustrated in Figure 5 and Figure 6.

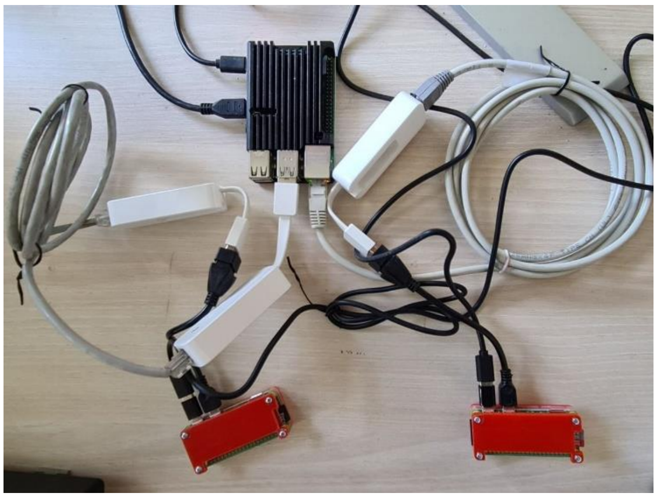


Figure 5. The second testbed consists of Raspberry Pi boards

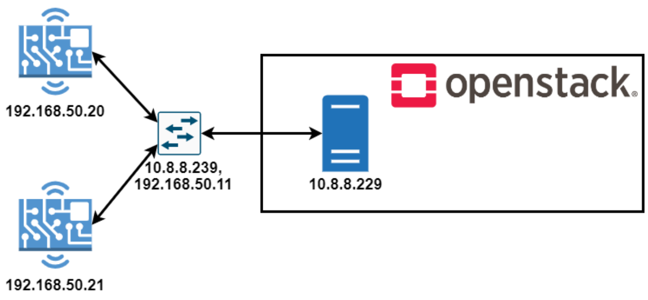


Figure 6. The logical architecture of the second testbed

As shown in Figure 5, we used three TP-Link USB 3.0 to Gigabit Ethernet Network adapters [17]. Theoretically, we could realize the deployment in fully wireless fashion without using additional components. However, we considered the wireless interface of the devices as a management network to connect to them via SSH. This is important for controlling the devices without a display, but also because Ansible configures devices through SSH.

IV. EXPERIMENTAL RESULTS

We validated our solution using two methods: first we created a TCP connection between hosts with the iperf [18] tool to check the connectivity and measure the throughput. The second experiment was to measure the time needed for both scenarios to complete. We used the default TCP windows size in iperf and a timeframe of 100 seconds. For the first scenario, we measured a throughput between 1.3 and 1.7 Gbps between hosts. Since all the virtual machines were created with the same hypervisor under the same host, the obtained values are strictly dependent on the host capabilities, which were presented in Table 1. For the second scenario however, the obtained values were dependent on the components. Since the Zero 2 W model has only one micro USB port, we connected an On the Go (OTG) cable to it and cascaded also to the TP-Link adapter. The throughput varies between 310 and 360 Mbps, which is below 480 Mbps, the maximum achievable data transfer with USB 2.0. The measured values were captured together with the IP addresses of the hosts for both scenarios in the ONOS GUI as illustrated in Figure 7 and 8. Note that the values in the figures are instantaneous values obtained during the measurement, which vary within the specified range.

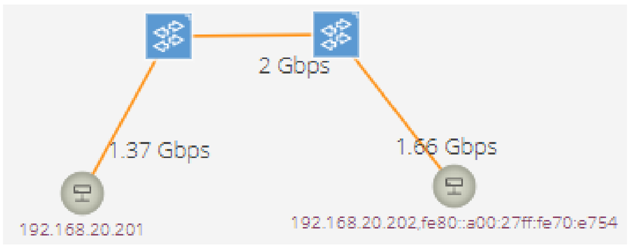
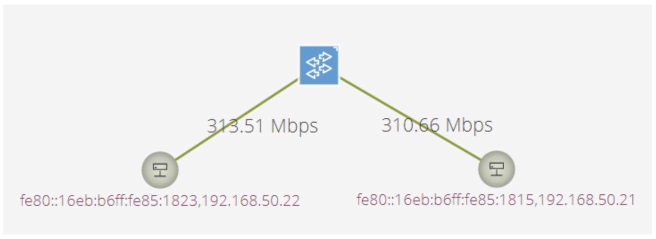
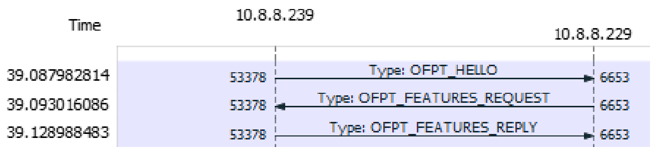


