Enhanced Failure Recovery Mechanism using OpenState Pipeline In SDN

Abdullah Soliman Alshra'a, Parag Sewalkar, and Jochen Seitz
Communication Networks Group, Technische Universität Ilmenau, Germany
Email: {Abdullah.Alshraa, Parag-Vishwas.Sewalkar, Jochen.Seitz}@tu-ilmenau.de

***Abstract*—In Software Defined Networking (SDN), link failure situations require switches to make optimized rerouting decisions locally i.e. without controller intervention. However, such optimized rerouting decisions must be made after the link failure situation has been confirmed. The mechanisms that confirm such link failure situations require a certain amount of processing time which may introduce a bottleneck in the performance. This paper proposes an enhanced approach that employs a mechanism to make the re-routing decision without the confirmation of link failure. The approach uses the OpenState Pipeline to manage the network in failure situations, and the experiment results show notable enhancements in packets loss rate and usage of Ternary Content Addressable Memory (TCAM).**

***Index Terms*—Software Defined Network, OpenState Pipeline, Fault Tolerance, Ternary Content Addressable Memory, TCAM, SDN**

I. Introduction

Various types of connected mobile devices have been launched in the last few years, which have led to an increased demand for connected applications. These connected applications have various dynamically changing Quality-of-Service (QoS) and performance requirements. Networks must employ mechanisms that can respond to such dynamic requirements. The SDN technique has been introduced to allow networks to be flexible and adaptable to the dynamic requirements of connected applications. It introduces a new layer that separates the control plane and data plane functions in the network. SDN enables the control plane to be programmable and turns it into a framework for applications and network services [1].

In the SDN architecture, the control plane (controller) is responsible for making the routing decisions by installing the forwarding rules and packet processing instructions, collectively known as flows, for the data plane (switch). Thereafter, the switch uses the flow table to determine the packet processing instructions and the next switch for forwarding the packet. However, if the switch does not find any appropriate entry in the flow table, it sends a query to the controller which then defines and updates the flow entry in the switch accordingly [2]. In fact, the query could need a long time (200 ms) to update the flow entry [3]. OpenFlow has been proposed as SDN standard that defines the communication protocol between the controller and the switch [1].

As the SDN architecture makes networks more dynamic, quick recovery from link failures is an essential aspect of the SDN architecture. It must recognize the link failure quickly and establish an alternate path. In carrier-grade networks, the failure recovery must be achieved within 50ms [4]. The recovery can be achieved either in a centralized way (i.e. controller to track switches) or decentralized way (i.e. switches to track the other switches) [5, 6]. In a centralized way, the switches send "HELLO" packets periodically and the controller tracks the switches for their liveness using the ports to these switches [7]. The controller is also responsible for establishing an alternate path. This mechanism increases the load on the network as well as the controller. Also, the controller may not always be reachable which poses another challenge. In a decentralized way, reliance on the remote controller is minimized and link failure situations are recovered locally as much as possible [18]. OpenState Pipeline or SPIDER pipeline [5, 6] are examples of such efforts. [5, 6] propose mechanisms to add more processing locally (at the switch) and reduce the dependence on the controller. These mechanisms implement a Finite State Machine (FSM) inside the switches in order to track the states of other switches and to recognize and recover from link failures. Decentralized mechanisms overcome the limitations of centralized mechanisms and, hence, may be more appealing to the researchers working on failure recognition/recovery mechanisms.

However, all the decentralized mechanisms use an alternate path (for recovery) *after* they ensure that link failure has *actually* occurred. In order to ensure the link failure situation, switches have to access their flow tables. As these flow tables are stored in TCAM, unoptimized use of flow tables require higher TCAM capacity. TCAM is known to be expensive and power hungry and hence their usage must be optimized in order to keep the equipment and operating costs low [8, 9]. Also, due to the delay between link failure and link recovery, the network may suffer from packet loss. A link recovery mechanism must handle the packets flowing over the broken link appropriately e.g. reject the packets over the broken link [10]. This may be necessary to minimize the packet loss.

