L2OVX: An On-demand VPLS Service with Software-Defined Networks

Jen-Wei Hu*,†, Chu-Sing Yang† and Te-Lung Liu*
*National Center for High-performance Computing, NARLabs, Tainan, Taiwan
Email: {hujw, tlliu}@narlabs.org.tw
†Institute of Computer and Communication Engineering, NCKU, Tainan, Taiwan
Email: csyang@mail.ee.ncku.edu

***Abstract*—Virtual Private LAN Service (VPLS) is a widely used network technology which connects geographically distributed customer sites as a local area network. However, the process of provisioning a new VPLS circuit is complicated because service provider has to check device configurations one by one. Therefore, it is difficult to provide the service on demand. OpenFlow, the most notable protocol in Software Defined Networking (SDN), manage the network devices via a well-defined set of instructions and exposes the handling capability of flows to a centralized controller. By this logically centralized programmatic model, we have a consistent and convenient way to dynamically allocate an end-to-end path for a VPLS service. In this paper, we present a system L2OVX which leverages OpenVirteX, a network virtualization platform, to realize the VPLS service on-demand in SDN. This solution provides layer 2 translation to achieve virtualization process with line-rate performance, and enables the load balance function to improve the transfer bandwidth among tenants. The result of evaluation shows L2OVX is competitive with OpenVirteX according to the TCP throughput on software-based OpenFlow switches and moreover, supports most hardware OpenFlow-enabled switches to provide the line-rate throughput. Furthermore, for scalability, the number of flow entries for one tenant remains constant in L2OVX while grows exponentially in OpenVirteX.**

***Index Terms*—Software-defined networking; network virtualization; VPLS; OpenFlow**

I. Introduction

With the growth of scale or business, there are several scenarios that require multiple distributed sites to work together and the demand for cross-site data-communication arises. For example, an institute or university with two or more campus areas; a data center that needs to interconnect the infrastructure resources (e.g., computing nodes and storage nodes) among different sites. The primary motivation behind *Virtual Private LAN Service (VPLS)* is to provide connectivity between geographically dispersed customer sites across MANs or WANs, as if they were connected using a local area network [1].

VPLS is an importance service of *TaiWan Advanced Research and Education Network (TWAREN)* network. Currently, there are more than 30 projects or institutes subscribed to this service. TWAREN uses line-rate interfaces on edge devices to provide VPLS services. However, the network interface is not cheap and vendor-dependent. Moreover, there is no single unifying abstraction that can be leveraged to configure the network devices. As a consequence, provisioning a network service can take months [2]. Therefore, there are challenges to design on-demand reliable services with efficient configuration, flexible deployment, and low cost. Recently, *Software Defined Networking (SDN)* achieves these requirements and provides unified interfaces to network devices that is free from vendor lock-in.

The concept of SDN is based on the idea of separating the control plane from the data plane of network equipments (e.g., switches and routers). Hence, the control and data planes are decoupled, network intelligence and state are logically centralized, and the underlying network infrastructure is abstracted from applications. The control plane is implemented in software that runs on separate servers while the data plane might be realized in software or networking hardware. *OpenFlow* [3] is the most popular protocol of SDN that enables networks by giving a remote controller the access to modify the behaviour of network switches through a well-defined forwarding instruction set. Therefore, applying SDN concepts, we can centralize the control of these devices and easily create a path between end-points on demand for a VPLS service.

In this article, we implement a system based on *OpenVirteX* [4], call L2OVX, which provides VPLS-like service in SDN. Our main contributions are as follows:

1) L2OVX provides VPLS-like functions with lower cost and on-demand configuration to improve the efficiency.
2) With modifications to layer 2 fields, L2OVX is compatible with hardware of OpenFlow switches for line-rate performance as compared to OpenVirteX which requires layer 3 modifications.
3) L2OVX enables link load balancing over multiple paths.

