Improvement of Virtual Network Mapping Based on OpenVirteX

Yuan ZHANG¹, Jian QIN²*, Xiaoping LI¹ and Shiling ZHANG¹

¹ State Grid Chongqing Electric Power Co., Electric Power Research Institute, Chongqing, 401123, China
² School of Microelectronic and Communication Engineering, Chongqing University, Chongqing, 400030, China
*Corresponding author’s e-mail: qinjian@cqu.edu.cn

Abstract. Network virtualization allows the flexible sharing of physical networking resources by multiple users (tenants). Each tenant runs its own applications over its virtual network. OpenVirteX, a network virtualization platform, provides virtual Software Defined Networks (vSDNs). Each vSDN is customizable in terms of topology as well as addressing scheme. However, the virtual network mapping process of OpenVirteX does not take into account resource constraints and deployment cost, just to complete the virtual network mapping. To solve this problem, this paper proposes to deploy the classic mapping algorithm named Improved-vnmFlib on the OpenVirteX platform and designs the overall process of implementation. Finally, we build a complete simulation environment to test the new platform, the test results show that the new platform can take a virtual network mapping combined with the deployment cost under the premise of resource constraints, and can create the virtual network automatically according to the mapping result.

1. Introduction

Network Virtualization has been hailed as a key enabler technology to escape from the current limitations of the Internet. It was conceived to “slice” a given physical network infrastructure into multiple virtual networks [1]. There are several related techniques that can create network “slices” in the networking domain. For instance, wavelength division multiplexing (WDM) [2] creates slices at the physical layer, while virtual local area networks (VLANs) [3] create slices at the link layer. Multiple protocol label switching (MPLS) [4] creates slices of forwarding tables in switches. In contrast, network virtualization seeks to create slices of the entire network.

Software defined networking (SDN) has emerged as a promising paradigm for making the control of communication networks flexible. SDN separates the data packet forwarding plane from the control plane and employs a central controller. The virtualization of SDN networks promises to allow networks to leverage the combined benefits of SDN networking and network virtualization. Therefore, it has attracted significant research attention in recent years. A critical component for virtualizing SDN networks is a network virtualization platform that abstracts the underlying physical SDN network into multiple logically isolated virtual SDN networks (vSDNs), each with its own controller.

In recent years, the academic community has proposed several typical network virtualization platforms, including FlowVisor[5], ADVisor[6], VeRTIGO[7], OpenVirteX [8-10] and CoVisor. OpenVirteX is a network virtualization platform developed by Stanford University. It can overcome
FlowVisor’s shortcomings, including that virtual topology and physical topology are not independent, the addressing space of virtual network can’t overlap and so on. In addition, OpenVirteX can combine with the way to expand network topology virtualization of the Advisor and VeRTIGO to specify their virtual network topology completely independent of the physical network topology. However, the entire mapping process of OpenVirteX does not take into account the resource constraints and the deployment cost. Moreover, the process of creating a virtual network is manually operated, which efficiency is low and have higher labor costs.

In this paper, we deploy the Improved-vnmFlib algorithm that can solve the problem of resource allocation in accordance with the deployment cost on the OpenVirteX. When the virtual network request arrives, OpenVirteX can call the Improved-vnmFlib algorithm to complete the mapping, and create a virtual network automatically according to the mapping results.

The rest of the paper is organized as follows. In Section II, we discuss the system architecture of the OpenVirteX platform and mapping algorithm. In Section III, we design the overall process for deploying Improved-vnmFlib algorithm on OpenVirteX platform. We discuss the specific implementation of each module in Section IV. We build a simulation environment to test the new platform in Section V, and we conclude the paper in Section VI.

2. Related work

2.1. OpenVirteX Architecture
Like FlowVisor, OpenVirteX functions as a proxy within the control channel, presenting OpenFlow networks to tenants, while controlling the underlying physical infrastructure via the southbound OpenFlow interface. Figure 1 depicts this architecture.

OpenVirteX is a network virtualization platform capable of spawning virtual networks with OpenFlow protocol. These virtual networks may have arbitrary topology and addressing schemes, configured as per tenant request. Requests are conveyed via API calls to OpenVirteX, with a tool such as a network embedder. Figure 1 shows how a network embedder and OpenVirteX would work in concert to realize a requested virtual network. First, a user species a virtual network's addressing scheme (e.g. vSDN request) to the embedder, which generates a virtual-to-physical mapping using information from OpenVirteX. Next, this mapping is passed to OpenVirteX, which in turn instantiates the virtual network on the physical topology.

