Interconnected Multiple Software-Defined Network Domains with Loop Topology

Jen-Wei Hu
National Center for High-performance Computing &
Institute of Computer and Communication Engineering
NARLabs & NCKU
Tainan, Taiwan
hujw@narlabs.org.tw

Chu-Sing Yang
Institute of Computer and Communication Engineering
NCKU
Tainan, Taiwan
csyang@mail.ee.ncku.edu

Te-Lung Liu
National Center for High-performance Computing
NARLabs
Tainan, Taiwan
tlliu@narlabs.org.tw

***Abstract*—With the trends of software-defined networking (SDN) deployment, all network devices rely on a single controller will create a scalability issue. There are several novel approaches proposed in control plane to achieve scalability by dividing the whole networks into multiple SDN domains. However, in order to prevent broadcast storm, it is important to avoid loops in connections with OpenFlow devices or traditional equipments. Therefore, one SDN domain can only have exactly one connection to any other domains, which will cause limitation when deploying SDN networks. Motivated by this problem, we propose a mechanism which is able to work properly even the loops occurred between any two controller domains. Furthermore, this mechanism can also manage link resources more efficiently to improve the transfer performance. Our evaluation shows that the transmissions between hosts from different areas are guaranteed even if the network topology contains loops among multiple SDN domains. Moreover, the proposed mechanism outperforms current method in transferring bandwidth.**

***Keywords-Software-defined networking; OpenFlow; multiple domains; loop topology*.**

I. INTRODUCTION

During the last decades, numbers of innovative protocols are proposed by researchers in network area. However, it is hard to speed up the innovation because network devices are non-programmable. The software defined networking (SDN) approach is a new paradigm that separates the high-level routing decisions (control plane) from the fast packet forwarding (data plane). Making high-speed data plane still resides on network devices while high-level routing decisions are moved to a separate controller, typically an external controller. OpenFlow [1] is the leading protocol in SDN, which is an initiative by a group of people at Stanford University as part of their clean-slate program to redefine the Internet architecture. When an OpenFlow switch receives a packet it has never seen before, for which it has no matching flow entries, it sends this packet to the controller. The controller then makes a decision on how to handle this packet. It can drop the packet, or it can add a flow entry directing the switch on how to forward similar packets in the future.

Moving local control functionalities to remote controllers brings numerous advantages, such as device independency, high flexibility, network programmability, and the possibility of realizing a centralized network view [2]. However, with the number and size of production networks deploying OpenFlow equipments increases, there have been increasing concern about the performance issues, especially scalability [3].

The benchmarks on NOX [7] showed it could only handle 30,000 flow installs per second. However, in [2][4][5][6], authors mention fully physically centralized control is inadequate because relying on a single controller for the entire network might not be feasible. In order to alleviate the load of controller and achieve more scalability, there are several literatures proposed their solutions. DevoFlow [8] which addresses this problem by proposing mechanisms in data plane (e.g., switch) with the objective of reducing the workload towards the controller [6]. In contrast to request reducing in data plane, the other way is to propose a distributed mechanism in control plane. A large-scale network should be divided into multiple SDN domains, where each domain manages a relatively small portion of the whole network, such like that many data centers may be located on different areas for improving network latency. However, if separating to multiple SDN domains, we will lose the consistent centralized control. Currently, there is no protocol for solving this issue [9]. Thus, there are some proposed frameworks [4][5][6] in which create a specific controller to collect information (e.g., states, events, etc.) from multiple domain controllers. They all focus on solving controller scalability issues and facilitating a consistent centralized control among multiple controller domains.

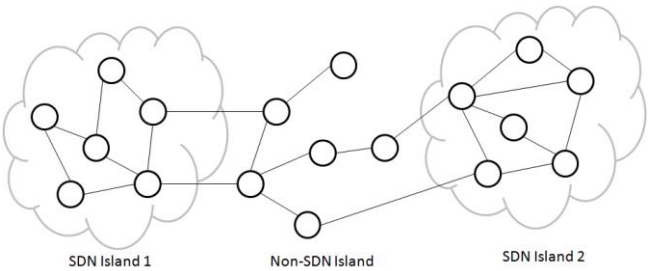

Figure 1. Looped example of a topology with SDN devices and traditional equipments.