Figure 7. Connectivity between the hosts for the first scenario


313.51 Mbps
310.66 Mbps
fe80::16eb:b6ff:fe85:1823,192.168.50.22
fe80::16eb:b6ff:fe85:1815,192.168.50.21
Figure 8. Connectivity between the hosts for the second scenario

The OpenFlow handshake was captured between the Raspberry Pi 4, which acts as an OvS (10.8.8.239), and ONOS controller (10.8.8.229), as shown in Figure 9. For the sake of brevity, we only included the captures for one of the possible scenarios.


```
Time            10.8.8.239                                      10.8.8.229
39.087982814    53378  Type: OFPT_HELLO ---------------------->  6653
39.093016086    53378  <-------- Type: OFPT_FEATURES_REQUEST      6653
39.128988483    53378  Type: OFPT_FEATURES_REPLY ------------->  6653
```
Figure 9. OpenFlow handshake

Then, we measured the configuration time needed to deploy the entire testbed for both scenarios (Figure 10). The measured time for the virtualized scenario (depicted with orange) ranged between 563s up to 590s, with an average value of 575.8 seconds. For the Raspberry Pi deployment, the measured time (depicted with blue) varied from 328s up to 352s, with an average value of 341.8 seconds. We observed that the average configuration time for the Raspberry Pi deployment was smaller with almost 4 minutes (234 seconds) than the other one. This is because the provisioning step was omitted, and we used fewer devices. However, the average provisioning time was 1 minute and 5 seconds per instance. In our case, the provisioning time increases with the number of instances. Note that if we had chosen a cloud-based environment, we could have parallelized this step, thus obtaining a constant provisioning time. The maximum obtained time was 590 seconds (almost 10 minutes) for the virtualized testbed. But if a manual configuration were used to deploy that testbed the time required would be on average 20 minutes even for an experienced user. The time difference becomes much more noticeable if the required number of configuration instances is taken into account.

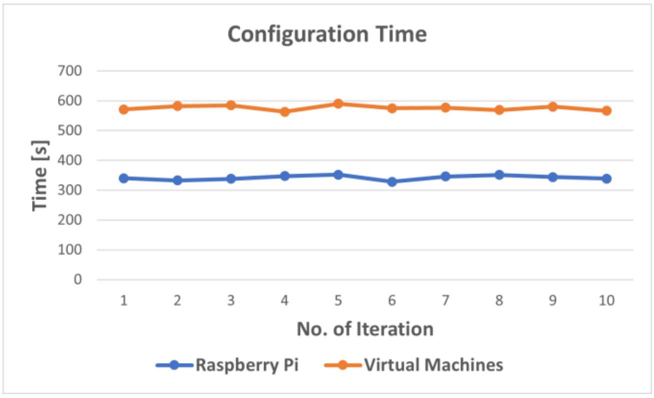


Figure 10. Configuration time for both scenarios

V. CONCLUSIONS AND FUTURE WORK

This paper presents a general solution for automatically deploying test benches for SDN/SD-WAN architectures. We involved Vagrant and Ansible as IaC and CM tools in order to achieve the automation degree needed for such an application. The testbed was validated on a virtualized environment using VirtualBox as a hypervisor, but also on a physical topology consisting of Raspberry Pi boards, thus proving in this way the high degree of interoperability that our solution possesses. Finally, we measured the configuration time for both the Raspberry Pi and virtual machines scenarios. We observed that the configuration time is higher for deployments that need resources to be provisioned prior to implementation. If in our case the provisioning time increases with the number of instances, in a cloud-based environment this step can be parallelized in order to obtain a constant provisioning time, regardless of the number of instances. However, we estimated that for the maximum value of 590 seconds obtained for the virtualized testbed, a manual approach should take at least a double amount of configuration time.

For future work, we plan to focus more on SD-WAN deployments and extend our experiments for multi-domains interconnectivity. We are also working on an application for monitoring metrics such as Available Transfer Rate (ATR), One-Way Delay (OWD) and Packet Loss (PL) in SDN topologies. By analyzing these metrics, we want to create an algorithm for calculating the optimal path in SDN and SD-WAN networks. The algorithm can be integrated into 5G backhaul networks to create network slicing mechanisms.

