International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:10, No:9, 2016

Implementation and Demonstration of Software-Defined Traffic Grooming

Xu Zhang, Lei Guo, Weigang Hou

***Abstract*—**Since the traditional network is closed and it has no architecture to create applications, it has been unable to evolve with changing demands under the rapid innovation in services. Additionally, due to the lack of the whole network profile, the quality of service cannot be well guaranteed in the traditional network. The Software Defined Network (SDN) utilizes global resources to support on-demand applications/services via open, standardized and programmable interfaces. In this paper, we implement the traffic grooming application under a real SDN environment, and the corresponding analysis is made. In our SDN: 1) we use OpenFlow protocol to control the entire network by using software applications running on the network operating system; 2) several virtual switches are combined into the data forwarding plane through Open vSwitch; 3) An OpenFlow controller, NOX, is involved as a logically centralized control plane that dynamically configures the data forwarding plane; 4) The traffic grooming based on SDN is demonstrated through dynamically modifying the idle time of flow entries. The experimental results demonstrate that the SDN-based traffic grooming effectively reduces the end-to-end delay, and the improvement ratio arrives to 99%.

***Keywords*—**NOX, OpenFlow, software defined network, traffic grooming.

I. Introduction

WITH the rapid development of Internet technology, there will be the growing demand for new services such as cloud computing and mobile device-to-device communications, which makes the traditional network expose shortcomings. The traditional network and service are separated of each other. When the new service requires the network to make prompt adjustments, it will be inefficient or even impossible. We have to adjust network properties for several years or introduce high-cost equipment to satisfy the requirements of new services, which motivates us to deal with this problem from the perspective of network architecture.

SDN [1], [2] decouples the control function from the data forwarding plane using OpenFlow [3], with the objective of achieving a flexible control of data forwarding equipment and network traffic. SDN also provides a good platform for innovative services. The main features of SDN include: separation of data forwarding and control planes, logically centralized control, and open programming interfaces. After separation, the data forwarding plane is only responsible for packet transferring, while the network control will be done by the control plane. In this way, the data forwarding plane becomes generic and simple, and the operational cost decreases. Meanwhile, the control plane will become centralized. Due to the powerful performance, the centralized controller is able to maintain the global network topology, execute global routing and optimization, and achieve the unified management of the entire network. All of control commands should be sent to the data forwarding plane in the form of flow tables. The format of the flow table is standardized by the open programming southbound interface (e.g., OpenFlow).

In recent years, SDN has enabled some researches to grow rapidly, especially for control scalability, network virtualization, packet circuit integration [4], [5], network source address validation, IPv6, network security [6], wireless embedded OpenFlow technology [7], and optical networks [8]-[12]. However, in existing SDN network, the value of idle timeout is usually very small, i.e., the flow entry quickly expires after idle_timeout seconds if no received traffic there is. So, when the packets arrive at the switch, the flow entry matched with them has been removed, and the flow tables have to be re-established for forwarding these packets, which results in a long end-to-end delay. To solve this problem, we build a real SDN environment by using NOX controller [13], Open vSwitch [14] and OpenFlow. Open vSwitch constructs a virtual network topology. NOX controller manages all virtual switches within the topology, and implements the routing of network traffic using internal applications designed by us.

The key contributions of this paper are as follows. Based on our SDN architecture, we implemented the routing function and made an extensive experimental analysis. The routing application has the function of traffic grooming that dynamically modifies the idle time of flow entries, and the improvement ratio of reducing the end-to-end delay arrives to 99%.

The rest of this paper is organized as follows. In Section II, we introduce technologies of enabling SDN, mainly including OpenFlow protocol, NOX controller, and Open vSwitch. In Section III, our SDN architecture, OpenFlow communication process, and SDN-based traffic grooming are designed. In Section IV, we present experimental results before concluding this paper in Section V.

II. Technologies of Enabling SDN

A. OpenFlow Protocol

OpenFlow protocol defines a communication standard between the centralized controller (e.g., NOX) and the switches in the data forwarding plane. OpenFlow makes traditional L2

X. Zhang and L. Guo are with the School of Computer Science and Engineering, Northeastern University, Shenyang 110819, P. R. China.

W. Hou is with the School of Computer Science and Engineering, Northeastern University, Shenyang 110819, P. R. China (Corresponding author e-mail: houweigang@cse.neu.edu.cn).

and L3 switches have powerful forwarding capability, e.g., the fine-grained flow forwarding integrating MAC- and IP-based formats in the multi-domain header of the packet. Table I lists the main types of messages in OpenFlow 1.0, including controller-to-switch, asynchronous, and symmetric. Each message has multiple sub-types. Especially, the controller initiates controller-to-switch message to acquire switch status; the switch initiates asynchronous message to send status report to the controller; symmetric message is initiated by either switch or controller.

In the following, we focus on two sub-messages: OFPT_PACKET_IN and OFPT_FLOW_MOD. When receiving a packet, if the switch cannot find any flow entries to match this packet in the local flow table, it sends OFPT_PACKET_IN message to the controller, and the packet is temporarily stored in the switch buffer. So, both partial packet information and the serial number of the buffer are also sent to the controller. If the buffer size is not large enough to store the packet in the switch, the whole packet is sent to the controller along with the attached content of the OFPT_PACKET_IN message.

TABLE I
MESSAGES IN OPENFLOW 1.0

Controller	Message Type	Switch
OFPT_HELLO	symmetric	OFPT_HELLO
OFPT_ECHO_REQUEST	symmetric	OFPT_ECHO_REPLY
OFPT_VENDOR	symmetric	OFPT_VENDOR
OFPT_ERROR	symmetric	OFPT_ERROR
OFPT_FEATURES_REQUEST	controller-to-switch	OFPT_FEATURES_REPLY
OFPT_GET_CONFIG_REQUEST	controller-to-switch	OFPT_GET_CONFIG_REPLY
OFPT_SET_CONFIG	controller-to-switch	
	asynchronous	**OFPT_PACKET_IN**
	asynchronous	OFPT_FLOW_REMOVED
	asynchronous	OFPT_PORT_STATUS
OFPT_PACKET_OUT	controller-to-switch	
OFPT_FLOW_MOD	controller-to-switch	
OFPT_PORT_MOD	controller-to-switch	
OFPT_STATS_REQUEST	controller-to-switch	OFPT_STATS_REPLY
OFPT_BARRIER_REQUEST	controller-to-switch	OFPT_BARRIER_REPLY
OFPT_QUEUE_GET_CONFIG_REQUEST	controller-to-switch	OFPT_QUEUE_GET_CONFIG_REPLY

