Comparisons of SDN OpenFlow Controllers over EstiNet: Ryu vs. NOX

Shie-Yuan Wang
Department of Computer Science
National Chiao Tung University
Hsinchu, Taiwan
Email: shieyuan@cs.nctu.edu.tw

Hung-Wei Chiu
Department of Computer Science
National Chiao Tung University
Hsinchu, Taiwan
Email: hwchiu@cs.nctu.edu.tw

Chih-Liang Chou
EstiNet Technologies, Inc.
Hsinchu, Taiwan
Email: clchou@estinet.com

***Abstract*—SDN (Software-defined Networks) is a new approach to networking in which the control plane is extracted from the switch and put into the software application called the controller. In an SDN, the controller controls all networking switches and implements specific network protocols or functions. So far, the OpenFlow protocol is the most popular protocol used to exchange messages between the controller and OpenFlow switches. In this paper, we use the EstiNet OpenFlow network simulator and emulator to compare two open source popular OpenFlow controllers — Ryu and NOX. We studied the behavior of Ryu when it controls a network with loops and how quickly Ryu and NOX can find a new routing path for a greedy TCP flow after a link's status has changed. Our simulation results show that (1) Ryu results in the packet broadcast storm problem in a network with loops; (2) Ryu and NOX have different behavior and performance in detecting link failure and changing to a new routing path.**

Keywords*–*SDN; OpenFlow; Network Simulator.

I. Introduction

SDN (Software-Defined Networks) [1] is an emerging network architecture. It is a programmable, dynamic, adaptable, and well-managed network architecture. SDN extracts the control-plane functions from legacy switches and implements them into a software application called a controller. Nowadays, the OpenFlow [2] [3] protocol is the most popular protocol used for a controller to control SDN switches. Via this protocol, an OpenFlow switch learns the forwarding information from the controller and forwards incoming packets based on the received information. In an SDN network, people often call a network function implemented by a controller a "controller application." A controller application can implement a useful network function such as network virtualization. With these applications, network administrators can more easily manage an SDN network.

In an SDN network, the OpenFlow controller and its various applications work together to control the network and provide services. Before one introduces a new controller application into an SDN network, however, one must validate and evaluate its correctness, efficiency, and stability. There are several approaches that can be used for this purpose. One approach is creating an OpenFlow network testbed with real OpenFlow switches. Although the results of this approach are convincing, it incurs very high cost and the network settings cannot be very flexible. Another approach is via simulation in which all network switches, links, protocols, their operations, and the interactions between them are all simulated by a software program. Generally speaking, if the simulation modeling is correct enough, the simulation approach is a low-cost, flexible, scalable, and repeatable approach. This explains its wide uses in the research communities.

In this paper, we used the EstiNet OpenFlow network simulator and emulator [4] [5] to compare the behavior and performance of two popular OpenFlow controllers — Ryu [6] and NOX [7]. EstiNet uses an innovative simulation methodology called the “”kernel-reentering” simulation methodology to provide many unique advantages. When EstiNet simulates a network, each simulated host uses the (shared) real-world Linux operating system and allows any real-world Linux programs to run over it without any modification. For this property, the real-life Ryu and NOX controller programs can readily run over EstiNet to control many simulated OpenFlow switches and we can create simulation test cases to study the details of their behavior.

There are several other popular open source OpenFlow controllers [8] [9] [10]. The reason why we chose to study NOX and Ryu is because NOX is the world's first OpenFlow controller and Ryu is widely used with the OpenStack cloud operating system for cloud orchestration. Both Ryu and NOX are real-world applications written in the python language and they are runnable on any operating system supporting python. In this paper, we chose the learning bridge protocol (LBP) and spanning tree protocol (STP) controller applications to study. We observed the phenomenon when using Ryu as the OpenFlow controller in a network with loops and studied how quickly Ryu and NOX can find a new path for a greedy TCP flow after a link's status has changed. We also studied the impact of the address resolution protocol (ARP) on the path-changing time for a greedy TCP flow under the control of Ryu and NOX. Our results reported in this paper reveal the performance, behavior, and implementation flaws of NOX and Ryu over the tested network settings.