The remainder of this paper is organized as follows. In Section II, we discuss related works on the network virtualization and examine two novel platforms which implement this concept in SDN. In Section III, we present the architecture of L2OVX and its components for solving the challenges encountered in previous network virtualization platforms. In Section IV, we evaluate our system in terms of the throughput for a TCP connection, the impact with the number of services increases, and link load sharing. Finally, we conclude the paper and present future works in Section V.

II. Related Work

Network Virtualization [5], [6] provides the concept of a virtual network which is decoupled from the underlying physical infrastructure. With this technology, multiple isolated virtual networks with different addressing and forwarding mechanisms can share the same physical infrastructure. *FlowVisor* [7] enables multiple virtual networks by slicing network resources and delegating the control of each slice to a single OpenFlow controller. It acts as a transparent proxy between OpenFlow controllers and OpenFlow switches. Slices can be defined by any combination of the fields from layer 1 to layer 4 (e.g., switch ports, MAC address, IP address, and TCP/UDP port). FlowVisor guarantees isolation among all slices in its controlled scope if non-overlapping flow spaces exist. This arrangement allows multiple OpenFlow controllers to run virtual networks on the same physical infrastructure while ensuring that each controller touches only the switches and resources assigned to it. However, using these slices need to follow the pre-defined flow spaces that leads to FlowVisor had several limitations for tenants to develop their upper application on controllers. Moreover, the slice manager has to manually configure each of flow spaces through a full or a subset of network equipments in physical topology. Hence, configuration complexity increases exponentially with number of tenants and controlled equipment.

FlowVisor realizes the network isolation but lacks of having full configurable topologies and on-demand creation, destroy, and other maintenances on virtual networks. Therefore, authors in [4] propose another network virtualization platform, called OpenVirteX, which can i) provide address virtualization to keep tenant traffic separated, ii) provide topology virtualization to enable tenants to specify their topology. Similar to FlowVisor, they present OpenVirteX as a network hypervisor that enables operators to provide network virtualization to their customers. By exposing OpenFlow networks, OpenVirteX allows tenants to use their own NOS to control the network resources corresponding to their virtual network. In other words, OpenVirteX creates multiple virtual software defined networks out of one. Unlike FlowVisor, which simply slices the entire flow space amongst the tenants, OpenVirteX provides each tenant with a fully virtualized network featuring a tenant specified topology and a full header space.

However, in order to achieve these features, the edge switches in OpenVirteX need to enable layer 3 (e.g., IP address) modification in OpenFlow action field. Currently, most OpenFlow-enabled network equipment in the market does not support this feature or implements it by software. This leads to poor performance in processing the frames, and also limits the use of OpenVirteX for realizing network virtualization in physical network infrastructure. Moreover, if a tenant adds new devices in its virtual network, OpenVirteX needs to manually bind these devices' MAC addresses to specific ports for the correctness in address translation. This leads the complexity to configure and maintain the virtual networks.

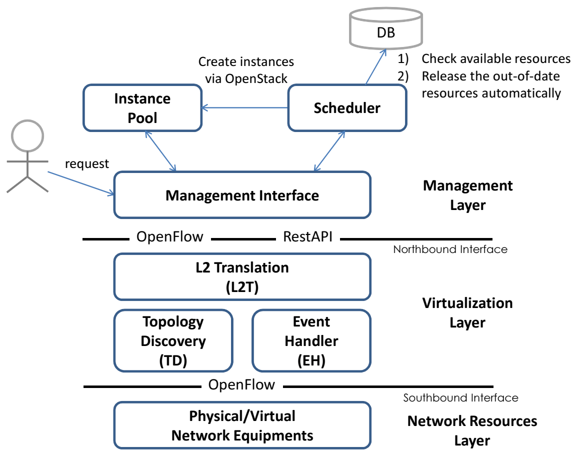


Fig. 1: The architecture of L2OVX approach

III. The L2OVX Approach

Currently, the network virtualization is considered a fundamental enabler in several areas, such as cloud computing, *Network as a Service (NaaS)* [8], and experimental testbeds [9]. We design the architecture of L2OVX as a three-layered model, which is depicted in Figure 1, to map these three basic components, called *Network Resources layer*, *Virtualization layer*, and *Management layer*. We show an overall view of L2OVX by summarizing the functionalities of these layers in this section.