2.2. Mapping Algorithm
The virtual network created by OpenVirteX is mainly composed of virtual nodes and virtual links. The mapping schematic diagram is shown in Figure 2, which is divided into two phases, node mapping and link mapping.

(1) Node mapping: The method used by node mapping is port mapping. Port mapping refers to the virtual network interface mapping according to the physical switch ID number, port number. The virtual node must be embedded to the physical node to complete the forwarding of the internal data within the virtual network.
(2) Link mapping: Once the virtual node mapping is over, the path of the virtual link mapped to the physical network refers to the path between the embedded physical nodes, which is calculated by the SPF algorithm based on the underlying physical network topology detected by the OpenVirteX. The result is sent to the underlying switch in the form of a flow table, thus enabling the switch to complete the data forwarding.

A virtual network is created, configured, and initialized via API calls. Procedurally, OVXNetwork creation involves the following:

1. Declare the address block used by the virtual network, and connect the virtual network to the tenant controller;
2. CreateOVX Switches from available Physical Switches;
3. Add OVX Ports to the OVX Switches;
4. Add OVXLinks, Hosts;
5. If you select SPF, it is the shortest path between the embedded physical nodes. If manual, specify paths for OVXLinks;
6. Optionally, add backup paths for OVXLinks;
7. Initialize the OVXNetwork.

From the above OpenVirteX mapping algorithm and steps can be seen, the inherent algorithm of OpenVirteX platform is the shortest path algorithm SPF, the algorithm is based on the number of hops to select the path. In the real network environment, just seeking the shortest path may not be the best choice. For example, some links may have been very congested or the deployment cost is high, and SPF does not consider these issues, so that the paths mapped by multiple virtual links may include the congested or costly link, resulting in a significant increase in the data flow delay and packet loss rate or a high deployment cost. In addition, the process of creating virtual network is manually, when we need to deploy large-scale topology to verify the new technology, the process will bring a lot of inconvenience to the test.

To solve this problem, this paper proposes to deploy the classic mapping algorithm named Improved-vnmFlib on the OpenVirteX platform. The algorithm is described in detail in the next section.

2.3. Improved-vnmFlib algorithm

2.3.1. Network model and problem description

**Physical network:** We denote the physical network by an undirected graph \( G_s = (V_s, E_s, C_s(n_s), B_s(l_s)) \), where \( V_s \) and \( E_s \) refer to the set of nodes and links, substrate nodes and links are associated with their attributes, denoted by \( C_s(n_s) \) and \( B_s(l_s) \), respectively. In this paper, we consider CPU capacity for node attributes, and bandwidth capacity for link attributes.

**Virtual network request:** We denote by an undirected graph \( G_v = (V_v, E_v, C_v(n_v), B_v(l_v)) \), where \( V_v \) and \( E_v \) refer to the set of nodes and links, a virtual network request typically has link and node constraints that are specified in terms of attributes of the physical network. We denote by \( C_v(n_v) \) and \( B_v(l_v) \) the set of link and node constraints, respectively.

**Virtual network mapping:** A virtual network mapping for a virtual network request is defined as a mapping from \( G_v \) to a subset of \( G_s \), such that the constraints in \( G_v \) are satisfied. The virtual network mapping can be naturally decomposed into node and link mapping as follows:

**Node Mapping:** Each virtual node from the same virtual network request is assigned to a different substrate node. We use \( M_s(\ ) \) represent the node mapping function. And \( n_v \in V_v, n_s \in V_s \):

\[ n_v \rightarrow n_s : M_s(n_s) = n_s \]

Subject to
\[ C_S \left(M_N (n_v) \right) \geq C_V (n_v), \forall n_v \in V_v \]

**Link Mapping:** Each virtual link is assigned to a substrate path (indivisible flow) or a set of substrate paths \( p \) (divisible flow) between the corresponding substrate nodes that host the end virtual nodes of the virtual link. We use \( M_p (\cdot) \) represent the link mapping function. And \( p \in E_S, I_v \in E_s \):

\[ l_v \rightarrow p : M_p (l_v) = p \]