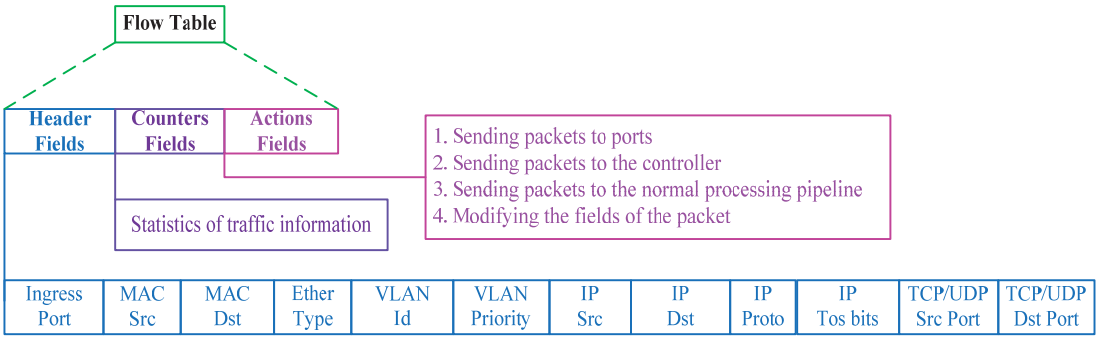

Fig. 1 Structure of flow table

As mentioned above, for each switch, the corresponding forwarding rule is determined by matching flow entries of the local flow table. Consequently, the OFPT_FLOW_MOD message includes operations of adding, modifying, and deleting flow entries, in order to update forwarding rules. Obviously, whether controller or switch, the actually processed information is the traffic flow rather than a single packet. We can see that the flow table is the most important data structure of setting forwarding rules. As shown in Fig. 1, each flow entry includes three parts, Header Fields, Counters Fields, and Actions Fields. Among which, Header Fields include 12 domains: Ingress Port, Ethernet source address, Ethernet destination address, Ethernet type, VLAN Id, VLAN priority, IP source address, IP destination address, IP protocol type, IP ToS bits, TCP/UDP source port, and TCP/UDP destination port; Counters Fields can maintain any combination of flow tables, flow entries, ports and queues, and they are mainly used to make a statistical analysis of the traffic information such as the number of active flow entries, and the number (bytes) of packets sent, etc. In this paper, we achieve the flow statistics by using OFPT_STATS_REQUEST and OFPT_STATS_REPLY messages. Each flow entry has a list of actions executed according to the priority queue. Generally, the actions are divided into four parts: sending packets to ports, sending packets to the controller, sending packets to the normal processing pipeline, and modifying the fields of the packet.

B. NOX Controller

In fact, NOX is the first controller with the development of OpenFlow protocol, and Nicira team donated it to the open source community in 2008. NOX mainly has two parts of functions. On the one hand, via the southbound interface (i.e., OpenFlow), NOX manages or parses forwarding behaviors of switches in the data forwarding plane. On the other hand, it is the platform (operating system) of building new applications by virtue of providing feasible Application Programming Interfaces (APIs). API is actually the northbound interface. These applications can generate new traffic flows through network events. For example, we can develop a new application to determine how to forward a new traffic flow in the network because the application can guide NOX to modify the traffic flow according to the collected statistic information.

NOX monitors the specific Transmission Control Protocol (TCP) port so that a switch can establish the connection with the NOX controller via the TCP port. However, the NOX core merely provides the aforementioned simple network connections. The high-level services are performed by NOX components, i.e., applications such as our designed routing modular with traffic grooming. As a result, NOX forwards the received traffic flow to the running component, and it then sends the modified traffic flow to the switch. In fact, each application is a collection of functions and statements developed by C++ or Python. The former provides better performances, while the latter has a more user-friendly API.

C. Open vSwitch

Open vSwitch is the virtual switch achieved by software, and it can well integrate with KVM and Xen virtualization platforms. The working principle is very similar to that of the physical switch. More specifically, the two ends of the Open vSwitch connect respectively with a physical network card and multiple virtual network cards. Note that each virtual network card corresponds to a virtual machine. Meanwhile, the Open vSwitch maintains an internal mapping table so that it can find the corresponding virtual link (port) to forward packet based on MAC address. When a packet passes through the virtual network card configured in a virtual machine, the virtual network card determines how to deal with this packet based on forwarding rules. The released packet will then be forwarded to the Open vSwitch. Different from the traditional virtual switch, the OpenFlow-enabled Open vSwitch will find the flow entries to match this packet in the local flow table: if it cannot find any flow entries to match this packet, this packet will be sent to the controller that will send a new flow table to guide the relative Open vSwitch to forward this packet (case 1); otherwise, this packet will be directly processed in accordance with the corresponding actions in matching flow entries (case 2).

When a packet is forwarded to the physical network card connected with the Open vSwitch, this Open vSwitch will send this packet to the local physical network equipment. Aside from OpenFlow protocol, the Open vSwitch also has data forwarding paths (datapaths), each of which is mainly used to transfer packets according to the actions in matching flow entries. As shown in Fig. 2, the Open vSwitch provides two types of datapaths: the slow channel operating at case 1, and the fast channel with the special Linux kernel module under case 2.

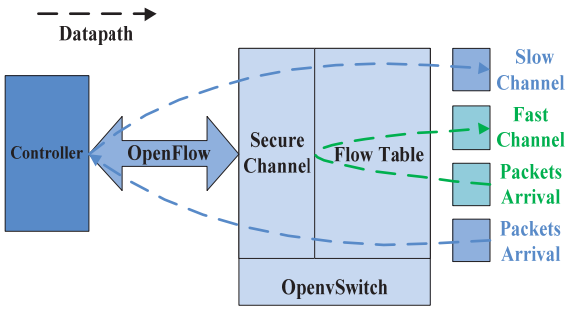

Fig. 2 Open vSwitch architecture

Ovs-vswitchd is the most important component in the Open vSwitch because it realizes the core functions of Open vSwitch. Ovs-vswitchd directly communicates with the kernel modules of the Open vSwitch via the netlink protocol. During the run time of the Open vSwitch, ovs-vswitchd keeps the information of switch configuration on the database called ovsdb. Since ovsdb is directly managed by ovsdb-server, ovs-vswitchd needs to communicate with ovsdb-server through Linux socket mechanism, aiming at obtaining or saving the information of switch configuration. In other words, the information of switch configuration can be persistent to ovsdb, even if the Open vSwitch is restarted.