A. Network Resources Layer

This layer contains the number of available network resources which are shared to tenants according to different requests. The only requirement of these network resources is need to support OpenFlow protocol. In general, we can divide the network resources into two types: OpenFlow software switch (e.g., virtual switch) and OpenFlow hardware switch. Software switch is commonly applied in cloud computing to virtualize the underlying network. These switches run on the hypervisor and provide network isolation and communication among virtual machines. Popular open source examples which support OpenFlow protocol are *Open vSwitch* [10], *Lagopus* [11]. The other type is hardware-based OpenFlow switch. Currently, more and more network manufactures, such as Brocade, Edge-core, HP, Pica8, and etc., implement OpenFlow protocol in their equipment. For virtual switches, we do not consider the hardware acceleration in the flow matching process. The reason is all flows are handled by the server. As a result, the performance is dependent on the CPU power. However, hardware-based switches rely on TCAM to process packets instead of CPU. Switches of this type present two known issues in regard to the number of flow entries and line-speed modification.

1) Flow Entries Limitation: Because every hardware-based switch has a finite amount of TCAM, it can only support a limited number of flow entries. The size of the flow table, which stores these entries, is from 1500 to 4000 among

TABLE I: A Comparison of supported actions in OpenFlow switches.

	Brocade 6610	Edge Core AS4600	HP 2920/5400	Pica8 3297
L1 (Switch Port)	Yes	Yes	Yes	Yes
L2 (MAC address)	Yes	Yes	Yes	Yes
L2 (VLAN)	Yes	Yes	Yes	Yes
L3 (IP address)	No	No	Yes(S/W)[a]	Yes(S/W)[a]

[a] S/W: The supported action is implemented by software.

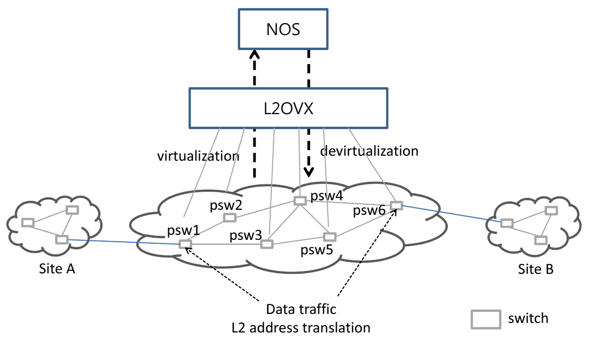


Fig. 2: The mapping between physical and virtual components in L2OVX.

different vendor switches. Although some switches have the ability to support more flows if the inserted flow entries only required to match Layer 2 fields, the flow table is a very limited resource that should be considered when designing our system.

2) Support for the Match Fields and the Actions in OpenFlow: OpenFlow flow entries contains two important components: match fields and actions. Each incoming packet is matched against a set of rules (e.g., match fields), and the action list (e.g., actions) associated with the matching rule is executed. Most current OpenFlow switches have already supported to process match fields (e.g., 12-tuple in OpenFlow 1.0 and more than 30-tuple in OpenFlow 1.3) in the flow entries by hardware. But for actions, only few are supported or implemented by software. In our verification of OpenFlow-enabled switches from various vendors, most of them only support a number of actions which allow the modification of layer 1 and layer 2 field in packet headers, as shown in Table I. This means that our solution should consider not only efficiently using limited flow table but also separating different tenants by using layer 2 mechanisms.