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-377-3

For increasing reliability and transmission rate, multiple links are often deployed between nodes in ordinary network design, which practically create loops in topology. The loop leads to broadcast storms as broadcasts are forwarded by switches out of every port, the switches will repeatedly rebroadcast the broadcast packets flooding the network. Since the Layer 2 header does not support a time to live (TTL) value, if a packet is sent into a looped topology, it can loop forever and bring down the entire network. Border gateway protocol (BGP) can handle the loop topology in current Internet, but it is designed based on Layer 3. Thus, BGP still cannot prevent broadcast storm. In order to tackle broadcast storm issue, the spanning tree protocol (STP) is usually used to only allow broadcast packets to be delivered through the STP tree. This design avoids broadcast storm but reduces the overall utilization of the links as a consequence. When in a single SDN domain, controller has the complete knowledge over the entire network topology, thus the tree can be easily built. However, when the SDN network has been split into SDN islands, with the traditional network in between, the controller no longer knows the complete topology, resulting in the inability to build an effective tree, as shown in Figure 1. In this case, the existence of the loop may block the communication between two islands because the broadcast packets transmitted in the topology would mislead the controller to believe that the two hosts in communication are from the same island, thus wrong flow entries are incorporated. The situation will become even more complex in multiple SDN domains. In this paper, we focus on Layer 2 and propose a mechanism which can work properly even the loops occurred between any two SDN domains. Furthermore, this mechanism can also more efficiently in using link resources to improve the transfer performance.

The remainder of the paper is organized as follows. Section 2 presents a brief review of relevant research works focus on solving scalability issue in control plane. In Section 3, we define the preliminaries which will be used in proposed mechanism. Then, we briefly describe proposed mechanism for solving the loop limitation between multiple SDN domains and traditional networks in Section 4. In Section 5, we evaluate our mechanism in a real environment among multiple SDN domains and experiments are reported and compared with the current approach. Finally, the paper is concluded.

II. RELATED WORKS

Onix [5] is a control plane platform, which was designed to enable scalable control applications. It is a distributed instance with several control applications installed on top of control plane. In addition, it implements a general API set to facilitate access the Network Information Base (NIB) data structure from each domain. However, authors mention this platform does not consider the inter-domain network control due to the control logic designer needs to adapt the design again when changed requirements.

HyperFlow [4] is a distributed event-based control plane for OpenFlow. It facilitates cross-controller communication by passively synchronizing network-wide view among OpenFlow controllers. They develop a HyperFlow controller application and use an event propagation system in each controller. Therefore, each HyperFlow controller acts as if it is controlling the whole network. In addition, each HyperFlow controller processes and exchanges these self-defined events and the performance gets poor when the number of controllers grows [4].

HyperFlow and Onix assume that all applications require the network-wide view; hence, they cannot be of much help when it comes to local control applications. In Kandoo [6], authors design and implement a distributed approach to offload control applications over available resources in the network with minimal developer intervention without violating any requirements of control applications. Kandoo creates a two-level hierarchy for control planes. One is local controller which executes local applications as close as possible to switches in order to process frequent requests, and the other is a logically centralized root controller runs non-local control applications. It enables to replicate local controllers on demand and relieve the load on the top layer, which is the only potential bottleneck in terms of scalability.

The above proposals focus on network control, which proactively create the inter-domain links. Thus, these approaches are able to provision cross-domain paths but the complexity of maintaining global proactive flow rules can be minimized. However, loops may form between controller domains and need to be dealt with carefully. To meet this requirement, we develop a mechanism which dynamically forwards packets in the reactive way and solves the loop limitation between multiple controller domains and traditional networks.

III. PRELIMINARIES

We assume the network topology as an undirected graph $G = (V, E)$, where $V = \{\nu_1, \dots, \nu_n\}$ is a finite set, the elements of which are called vertices, and $E = \{e_1, \dots, e_n\} \subset V \times V$ is a finite set, the elements of which are called edges, where each edge e_k can be represented by (ν_i, ν_j), with $\nu_i \neq \nu_j$.