III. Network Architecture

A. SDN Architecture

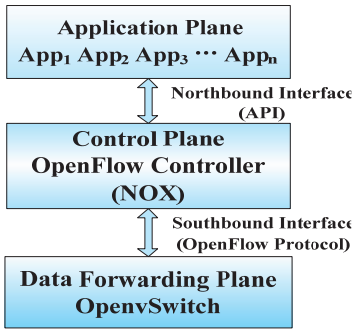

Fig. 3 Diagram of SDN architecture

Fig. 3 demonstrates our three-layer SDN architecture with including data forwarding plane, control plane and application plane. The control and data forwarding planes communicate with each other through the southbound interface (OpenFlow). It indicates that the controller does not have to run on the physical switch, which enables the flexible programmability of the network.

The centralized controller NOX provides the northbound API for each network application. In the application plane, the user can customize the application that triggers the traffic flow definition, and the application can communicate with the controller via the northbound API. NOX controller then sends

the new traffic flow to the data forwarding plane. We customize the routing application in accordance with the service requirements (e.g., traffic grooming), which makes the whole network upgraded quickly without reconfiguring all switches independently.

B. SDN Topology

To demonstrate the effectiveness of our routing function with traffic grooming and the interactive process between controller and switch, we establish a real SDN topology in Fig. 4. Our SDN topology is composed of a NOX controller, seven OpenFlow-enabled switches and two hosts responsible for sending and receiving packets. Host_1 sends packets to Host_2 by using Ping command, and two hosts both can send the connection request to the switch. The NOX controller is used to dynamically monitor the status of the network topology. Both NOX controller and switch can send OFPT_ECHO_REQUEST message, but the receiver of this message is required to send back OFPT_ECHO_REPLY message for checking the connection availability between controller and switch. If the connection is lost, the NOX controller will dynamically update the current status of the network topology before we trigger our routing application. Similarly, the switch will send OFPT_PORT_STATUS message to the NOX controller if we vary the configuration of ports in the switch. As discussed in Subsection II.C, each switch maintains a flow table, aiming at performing the flow-based packet forwarding.

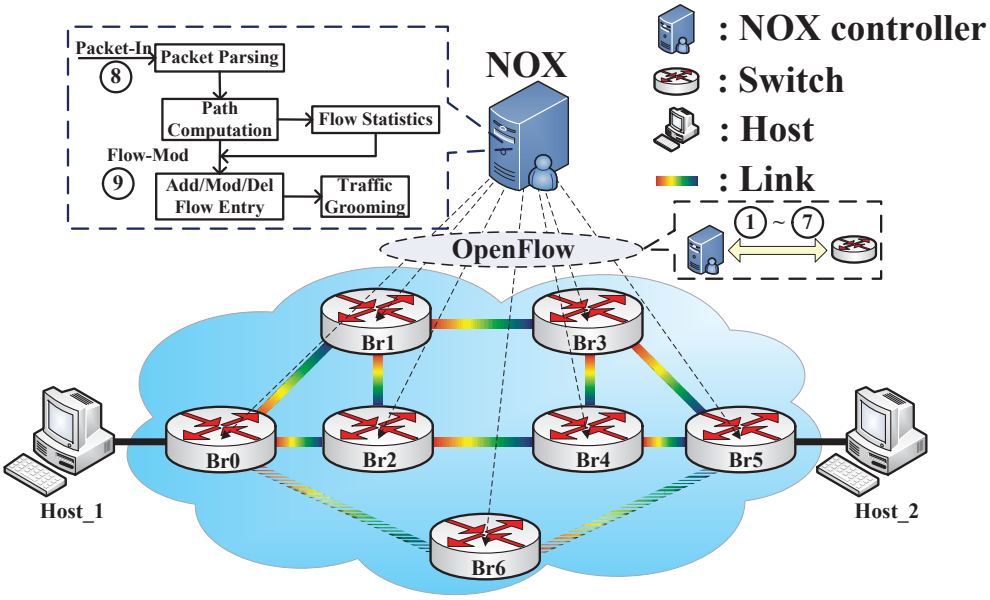

Fig. 4 Diagram of our SDN topology

C. SDN Environment

Fig. 5 shows the internal connection relationship of all elements in our SDN environment. The IP address of the NOX controller is 192.168.100.100, the IP address of Host_1 is 192.168.100.1, and the IP address of Host_2 is 192.168.100.2.

For network cards (Eth0, Eth1, and Eth2), the corresponding IP addresses are 192.168.100.10, 192.168.100.11, and 192.168.100.12, respectively. For OpenFlow-enabled switches Br0-Br6, the corresponding IP addresses are 192.168.100.20, 192.168.100.21, 192.168.100.22, 192.168.100.23, 192.168.100.24, 192.168.100.25, 192.168.100.26, respectively.

Host_1 is the source node, while Host_2 is the destination node. The NOX controller implements our routing application with traffic grooming. When the switch Br0 receives packets from Host_1, Br0 will send OFPT_PACKET_IN message to the NOX controller. In a word, the NOX controller analyzes the global status of the whole network topology, and it then computes the shortest path between two hosts. In Fig. 5, the shortest path from Host_1 to Host_2 is Br0 → Br6 → Br5. Next, the NOX controller distributes OFPT_FLOW_MOD message to Br0, Br6, and Br5. Finally, the NOX controller builds a complete datapath for packets from Host_1 to Host_2 based on flow tables.

D. SDN Communication Process

The interactive process between controller and switch can be described as follows:

1) The switch sends OFPT_HELLO message to the NOX controller.
2) The NOX controller returns OFPT_HELLO message back to the corresponding switch.
3) The NOX controller sends OFPT_FEATURES_REQUEST message to the switch for acquiring the feature and function of this switch.
4) The switch sends the report of switch feature and function to the NOX controller through returning OFPT_FEATURES_REPLY message.
5) The NOX controller sends OFPT_SET_CONFIG message to the switch for configuring this switch. For example, the

NOX controller informs that the maximal length of each packet is 128 bytes.

6) The NOX controller sends OFPT_FLOW_MOD message to the switch. Initially, the NOX controller deletes flow tables by sending this message.
7) The NOX controller sends OFPT_PORT_MOD message to the switch for determining the behavior of physical ports in the switch.
8) When the packet arrives at the switch, if this switch cannot find any flow entries to match the packet in the local flow table, it sends OFPT_PACKET_IN message to the NOX controller.
9) After receiving OFPT_PACKET_IN message, the NOX controller triggers the routing application to find the shortest path for the packet, and it then distributes flow tables to relative switches by using OFPT_FLOW_MOD message. Note that if the result of flow statistics is greater than threshold, the routing application with traffic grooming will dynamically modify the idle time of flow entries, in order to reduce the end-to-end delay.