B. Virtualization Layer

The network virtualization allows the same physical resources to be shared by various services at the same time. Although the current OpenVirteX works properly in software-based OpenFlow switch, it does not fit in most OpenFlow-enabled hardware switches due to its layer 3 dependent translation mechanism. Therefore, by modifying several components in OpenVirteX, we propose the core of L2OVX - Virtualization Layer, which is composed of three major parts: *Topology Discovery (TD)*, *Event Handler (EH)*, and *L2 Translation (L2T)*.

1) Topology Discovery (TD): The TD is responsible for collecting all information from the underlying network resources. These information is including managed devices, all ports of each device, and links between devices. The first two parts can be retrieved via the initial OpenFlow protocol handshake. Each OpenFlow switch initiates a connection to the controller in the beginning, thus in this way we know all controlled devices. Then, the controller sends a *OFPT_FEATURES_REQUEST* message to each connected switch, asking the ability and current status including active ports. Finally, the links information learned from received LLDP packets which is generated periodically by the controller. Becuase OpenVirteX is a specific controller, it has already implemented this service. Therefore, our TD leverages the topology discovery in OpenVirteX to discover not only the software OpenFlow switch but also hardware one.

2) Event Handler (EH): Because virtualization layer acts as a transparent layer, it should ensure multiple tenants can transmit any type of packet (e.g., ARP, LLDP, IP, ICMP, TCP, and etc.) on their virtual network with no restriction. As described above, most of OpenFlow controllers use LLDP to discover links information among the underlying devices. But in OpenVirteX, it ignores all LLDP packets from controllers of its tenants. Currently, it only handles the LLDP packet generated by itself. If tenants would like to deploy OpenFlow-enable devices in their virtual networks, they cannot properly find out the topology of controlled OpenFlow-enable devices. For solving this problem, we design a process to handle it. When L2OVX receives OpenFlow message containing a LLDP information in *PACKET_IN* event, the process checks if this LLDP is generated by L2OVX, then it will update the topology. Otherwise, we forward these packets to the ports belonging to specific tenants. In this way, our system separates different LLDP packets and assures tenants can properly discover the network topology.

3) L2 Translation (L2T): The L2T is the core component in our L2OVX architecture. It is responsible for guaranteeing the isolation among all tenants to protect each one from interference caused by the others. As stated in [9], using a common layer to define a flow entry can simplify and ensure the isolation among tenants. In addition, we previously mentioned that most OpenFlow-enabled network devices only support layer 2 actions modification by hardware, as shown in Table I. As a result, our L2T chooses layer 2 (e.g., VLAN, source and destination MAC address) as the common layer to separate its tenants. In this way, L2T not only simplifies the isolation process, but also achieves line-rate performance for forwarding tenants' packets.

There are two core processes, the *virtualization* and the *devertualization*, in L2T. The former is the traffic from physical network to virtual one and the devertualization is defined the reverse direction of the virtualization, as illustrated in Figure 2. We will discuss in more detail in the following paragraph.

4) Virtualization and Devirtualization: First, we present the virtualization algorithm, as shown in Algorithm 1. This procedure dispatches the incoming packet to proper tenant. In

OpenVirteX, it has to bind the MAC address of each device to one specific port. As a result, OpenVirteX requires an extra effort to update the mapping if a user adds or removes devices. Furthermore, the number of flow entries is increased with the connections among tenants' devices. L2OVX identifies the tenant by the switch edge ports instead of the source MAC address. In this way, L2OVX is independent of the devices connected on the physical switch port. Compared to OpenVirteX, our system reduces the number of flow entries and increases the scalability of user size.