In order to limit the propagation of edge information that is only relevant in certain portions of the network and to improve scalability, we group vertices into structures called areas, denoted as A. Each vertex $\nu \in V$ is assigned a label $L(\nu) = l_k$ that is taken from a set L, describing the area that ν belongs to. Each area has a controller which is charged to coordinate vertices to exchange edge information with its connected areas. As shown in Figure 2, vertices ν_1, ν_2, and ν_3 are on the same area $A_{(0\,1)}$. $C_{(0\,1)}$ is the controller of area $A_{(0\,1)}$. φ is a special label used to represent the area is not belong to any controller domains (e.g., legacy network). That is, area $A_{(\varphi)}$ do not have a controller which can communicate with other areas. There is another special area, called Root Area (RA), which is the root of all areas whose starting item of label match to the label of RA. The controller of RA, called Rooter, collects edge information from its area controllers. Therefore, the Rooter owns a global relationship among its controlled areas.

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-377-3

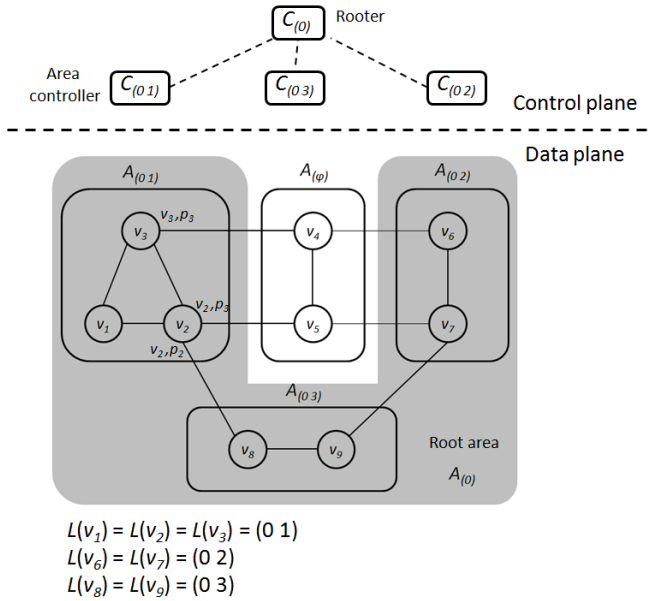

Figure 2. A sample network where vertices have been assigned to areas, represented by rounded boxes.

A vertex $u \in A_{L(u)}$ incident on an edge (u, v), such that $v \notin A_{L(u)}$ is called a border vertex for $A_{L(u)}$, and is in charge of exchanging the edge information to other connected area $A_{L(v)}$where $L(v) \neq L(u)$. If a vertex $u \in A_{L(u)}$ has an edge (u, v), such that $v \in A_{L(u)}$, we call u and v as inner vertices. An inner vertex only exchanges edge messages to other inner vertices in the same area.

There are two different edge information messages. one is exchanged among border vertices in network $G = (V, E)$, called $\Pi =< v, p(v), L(v), \delta >$ where $v \in V$ is the vertex which fires this edge information message; $p(v)$ is the port of vertex v that sends the outgoing edge information; $L(v)$ is the label of v to represent the area that v belongs to; δ (possible empty) is the set of parameters of the port (e.g., priority), which can realize some simple link utilization functions. Each area controller disseminates the other edge message, called $M =< \sigma(v), \sigma(w), \delta >$, to its Rooter. As illustrated in Figure 2, $C_{(0)}$ will receive M from $C_{(0\,1)}$, $C_{(0\,2)}$, and $C_{(0\,3)}$ respectively. There are three fields in M, $\sigma(v)$ is a composition field which includes vertex v, its label $L(v)$, and port $p(v)$; the definition of $\sigma(w)$ is same as $\sigma(v)$ but the vertex $w \in V$ such that $v \neq w$ is the end vertex; δ is the set of parameters of the port.

Each area controller owns full edge information of neighbor areas via all its border vertices. If there exists a port on one border vertex, it receives the edge information of a specific area which is the same as its area controller, we call this port of the vertex is a Representative Port (*RP*) for this neighbor area. As illustrated in Figure 2, (v_2, p_3) and (v_3, p_3) are two *RP*s for $A_{(0\,2)}$ in area $A_{(0\,1)}$.

IV. PROPOSED MECHANISM

In this section, we describe the design philosophy and implementation of our approach. In general SDN network environment, all vertices in the same area will exchange edge information to each other. This is well defined in OpenFlow specification and all popular controllers have already implemented it. However, extended edge information that beyond this area will not be exchanged. In order to compose edge information across different areas, the border vertex which resides in one area will exchange the edge information to its neighbor border vertices that may be directly connected or through one or more area $A_{(\varphi)}$. These operations are formalized in Figure 3. First of all, area controller $C_{L(u)}$ calls sub-procedure to update its data structures according to the received message Π_{new}, such as its neighbor area list and connected edges list for areas. Then, it creates and disseminates edge information that are newly appeared in Π_{new}. After processing all new edge information, $C_{L(u)}$ disseminates edge information that updated the set of parameters. Finally, the procedure updates the list of *RP*, that is used when area controller determines which ports allowed to forward broadcast packets to all connected areas. Note that, in each area controller we have a background process to check the validity of edge information. If it exceeds the timeout $T_{L(u)}(\pi)$, all related information of edge will be removed.

01: **procedure** UpdateBorderVertexEdges (u, Π_{new})
02: BVP is a set stores all border vertices and ports in $C_{L(u)}$
03: A is the set of all connected areas in $C_{L(u)}$
04: $ABVP$ is the map of $area \rightarrow (vertex, port, \delta)$ that stores the information of all discovered border vertices according to different connected areas.
05: BVP^2 is the map of $(u, p(u)) \rightarrow (v, p(v), \delta)$, where u and v are resided in two different controller areas respectively.
06: RP is the map of $area \rightarrow list\ of\ (vertex, port, \delta)$ which represents if sending packets to specific area, we can choose one vertex-port pair in the list.
07: **for each** $\pi_{new} =< v, p(v), L(v), \delta > \in \Pi_{new} \backslash \Pi_{L(u)}$
08: UpdateElements (π_{new})
09: Initialize $T_{L(u)}(\pi_{new})$ to a configured edge timeout
10: Create a new M
11: $M.\sigma(v) = (u, p(u), L(u))$
12: $M.\sigma(w) = (v, p(v), L(v))$; $M.\delta = \delta$
13: Send M to the Rooter
14: UpdateRepresentativePorts $(BVP, ABVP, BVP^2)$
15: **end for**
16: **for each** $\pi =< v, p(v), L(v), \delta_{old} > \in \Pi_{L(u)} \backslash \Pi_{new}$
17: **if** $\exists\ \pi_{new} =< v, p(v), L(v), \delta_{new} > \in \Pi_{new}$
18: π_{new} is an updated instance of π in $\Pi_{L(u)}$
19: $\pi.\delta_{old} = \delta_{new}$; $M.\delta = \delta_{new}$; renew $T_{L(u)}(\pi)$
20: Send M to the Rooter
21: **end if**
22: **end for**
23: **end procedure**
24: **subprocedure** UpdateElements (π)
25: $\Pi_{L(u)} \cup \{\pi\}$
26: **if** $L(v) \notin A$
27: $A \cup \{L(v)\}$

28: **if** $(v, p(v), \delta) \notin BVP^2(u, p(u))$
29: $BVP^2(u, p(u)) \cup \{v, p(v), \delta\}$
30: **if** $(v, p(v), \delta) \notin ABVP(L(v))$
31: $ABVP(L(v)) \cup \{v, p(v), \delta\}$
32: **end subprocedure**
33: **function** UpdateRepresentativePorts ($BVP, ABVP, BVP^2$)
34: **for each** $bvp \in BVP$
35: **for each** $a \in A$
36: **if** $BVP^2(bvp) = ABVP(a)$
37: $RP(a) \cup \{BVP^2(bvp), \delta\}$
38: **end for**
39: **end for**
40: **end function**

Figure 3. Algorithm used for updating the set $\Pi_{L(u)}$ of known edge information in area controller $C_{L(u)}$ when the new edge information Π_{new} arrived at a border vertex u.

As we described in Section 3, each area controller will send edge information M to its Rooter periodically. In our proposal, this Rooter is responsible for providing the network-wide topology. Although our mechanism works well even eliminating the Rooter, we still keep this element for preserving the control flexibility to network management system in the upper layer, such as altering the original flow path and so on. We now illustrate the operations undertaken by Rooter to update its controlled area topology according to received M. These operations are formalized as the procedure UpdateAreaTopology (M_{new}) in Figure 4. First of all, Rooter composes any newly edge information from area controllers. If received M is already existed in edge information of Rooter, this process only updates the new parameter and the timeout of this edge. Last, the Rooter will refresh the area topology according to updated M_{Rooter} and keep this data structure for computing the area path in the future.

Both area $A_{(0\,1)}$ and $A_{(0\,2)}$ in Figure 2 have two border vertices. Take vertex v_3 in area $A_{(0\,1)}$ as an example, it receives edge information from area $A_{(0\,2)}$ by two paths, $(v_7 - v_5 - v_4 - v_3)$ and $(v_6 - v_4 - v_3)$. Similar to vertex v_2, we also can discover two paths. Therefore, four edges are discovered between the area $A_{(0\,1)}$ and $A_{(0\,2)}$ in Rooter. After processing the algorithm of Figure 4, the area topology is delivered, as shown in Figure 5.

01: **procedure** UpdateAreaTopology (M_{new})
02: m is a new or an updated edge from controlled areas
03: **for each** $m =< \sigma(v), \sigma(w), \delta > \in M_{new} \backslash M_{Rooter}$
04: Update M_{Rooter} by m and initialize $T_{Rooter}(m)$
05: **end for**
06: **for each** $m =< \sigma(v), \sigma(w), \delta_{old} > \in M_{Rooter} \backslash M_{new}$
07: **if** $\exists\, m_{new} =< \sigma(v), \sigma(w), \delta_{new} > \in M_{new}$
08: $m.\delta_{old} = \delta_{new}$; renew $T_{Rooter}(m)$
09: **end for**
10: Refresh area topology according to M_{Rooter}
11: **end procedure**

Figure 4. Algorithm to update area topology at Rooter according to the received edge information from controller areas.

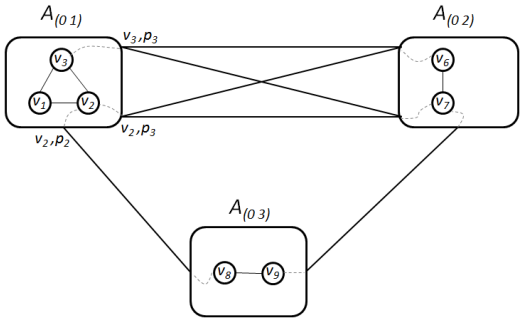

Figure 5. The logical network reduced from the sample network in Figure 2.

We illustrate the operation undertaken by area controller $C_{L(u)}$ when it receives a broadcast packet from any vertex in the controlled domain. These operations are formalized in Figure 6. The input parameters are composed of all connected areas (A), the representative port list generated in Figure 3, and the broadcast packet (*packet*). This procedure goes through the set A and checks how many vertex-port pairs in RP of A. If there contains two or more pairs, it uses the function PickForwardingVertexPorts (RP) to determine the ports of border vertices for forwarding this broadcast packet.

01: **procedure** HandleBroadcastPacket ($A, RP, packet$)
02: $FT(u)$ is the flow tables of vertex u
03: A is the set of all connected areas in $C_{L(u)}$
04: RP is the map of $area \rightarrow list\ of\ (vertex, port, \delta)$
05: MP is the map of $vertex \rightarrow (mac, port)$ in $C_{L(u)}$
06: **for each** $a \in A$
07: **if** size of $RP(a) > 1$
08: $(vertex, port) =$ PickForwardingVertexPorts ($RP(a)$)
09: **else** // only one item in $RP(a)$
10: $(vertex, port) = RP(a)$
11: $mac = packet.src.mac$
12: $FT(vertex) \cup \{mac, port\}$
13: **end if**
14: **end for**
15: **end procedure**
16: **function** PickForwardingVertexPorts (RP)
17: $\varepsilon = \phi$
18: **for each** $rp \in RP$
19: **if** $\varepsilon = \phi$ **or** $rp.\delta.prio \geq \varepsilon.\delta.prio$
20: $\varepsilon \leftarrow rp$
21: **end for**
22: $\varepsilon.\delta.prio = \varepsilon.\delta.prio - 1$
23: **return** ε
24: **end function**

Figure 6. Algorithm used for determining which ports on its all border vertices will be used to forward broadcast packets to other areas.

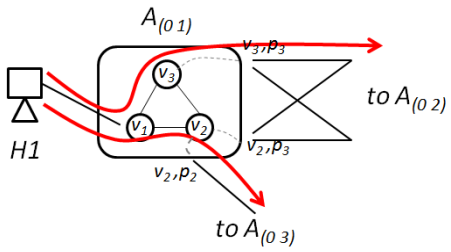

Figure 7. An example of forwarding packets from a host to different areas.

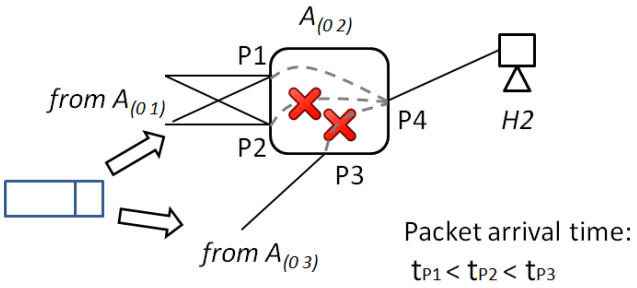

Figure 8. An example of selecting one port in the receiver according to arriving time of the same request packet.

To show an example of process in Figure 6, we consider the area $A_{(0\,1)}$ in Figure 5 and let one host H1 connect to vertex v_1 in this area, as shown in Figure 7. There are three vertex-port pairs on all border vertices in $A_{(0\,1)}$, two of them can reach to area $A_{(0\,2)}$ and the other is for area $A_{(0\,3)}$. Assume that H1 sends a broadcast packet (e.g., an ARP request). In the meantime, the area controller triggers the procedure HandleBroadcastPacket and determines the forwarding ports on its border vertices to transmit the packet to other areas. Note that we only consider *RP*s on all border vertices, if there exists other type of ports, such as access ports (e.g., connecting to hosts) or intra-switch ports (e.g., the port on v_3 connecting to v_2), our mechanism also forwards the broadcast packet to these ports.

In Figure 7, both (v_2, p_3) and (v_3, p_3) are *RP*s for area $A_{(0\,2)}$, according to the algorithm in Figure 6 we will select only one *RP* and forward the broadcast packet. As shown in Figure 7, we choose the port 3 of vertex v_3 to forward the broadcast packet of host H1 to area $A_{(0\,2)}$ while using the port 2 of vertex v_2 to transmit the same packet to area $A_{(0\,3)}$. In this way, we can decrease the number of packets in network to offload area controllers. Moreover, selecting the forwarding edge from one single area can ensure the effective usage of the links without conflicts with other hosts which target the same area. We choose the *RP* to forward packets according to the parameter on the port (e.g., δ). If this port is selected this time, it will adjust the priority to ensure we can choose another ports next time. This priority value will be restored when the flow is released.

In addition, to ensure that the receiver H2 only replies via one of the ports, the arrival time of the request is recorded and used to determine the returning port of H2's reply. In Figure 8, area $A_{(0\,2)}$ receives the same broadcast packet from three different ports, P1, P2, and P3. Assume the arrival times are t_{P1}, t_{P2}, and t_{P3} respectively and t_{P1} is the minimum of them. Thus, host H2 chooses the P1 to send its reply packet.

V. EVALUATION

In this section, we describe the performance evaluation of our mechanism. We simulate physical network connection between TWAREN and Internet2 as our experiment topology. There are 2 physical servers equipped with 64G of RAM and 2 Intel Xeon(R) L5640 CPUs. Each of them runs a Mininet [10] to emulate OpenFlow network topology in TWAREN (e.g., $A_{(0\,1)}$) and Internet2 (e.g., $A_{(0\,2)}$). In addition, we create another domain, called $A_{(0\,3)}$, with 2 physical OpenFlow switches and 1 physical host. There are 4 controllers, one of them represents the Rooter controller and the others install Floodlight (version 0.9) and manage their own areas. The topology of our experiment is shown in Figure 9.

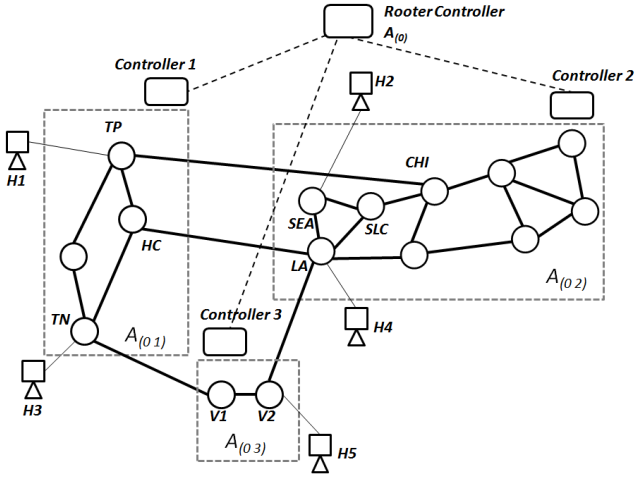

Figure 9. Experiment topology.

As we described in Section 1, controller can handle loop topology in a single domain but not multiple domains. Therefore, we only consider the loops across two or more domains. A loop is represented as a series of edge nodes (e.g., TP-CHI-LA-HC-TP).

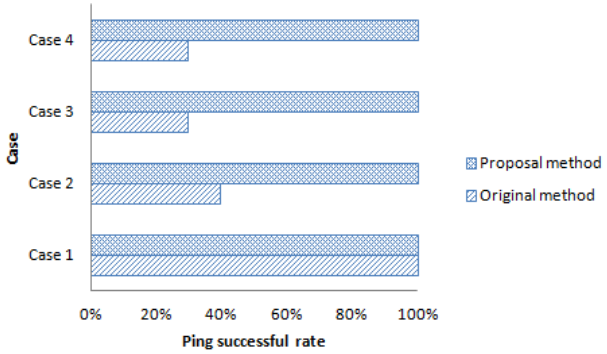

Figure 10. Ping successful rate with four different cases.

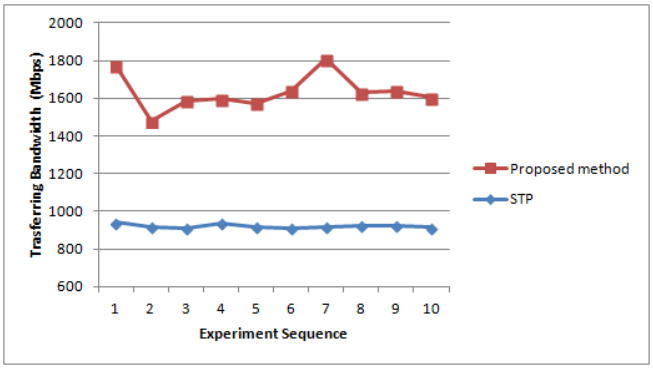

Figure 11. Comparison of transferring bandwidth in two methods when there are two pairs of hosts transferring packets at the same time.

In the first experiment, we evaluate the ping successful rate of our method by comparing with the original forwarding method. There are four cases in this experiment. In each case, we run 10 times on host pairs (e.g., H1-H2, H1-H3, H1-H4, and H1-H5) and compute the ping successful rate. In Case 1, we remove two inter-domain links (e.g., HC-LA and V2-LA) to construct the topology with no loops. As depicted in Figure 10, both our proposed method and the original method have 100% ping successful rate. In the second case, we add the link HC-LA to form one loop (e.g., TP-CHI-LA-HC-TP) between domain $A_{(0\,1)}$ and $A_{(0\,2)}$. We note that there are 2 paths from CHI to LA (e.g., CHI-SEA-LA and CHI-SLC-LA). But in our experiment, we only require that there is at least one path between these 2 edge nodes to assure the connectivity in a single domain. Thus, this is not regarded as a failure. The Case 2 in Figure 10 shows our method reaching 100% ping successful rate while the original method is only 40%. We add link V2-LA in the third case to create the topology with 3 loops (e.g., TP-CHI-LA-HC-TP, HC-LA-V2-V1-TN-HC, and TP-CHI-LA-V2-V1-TN-HC-TP). Our proposed method still reaches 100% but the original forwarding method is lower than 30%, shown as Figure 10. In the final case, we add an extra link between HC and LA. The result is illustrated in Case 4 of Figure 10. This experiment shows that our method is working regardless the number of loops in the topology. We can guarantee any host can successfully communicate with hosts in other areas.

Next, we compare our proposal with STP protocol. We use two pairs of hosts (e.g., H1-H2 and H3-H4) to measure the performance when these hosts transferring packets at the same time. Host H2 and H4 are selected as the iperf servers while the others are the clients of iperf. In order to simulate STP protocol, we remove two inter-domain links (e.g., HC-LA and V2-LA) to build a tree structure in a looped topology. Figure 11 demonstrates the throughput results from STP and our method. We observe our proposed is performing considerably better than STP, offering 77% increasing throughput compared to STP. The reason is STP has only one path between domain $A_{(0\,1)}$ and $A_{(0\,2)}$. Thus, two pairs of hosts share this inter-domain link TP-CHI. But in our proposed method, if there exists another link between these two domains, they will all be used to improve the transfer performance.

VI. CONCLUSION

In this paper, we have proposed a mechanism for solving transmission problem among SDN domains with loops. The proposed algorithms select one port for each connected area to forward broadcast packets. It decreases the number of packets in network to offload area controllers. In addition, the area controller uses our method to filter the repeated broadcast packets at a border vertex and do not forward these packets to avoid broadcast storm. Besides, as compared with original forwarding method, our method can efficiently use multiple edges in loops topology to improve the transferring bandwidth.

Our future work is to improve path compute by defining more granular parameters across multiple SDN domains. In addition, our proposal use the Advanced Message Queuing Protocol (AMQP) to exchange edge information between the Rooter controller and area controllers. We can integrate our approach with other event-based frameworks (e.g., Kandoo) for resources management among multiple domains.

REFERENCES

[1] N. McKeown et al., “Openflow: Enabling Innovation in Campus Networks,” ACM SIGCOMM Computer Communication Review, vol. 38, no. 2, pp. 69-74, Apr. 2008.

[2] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali, “On Scalability of Software-Defined Networking,” IEEE Commun. Mag., vol. 51, no. 2, pp. 136-141, Feb. 2013.

[3] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann, “Logically Centralized? State Distribution Trade-offs in Software Defined Networks,” Proc. ACM Workshop Hot Topics in Software Defined Networks (HotSDN '12), Aug. 2012, pp. 1-6, ISBN: 978-1-4503-1477-0.

[4] A. Tootoonchian and Y. Ganjali, “HyperFlow: A Distributed Control Plane for OpenFlow,” Proc. Internet Network Management Workshop/Workshop on Research on Enterprise Networking (INM/WREN '10), USENIX Association, Apr. 2010, pp. 3-3.

[5] T. Koponen et al., “Onix: A Distributed Control Platform for Large-scale Production Networks, ” Proc. USENIX Symp. on Operating Systems Design and Implementation (OSDI '10), Oct. 2010, pp. 351-364, ISBN: 978-1-931971-79-9.

[6] S. H. Yeganeh and Y. Ganjali, “Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications,” Proc. ACM Workshop Hot Topics in Software Defined Networks (HotSDN '12), Aug. 2012, pp. 19-24, ISBN:978-1-4503-1477-0.

[7] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker, “Applying NOX to the Datacenter,” Proc. ACM Workshop on Hot Topics in Networks (HotNets '09), Oct. 2009, pp. 1-6.

[8] A. R. Curtis et al., “DevoFlow: Scaling Flow Management for High-Performance Networks,” Proc. ACM SIGCOMM 2011 Conference (SIGCOMM '11), Aug. 2011, pp. 254-265, ISBN: 978-1-4503-0797-0.

[9] H. Yin et al., “SDNi: A Message Exchange Protocol for Software Defined Networks (SDNS) across Multiple Domains,” IETF Internet-Draft, draft-yin-sdn-sdni-00, Jun. 2012.

[10] B. Lantz, B. Heller, and N. McKeown, “A network in a Laptop: Rapid Prototyping for Software-Defined Networks,” Proc. ACM Workshop on Hot Topics in Networks (HotNets '10), Oct. 2010, pp. 1-6.

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-377-3