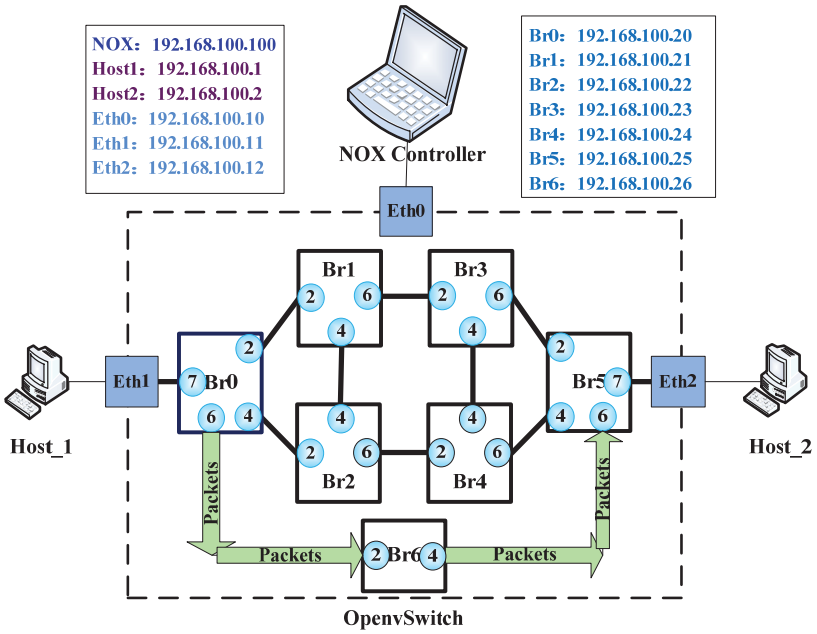

Fig. 5 Diagram of our SDN environment

E. SDN-Based Traffic Grooming

Fig. 6 shows the process of traffic grooming. Each flow entry has an idle_timeout and a hard_timeout associated with it. The idle_timeout and hard_timeout fields control how quickly flows expire. If no packet has matched the flow entry in the last idle_timeout seconds, or it has been hard_timeout seconds since the flow was inserted, the flow entry will be deleted, and the switch sends a flow removed message to controller. In this paper, the idle_timeout is set by us, but the hard_timeout is zero. The flow entry expires after idle_timeout seconds if no received traffic there is. Host_1 sends packets to Host_2 with constant interval time, six seconds. Initially, the idle_timeout of the flow entry is set as five seconds. Thus, when the packets arrive at the switch, the flow entry matched with them has been removed, and the flow tables have to be re-established for forwarding these packets, which results in a long end-to-end delay. For this end, we groom this kind of traffic in the manner of flow statistics, and the NOX controller sends Aggregate_Flow_Statistics message to switches for traffic statistics. If the value of packet_count field in the reply message is greater than a threshold value, we will dynamically extend idle_timeout (i.e. eight seconds), and re-insert flow entries into the flow tables of relative switches along the computed route by using Flow_Mod message, with the objective of reducing the end-to-end delay.

IV. Experimental Results and Discussion

A. Capture of Openflow Messages

To demonstrate the interactive communication process between NOX controller and switch, we capture OpenFlow messages using WireShark. The IP address of the NOX controller is 192.168.100.100, while the IP address of the network card Eth0 connected directly with the NOX controller is 192.168.100.10. In Fig. 9, we can clearly see that the interactive communication processes mainly include: 1) the NOX controller establishes connections with switches (such as Hello messages in Fig. 9); 2) the NOX controller configures the switches in the initial run time (such as Features Request, Set Config, Flow Mod, Port Mod and other messages in Fig. 9).

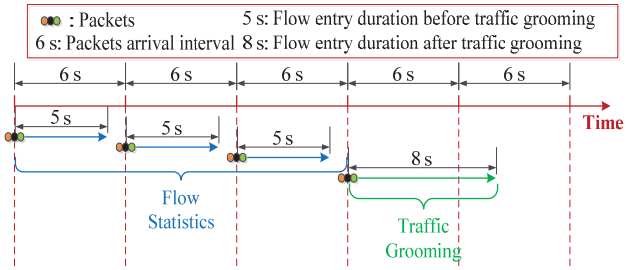

Fig. 6 Diagram of traffic grooming

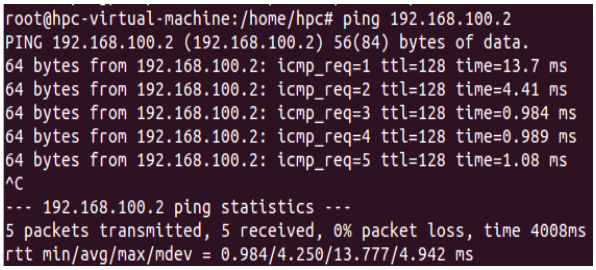

Fig. 7 Experimental result of successful packet forwarding

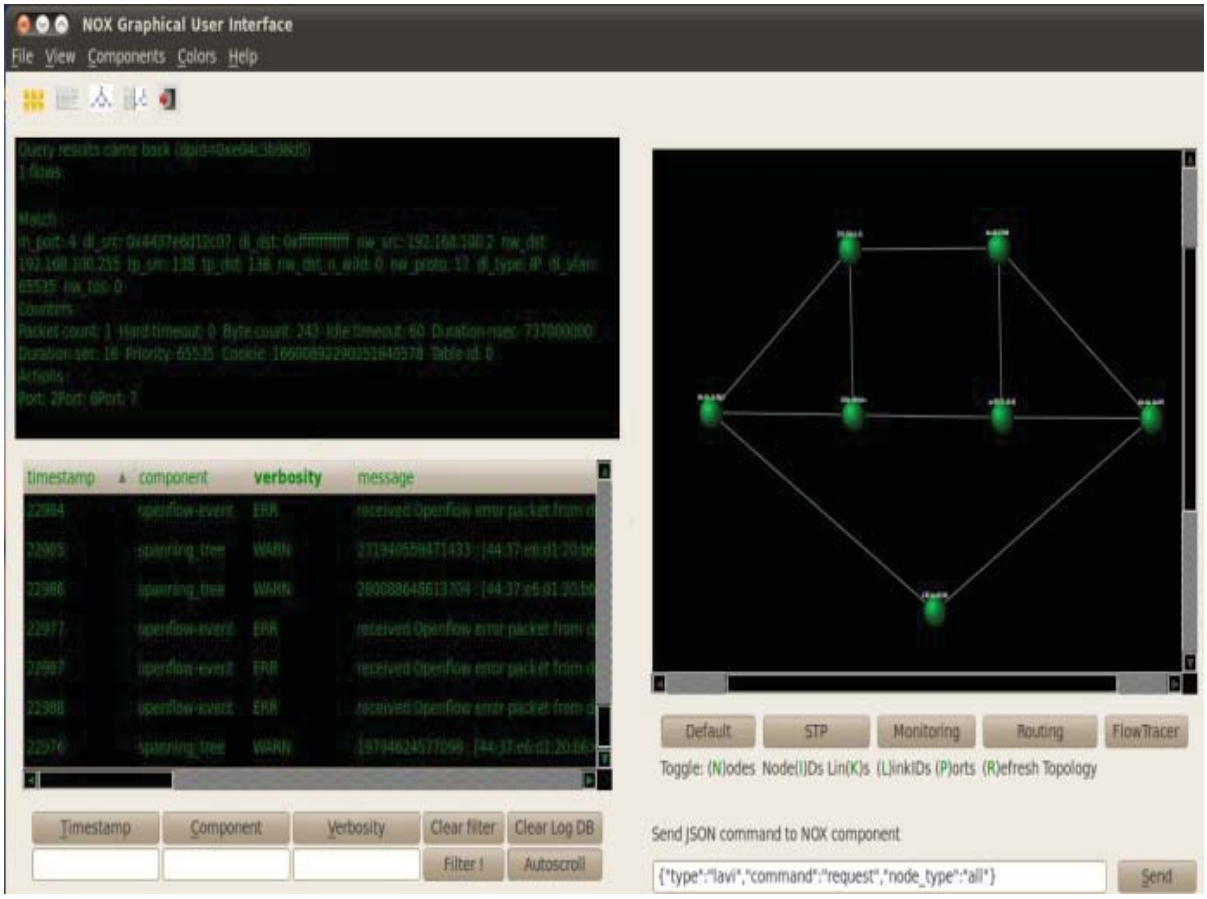

Fig. 8 GUI information of the NOX controller

Figs. 10 and 11 show the WireShark captures of Flow_Mod message before and after traffic grooming, respectively. The source IP address is 192.168.100.1, while the destination address is 192.168.100.2. The input port is No. 6, while the output port is No. 7 for forwarding packets. Fig. 10 demonstrates that the idle time of flow entry is five seconds before executing traffic grooming, while eight seconds after executing traffic grooming in Fig. 11. These results prove the feasibility and effectiveness of our developed routing application with the function of traffic grooming.

In our routing application, the data structures of routing information mainly include: 1) Identifier (ID) of the datapath (i.e., RouteId), and it also includes the source node of the datapath (i.e., the source node Sdp) and the destination node of the datapath (i.e., the destination node Edp); 2) Path links, i.e., a link list (list<Link>), and each link also includes the destination datapath (i.e., dpdst), as well as output and input ports (i.e., outport and inport) of the destination datapath. For example, Table II shows the data structure of routing information for the shortest path Br0 → Br6 → Br5 from Host_1 to Host_2.

When the NOX controller receives the routing request from Br0, it will assign No. 6 output port to Br0 by using Flow-Mod (OFPT_FLOW_MOD) message. Similarly, the NOX controller will assign No. 4 output port to switch Br6. As a result, a complete datapath has been established.

Filter: of Expression... Clear Apply

No.	Time	Source	Destination	Protocol	Length	Info
14	0.224566	192.168.100.10	192.168.100.100	OFP	74	Hello (SM) (8B)
16	0.224653	192.168.100.10	192.168.100.100	OFP	74	Hello (SM) (8B)
17	0.224784	192.168.100.10	192.168.100.100	OFP	74	Hello (SM) (8B)
21	0.226054	192.168.100.100	192.168.100.10	OFP	74	Hello (SM) (8B)
22	0.226275	192.168.100.100	192.168.100.10	OFP	74	Features Request (CSM) (8B)
25	0.226596	192.168.100.10	192.168.100.100	OFP	242	Features Reply (CSM) (176B)
26	0.226837	192.168.100.100	192.168.100.10	OFP	74	Hello (SM) (8B)
28	0.227185	192.168.100.100	192.168.100.10	OFP	74	Features Request (CSM) (8B)
30	0.227475	192.168.100.100	192.168.100.10	OFP	74	Hello (SM) (8B)
31	0.227567	192.168.100.10	192.168.100.100	OFP	290	Features Reply (CSM) (224B)
33	0.227980	192.168.100.100	192.168.100.10	OFP	74	Features Request (CSM) (8B)
35	0.228298	192.168.100.10	192.168.100.100	OFP	290	Features Reply (CSM) (224B)
36	0.228472	192.168.100.100	192.168.100.10	OFP	78	Set Config (CSM) (12B)
37	0.228824	192.168.100.100	192.168.100.10	OFP	78	Set Config (CSM) (12B)
38	0.229043	192.168.100.100	192.168.100.10	OFP	78	Set Config (CSM) (12B)
39	0.229358	192.168.100.100	192.168.100.10	OFP	90	Vendor (SM) (24B)
40	0.229591	192.168.100.100	192.168.100.10	OFP	90	Vendor (SM) (24B)
42	0.229824	192.168.100.10	192.168.100.100	OFP	102	Error (SM) (36B)
44	0.229936	192.168.100.10	192.168.100.100	OFP	102	Error (SM) (36B)
45	0.230172	192.168.100.100	192.168.100.10	OFP	90	Vendor (SM) (24B)
47	0.230536	192.168.100.10	192.168.100.100	OFP	102	Error (SM) (36B)
48	0.230553	192.168.100.100	192.168.100.10	OFP	138	Flow Mod (CSM) (72B)
49	0.230969	192.168.100.100	192.168.100.10	OFP	138	Flow Mod (CSM) (72B)
50	0.233146	192.168.100.100	192.168.100.10	OFP	98	Port Mod (CSM) (32B)
51	0.233462	192.168.100.100	192.168.100.10	OFP	98	Port Mod (CSM) (32B)
57	0.257566	192.168.100.100	192.168.100.10	OFP	98	Port Mod (CSM) (32B)
58	0.257875	192.168.100.100	192.168.100.10	OFP	98	Port Mod (CSM) (32B)
60	0.258120	192.168.100.100	192.168.100.10	OFP	98	Port Mod (CSM) (32B)
62	0.262093	192.168.100.100	192.168.100.10	OFP	138	Flow Mod (CSM) (72B)
63	0.263125	192.168.100.100	192.168.100.10	OFP	98	Port Mod (CSM) (32B)
64	0.263298	192.168.100.100	192.168.100.10	OFP	98	Port Mod (CSM) (32B)
66	0.263468	192.168.100.100	192.168.100.10	OFP	98	Port Mod (CSM) (32B)

Fig. 9 Wireshark capture of OpenFlow messages

```
⊞ Frame 934: 146 bytes on wire (1168 bits), 146 bytes captured (1168 bits)
⊞ Ethernet II, Src: Vmware_f3:9c:12 (00:0c:29:f3:9c:12), Dst: HonHaiPr_15:d8:4c (90:fb:a6:15:d8:4c)
⊞ Internet Protocol Version 4, Src: 192.168.100.100 (192.168.100.100), Dst: 192.168.100.10 (192.168.100.10)
⊞ Transmission Control Protocol, Src Port: 6633 (6633), Dst Port: 57764 (57764), Seq: 2329, Ack: 2509, Len: 80
⊟ OpenFlow Protocol
  ⊟ Header
      Version: 0x01
      Type: Flow Mod (CSM) (14)
      Length: 80
      Transaction ID: 284
  ⊟ Flow Modification
    ⊟ Match
      ⊞ Match Types
        Input Port: 6
        Ethernet Src Addr: AsustekC_f9:0f:27 (f4:6d:04:f9:0f:27)
        Ethernet Dst Addr: Vmware_8f:eb:fe (00:0c:29:8f:eb:fe)
        Input VLAN ID: 65535
        Input VLAN priority: 0
        Ethernet Type: ARP (0x0806)
        IPv4 DSCP: 0
        ARP Opcode: reply (2)
        IP Src Addr: 192.168.100.1 (192.168.100.1)
        IP Dst Addr: 192.168.100.2 (192.168.100.2)
        TCP/UDP Src Port: 0 (0)
        TCP/UDP Dst Port: 0 (0)
      Cookie: 0x0000000000000000
      Command: New flow (0)
      Idle Time (sec) Before Discarding: 5
      Max Time (sec) Before Discarding: 0
      Priority: 32768
      Buffer ID: 288
      Out Port (delete* only): None  (not associated with a physical port)
    ⊞ Flags
    ⊟ Output Action(s)
      ⊟ Action
          Type: Output to switch port (0)
          Len: 8
          Output port: 7
          Max Bytes to Send: 0
        # of Actions: 1

0000  90 fb a6 15 d8 4c 00 0c  29 f3 9c 12 08 00 45 00   .....L.. ).....E.
0010  00 84 36 ea 40 00 40 06  b9 ca c0 a8 64 64 c0 a8   ..6.@.@. ....dd..
0020  64 0a 19 e9 e1 a4 f9 fc  39 0a 33 3f 1f b5 80 18   d....... 9.3?....
0030  00 36 e8 d1 00 00 01 01  08 0a 00 1d 20 21 00 91   .6...... .... !..
0040  ae 0a 01 0e 00 50 00 00  01 1c 00 00 00 00 00 06   .....P.. ........
0050  f4 6d 04 f9 0f 27 00 0c  29 8f eb fe ff ff 00 00   .m...'.. ).......
0060  08 06 00 02 00 00 c0 a8  64 01 c0 a8 64 02 00 00   ........ d...d...
0070  00 00 00 00 00 00 00 00  00 00 00 00 00 05 00 00   ........ ........
0080  80 00 00 00 01 20 ff ff  00 01 00 00 00 08 00 07   ..... .. ........
0090  00 00                                              ..
```

Fig. 10 Flow-mod message before traffic grooming

```
⊞ Frame 25881: 146 bytes on wire (1168 bits), 146 bytes captured (1168 bits)
⊞ Ethernet II, Src: Vmware_f3:9c:12 (00:0c:29:f3:9c:12), Dst: HonHaiPr_15:d8:4c (90:fb:a6:15:d8:4c)
⊞ Internet Protocol Version 4, Src: 192.168.100.100 (192.168.100.100), Dst: 192.168.100.10 (192.168.100.10)
⊞ Transmission Control Protocol, Src Port: 6633 (6633), Dst Port: 57764 (57764), Seq: 58441, Ack: 333653, Len: 80
⊟ OpenFlow Protocol
  ⊟ Header
      Version: 0x01
      Type: Flow Mod (CSM) (14)
      Length: 80
      Transaction ID: 536
  ⊟ Flow Modification
    ⊟ Match
      ⊞ Match Types
        Input Port: 6
        Ethernet Src Addr: AsustekC_f9:0f:27 (f4:6d:04:f9:0f:27)
        Ethernet Dst Addr: Vmware_8f:eb:fe (00:0c:29:8f:eb:fe)
        Input VLAN ID: 65535
        Input VLAN priority: 0
        Ethernet Type: ARP (0x0806)
        IPv4 DSCP: 0
        ARP Opcode: request (1)
        IP Src Addr: 192.168.100.1 (192.168.100.1)
        IP Dst Addr: 192.168.100.2 (192.168.100.2)
        TCP/UDP Src Port: 0 (0)
        TCP/UDP Dst Port: 0 (0)
      Cookie: 0x0000000000000000
      Command: New flow (0)
      Idle Time (sec) Before Discarding: 8
      Max Time (sec) Before Discarding: 0
      Priority: 32768
      Buffer ID: 1140
      Out Port (delete* only): None  (not associated with a physical port)
    ⊞ Flags
    ⊟ Output Action(s)
      ⊟ Action
          Type: Output to switch port (0)
          Len: 8
          Output port: 7
          Max Bytes to Send: 0
        # of Actions: 1

0000  90 fb a6 15 d8 4c 00 0c  29 f3 9c 12 08 00 45 00   .....L.. ).....E.
0010  00 84 3e 5a 40 00 40 06  b2 5a c0 a8 64 64 c0 a8   ..>Z@.@. .Z..dd..
0020  64 0a 19 e9 e1 a4 f9 fd  14 3a 33 44 2d 3d 80 18   d....... .:3D-=..
0030  01 f5 92 45 00 00 01 01  08 0a 00 1d 86 ea 00 92   ...E.... ........
0040  ae fd 01 0e 00 50 00 00  02 18 00 00 00 00 00 06   .....P.. ........
0050  f4 6d 04 f9 0f 27 00 0c  29 8f eb fe ff ff 00 00   .m...'.. ).......
0060  08 06 00 01 00 00 c0 a8  64 01 c0 a8 64 02 00 00   ........ d...d...
0070  00 00 00 00 00 00 00 00  00 00 00 00 00 08 00 00   ........ ........
0080  80 00 00 00 04 74 ff ff  00 01 00 00 00 08 00 07   .....t.. ........
0090  00 00                                              ..
```

Fig. 11 Flow-mod message after traffic grooming

```
OFPT_FEATURES_REPLY (xid=0x1): dpid:000000e04c3b98d5
n_tables:255, n_buffers:256
capabilities: FLOW_STATS TABLE_STATS PORT_STATS QUEUE_STATS ARP_MATCH_IP
actions: OUTPUT SET_VLAN_VID SET_VLAN_PCP STRIP_VLAN SET_DL_SRC SET_DL_DST SET_NW_SRC SET_NW_DST SET_NW_TOS SET_TP_SRC SET_TP_DST EN
QUEUE
 2(patch-0-1): addr:a6:5c:d5:0f:aa:a5
     config:     NO_FLOOD
     state:      0
     speed: 100 Mbps now, 100 Mbps max
 4(patch-0-2): addr:1e:3e:0e:85:3a:0d
     config:     0
     state:      0
     speed: 100 Mbps now, 100 Mbps max
 6(patch-0-6): addr:92:10:f8:f8:2d:94
     config:     NO_FLOOD
     state:      0
     speed: 100 Mbps now, 100 Mbps max
 7(eth1): addr:00:e0:4c:3b:98:d5
     config:     0
     state:      0
     current:    100MB-FD AUTO_NEG
     advertised: 10MB-HD 10MB-FD 100MB-HD 100MB-FD COPPER AUTO_NEG
     supported:  10MB-HD 10MB-FD 100MB-HD 100MB-FD COPPER AUTO_NEG
     speed: 100 Mbps now, 100 Mbps max
 LOCAL(br0): addr:00:e0:4c:3b:98:d5
     config:     0
     state:      0
     speed: 100 Mbps now, 100 Mbps max
OFPT_GET_CONFIG_REPLY (xid=0x3): frags=normal miss_send_len=0
```

Fig. 12 Detailed information of Br0

B. Routing Analysis Based on Flow Tables

In this subsection, we demonstrate the effectiveness of our routing application by viewing the detailed information of all switches and their local flow tables in the SDN environment. After executing the command *ovs-ofctl show br0*, we can obtain the detailed information of Br0, which can be seen in Fig. 12.

We can see that the datapath id (dpid) of Br0 is 000000e04c3b98d5, the number of flow tables is 255, and the maximal number of packets is 256 in the buffer. In Br0, No. 2 port connects with Br1, No. 4 port connects with Br2, No. 6 port connects with Br6, and No. 7 port connects with Eth1. This information is consistent with that of Fig. 5. After executing the command *ovs-ofctl dump-flows br0*, we can obtain the flow table of Br0. Similarly, we can obtain the flow tables of Br1-Br6. Finally, the summary information of all flow tables from Br0-Br6 is in Table III.

The IP address of Host_1 is 192.168.100.1, and the IP address of Host_2 is 192.168.100.2. When Host_1 sends packets to Host_2, there is no flow table in Br0 initially. But when the packet arrives at Br0, Br0 sends OFPT_PACKET_IN message to the NOX controller. After receiving this message, the NOX controller analyzes the global status of the whole network topology. Next, the NOX controller establishes appropriate flow tables for the routing request, and it then distributes flow tables to relative switches. So, the distributed flow tables can be seen in Table III. After that, the packet will be forwarded according to the bold routing information in Table III. More specifically, Br0 first receives the packet from No. 7 port and sends the packet to Br6 from No. 6 port. Br6 receives the packet from No. 2 port and sends the packet to Br5 from No. 4 port. Finally, Br5 receives the packet from No. 6 port and sends the packet to Eth2 from No. 7 port. Obviously, the shortest path from Host_1 to Host_2 is Br0 → Br6 → Br5.

TABLE II
DATA STRUCTURE OF SAVING ROUTING INFORMATION

Path	RouteId		Path		
From Br0 to Br5	Sdp	Edp	dpdst	outport	inport
Br0 → Br6 → Br5	Br0	Br5	Br0 → Br6	6	2
			Br6 → Br5	4	6

TABLE III
SUMMARY INFORMATION OF FLOW TABLES

Open vSwitch	Input port	Source IP address	Destination IP address	Output port
Br0	**7**	**192.168.100.1**	**192.168.100.2**	**6**
	6	192.168.100.2	192.168.100.1	7
Br1	No flow table			
Br2	No flow table			
Br3	No flow table			
Br4	No flow table			
Br5	**6**	**192.168.100.1**	**192.168.100.2**	**7**
	7	192.168.100.2	192.168.100.1	6
Br6	**2**	**192.168.100.1**	**192.168.100.2**	**4**
	4	192.168.100.2	192.168.100.1	2

C. Simulation Results

In this paper, we consider two performance metrics of our routing application in the real SDN environment: CPU utilization of NOX and average delay of sending packets from Host_1 to Host_2 before and after executing traffic grooming. We first evaluate the CPU utilization of NOX with different flow interval times, and the corresponding results are shown in Fig. 13. The NOX controller accounts for path computation and provisioning on a virtual machine with 1 Processor and 1 GB Memory.

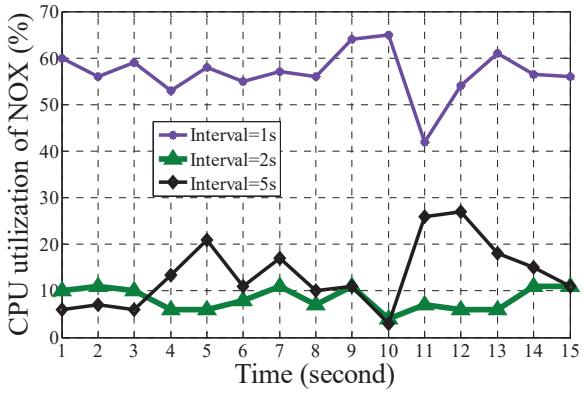

Fig. 13 CPU utilization of NOX

To evaluate the CPU utilization of NOX in the case of different loads, Host_1 sends 15 packets to Host_2 with the flow interval time of one, two and five seconds, respectively. From Fig. 13, we can see that when the flow interval time is one second, the NOX controller will be very busy and highly CPU loaded. But when the flow time interval increases to two or five seconds, the CPU load is sharply alleviated.

Fig. 14 compares the end-to-end delay before and after executing traffic grooming, the horizontal axis is the packet arrival sequence, while the vertical axis records the end-to-end delay including the propagation time of OpenFlow message and the processing latency of the NOX controller. The average end-to-end delay is shown in Fig. 15, and we can clearly observe that the delay quickly decreases after executing traffic grooming. Therefore, the routing application with traffic grooming achieves the objective of reducing the end-to-end delay by dynamically modifying the idle time of flow tables.

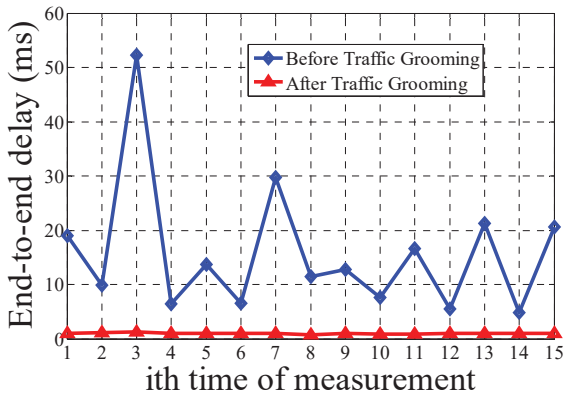

Fig. 14 End-to-end delay

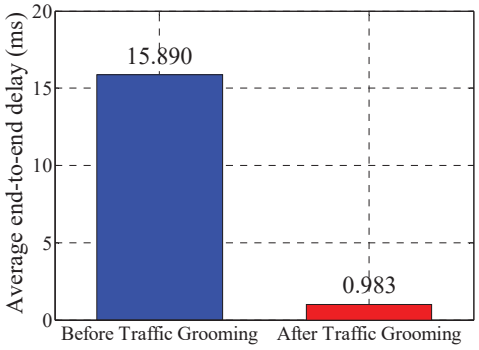

Fig. 15 Average end-to-end delay

V. Conclusion

In this paper, we have experimentally setup a real SDN environment by using NOX controller, Open vSwitch and OpenFlow protocol, and the routing application with traffic grooming was implemented for reducing end-to-end delay. The NOX controller has been installed in Ubuntu-10.04, and it has been successfully used to control all switches in the topology; Open vSwitch has been installed in Ubuntu-12.04. The host with including Open vSwitch has three network cards, and these network cards connect NOX controller, Host_1 and Host_2, respectively. Open vSwitch has been successfully used to build a virtual network topology. Finally, our routing application with traffic grooming has been performed within the NOX controller, and the experimental results have demonstrated the effectiveness of our routing services.

Acknowledgement

This work was supported in part by Fundamental Research Funds for the Central Universities (Grant Nos. N130817002, N140405005, N150401002), Foundation of the Education Department of Liaoning Province (Grant No. L2014089), National Natural Science Foundation of China (Grant Nos. 61302070, 61401082, 61471109), Liaoning BaiQianWan Tal-ents Program, and National High-Level Personnel Special Support Program for Youth Top-Notch Talent.

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, et al. OpenFlow: Enabling Innovation in Campus Networks. ACM SIGCOMM Computer Communication Review, vol. 38, no. 2, pp. 69-74, 2008.

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

[3] OpenFlow Protocol Specification v1.0. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf, 2009-12-31.

[4] S. Das, G. Parulkar, N. McKeown, et al. Packet and Circuit Network Convergence with OpenFlow. Proceedings of OFC, San Diego, 2010, pp. 1-3.

[5] S. Das, Y. Yiakoumis, G. Parulkar, et al. Application-aware Aggregation and Traffic Engineering in a Converged Packet-circuit Network. Proceedings of OFC, Los Angeles, 2011, pp. 1-3.

[6] L. Xiong, X. H. Wei, F. X. Ping, et al. Design of the Multi-Level Security Network Switch System with Restricts Covert Channel. Proceedings of ICCSN, Xi an, 2011, pp. 233-237.

[7] L. H. Zhuo, L. S. Jing, Y. F. Tao, et al. Apply Embedded OpenFlow MPLS Technology on Wireless OpenFlow-OpenRoads. Proceedings of CECNet, Yi chang, 2012, pp. 916-919.

[8] L. Lei, Y. C. Hyeon, C. Ramon, et al. Demonstration of a Dynamic Transparent Optical Network Employing Flexible Transmitters/Receivers Controlled by an OpenFlow–Stateless PCE Integrated Control Plane. IEEE/OSA Journal of Optical Communications and Networking, vol. 5, no. 10, pp. 66-75, 2013.

[9] C. Ramon, M. Ricardo, L. Lei, et al. Control and Management of Flexi-grid Optical Networks with an Integrated Stateful Path Computation Element and OpenFlow Controller. IEEE/OSA Journal of Optical Communications and Networking, vol. 5, no. 10, pp. 57-65, 2013.

[10] R. R. Bijan, Z. George, Y. Yan, et al. All Programmable and Synthetic Optical Network: Architecture and Implementation. IEEE/OSA Journal of Optical Communications and Networking, vol. 5, no. 9, pp. 1096-1110, 2013.

[11] D. Z. Xu, L. Lei, L. H. Feng, et al. Experimental Demonstration of OBS/WSON Multi-Layer Optical Switched Networks with an OpenFlow-based Unified Control Plane. Proceedings of ONDM, Colchester, 2012, pp. 1-6.

[12] D. Simeonidou, R. Nejabati, S. Azodolmolky. Enabling the Future Optical Internet with OpenFlow: a Paradigm Shift in Providing Intelligent Optical Network Services. Proceedings of ICTON, Stockholm, 2011, pp. 1-4.

[13] N. Gude, T. Koponen, J. Pettit, et al. Nox: Towards an Operating System for Networks. ACM SIGCOMM Computer Communication Review, vol. 38, no. 3, pp. 105−110, 2008.

[14] H. W. Chiu, S. Y. Wang. Boosting the OpenFlow Control-Plane Message Exchange Performance of Open vSwitch. Proceedings of ICC, London, 2015, pp. 5284-5289.