Algorithm 1 Algorithm of Virtualizing a Packet to the Specific Tenant

```
1: procedure VIRTUALIZE(psw,inport,data)
2:     $tid = fetchTenant(inport)$;
3:     if $tid \neq \emptyset$ then
4:         $vsw = fetchVirtualSwitch(psw, tid)$;
5:         $vp = fetchVirtualPort(inport, tid)$;
6:         $sendPacket(vsw, vp, data)$;
7:     end if
8: end procedure
```

The procedure *virtualize(psw,inport,data)* is executed when an incoming packet appears on the port (e.g., $inport$) of switch psw. Line 2 retrieves the tenant by passing the parameter $inport$. According to the tenant, we can find the virtual switch and virtual port in Line 4 and Line 5 respectively. Finally, Line 6 sends the incoming packet $data$ to the port vp on the virtual switch vsw.

Next, we describe the devirtualization process of L2OVX, as shown in Algorithm 2. In physical switches, each edge port belongs to exact one user while the core ports are shared among the users of L2OVX. When a user sends a packet to the core ports, L2OVX chooses a unique identifier for the user and attaches it on this packet to achieve the isolation in physical switches. L2OVX makes use of VLAN id field as this identifier because most OpenFlow-enable hardware devices support only layer 2 modification in OpenFlow action, as shown in Table I. It is important to note that this modification is performed in L2OVX and does not expose to the service users. That means there is no configuration overhead in user side.

By these two algorithms, we can properly separate different VPLS services by VLAN. That means each service is isolate with the others. Although the maximum number of using 4096 VLANs, it may be the limitation of our system. However, as described in [9], the major limitation factor is the number of flow entries supported by the OpenFlow devices. For example, in Pica8, the flow number for 1Gbps-interface switch is 2048 and 10Gbps-interface switch is 1024. Furthermore, we will replace new OpenFlow library which supports OpenFlow 1.3 in the future. At that time, we can use Q-in-Q encapsulation to extend the number of VLAN.

5) Link Load Balance: For improving the link utilization and load balancing tenants' traffic, we implement a mechanism to achieve this feature. As stated in Line 3 of Algorithm 2, L2OVX allocates paths for a tenant. Assume two or more tenants request VPLS services from the same two edge switches, they will have the same primary path. It causes poor data transfer performance because tenants may use the specific links at the same time. Therefore, we add an attribute of a link, called the utilization, which represents the number of primary paths which contain it. The value will be increased when a primary path containing this link is created by L2OVX. When one primary path is released, we decrease the value of all links in this path. In this way, L2OVX is able to simply and efficiently load sharing the traffic among tenants.

Algorithm 2 Algorithm of Devirtualizing a Packet to the OpenFlow Switch

```
1: procedure DEVIRTUALIZE(vsw, vmatch, vactions)
2:     $tid = getTenant(vsw)$
3:     $paths = getAllPath(vmatch.inport, vactions.ports)$
4:     $stp = getShortestPath(paths)$
5:     for $(s, t) \in stp$ do
6:         $psw_s = fetchPhysicalSwitch(s)$
7:         if $(s, t)$ is the first link of $stp$ then
8:             $match = [in\_port : s]$
9:             $actions = [set\_vlan : tid, output : t]$
10:        end if
11:        if $(s, t)$ is a intermediate link of $stp$ then
12:            $match = [in\_port : s, vlan\_id : tid]$
13:            $actions = [output : t]$
14:        end if
15:        if $(s, t)$ is the last link of $stp$ then
16:            $match = [in\_port : s]$
17:            $actions = [strip\_vlan, output : t]$
18:        end if
19:        $installFlow(psw_s, match, actions)$
20:    end for
21:    $updateLink(stp, assign)$
22: end procedure
```

C. Management Layer

The management layer is composed of two modules: *Instances Pool* and *Scheduler*. Instances Pool maintains a number of available instances, which are used to serve the VPLS service requests from tenants. We create a image in which an Linux OS and a OpenFlow controller are installed. What kinds of controllers (e.g., NOX [12], Ryu [13], and etc.) running on this image are independent, but each has to match two requirements, supporting OpenFlow 1.0 and enabling switch application. Based on this image, the instance is created and automatically assigned a private and fixed IP address. This IP address is permanently associated with the instance until the instance is terminated.

Scheduler is a standalone and subscribing service in our architecture. It periodically executes all subscribed tasks according to their configuration. Currently, we have registered two tasks in Scheduler. The first task that is to create instances when finding the number of available instances is not enough in the pool. We define two parameters, one is an available

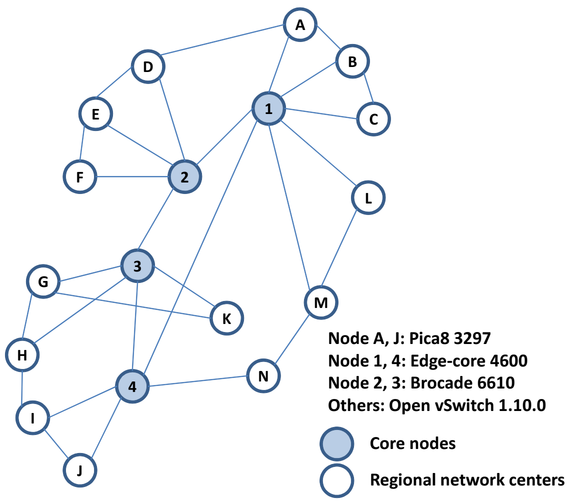


Fig. 3: The experiment topology.

threshold T and the other is a number of creating instances N. If the number of available instances is lower than T, Scheduler will automatically create instances according to N for use in the future. The second task in Scheduler is responsible to check if L2OVX contains any expiring VPLS services. This information will be provided to service manager, for notifying related tenants or doing optional processes including stopping the VPLS service automatically.

IV. Evaluation

For evaluating L2OVX, we emulate physical network connection of TWAREN as our experiment topology. As illustrated in Figure 3, there are four core nodes and twelve regional network centers. In our network laboratory, we have six OpenFlow-enable switches including two Pica8 3297 switches, two Edge-core AS4600 switches, and two Brocade 6610 switches. We choose four of them to be the core nodes, and the other two switches combines five servers installing Open vSwitch to represent as the regional network centers. In this way, this experiment environment is constructed by both virtual and physical network.

We present following experiments in the subsections below to evaluate our solution: the comparison between OpenVirteX and L2OVX in terms of the throughput for a TCP session on software-based and hardware-based OpenFlow switches in Section IV-A; the throughput with link load balancing function enabled in Section IV-B; finally, the number of flow entries needed as the number of services or devices increases, in Section IV-C.

A. The Throughput for a Single TCP Session

In this subsection, we use the Iperf tool to show the comparison of TCP throughput between OpenVirteX and L2OVX. As described above, OpenVirteX achieves the isolation of tenants by using layer 3 translation, which is only implemented in software-based OpenFlow switch, such as Open vSwitch. Therefore, in the first experiment we use Mininet, developed based on Open vSwitch, to simulate the TWAREN network topology and compare these two systems. The TCP throughput is measured from host H1 on node A to host H2 on node J (Figure 3). The experiment result is shown as Figure 4a. The average throughputs between hosts are 617 Mbps over 60s in OpenVirteX and 621 Mbps in L2OVX. This result shows that there is no difference between OpenVirteX and L2OVX in virtual network environment.

In the second experiment, we measure the TCP throughput in physical network environment. Most physical OpenFlow-enable switch currently support layer 2 modification but not layer 3 in OpenFlow action, so we only show the performance result of L2OVX in Figure 4a. With regard to the second experiments, the physical switch outperforms the software-based switch in the TCP throughput. Therefore, for both virtual and physical OpenFlow devices, L2OVX is able to support and has a outstanding performance.