Subject to

\[ B_s (M_p (l_v)) \geq B_s (l_v), \forall l_v \in E_s \]

**Objectives:** Our main interest in this paper is to decrease mapping cost of the Infrastructure Providers. Which can be defined as follows:

\[ F = u_S (n_v) \times C_S (n_v) + \sum_{l_v \in n_v} B_s (l_v) = u_S (n_v) \times C_S (n_v) + \sum_{l_v \in n_v} B_s (l_v) \]

Where \( u_S (n_v) \) denotes unit CPU cost for \( n_v \in V_S \), \( u_S (l_v) \) denotes unit bandwidth cost for \( l_v \in E_s \).

### 2.3.2 Algorithm Steps

The main goal of the Improved-vnmFlib algorithm is how to embed the virtual network \( G_v \) to the physical network \( G_s \) with the minimization of the mapping cost under the constraints of the virtual node and link. Proceed as follows:

**Input:** Physical network, virtual network request

**Function:** Virtual network requests are embedded to physical network with the goal of minimizing mapping cost.

**Output:** Mapping results

Step 1: Initialize \( M = \emptyset, i = 0 \); \( M \) represents the embedded virtual node collection;

Step 2: Select the virtual node \( n'_i (n'_i \in M) \) and the associated link that are connected to the virtual node in \( M \). If \( M = \emptyset \), select a virtual node randomly;

Step 3: Find all feasible mapping physical nodes for \( n'_i \). Then sort them according to \( F \) incrementally, and record them in \( D_i \);

Step 4: If \( D_i = \emptyset \) and \( i = 0 \), the mapping fails;

Step 5: If \( D_i = \emptyset \) and \( i \neq 0 \), Delete the mapping result of the virtual node \( n'_i \), and remove it from \( D_{i-1} \). \( i-1 \), go to step 4;

Step 6: In \( D_i \), map \( n'_i \) to the first feasible physical node, map the related links, and update the resource information;

Step 7: Place \( n'_i \) in \( M \);

Step 8: If all virtual node mapping are complete, the algorithm ends. Otherwise, \( i +1 \), goto step 2.

### 3. Design scheme of developed OPENVIRTEX

#### 3.1 System overall design

In order to achieve the above functions, we modify or add the main modules in the OpenVirteX platform are as follows:

1. User Interaction module (UI): This module provides user interaction interface for user to operate the virtual network friendly.
2. Add Resource Constraints module (ARC): This module is used to provide a functional interface for adding resource limits and costs to the physical network.
(3) Virtual Network Mapping module (VNM): This module is the most important module to achieve the above functions, call the Improved-vnmFlib algorithm to deal with the mapping relationship between the virtual network and physical network.

(4) Create Network Automatically module (CNA): This module provides a functional interface for creating virtual networks automatically based on the mapping results.

3.2. Interaction between the key modules

The interaction relationship between the key modules is shown in Figure 3.

![Figure 3. Key modules on the platform](image)

First, a user enters the resource constraints file through the UI module to add resource constraints for the physical topology, the UI module recognizes the command of add resource constraints, reads the file, and then calls the ARR module to add physical constraints for the physical topology. Second, the user enters the virtual network request file through the UI module, the UI module recognizes the command of adding virtual network request, reads the file, and then calls the CNA module, which calls the VNE module to complete the virtual network mapping, and returns the resource mapping result to the CNA module. The CNA module instantiates the virtual network based on the resource mapping result by calling the local method of OpenVirteX, then creating multiple virtual networks. Finally, the resource allocation result is displayed through the control terminal and the topology of the virtual network is displayed through the controller web page.

4. Implementation of the key module

4.1. Design and implementation of adding resource constraints module

The main function of this module is to provide the functional interface to add resource constraints for the physical network. The physical network resources mainly include the node CPU capacity and the maximum bandwidth of the link and the unit CPU and unit bandwidth cost. The virtual network request resources mainly include the node CPU demand and the link bandwidth requirement. The design flow is shown in figure 4.

A user enters physical resource configuration file via the addNetworkResourceCfg command. Ovxctl.py parses the addNetworkResourceCfg command first, then reads the physical resource configuration file, and calls the functional interface addNetworkResourceCfg to achieve the function of adding resource constraints for the physical network, and finally outputs the result to the console terminal to the user.