The rest of the paper is organized as follows. In Section II, we present some information about EstiNet OpenFlow network simulator and emulator to let readers know more about its special capabilities in conducting SDN researches. In Section III, we show the simulating settings used in this study. In Section IV, we explain the path-finding and packet-forwarding functions in Ryu and NOX. Performance evaluation results are presented in Section V. Finally, we conclude the paper in Section VI.

II. EstiNet OpenFlow Network Simulator and Emulator

The current version (as of the publication date) of EstiNet OpenFlow network simulator and emulator is 9.0. It can accurately simulate thousands of Ver 1.4.1 OpenFlow switches. It supports both of the simulation mode and the emulation mode. In the simulation mode, a real-world open source OpenFlow controller such as NOX, POX, Floodlight, or Ryu application program can directly run up on a controller node in the simulated network to control these simulated OpenFlow switches without any modification. In the emulation mode, these controller application programs can run up on an external machine that is different from the machine used to simulate OpenFlow switches to control these simulated OpenFlow switches. In additiion, in the emulation mode, if an OpenFlow controller has been implemented as a dedicated hardware device, it can remotely control the simulated OpenFlow switches in EstiNet via an Ethernet cable.

EstiNet, when running in the simulation mode, can accurately simulate the properties of the links that connect simulated OpenFlow switches. These properties include link bandwidth, link delay, link downtime, and the medium access control (MAC) protocol used over the link (e.g., IEEE 802.3 or IEEE 802.11, etc.). As a result, performance evaluation of traffic flows or the whole OpenFlow network can be accurately studied in EstiNet. Furthermore, since during simulation, the advancement of the simulation clock is accurately controlled by EstiNet, the performance simulation results of EstiNet are always realistic and repeatable, totally unaffected by the number of OpenFlow switches and hosts simulated by it. These unique and important capabilities enable us to conduct SDN research for various purposes. In this paper, we used these capabilities to compare the in-depth differences in the protocol operations of the NOX and Ryu SDN controllers.

III. Simulation Settings

Figure 1 shows the network topology that we used for this study. Each of node 3, 4, 5 and 11 simulates a host running the real-world Linux operating system (Fedora 14). On top of these nodes, any real-world Linux program can run without modification. Each of node 6, 7, 8, 9 and 10 simulates an OpenFlow switch supporting the OpenFlow protocol version 1.0. Node 1 is a simulated host on top which the Ryu or NOX OpenFlow controller program will run. Node 2 is a simulated legacy switch that connects all simulated OpenFlow switches together with the OpenFlow controller node. This formed network is the control-plane network. All TCP connections between simulated OpenFlow switches and the OpenFlow controller are set up over the control-plane network and the messages between Ryu/NOX and simulated OpenFlow switches are all exchanged over this control-plane network. In contrast, all simulated hosts, simulated OpenFlow switches and all links connecting them together form the data-plane network and the real applications running on simulated hosts will exchange their information over the data-plane network.

We studied two simulation cases on the network topology shown in Figure 1. We set the link delay and bandwidth to 10 ms and 100 Mbps for each link in these simulation cases. Both cases start at 0'th sec and ends at 120'th sec. In case 1, we only used Ryu as the OpenFlow controller. Because Ryu only uses LBP (Learning Bridge Protocol) to forward packets

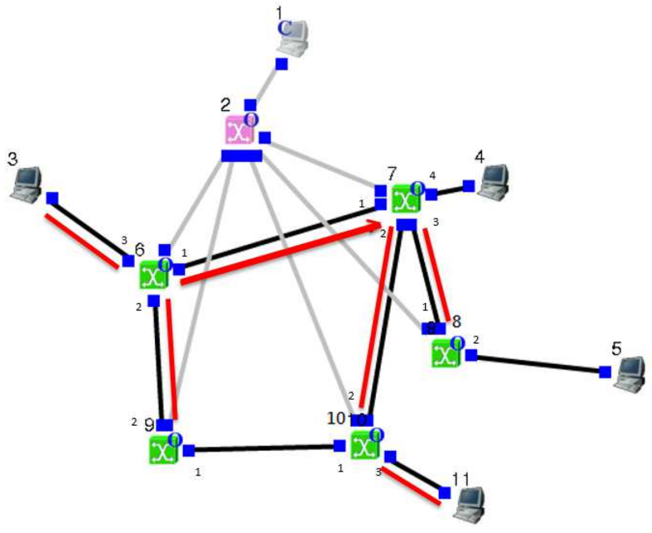

Figure 1. Spanning Tree in NOX before the link(6,7) breaks

without using SPT (Spanning Tree Protocol), we wanted to observe the phenomenon of running Ryu in a network with loops.

In case 2, we studied the required time to find a new path after a links status has changed when Ryu or NOX was used as the OpenFlow controller. We purposely broke the link between nodes 6 and 7 between 40'th sec and 80'th sec and shutdown the link between nodes 9 and 10 from 0'th sec to 40'th sec and from 80'th sec to 120'th sec. We studied the required time that the TCP flow can continue its transmission after a link on its path fails. We chose node 3 as the TCP sender and node 11 as the TCP receiver and generated endless TCP traffic from node 3 to node 11. Because over EstiNet one can directly run real-world Linux applications on simulated hosts without modification, we chose the open source programs "stcp" and "rtcp" as the TCP sender and TCP receiver programs. Once the stcp successfully sets up a real TCP connection with rtcp, it generates and sends endless TCP data to the rtcp. Both stcp and rtcp are set to start at 30'th sec rather than 0'th sec. This is because the OpenFlow controller needs enough time to discover the topology of the data-plane network and compute the path-finding and packet-forwarding information before any packet enters into the data-plane network.

We also studied the effects of the ARP protocol on the required time that the TCP flow finds a new path. Normally, on a real network the ARP protocol is enabled and the host will issue an ARP request to learn (MAC address, IP address) mapping information. However, avoid the ARP request/reply latency and bandwidth consumption, the ARP protocol can be disabled under some circumstances. Our simulation results show that when ARP is disabled, the path-finding capability and speed of NOX will significantly reduce.

IV. Path-Finding and Packet-Forwarding Functions in Ryu/NOX

In this section, we briefly explain how Ryu and NOX implement the path-finding and packet-forwarding functions.

A. LBP Component in Ryu

Ryu uses its LBP component to perform path-finding and packet-forwarding functions. When a switch issues a PacketIn message to Ryu due to a table-miss event, Ryu learns a "(switch ID, MAC address) = output port" mapping information from the packet that causes this table miss. This learned mapping information helps Ryu know through what "output port" the switch should forward a packet when its destination MAC address is the specified "MAC address" here. If Ryu currently has the mapping information for the destination host, it will send a FlowModify message to the switch to add a new flow entry and send a PacketOut message to the switch to forward the packet out of the specified port. Otherwise it will send a PacketOut message to the switch to flood the packet out of all of its ports. Here we use Figure 1 to explain how Ryu's LBP works. In order to explain how it works clearly, we assume that the link connecting nodes 9 and 10 is down on Figure 1 so that there is no loop in the network.

Suppose that the source host sends a TCP DATA packet to the destination host. When the packet arrives at node 6, because there are no flow entries in the flow table that can match it, node 6 sends a PacketIn message to Ryu and asks it how to process this packet. After Ryu receives the packet, it first learns (node 6, node 3's MAC) = port 3 mapping information. Because there are no mapping information about the destination host yet, it sends a PacketOut message to node 6 to flood the packet. After node 7 receives the packet, it issues a PacketIn message to Ryu and Ryu learns (node 7, node 3's MAC) = port 1. It then sends a PacketOut message to instruct node 7 to flood this packet. On the other hand, after node 9 receives the packet from node 6, the same scenario happens. It issues a PacketIn message to Ryu. Then, Ryu learns (node 9, node 3's MAC) = port 2 and sends a PacketOut message to node 9 asking it to flood this packet. However, because the link between nodes 9 and 10 is shutdown, node 10 cannot receive the packet from node 9. After node 10 receives the packet from node 7, the same scenario happens. Ryu learns (node 10, node 3's MAC) = port 2 and sends a PacketOut to instruct node 10 to flood this packet. When the destination host receives the packet, it sends a TCP ACK packet to the source host. When the packet arrives at node 10, because there are no flow entries that the packet can match, node 10 issues a PacketIn to Ryu and Ryu learns (node 10, node 11's MAC) = port 3. Now, with the mapping information learned before, Ryu issues a FlowModify message to node 10 instructing it to add a new flow entry of (destination MAC = host 3's MAC, ingress port = port 3, output port = port 2) and issues a PacketOut message to node 10 asking it to forward the packet out of port 2. After node 7 receives the packet, the same scenario happens. Ryu learns (node 7, node 11's MAC) = port 2. Then, Ryu issues a FlowModify message to node 7 instructing it to add a new flow entry of (destination MAC = host 3's MAC, ingress port = port 2, output port = port 1) and issues a PacketOut message to node 7 asking it to forward the packet to node 3 out of port 1. After node 3 receives the packet, the same scenario happens. Ryu sends a PacketOut message to node 3 asking it to forward the packet to node 3 out of port 3. After the TCP ACK packet enters node 3, the route from the destination host to the source host has completed and the related flow entries have been added into the switches. However, the route from the source host to the destination host is not completed yet. Later on, when node 3 sends the second TCP DATA packet to the destination host, after the packet enters node 6, the same scenario happens. Node 6 issues a PacketIn message to Ryu and Ryu issues a FlowModify message and PacketOut message to node 6. After the packet finally enters node 11, the route from the source host to the destination host finally is completed and the related flow entries have been added into the switches on the path. At this moment, the TCP flow can bidirectionally send its data along the path composed of nodes 6, 7, and 10 without bothering the Ryu controller.

B. STP and LBP Components in NOX

NOX's STP uses LLDP [11] packets to discover the topology of an OpenFlow network and build a spanning tree over the network. When an OpenFlow switch establishes a TCP connection to NOX, NOX immediately sends a FlowModify message to it and add a flow entry into its flow table. This flow entry will match future received LLDP packets and forward them to NOX. To discover the whole network topology, NOX sends LLDP packets to all switch ports in the network periodically. The LLDP transmission interval is 5 seconds. If there are N switch ports in the network, NOX will send a LLDP packet every (5 divided by N) seconds to evenly spread the LLDP traffic load. For each port of a switch, NOX will send a PacketOut message to the switch and ask it to send the LLDP packet carried in the PacketOut message out of the specified port every 5 seconds. Because NOX has taught the switch how to process LLDP packets before, when a switch receives a LLDP packet from other switches, it will send the received LLDP packet to NOX. With these received LLDP packets from switches, NOX knows the complete network topology and can build a spanning tree over it. For each link in the topology, NOX sends a PortModify message to the switches at the two endpoints of the link. This message sets the flooding status of the port connected to the link to FLOOD/NO_FLOOD according to whether the link is included/excluded in the spanning tree. NOX sets up a 10-second timer that is two times of the LLDP transmission interval to monitor a link's connectivity when it has been detected. When a link's timer expires, NOX thinks that this link is currently down and will build a new spanning tree. Then it sends a PortModify message to switches to change the flooding status of the affected ports.

NOX's LBP implementation is similar to Ryu's LBP implementation. The only difference is that NOX uses the spanning tree to prevent the packet broadcast storm problem. We use Figure 1 to explain how it works on this network topology. NOX uses STP to build a spanning tree as shown in Figure 1 and the spanning tree is composed of the links in red color and all links connecting a host to a switch . Suppose that the source node 3 sends a TCP DATA packet to the destination node 11. As discussed previously, when node 10 receives the flooded packet from node 7, it will issue a PacketIn message to NOX and then NOX issues a PacketOut message to node 10 to instruct it to flood the packet. However, because the status of port 1 of node 10 is set to NO_FLOOD, node 10 will not flood the packet on port 1.

V. Performance Evaluation

In this section, we studies two simulation cases. We first observed the phenomenon of Ryu when it operates in a network with loops. Then, we studied how quickly a TCP flow can change its path to a new path under the control of Ryu and NOX when the topology changes.

A. Case 1

Our simulation results show that critical problems result when using Ryu as the OpenFlow controller in a topology with loops. These problems are the packet broadcast storm problem and insertion of incorrect flow entries into OpenFlow switches. We use Figure 1 to explain why these problems occur. Before the source host sends a TCP data packet to the destination host, it must broadcast the ARP request to learn the destination host's MAC address. After node 6 receives this ARP packet, it issues a PacketIn message to Ryu. Ryu learns the (node 6, node 3's MAC) = port 3 mapping information from this ARP packet but Ryu does not know any information about the destination MAC address, which is the broadcast address. As a result, Ryu sends a PacketOut message to node 6 asking it to flood the ARP packet. After node 7 and node 9 receive the packet, the same scenario happens and both nodes 7 and 9 flood the packet. Later on, node 10 receives the two ARP packets, one from node 7 and the other from node 9. Here we assume that the packet from node 9 arrives earlier than the packet from node 7. For each of these ARP packets, node 10 sends a PacketIn message to Ryu and Ryu sends a PacketOut message to node 10 asking it to flood the ARP packet.

When node 9 receives the ARP packet flooded from node 10, Ryu learns (node 9, node 3's MAC) = port 1 and overrides its information learned before. When node 7 receives the ARP packet flooded from node 10, Ryu learns (node 7, node 3's MAC) = port 2 and overrides its information learned before. When node 11 receives the ARP packet, it responds an ARP reply to node 3. When node 10 receives this ARP reply packet, node 10 issues a PacketIn message to Ryu and Ryu learns (node 10, node 11's MAC) = port 3. Ryu sends a FlowModify message to node 10 instructing it to add a new flow entry of (destination MAC = host 3's MAC, ingress port = port 3, output port = port 2) according to the information learned before. The idle timeout value and the hard timeout value associated with the flow entry are set to 0 by Ryu. Ryu then sends a PacketOut message to node 10 asking it to forward the packet out of port 2. This is because the ARP packet from node 7 was flooded on node 10 after the ARP packet from node 9 was flooded on node 10, which causes Ryu to learn that node 3 can be reached from port 2 of node 10.

According to the OpenFlow protocol, an idle timeout value specifies after how long the entry should be removed if no packet of this flow enters the switch to match this flow. On the other hand, a hard timeout value specifies after how long the entry should be removed after it has been added to the flow table. Since Ryu sets both the idle timeout and the hard timeout values to zero, it means that the flow entry is permanent and should never expire.

After node 7 receives the ARP reply packet, Ryu sends a FlowModify message to node 7 instructing it to add a new flow entry of (destination MAC = host 3's MAC, ingress port = port 2, output pot = port 2) and sends a PacketOut message to node 7 asking it to forward the packet out of port 2. (This is because node 7 received the ARP broadcast packet flooded from node 10, which made Ryu learn that node 3 can be reached from port 2 of node 7.) However, because the packet ingress port and output port are the same and the flow entry never expires, node 7 decides to drop the ARP reply packet. Because this incorrect flow entry has been added into node 7 and it will never expire due to the timeout value settings, from now on, any packet sent from node 11 to node 3 will be dropped at node 7. On the other hand, because Ryu cannot learn the forwarding information about the broadcast MAC address, it will ask OpenFlow switches to flood any received ARP request packet. As a result, for each ARP request packet received on any OpenFlow switch, it will be copied and flooded many times in the network. Worse yet, because there is no STP in the network, these ARP packets will spawn themselves repeatedly and eventually exhaust all the network bandwidth and cause the packet broadcast storm problem.

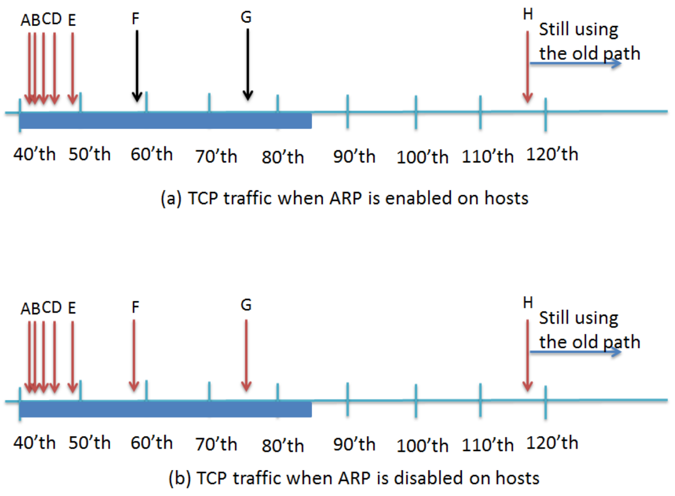

Figure 2. The timeline for a TCP flow to change/keep its path after the link between nodes 6 and 7 breaks at 40'th sec while using Ryu

B. Case 2 - Ryu

We studied how quickly the TCP flow can change its path to a new path under the control of Ryu. Our results show that the TCP flow never changes its path to a new path during the link(6,7) downtime and it becomes active again over the original path after the link downtime no matter whether ARP is enabled or disabled.

We use a timeline to display the significant events of TCP flow and ARP packet when ARP is enabled on hosts in Figure 2(a). Because we want to observe the related events of changing to the new path, our timeline only focuses on the interval from 40'th sec to 120'th sec. We denote the events A, B, C, D, E and H as the timestamps when the TCP flow tried to resend a lost packet due to the TCP reliable transmission design and the events F and G as the timestamps when the source host broadcasts the ARP request to learn the ARP record again. These ARP request transmissions occur because on the source host the ARP record for the destination host had expired during the long TCP retransmission interval. We found that there are two ARP packets but not TCP packets at the timestamps of events F and G. The H event represents the successful retransmission of the lost packet over the original path after the link between nodes 6 and 7 becomes up again at 80'th sec.

In the following, we explain (1) why the TCP flow cannot change to a new path during the link downtime and (2) why there are two ARP packets but no TCP packets at the timestamps of events F and G. The reason why the TCP retransmission fails at events A, B, C, D, and E are the same. Before the link breaks at 40'th sec, Ryu uses its LBP to find a path from the source host to the destination host, which traverses nodes 6, 7, and 10. Ryu also sends the FlowModify message and PacketOut message to these switches and set the idle timeout value and the hard timeout value associated with the flow entry to 0. At the timestamp of event A, when a TCP data packet enters node 6 after the link between nodes 6 and 7 breaks, because there is a flow entry in the flow table that the TCP data packet can match, node 6 forwards the packet out of port 1 to the broken link. Therefore, the destination host cannot receive any packets sent from the source host.

At the timestamp of event F, the source host broadcasts an ARP request before it retransmits the packet lost on the broken link. When the ARP request enters node 6, because there is no flow entry that can match a broadcast packet, node 6 sends a PacketIn message to Ryu. Ryu then sends a PacketOut message to node 6 asking it to flood the packet. As discussed previously, the ARP request will arrive at the destination host and the destination host will reply a unicast ARP reply packet to the source host. When the ARP reply enters node 11, node 11 will forward the packet out of port 2 to node 7 according to the flow entry that it learned before. When the ARP reply enters node 7, node 7 will forward the packet out of port 1 over the broken link. Therefore, the source host cannot receive the ARP reply, which made the source host unable to resend the TCP data packet. At the timestamp of event H, because the link downtime has passed, the source host can receive the ARP reply from the link between nodes 6 and 7 and successfully resend the TCP data packet. Finally, the TCP flow becomes active again over the original path.

When ARP is disabled on hosts, as shown in Figure 2(b), the TCP flow still cannot change to the new path during the link downtime. The reasons and the phenomenon are the same as when ARP is enabled on hosts, except for the events F and G. Because ARP is disabled on hosts, the source host uses the pre-built ARP table instead of broadcasting ARP request packets. Therefore, at the timestamps of events F and G, the source host tried to resend the TCP packet lost on the broken link rather than broadcasting the ARP request to learn the ARP record.

We found that the problem that a TCP flow cannot changes its path to a new path is caused by the improper settings of values of flow idle timeout and flow hard timeout. To fix this design flaw, we suggest to modify Ryu so that an installed flow entry can be expired after some idle period.

C. Case 2 - NOX

The NOX's spanning tree before the link(6, 7) fails is shown in Figure 1. After the link(6, 7) breaks at 40'th sec, NOX rebuilds the spanning tree (to save space, the new spanning tree is not shown in this paper). In the following, we show that (1) when ARP is enabled, the TCP flow will change its path to a new path traversing nodes 3, 6, 9, 10, and 11 at 58'th sec. However, the TCP flow never changes back to the original path after the link between nodes 6 and 7 becomes up again; and (2) when ARP is disabled. The TCP flow never changes its path to the new path when the link between nodes 6 and 7 was down from 40'th to 80'th and it becomes active again over the old path at 112.3'th sec. These results are caused by the settings of the idle timeout and hard timeout values used in flow entries and NOX's LBP and STP implementations.

We use the timeline in Figure 3(a) to display the significant events of a TCP flow when ARP is enabled on hosts. Because we want to observe the related events of changing to the new path, our timeline only focuses on the interval from 40'th sec to 120'th sec. Since the link between nodes 6 and 7 breaks at 40'th sec, as discussed previously, because NOXs STP monitors a link's status every 10 seconds, we expected to see the new spanning tree be built at around 50+'th sec and the TCP flow change to a new path after the new spanning tree is built. However, our simulation result shows that the TCP flow changes its path to the new path at around 59'th sec rather than at 50+'th sec.

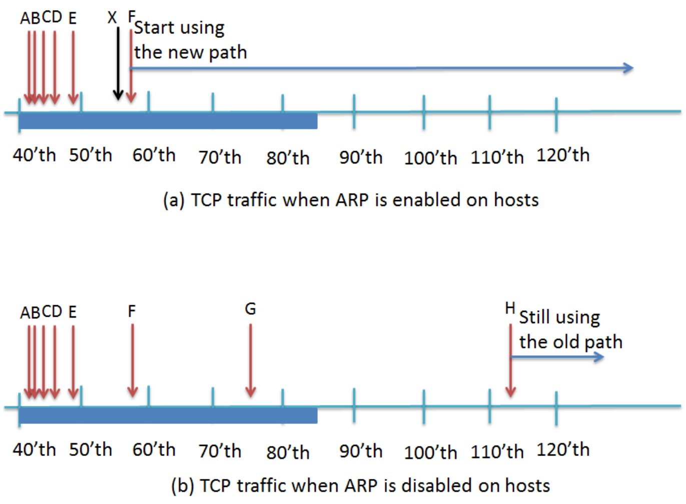

Figure 3. The timeline for a TCP flow to change/keep its path after the link between nodes 6 and 7 breaks at 40'th sec while using NOX

We denote the events A, B, C, D, E, and F as the timestamps when the TCP flow tried to resend a packet due to the TCP reliable transmission. The F event represents the successful retransmission of the lost packet over the new path during the link downtime. The X event represents the ARP request, which is triggered by the TCP flow retransmission at the timestamp of event F. Because the ARP record expires during the long TCP retransmission interval on the source host, the source host has to broadcast the ARP request to the network to learn the ARP record again.

In the following, we explain why the TCP flow changes to the new path at 50+'th sec when ARP is enabled. The reason why the TCP retransmission fails at events A, B, C, D, and E are the same and explained below. When NOX added a flow entry into a switch's flow table, the idle timeout value associated with the entry is set to 5 seconds. According to EstiNet and NOX logs, we found that NOX formed the new spanning tree at 51'th sec, which is after the timestamp of event E. Therefore, all of the resent TCP packets are forwarded over the broken link and got lost before the new spanning tree is formed. After NOX rebuilds the spanning tree, it sends

PortModify messages to switches instructing them to modify the statuses of their ports to FLOOD/NO_FLOOD according to the spanning tree's status.

As for the retransmission at event F, it succeeds and the reason is explained below. Because the interval between the timestamps of events E and F is larger than the value of flow entry's idle timeout, the flow entries on all switches will have expired by event F. Because the ARP record has expired as well on the source host, the source host broadcasts a ARP request at the timestamp of event X to learn the mapping information. After the ARP request enters node 6, because there is no flow entry in the flow table, node 6 sends a PacketIn message to NOX asking for forwarding instructions. NOX then sends a PacketOut message to node 6 asking it flood the ARP request out of all of its FLOOD ports. As a result, the ARP request traverses nodes 6, 9, and 10 to reach the destination host, and the destination host sends back the unicast ARP reply back to the source host.

We found that the ARP request and reply packets play very important roles. They not only let NOX learn the latest forwarding information but also install new and correct flow entries for ARP packets into all switches. Later on, when the resent TCP packet enters node 6, because there are no entries for this TCP flow in its flow table, node 6 sends a PacketIn message to NOX asking for instructions. NOX then sends a FlowModify message and PacketOut message to node 6 asking it to add a new flow entry for this TCP flow and forward the TCP packet out of port 2. After that, the following TCP DATA packets and their ACK packets follow the LBP scenario described before and starts to flow smoothly over the new path at 58'th sec.

In contrast to the fact that the TCP flow can change its path to the new path when ARP is enabled, we found that the TCP flow never changes its path to the new path when ARP is disabled, as shown in Figure 3(b). The reason why the TCP retransmission fails at events A, B, C, D, and E are the same as that described before. At the timestamp of event E, when the TCP DATA packet enters node 6, node 6 still forwards the packet out of port 1 due to the (old) matched flow entry. However, this entry has become incorrect now as the link between nodes 6 and 7 is already broken. At the timestamp of event F, because the interval between the timestamps of events E and F is larger than the value of the flow entry's idle timeout, each flow entry on every switches will have expired by event F. When the TCP DATA packet enters node 6 at event F, because the flow entry for this TCP flow has expired, node 6 sends a PacketIn message to NOX asking for instructions. However, because there were no broadcast ARP packets flooded over the network, NOX has no chance to update and correct its forwarding information. As a result, NOX sends a FlowModify message and PacketOut message to node 6 with incorrect forwarding information and instructs node 6 to forward the TCP packet over the broken link. Clearly, the packet cannot reach the destination host. For the retransmission event G, the same scenario occurs. Finally, at the timestamp of event H, because the link between nodes 6 and 7 has become up again at 80'th sec, the TCP flow uses its old path to successfully retransmit its lost packet and starts to flow smoothly.

Another significant problem we found in Figure 3(a) is that the TCP flow does not change its path from the new path to its original path after the link downtime, even though the spanning tree has been restored to the original one. This problem is caused by the improper settings of the idle timeout and hard timeout, which are set to 5 seconds and infinite, respectively. As long as the TCP flow sends some packets over the new path every 5 seconds, the flow entries in these switches will never expire. Since there are flow entries that can match the incoming TCP packet, these switches need not send PacketIn messages to NOX to ask for forwarding information. Therefore, NOX has no chance to install new flow entries into the flow tables of these switches and the TCP flow still uses the new path without changing back to its original path.

VI. CONCLUSION

In this paper, we used the EstiNet OpenFlow network simulator and emulator to compare the path-finding and packet-forwarding behavior of two widely used OpenFlow controllers — Ryu and NOX. NOX is chosen because it is the world's first OpenFlow controller; Ryu is chosen because it is widely used for cloud orchestration controller applications. Our simulation results show that Ryu lacks the spanning tree protocol implementation and will result in the packet broadcast storm problem when controlling a network with loops. When Ryu controls a network without a loop, after a link failure, we found that a TCP flow cannot change to a new path whether ARP is enabled or disabled. In contrast, we found that NOX enables a TCP flow to change to a new path when ARP is enabled but the TCP flow cannot change to a new path when ARP is disabled. As shown in our studies, the behavior of Ryu and NOX are very different. In the future, we plan to use EstiNet to compare more SDN controllers. Two other SDN controllers that are widely used are POX and Floodlight, each of which has a different implementation for path-finding and packet-forwarding functions. It is interesting to see how these four SDN controllers differ in these important functions.

REFERENCES

[1] ONF, "Software-defined networking: The new norm for networks," Apr 2012, White Paper. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/ sdn-resources/white-papers/wp-sdn-newnorm.pdf

[2] N. Mckeown, T. Anderson, H. BalaKrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, Jennifer, Shenker, Scott, Turner, and Jonathan, "Openflow: enabling innovation in campus networks," ACM SIGCOMM Computer Communication Review, vol. 38, no. 2, 2008, pp. 69–74.

[3] "Open Networking Foundation," 2012, URL: https://www.opennetworking.org/ [accessed: 2014-01-02].

[4] S.-Y. Wang, C.-L. Chou, and C.-M. Yang, "Estinet openflow network simulator and emulator," Communications Magazine, IEEE, vol. 51, no. 9, 2013, pp. 110–117.

[5] "EstiNet 8.0 OpenFlow Network Simulator and Emulator," URL: http://www.estinet.com [accessed: 2014-01-02].

[6] "Ryu OpenFlow Controller," URL: http://osrg.github.io/ryu/ [accessed: 2014-01-02].

[7] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "Nox: towards an operating system for networks," ACM SIGCOMM Computer Communication Review, vol. 38, no. 3, 2008, pp. 105–110.

[8] "Pox: A python-based openflow controller," URL: http://www.noxrepo.org/ [accessed: 2014-01-02].

[9] "Floodlight OpenFlow Controller," URL: http://www.projectfloodlight.org/floodlight/ [accessed: 2014-01-02].

[10] "OpenDaylight Controller," URL: http://www.opendaylight.org/ [accessed: 2014-01-02].

[11] "Link Layer Discovery Protocol," IEEE Standard 802.1 AB, 2009.