B. Services Load Sharing among Multiple Paths

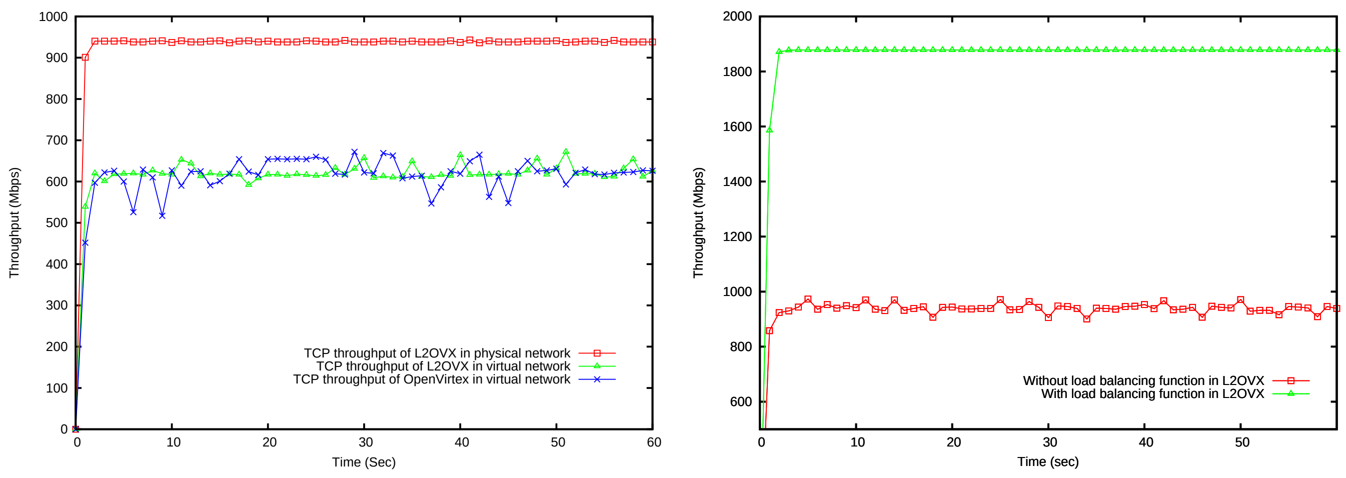
As described in Section III-B3, efficiently distributing incoming network traffic across links can improve data transfer performance among tenants. In this experiment, we show the comparison of TCP throughput between L2OVX with and without load balancing feature. There are two tenants in the scenario, and each of them requests a VPLS service between node 1 and node 4. If the load balancing feature in L2OVX is disabled, both tenants traverse along the same primary path (1,4) because it has the minimum node hops. However, if L2OVX with load balancing feature enabled, path (1,4) is still the primary path of tenant1 but tenant2's primary path is replaced by (1,2)-(2,3)-(3,4). As shown in Figure 4b, L2OVX with load balancing feature delivers almost double throughput in TCP session than disabled the feature in L2OVX. This is because tenant1 and tenant2 transmit data with line-rate performances at the same time along disjoint paths.

C. Reduce the Number of Flow Entries

Next, we consider the scalability impacts on OpenVirteX and L2OVX when the number of services increases. In general, a TWAREN user requests a VPLS service that connects two geographically dispersed sites. In this experiment, we assume each user owns two dedicated edge-ports for attaching end-hosts. There are two different equations which compute the number of flow entries in OpenVirteX and L2OVX separately.

$$N = t \times (\frac{m}{2})^2 \tag{1}$$

For OpenVirteX, the number of flow entries, denoted as N, which is dependent on the two parameters: t and m. The value of t is the number of requested services in the system, which is equal to the number of tenants. And the value of m, is the number of devices used in this service. Without loss of generality, we make an equal division of m, that is $\frac{m}{2}$, and let it be the number of devices in each site. Hence, the maximum number of flow entries N is computed as shown in Equation (1). On one hand, the maximum number



(a) (b)

Fig. 4: The Comparison between OpenVirteX and L2OVX in the TCP Throughput.

of flow entries N in OpenVirteX grows polynomially when the number of devices m increases. On the other hand, we note that the number of tenants t is another effect factor for the result, which indicates the rate of flow entry grows fast with the number of serviced tenants increases.

$$N' = t \tag{2}$$

Equation (2) is defined to compute the number of flow entries in L2OVX. It is similar to Equation (1) but the number of devices m can be ignored because L2OVX is a port-based virtualization system while OpenVirteX is based on end-hosts. Therefore, the number of flow entries in L2OVX remains constant regardless of an increase with the number of connected end-hosts.

V. Conclusion

In this paper, we presented a virtualization system L2OVX which provides a VPLS-like service with lower cost with SDN framework and supports on-demand configuration to improve the efficiency. The proposed solution is developed based on OpenVirteX but the main difference is L2OVX uses layer 2 instead of layer 3 modification in the virtualization process. The benefit of using L2OVX derives from the fact that is highly compatible with most OpenFlow-enabled hardware switches to achieve the wire-rate performance when transmitting the data. Furthermore, the number of installed flow entries of L2OVX is less than OpenVirteX. In addition, the proposed solution fixes the problem in OpenVirteX which does not properly display the OpenFlow devices topology in each tenant's virtual network, and provides a management layer in which routine tasks including checking the expired services and maintaining the service instances are invoked by *Scheduler*. Moreover, L2OVX enables the load balance function for each VPLS service to improve the transfer bandwidth among tenants.

For the future work, we will integrate L2OVX with OpenFlow 1.3 to provide more features available in the OpenFlow protocol and extend the number of VPLS services in TWAREN. Besides, we would like to propose more real-time and fine-grained load balancing mechanism to improve the link utilization and transmission bandwidth.

References

[1] M. Lasserre and V. Kompella, "Virtual Private LAN Service (VPLS) Using Label Distribution Protocol (LDP) Signaling," Internet Engineering Task Force, RFC 4762, Jan. 2007. [Online]. Available: https://tools.ietf.org/rfc/rfc4762.txt

[2] D. Kreutz *et al.*, "Software-Defined Networking: A Comprehensive Survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14-76, 2015.

[3] N. McKeown *et al.*, "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 6974, 2008.

[4] A. Al-Shabibi *et al.*, " OpenVirteX: make your virtual SDNs programmable," in *Proc. 3rd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2014.

[5] N. Chowdhury and R. Boutaba, "A survey of network virtualization," *Computer Networks*, vol. 54, no. 5, pp. 862876, 2010.

[6] G. Schaffrath, C. Werle, and P. Papadimitriou, "Network virtualization architecture: proposal and initial prototype," in *Proc. 1st ACM SIGCOMM Workshop on Virtualized Infastructure Systems and Architectures (VISA)*, 2009, pp. 63-72.

[7] R. Sherwood *et al.*, "FlowVisor: A Network Virtualization Layer," *Technology Report*, 2009.

[8] P. Costa, M. Migliavacca, P. Pietzuch, and A. L. Wolf, "NaaS: network-as-a-service in the cloud," in *Proc. 2nd USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE)*, 2012.

[9] J. Matias, A. Mendiola, N. Toledo, B. Tornero, and E. Jacob, "The EHU-OEF: An OpenFlow-based Layer-2 experimental facility," *Computer Networks*, vol. 63, pp. 101-127, 2014.

[10] Open vSwitch, 2014. [Online]. Available: http://openvswitch.org/

[11] Nippon Telegraph and Telephone Corporation, "Lagopus switch: a high-performance software OpenFlow 1.3 switch," 2015. [Online]. Available: https://lagopus.github.io/

[12] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker, "Applying NOX to the datacenter," in *Proc. 8th ACM Workshop on Hot Topics in Networks (HotNets-VIII)*, 2009.

[13] Nippon Telegraph and Telephone Corporation, "Ryu SDN Framework," 2015. [Online]. Available: http://osrg.github.io/ryu/