4.2. Design and Implementation of Virtual Network Mapping Algorithm Module

The virtual network mapping algorithm module is the core module that provides mapping results for virtual networks. That is, when the user enters the virtual network request, the virtual network can be embedded to the appropriate physical switch and physical link according to the improved-vnmFlib algorithm. Output includes the node mapping result, the link mapping result, the mapping cost, and the mapping time.

The overall design flow of the virtual network mapping algorithm module is as fellow:
Step 1: Read the physical resource constraints file, encapsulate the information as phyNetCfg; and then use the phyNetCfg as the parameter to establish simulation processing instance OVXApplication;

Step 2: Read the virtual network request file, encapsulate the information as virNetReqCfg, and then use the virNetReqCfg as parameter to establish resource request instance resourceRequest;

Step 3: Record the current system time startTime, as the start time of the mapping algorithm;

Step 4: Invoke the processResourceRequest method of OVXApplication to process resourceRequest, and save the resource mapping result in the resourceAllocation;

Step 5: Record the current system time endTime, as the end time of the mapping algorithm;

Step 6: Update the physical resource, print the mapping result to the console terminal. If there is no virtual network request, the algorithm ends. Otherwise, continue to enter the request file, go to Step 2.

4.3. Design and Implementation of Creating Virtual Network Automatically Module

This module provides the functional interface to create virtual network automatically according to the virtual network request and the mapping results. The design flow is shown in figure 5.

![Sequence diagram of creating virtual network automatically module](image)

Figure 5. Sequence diagram of creating virtual network automatically module

A user enters request network request file via the createNetworkAuto command. The ovxctl.py parses the createNetworkAuto command first, then reads the request network request file, and calls the functional interface createNetworkAuto to achieve the function of creating virtual network automatically, and finally outputs the mapping result to the console terminal to the user.

5. Performance evaluation

5.1. Simulation Environment

The overall architecture of the test platform is shown in Figure 6, mainly divided into three layers: the physical layer, virtualization layer and network operating system layer. The physical layer uses Mininet to customize the test topology; The virtualization layer uses the developed OpenVirteX platform to create and manage virtual networks; The network operating system layer uses an open source Floodlight controller to control the create virtual network.

The environment configuration we used in the development is as fellow: Ubuntu 16.04LTS as OS, JAVA as development language, Floodlight0.9 as controller, 0.0-MAINT as OpenVirteX platform and Mininet2.2.1 as topology generation tool.

5.2. Test topology

The physical topology generated by Mininet is loosely based on the Internet2 NDDI topology. It has 11 core switches in major cities as figure 7. The DPIDs are listed in Table1.
Figure 6. Architecture of the test platform

Figure 7. Physical network topology

Table 1. Physical DPID of each node

| Node | DPID              |
|------|-------------------|
| SEA  | 00:00:00:00:00:00:01:00 |
| SFO  | 00:00:00:00:00:00:02:00 |
| LAX  | 00:00:00:00:00:00:03:00 |
| ATL  | 00:00:00:00:00:00:04:00 |
| IAD  | 00:00:00:00:00:00:05:00 |
| EWR  | 00:00:00:00:00:00:06:00 |
| SLC  | 00:00:00:00:00:00:07:00 |
| MCI  | 00:00:00:00:00:00:08:00 |
| ORD  | 00:00:00:00:00:00:09:00 |
| CLE  | 00:00:00:00:00:00:0A:00 |
| IAH  | 00:00:00:00:00:00:0B:00 |

Table 2. Information of VN1

| VN1 nodes | Computation requirement | VN1 links | Bandwidth requirements |
|-----------|-------------------------|-----------|-----------------------|
| V1        | 10                      | V1-V2     | 70                    |
| V2        | 10                      | V2-V3     | 70                    |
| V3        | 10                      | V1-V3     | 70                    |
Two virtual network requests VN1 and VN2 are constructed on the physical network topology shown in Figure 8. The number and attributes of the node and link can be configured according to the requirements. In this paper, two virtual network requests shown in Figure 8 were used, and the request attributes mainly included the virtual node CPU and the link bandwidth. The details of the virtual network request are shown in Table 2 and Table 3, respectively.