REFERENCES

[1] R. Rake, V. Gaikwad & V. Kumar, "Software Defined Networking Market: Global Opportunity Analysis and Industry Forecast, 2020-2027", Allied Market Search, 2020, [Online], Available: https://www.alliedmarketresearch.com/software-defined-networking-market.

[2] R. Botez, J. Costa-Requena, I.-A. Ivanciu, V. Strautiu, and V. Dobrota, "SDN-Based Network Slicing Mechanism for a Scalable 4G/5G Core Network: A Kubernetes Approach," *Sensors*, vol. 21, no. 11, p. 3773, May 2021, doi: 10.3390/s21113773.

[3] "Mininet: An Instant Virtual Network on your Laptop (or other PC) – Mininet", Mininet Project Contributors, 2022, [Online], Available: http://mininet.org/.

[4] O. Flauzac, E. M. Gallegos Robledo and F. Nolot, "Is Mininet the Right Solution for an SDN Testbed?," 2019 IEEE Global Communications Conference (GLOBECOM), 2019, pp. 1-6, doi: 10.1109/GLOBECOM38437.2019.9013145.

[5] J. Ortiz, J. Londoño and F. Novillo, "Evaluation of performance and scalability of Mininet in scenarios with large data centers," 2016 IEEE Ecuador Technical Chapters Meeting (ETCM), 2016, pp. 1-6, doi: 10.1109/ETCM.2016.7750830.

[6] J. Alcorn, S. Melton and C. E. Chow, "SDN On-The-Go (OTG) physical testbed," 2017 IEEE Conference on Dependable and Secure Computing, 2017, pp. 202-208, doi: 10.1109/DESEC.2017.8073808.

[7] "GNS3 – The software that empowers network professionals", GNS3, 2022, [Online], Available: https://www.gns3.com/.

[8] "VirtualBox Documentation", Oracle, 2022, [Online], Available: https://www.virtualbox.org/.

[9] "Vagrant", Vagrant 2022, [Online], Available: https://www.vagrantup.com/.

[10] "Red Hat Ansible Automation Platform, Ansible, 2022, [Online], Available: https://www.ansible.com/.

[11] "ONOS Project", ONOS 2022, [Online], Available: https://wiki.onosproject.org/display/ONOS/ONOS.

[12] Open vSwitch. Available: https://www.openvswitch.org/.

[13] N. Kukreja, G. Maier, R. Alvizu and A. Pattavina, "SDN based automated testbed for evaluating multipath TCP," 2016 IEEE International Conference on Communications Workshops (ICC), 2016, pp. 718-723, doi: 10.1109/ICCW.2016.7503872.

[14] M. Raza, S. Chowdhury and W. Robertson, "SDN based emulation of an academic networking testbed," 2016 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE), 2016, pp. 1-6, doi: 10.1109/CCECE.2016.7726828.

[15] O. Salman, I. H. Elhajj, A. Kayssi and A. Chehab, "SDN controllers: A comparative study," 2016 18th Mediterranean Electrotechnical Conference (MELECON), 2016, pp. 1-6, doi: 10.1109/MELCON.2016.7495430.

[16] L. Mamushiane, A. Lysko and S. Dlamini, "A comparative evaluation of the performance of popular SDN controllers," 2018 Wireless Days (WD), 2018, pp. 54-59, doi: 10.1109/WD.2018.836.

[17] "USB 3.0 to Gigabit Ethernet Network Adapter", TP Link 2022, [Online], Available: https://www.tp-link.com/ro/home-networking/computer-accessory/ue300/.

[18] "iPerf - The ultimate speed test tool for TCP, UDP and SCTP", iPerf 2022, [Online], Available: https://iperf.fr/.

[19] C. Jackson, J. Gooley, A. Iliesiu and A. Malegaonkar, "Cisco Certified DevNet Associate DEVASC 200-901 Official Cert Guide", Cisco Press, 2020.

[20] "What Are SDN Southbound APIs?", SDX Central, 2014, [Online], Available: https://www.sdxcentral.com/networking/sdn/definitions/southbound-interface-api/.

[21] "What are SDN Northbound APIs (and SDN REST APIs)?", SDX Central, 2014, [Online], Available: https://www.sdxcentral.com/networking/sdn/definitions/north-bound-interfaces-api/.

[22] "OpenFlow Switch Specification", Open Networking 2012, [Online], Available: https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf.