# Fast Failover and Switchover for Link Failures and Congestion in Software Defined Networks

Ying-Dar Lin[1], Hung-Yi Teng[1], Chia-Rong Hsu[1], Chun-Chieh Liao[1] and Yuan-Cheng Lai[2]

[1]Dept. of Computer Science, National Chiao Tung University, Hsinchu, Taiwan
ydlin@cs.nctu.edu.tw, q0226@cs.nctu.edu.tw, cjhsu@cs.nctu.edu.tw, ccliao9615@cs.nctu.edu.tw
[2]Dept. of Information Management, National Taiwan University of Science and Technology, Taipei, Taiwan
laiyc@cs.ntust.edu.tw

***Abstract* —In this paper, we present a fast failover mechanism and a fast switchover mechanism to deal with link failure and congestion problems. In the fast failover mechanism, the controller pre-establishes multiple paths for each source-destination pair in the related OpenFlow-enabled (OF) switches. When a link becomes faulty, OF switches are able to failover the affected flows to another path. Based on the pre-established paths, in the fast switchover mechanism, the controller periodically monitors the status of each port of each OF switch. When the average transmission rate of a port consistently exceeds the rate threshold, the controller would decrease the transmission rate of the port by iteratively switching the flow with the minimum rate to another path. The emulation on Ryu controller and Mininet emulator shows the average recovery time of the fast failover mechanism is less than 40 ms, compared to hundreds of ms in the fast restoration mechanism. And, the fast switchover mechanism can reduce 47.5%-72.5% sustained time of link congestion depending on the parameter setting.**

***Keywords—software defined networking, multiple paths, network resilience, link failure, link congestion.***

## I. INTRODUCTION

In recent years, software-defined networking (SDN) has received increasing attention in both academia and industry. In SDN, the control plane is separated from the network devices and is managed by a logical centralized controller. The controller supports any kinds of control software to manage underlying network devices via an open and standardized application programming interface (API). OpenFlow [1][2] is a well-known example of such an API. An OpenFlow-enabled switch supports one or more flow tables and communicates with the controller through a secure channel. The controller can manage the packet traffic by reactively or proactively installing flow entries in the flow table. In each flow entry, certain instructions are specified to handle the matched packets. Therefore, an OpenFlow-enabled switch can perform different functionalities according to the flow entries installed by applications.

Due to its centralized control paradigm, SDN is being adopted in the various types of networks such as data center [3][4], mobile networks [5][6], transport networks [7][8], and enterprise networks [9]. Network resilience is crucial to SDN networks. In this paper, we focus on link failure and congestion problems. To cope with link failure, there are two typical approaches: *restoration* [10][11] and *protection* [12][13]. In *restoration*, when a switch detects a link failure (port-down event), a notification message is sent to the controller. For each affected flow, the controller then computes another path and writes an alternative flow entry into the related switches. Since the controller need to handle a large number of affected flows simultaneously, this approach would yield a long latency to recover all the affected flows. In *protection*, the controller computes multiple paths for each flow and installs the flow entries into the related switches in advance. In cases of link failure, the switch can directly forward the affected flows to another path without waiting for the response from the controller. But, this approach would yield a large number of rarely-utilized flow entries which slow down the pipeline processing speed of the switches.

In addition to link failure, link congestion is also a challenging problem for network operators. Link congestion occurs when a link is carrying too much data. A typical effect is packet loss which would cause an actual reduction in network throughput. In traditional computer networks, link conditions could be reflected by the link weights of interior gateway protocols (IGPs). Therefore, a heavy-loaded link would have a higher link weight. Each router then can select a light-loaded path to resolve link congestion after exchange of the link weights. However, this approach cannot be effectively handle link congestion since the link weights are exchanged among routers in every 30 seconds. There are very few studies focused on link congestion problem in SDN. In [14], each switch needs to record *pathLoad* information which is the transient link load for each outgoing link. When the *pathLoad* exceeds a predefined threshold, the switch will compute the new sending rate and notify the source to reduce link congestion. In this mechanism, some intelligences built on switches are required.

In this work, we propose a fast failover mechanism and a fast switchover mechanism to deal with link failure and congestion. In the fast failover mechanism, the controller periodically computes multiple paths for all source-destination pairs and proactively installs the flow and group entries on the related switches. When a link failure is detected, the switch can failover the affected flows to the backup path. In this way, the connectivity of hosts could be recovered in a very short time. In the fast switchover mechanism, the controller periodically performs congestion detection for each port of each switch. When a port becomes congested, the controller adaptively decreases the transmission rate by iteratively switching the flow with the minimum rate to the backup path. Therefore, the link congestion could be handled efficiently. We implemented the proposed mechanisms on Ryu controller [15] and evaluated the performance via extensive emulations on Mininet [16]. The results show the average recovery time of the fast failover mechanism is less than 40 ms and the fast switchover mechanism can efficiently reduce the sustained time of link congestion.

The remainder of this paper is organized as follows. Section II presents our fast failover and fast switchover mechanisms. Section III demonstrates our experimental results. Finally, the conclusions are given in Section IV.



978-1-4799-6664-6/16/$31.00 ©2016 IEEE

## II. Fast Failover and Fast Switchover Mechanisms

### A. Network Architecture

Figure 1 depicts our SDN architecture which is typically composed of a logical centralized controller and OpenFlow (OF) switches. The controller is regarded as the "*brain*" of the SDN network which manages the underlying OF switches via OpenFlow protocol. Our mechanisms, fast failover and fast switchover, are implemented as a controller module which runs on the controller. In the fast failover mechanism, the controller periodically acquires global topology information (topology discoverer). According to the topology information, it then computes multiple paths for each source-destination pair and proactively establishes flow entries and group entries in traversal OF switches (path computer). As a result, when the "*port down*" event is detected by OF switches, the OF switches can locally failover the affected flows to another path which greatly reduces the recovery time. Furthermore, the flow setup delay could be reduced significantly by preconfiguring the flow entries into the OF switches.

After the fast failover mechanism is executed, multiple paths are established in the traversal OF switches for each source-destination pair. In order to resolve link congestion, the fast switchover mechanism is designed to adaptively adjust the forwarding path of flows in a centralized manner. In the fast switchover mechanism, the controller periodically monitors the port statistics of underlying OF switches. If the average transmission rate of an egress port is continually higher than a predefined threshold, the controller will decrease the sending rate of the port by iteratively switching the least-loaded flow from the active path to the backup path (flow switchover). Therefore, link congestion could be resolved in an efficient fashion. All the required network states such as topology, hosts, routing information, and port statistics are maintained in an information base. We define a number of RESTful APIs as northbound interface for network management. Network administrators could insert host information and monitor all the network states via the defined RESTful APIs.

Our mechanisms are based on OpenFlow specification 1.3.2. In each OF switch, there are multiple flow tables and a group table. A flow table consists of flow entries. Each flow entry contains (1) "*match fields*" which define the flow, (2) "*counters*" which record flow statistics, and (3) "*instructions*" which define how the packets of the flow should be handled (drop, forward, or goto another table). When a packet arrives at a OF switch, the packet is matched against the flow entries in the flow table. If a flow entry is found, the instructions of the flow entry will be executed. Otherwise, a table-miss flow entry would be matched to process the packet. Typically, the instructions of the table-miss flow would send an OFP_PACKET_IN message to the controller for further processes.

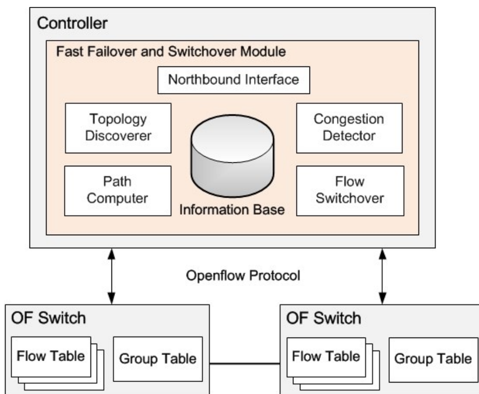

Figure 1. Overview of Network Architecture

The group table enables OF switches to employ additional forwarding methods. The group table also consists of group entries. Each group entry contains a unique group ID, a group type, and a number of action buckets. To execute any specific group entry, the flow entry forwards packets to a group entry having a specific group ID. In our mechanisms, we use an important group type, *fast failover*, to implement our multipath design. The group entries of this type could have multiple action buckets, but only the first alive action bucket is executed. Therefore, when the first action bucket is not alive, the OF switches can locally execute the second action bucket without involving the controller. Another advantage of using group table to implement multiple paths is that the flow table will not be enlarged significantly.

### B. Fast Failover Mechanism

Figure 2 shows the flowchart of the fast failover mechanism. We set a timer (FF timer) to define its execution interval. In our cases, the timer is set to 30 seconds. In the beginning, the controller performs topology discovery to acquire global topology information based on link layer discovery protocol (LLDP). Here, we briefly introduce the procedure of topology discovery. Firstly, the controller sends an OFP_PACKET _OUT message combined with a LLDP packet to each OF switch. Upon receiving the OFP_PACKET_OUT message, each OF switch floods the LLDP packet to its neighboring switches. Once a OF switch receives the LLDP packet from its neighbor, it will send an OFP_PACKET_IN message which the data field contains the received LLDP packet to the controller. According to the OFP_PACKET_IN message, the controller can discover a link between two OF switches. Eventually, the controller can acquire the information of global network topology by this way.

Next, the OF switches attached with hosts are formed into a switch set $N_s$ based on host information. The host information could be obtained manually (via RESTful API) or automatically (using learn bridge). For a OF switch $i$ in the

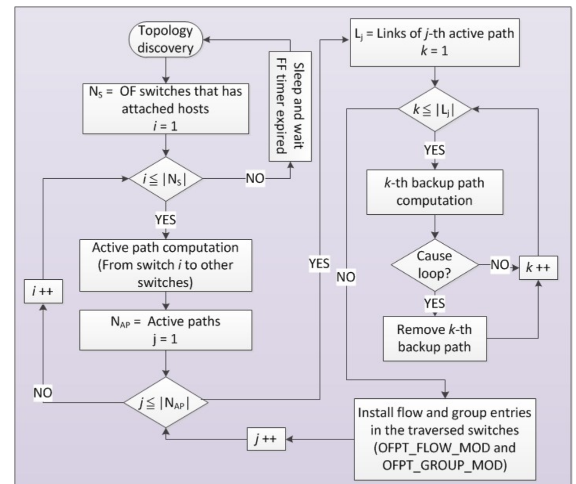

Figure 2. Flowchart of the fast failover mechanism

switch set $N_s$, the controller computes $|N_s| - 1$ active paths (From the selected switch to other OF switches) based on Dijkstra's algorithm. Then, for an active path $j$, the controller also computes $|L_j|$ backup paths considering a failure in each link traversed by the active path. In order to avoid routing loop, the controller checks whether there are path conflicts after computing a backup path. A backup path will be removed if it causes routing loop. Finally, the controller proactively establishes an active path and the backup paths by installing a flow entry and a fast failover group entry in the traversed OF switches for each source-destination pair. Two action buckets are specified in the group entry of the OF switches traversed by an active path. The first bucket and the second bucket indicate the output port of the active path and the backup path, respectively. Please note that the group entry will not be installed if the OF switches already have the same group entry.

Figure 3 illustrates an example of the fast failover procedure. For the sake of clarity, there are only two hosts (Host_1 and Host_2) in this example. For the flows from Host_1 to Host_2, the controller computes an active path <*ABC*>. Then, two backup paths <*AEFC*> and <*ABDEFC*> are also computed for a potential failure in link <*AB*> and link <*BC*>. Finally, the controller establishes these paths by configuring a flow entry and a fast failover group entry in all the OF switches. The group entry has two action buckets at OF switches A and B which enable the packets to be forwarded from Host_1 to the OF switch B, and from the OF switch B to the OF switch C. However, at other OF switches (OF switches C, D, E, and F), the group entry only has one action bucket because only a backup path is configured. Figure 4 shows how the backup path <*ABDEFC*> is used when a failure occurs in the link <*BC*>. When a failure of the output port 3 is detected by the OF switch B, the packets will immediately be forward to the OF switch D though the output port 4.

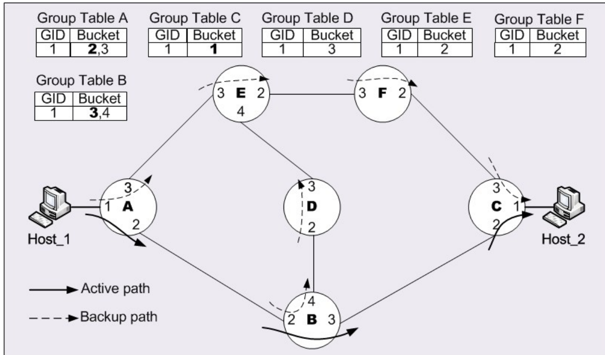

Figure 3. Example of the fast failover mechanism

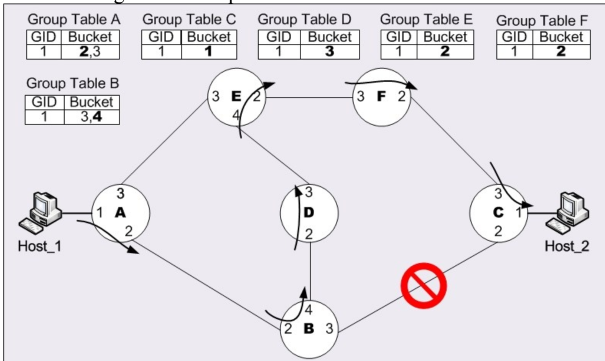

Figure 4. Backup path used upon failure of link <*BC*>

*C. Fast Switchover Mechanism*

Figure 5 shows the flowchart of the fast switchover mechanism. We set another timer, FS timer, to define the execution interval. Since this procedure is required to be executed very frequently, the execution interval is set to 1 or 2 seconds. The fast switchover mechanism consists of two stages: congestion detection and flow switchover. In the congestion detection stage, the first step is to acquire port statistics and group statistics from all the OF switches via the OFPT_MULTIPART_REQUEST and OFPT_MULTIPART _REPLY messages. The second step is to calculate the average transmission rate $R_t$ per port and the congestion window $W_{i,j}$ per port for each OF switch. At first, we compute the transmission rate $r_t$ per port as

$$r_t = \frac{tx_bytes * 8}{max_speed}, \quad (1)$$

where *tx_bytes* and *max_speed* denote the number of transmission bytes and the maximum port bitrate, respectively. We then employ exponential moving average (EMA) to calculate the average transmission rate $R_t$ per port as

$$R_t = \begin{cases} r_t & t = 1 \\ (1-\alpha)R_{t-1} + \alpha r_t & t > 1 \end{cases}, \quad (2)$$

where $\alpha$ denotes the weight of current transmitted rate. For a egress port $j$, the congestion window $W_{i,j}$ is increased by 1 if the average transmission rate $R_t$ is larger than the pre-defined rate threshold $T_r$. Otherwise, the congestion window $W_{i,j}$ is set to 0. Based on the congestion window, the last step is to decide whether the state of the egress port *j* is congestion or normal. The state of the egress port *j* will be set to be congestion only if the congestion window $W_{i,j}$ is also higher than the pre-defined window threshold $T_w$.

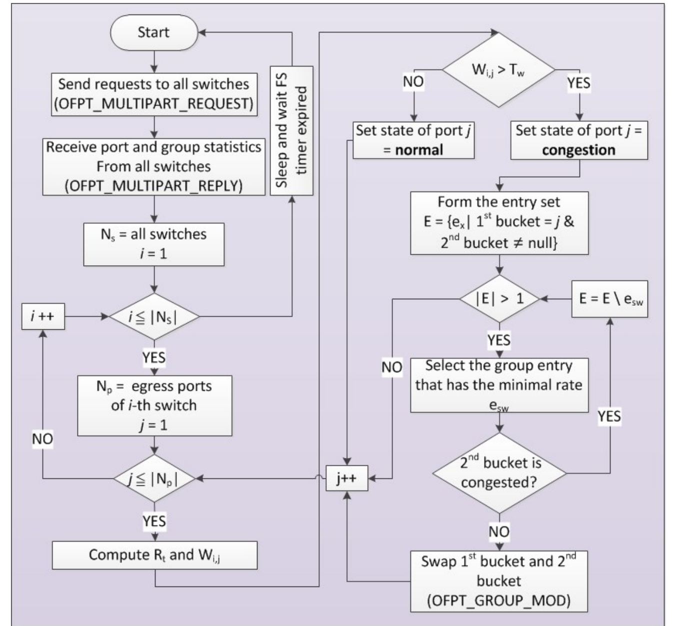

Figure 5. Flowchart of the fast switchover mechanism

Once the state of a port *j* is set to be congestion, the controller will execute the flow switchover stage to decrease the traffic sent by the port *j*. In the flow switchover stage, the first step is to form a set of the available group entries *E* which the first bucket is the port *j* and the second bucket is available from the group table. In order to avoid another congestion, the controller

selects the group entry with the minimum rate $e_{sw}$. Then, it checks the state of the second bucket of the group entry. If the state of the second bucket is normal, the controller will exchange the first bucket and the second bucket via the OFPT_GROUP_MOD message. Otherwise, the controller will remove the group entry and reselect again until there are no available entries.

Figure 6 illustrates an example of the fast switchover procedure. In this example, we assume that the forwarding paths from Host_1 and Host_2 to Host_3 are pre-configured by the fast failover procedure. Thus, at OF switch B, two flow entries and two group entries are configured in advance. The action buckets of the group entries are the same. When the time $t = T$, the controller calculates the average transmission rate per port and the congestion window per port for OF switch B. If it then detects that the congestion window of the egress port 3 exceeds the window threshold ($T_w = 2$), it will set the state of the port as congestion. The controller then executes the flow switchover stage to decrease the traffic sent by port 3. In this case, the controller selects the group entry with GID = 2 as the group entry with the minimum rate and checks the state of the second bucket (output port 4 of OF switch B). Since the state of the port is normal, the controller then swaps the first bucket and the second bucket of the group entry. When the time $t = T+1$, the controller performs the fast switchover procedure once again. The execution procedure is shown in Fig.7. Since the controller reduced the traffic sent by port 3 of OF switch B, the average transmission rate of port 3 is lower than the rate threshold ($T_r = 0.7$). As a result, the congestion window of the port 3 is set to 0 and the state of the port 3 is normal. Therefore, the flow switchover stage is not executed in this time.

## III. Performance Evaluation

### A. Emulation Environment

We investigate the performance of the proposed mechanisms through emulation of real implementation. The proposed mechanisms are implemented on Ryu controller as a Ryu application. In the emulation environment, two servers are employed. The first server (i.e., Intel core i3-2120 CPU 3.30GHz, Ubuntu 12.04 on VMware workstation) acts as the OpenFlow controller, running the Ryu version 3.6. The second

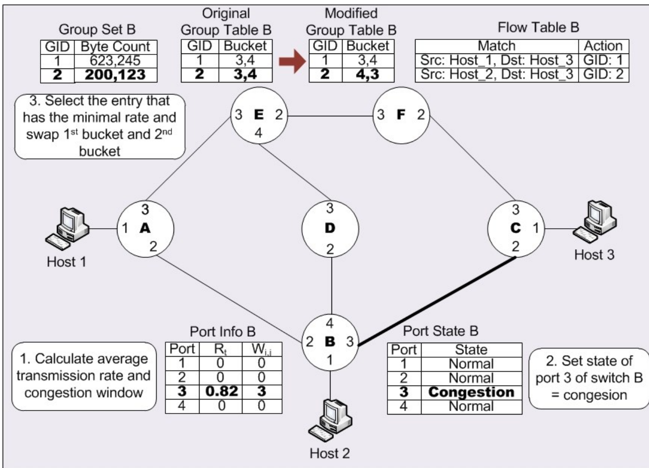


Figure 6. Example of the fast switchover mechanism ($t = T$)

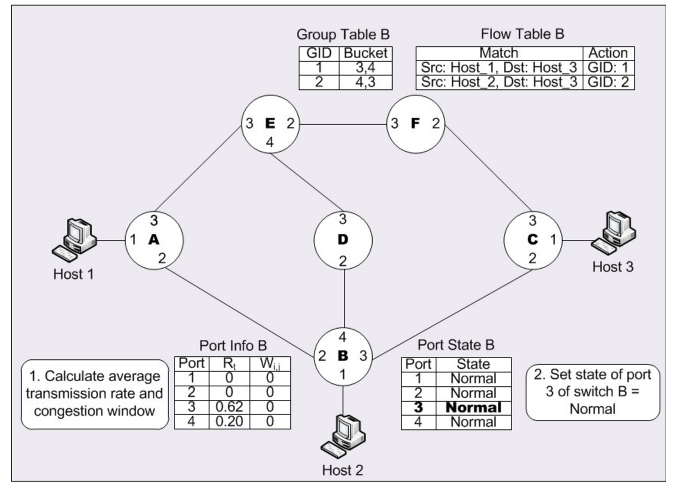


Figure 7. Example of the fast switchover mechanism ($t = T+1$)

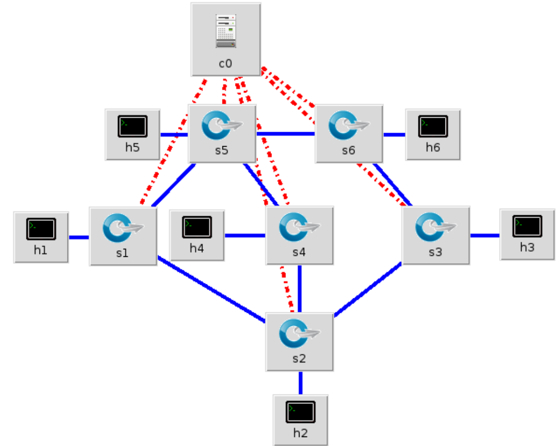


Figure 8. Emulation topology

server (i.e., Intel core i3-2120 CPU 3.30GHz, Ubuntu 12.04 on VMware workstation) emulates the network. As shown in Figure 8, a topology composed of $N$ = 6 OF switches is considered. We use Mininet to emulate the topology and $M$ hosts connected to each OF switch. The ping and iperf tools are used to generate traffic. In the fast failover mechanism, average recovery time, the number of required flow entries, and processing time are adopted as performance metrics. In the fast switchover mechanism, we investigate the impact of different parameter settings on packet loss rate.

### B. Numerical results

**Recovery time of link failure**

Firstly, we demonstrate the average recovery time of the fast failover mechanism. In the emulations, there are multiple hosts connected to the switch s1 (act as source host) and only one host attached to switch s3 (act as destination host). Each source host uses a ping application to generate a packet every 10 ms. The active path is s1-s2-s3. Packets are captured at the destination host using Wireshark [17]. Upon failure on the link s2-s3, the backup path is s1-s2-s4-s5-s6-s3. We compare the performance of the fast failover mechanism with that of the fast restoration mechanism [10]. In the fast restoration mechanism, the controller computes a backup path and installs the flow entries for each affected flows after the link failure occurs. The recovery time is estimated as the time between the reception of

the last packet before the link failure and the reception of the first packet after the link failure at the destination host. The failure on the link s2-s3 is repeated 30 times.

Figure 9 illustrates the average recovery time of the fast failover and the fast restoration mechanisms varying with different number of affected flows. In Fig. 9, the average recovery time of the fast restoration mechanism grows exponentially as the number of affected flows increases. It is because, in the fast restoration mechanism, the controller needs to handle each affected flow when the link failure occurs. Thus, the controller is heavy-loaded and becomes performance bottleneck when the number of affected flows becomes significantly large. When the number of affected flows is 48, the fast restoration mechanism requires 864.16 ms to recover all the affected flows, which is unacceptable for all types of networks. However, since the controller does not involve in the recovery procedure in the fast failover mechanism, the average recovery time only slightly increases and is less than 40 ms as the number of affected flows increases. This result indicates the fast failover mechanism is more effective than the fast restoration mechanism.

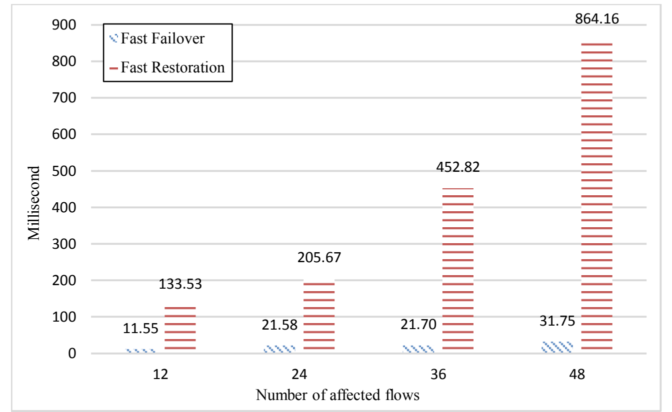

Figure 9. Average recovery time

**Control-plane and data-plane overhead**

Next, we investigate control-plane overhead (controller processing time) and data-plane overhead (the number of required flow entries) of the fast failover mechanism. Here, we consider two kinds of flow granularity: per source-destination host (fine-grained flow entry) and per source-destination subnet (coarse-grained flow entry). In the emulations, the hosts attached to the same switch are regarded as at the same subnet. That is, there are six subnets in the topology. The controller processing time comprises of three parts: active path computation, backup path computation, and entry insertion.

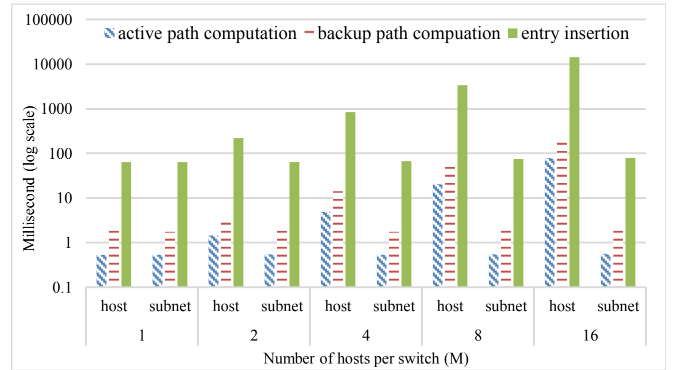

Figure 10. Controller processing time

Figures 10 and 11 show the controller processing time and the average number of flow entries of the fast failover mechanism with various number of hosts per switch $M$. In Fig. 10, as the number of hosts per switch $M$ increases, the entry insertion time also increases exponentially because Ryu applications are single-threaded entities. Moreover, we also observe that the path computation time increases significantly when the fine-grained flow entry is adopted. It is because the fast failover mechanism needs to access host information frequently in both active path computation and backup path computation. In Fig. 11, when the fine-grained flow entry is adopted, we can clearly observe that the average number of flow entries increases exponentially when the number of hosts per switch $M$ increases. In cases of $M$=16 hosts connected to each OF switch, the controller would maintain 6,416 flow entries on each OF switch. However, when the coarse-grained flow entry is adopted, the controller processing time is lower than 100 ms and the average number of flow entries is less than 50 no matter how many hosts are connected to each OF switch. Clearly, the fine-grained flow entry is not suitable to the fast failover mechanism since each OF switch has a limited TCAM and the controller should keep as light-loaded as possible.

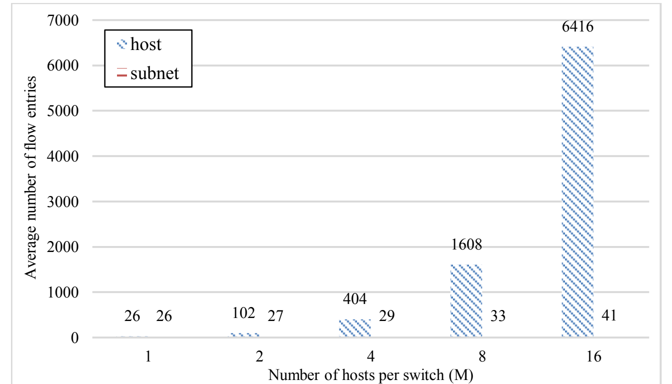

Figure 11. Average number of flow entries per switch

**Sustained time of link congestion**

Lastly, we demonstrate the performance of the fast switchover mechanism. In the emulations, there are three hosts, *h1*, *h2*, and *h3*, connected to the switch *s1* (act as source host) and only one host, *h4*, attached to switch *s3* (act as destination host). In order to simulate link congestion, each source host uses iperf tool to generate a flow with different configurations. The configurations of the three flows is shown in Table 1. From the hosts, *h1*, *h2*, and *h3*, to the host, *h4*, the active path and the backup path are *s1-s2-s3* and *s1-s2-s4-s5-s6-s3*, respectively. We set the capacity of the link s2-s3 to 600 Mb/s. Since the transmitted data rate exceeds the link capacity, we expect the link *s2-s3* would become congested from the 20th sec to the 40th sec. Here, we also investigate the performance of the fast switchover mechanism with various configurations of the execution interval and the congestion window (denoted by c_window). Four different cases: 1: [interval=1, c_window=2], 2: [interval=1, c_window=3], 3: [interval=2, c_window=2], and 4: [interval=2, c_window=3] are considered. In the emulations, we set the rate threshold $T_r$ and the weight α to 0.7 and 0.1, respectively.

Table 1. Flow Setting

| | Data rate | Start time | Duration |
|---|---|---|---|
| flow 1 | 128 Mb/s | $0^{th}$ sec | 30 sec |
| flow 2 | 256 Mb/s | $10^{th}$ sec | 30 sec |
| flow 3 | 384 Mb/s | $20^{th}$ sec | 30 sec |

Figure 12 shows the packet loss rate of the host h4 varying with five cases. In the no fast switchover case, the host h4 has high packet loss rate which sustains about 20 seconds (the sustained time of link congestion) due to the congested link s2-s3. When the fast switchover mechanism is employed, we can observe that the sustained time of link congestion is reduced significantly. In Fig. 12, the sustained time of link congestion of the four cases is 5.5 seconds, 6.5 seconds, 6.5 seconds, and 10.5 seconds, respectively. It is because the fast switchover mechanism can periodically switchover the least-loaded flow to the backup path when the link congestion is detected. Moreover, when the execution interval and the congestion window is set to be smaller, the fast switchover mechanism can resolve link congestion more quickly. However, this way would significantly increase the burden of the controller.

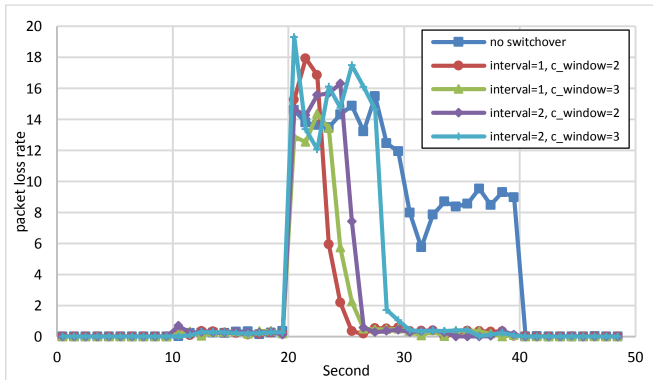

Figure 12. Packet loss rate of the host *h4*

## IV. CONCLUSION

In this work, we present a fast failover mechanism and a fast switchover mechanism to deal with link failure and link congestion in SDN networks. In the fast failover mechanism, the controller periodically computes multiple paths for each source-destination pair and proactively installs flow entries and group entries in related OF switches. When a link becomes faulty, the switch is able to failover the affected flows to the backup path. In the fast switchover mechanism, the controller periodically monitors the average transmitted rate of each port for each OF switch. When the average transmitted rate of a port consistently exceeds the rate threshold, the controller would decrease the transmitted rate of the port by iteratively switching the flow with the minimum rate from the active path to the backup path. The emulation results show the average recovery time of the fast failover mechanism is significantly lower than that of the fast restoration mechanism. Specifically, the average recovery time of the fast failover mechanism is less than 40 ms, compared to hundreds of ms in the fast restoration mechanism. Considering OF switches have a limited TCAM and the controller should keep as light-loaded as possible, the fast failover mechanism should employ coarse-grained flow entries to establish multiple paths. Our emulation results also demonstrate the fast switchover mechanism can reduce 47.5%-72.5% sustained time of link congestion depending on the parameter setting.

## ACKNOWLEDGMENT

The authors would like to thank the Ministry of Science and Technology, Taiwan for financially supporting this research under Contract No. MOST 103－2622－E－009－012 and MOST 103－2221－E－009－100－MY2.

## REFERENCES

[1] N. Mckeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," ACM SIGCOMM Computer Communication Review, vol. 38 no. 2, pp. 69-74, April, 2008.
[2] ONF, "OpenFlow Switch Specification version 1.3.2," April 25, 2013.
[3] F. Hao, T. V. Lakshman, S. Mukherjee, and H. Song, "Enhancing dynamic cloud-based services using network virtualization," ACM SIGCOMM Computer Communication Review, vol. 40, no. 1, pp. 67-74, Jan. 2010.
[4] V. Mann, A. Vishnoi, K. Kannan, and S. Kalyanaraman, "CrossRoads: Seamless VM Mobility Across Data Centers Through Software Defined Networking," in IEEE Network Operations and Management Symposium (NOMS), pp. 88-96, 2012.
[5] Kok-Kiong Yap, Masayoshi Kobayashi, Rob Sherwood, Te-Yuan Huang,Michael Chan, Nikhil Handigol, and Nick McKeown, "Openroads: empowering research in mobile networks," ACM SIGCOMM Computer Communication Review, vol. 40, no.1, pp. 125-126, 2010.
[6] Y. Wang, and W. Hu K. Pentikousis, "MobileFlow: Toward Software-Defined Mobile Networks," IEEE Communication Magazine, vol. 51, no. 7, pp. 44-53, July 2013.
[7] A. Giorgetti, F. Cugini, F. Paolucci, and P. Castoldi, "Open-Flow and PCE architectures in wavelength switched optical networks," in 16th Int. Conf. on Optical Network Design and Modeling (ONDM), pp. 1-6, Apr. 2012.
[8] J. Kempf, E. Bellagamba, A. Kern, D. Jocha, A. Takacs, and P. Skoldstrom, "Scalable fault management for OpenFlow," in IEEE Int. Conf. on Communications (ICC), pp. 6606-6610, June 2011.
[9] H. Kim et al., "Communicating with Caps: Managing Usage Caps in Home Networks," in Proc. ACM SIGCOMM' 11, pp. 470-71, 2011.
[10] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "Enabling fast failure recovery in OpenFlow networks," in 8th Int. Workshop on the Design of Reliable Communication Networks (DRCN), pp. 164-171, Oct. 2011.
[11] D. Staessens, S. Sharma, D. Colle, M. Pickaver, P. Demeester, "Software defined networking: Meeting carrier grade requirements," in Proc. of the 18th IEEE Workshop on Local and Metropolitan Area Networks, pp. 1-6, Oct. 2011.
[12] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "OpenFlow: Meeting carrier-grade recovery requirements," Computer Communication, vol. 36, no. 6, pp. 656-665, Mar. 2013.
[13] A. Sgambelluri, A. Giorgetti, F. Cugini, F. Paolucci, and P. Castoldi, "Openflow-Based Segment Protection in Ethernet Networks," Journal of Optical Communications and Networking, vol. 5, no. 9, pp. 1066-1075, Sep. 2013.
[14] S. Fang, Y. Yu, C. H. Foh, and K. M. M. Aung, "A Loss-free Multipathing Solution for Data Center Network using Software-defined Networking Appraoch," IEEE Transactions on Magnetics, vol. 49, no. 6, pp. 1-8, June 2013.
[15] RYU: http://osrg.github.io/ryu/
[16] Mininet: http://mininet.org/
[17] Wireshark: http://www.wireshark.org/