| VN2 nodes | Computation requirement | VN2 links | Bandwidth requirements |
|-----------|-------------------------|-----------|------------------------|
| V1        | 10                      | V1-V2     | 60                     |
| V2        | 10                      | V2-V3     | 60                     |
| V3        | 10                      | V3-V4     | 60                     |
| V4        | 10                      | V1-V4     | 60                     |

5.3. Test configuration

We use two virtual network requests VN1 and VN2 mapping to the physical topology as an example, to verify the platform’s newly features. The test procedure is as follows:

Step 1: Start Mininet to create a physical topology;
Step 2: Start the developed OpenVirteX;
Step 3: Start Floodlight;
Step 4: Entering resource constraints and costs files;
Step 5: Entering virtual network request.

The number on the left of the slash in the rectangle indicates the computing resource that the node can provide. The number on the right of the slash indicates the unit CPU cost. The number on the left side of the slash indicates the maximum available bandwidth provided by the link. The number on the right side of the slash indicates the link unit bandwidth cost.

5.4. Results

After entering the command of creating VN1 automatically, the mapping result will output to the control terminal to the user, including node mapping results, link mapping results, mapping costs, mapping time,
and the remaining resources of physical network. We can login to the Floodlight web interface to view the topology of VN1. At the same way, we can view the topology of VN2 as well.

The test results show that the new OpenVirteX platform addition the following two functions compared to the original OpenVirteX platform:

(1) When virtual network requests arrives, the new OpenVirteX platform can embed the virtual network request to the underlying physical network with the mapping cost under the premise of resource constraints.

(2) The new OpenVirteX platform can create a virtual network automatically based on the mapping results.

6. Conclusion
In this paper, we propose a scheme to deploy the improved-vnmFlib algorithm on the OpenVirteX platform for the existing problems of the virtual network mapping on the OpenVirteX platform. We designed the overall implement process of deploying the improved-vnmFlib algorithm on the OpenVirteX platform, analysed the interaction relationship between the key modules in detail and implemented each module by programming. Finally, we built a simulation environment to test the new platform, the test results show that the new platform can take virtual network mapping combined with deployment cost under the premise of resource constraints, and can create the virtual network automatically according to the mapping result.

References
[1] Ahmed M.F., Talhi C., Cheriet M. (2015) Towards flexible, scalable and autonomic virtual tenant slices. C. IEEE International Symposium on Integrated Network Management, Ottawa, ON, Canada, 720-726.
[2] Ishio H., Minowa J., Nosu K. (1984) Review and status of wavelength-division. J. Lightwave Technology Journal of, 2:448-463.
[3] Yu M., Rexford J., Sun X., et al. (2011) A survey of virtual LAN usage in campus networks. J. IEEE Communications Magazine, 49:98-103.
[4] Xiao X., Hanan A., Bailey B., et al. (2002) Traffic engineering with MPLS in the Internet. J. IEEE Network, 14:28-33.
[5] Sherwood R., Gibb G., Yap K., et al. (2009) FlowVisor: A network virtualization layer. J. OpenFlow Consortium, Palo Alto, CA, USA, OPENFLOW-TR-2009-1, 18:32-37.
[6] Salvadori E., Corin R.D., Broglio A., et al. (2011) Generalizing virtual network topologies in OpenFlow-Based networks. C. IEEE Conf. on Global Telecom., Kathmandu, Nepal, 1-6.
[7] Doriguzzi Corin R., Gerola M., Riggio R., et al. (2012) VeRTIGO: Network virtualization and beyond. C. European Workshop on Software Defined Networking, Darmstadt, Germany, 24-29.
[8] Al-Shabibi A., Leenheer M., Gerola M., et al. (2014) OpenVirteX: a network hypervisor. C. ACM Workshop on Hot Topics in SDN, Chicago, Illinois, USA, 15-20.
[9] Al-Shabibi A., Leenheer M., Gerola M., et al. (2014) OpenVirteX: make your virtual SDNs programmable. C. ACM Workshop on Hot Topics in SDN, Chicago, Illinois, USA, 25-30.
[10] Berde P., Gerola M., Hart J., et al. (2014) ONOS: towards an open, distributed SDN OS. C. ACM Workshop on Hot Topics in SDN, Chicago, Illinois, USA, 1-6.