In this paper, we propose a decentralized mechanism that is based on the OpenState Pipeline. Our approach aims to keep up the packet rate and to optimize the use of TCAM size by preemptively changing the data flow traffic. We achieve this by considering the link failure probability. The remainder of the paper is organized as follow: Section II discusses related work. Section III presents our approach. Section IV contains the simulations and validation of our approach. Section V provides the direction of our future work and conclusions.

II. Related Work

Various efforts have been made to propose recovery mechanisms that are based on controller's different roles, the timing of the establishment of alternate paths, and approaches for packet loss minimization [6, 10–15].

In a centralized mechanism for link failure recovery, the controller is required to perform tasks of polling switches and providing alternate paths. However, this creates a significant load on the controller. Efforts by Adrichem et al. [11] show that the increase in load on the controller affects the recovery time which cannot be achieved within 50ms. In order to reduce the dependence on the controller, a Fast-Failover group presents an approach to solve the recovery problem locally. They achieve this by adding an alternative path into the switch's forwarding table. However, if the alternative path is not available, then the switch asks the controller for an alternate path. The OpenState Pipeline enables failure detection and recovery at switches instead of the controller. Efforts by [6, 12] use the OpenState Pipeline approach to propose a recovery mechanism. Mohd Zahid and others [16] explain that Openstate recovery time is better than Openflow by estimating recovery times of Openflow and Openstate in multiple failure situations. In [5], a comprehensive approach has been introduced using the Openstate Pipeline and obtained obvious results show the advantage to use Openstate. However, their mechanisms relies on checking the reachability of the next node (after link failure) before the alternate path can be used. This leads to the loss of data before discovering the failure situation and hence is not an effective solution.

Few research efforts explore the timing of establishment of alternate paths to propose the recovery mechanism [10, 13, 14]. An alternate path may be established before the link failure (*protraction*) or after the actual link failure (*restoration*) occurs. In the *restoration* approach, an alternate path is established after the failure occurs. This establishment requires communication with the controller. This communication requires additional signaling and hence increases the recovery time. Efforts by [13, 14] show the *restoration* approach cannot establish an alternate path within 50ms.

The *protraction* approach has been introduced to decrease the shortcomings of the restoration approach. In this approach, alternative paths are supplied before the link failure occurs in the network (at boot time). This eliminates the need of additional signaling for controller intervention and speeds up the recovery process. The network can recover from the link failure within 50ms with the *protraction* approach [13, 14].

Sgambelluri et al. [10] present a *protraction* approach for the establishment of alternate paths. This approach installs the alternate paths into switches with lower priority (compared to primary paths). It also requires the switches to send periodic signaling packets to refresh the timers. Upon expiration of these timers, link entries corresponding to the switches may be deleted. This mechanism also proposes an auto-reject mechanism to avoid the loss of packets. Zhang and others in [17] investigated the single link failure recovery problem in SDN in order to minimize the flow entries by determining the importance level of a link, thus choosing the backup path nodes that help to minimize the average number of the flow entries in the whole network. However, upon link failure, the *protraction* approach waits for the confirmation of the failure. This waiting process causes the loss of packets. Also, switches increase the load on the network due to their signaling packets [18][16].

Ahmed et al. [15] use the Bidirectional Forwarding Detection protocol (BFD) to examine the communication between two nodes. In case a switch receives a packet and the primary path fails, the switch forwards the packet back in a recursive way to the previous switch until the packet finds an alternative valid path towards the destination. Once the packet reaches the destination, the switch, which has detected the failure, informs the controller to reconfigure the network. This mechanism suffers from the drawbacks of the centralized mechanism i.e. load on controller and network.

Efforts by Adrichem et al. [11] propose an approach that uses per-link BFD for failure detection. This decreases the Round-Trip Time (RTT) of link monitoring and also does not cause a significant overhead. However, it requires more TCAM due to its per-link monitoring mechanism. Also, due to its centralized nature, it suffers from its disadvantages i.e. controller reachability and load on the controller.

III. Research concept and the proposed failure recovery mechanism

We propose an approach that uses a decentralized way for link failure and recovery (re-routing). This reduces the dependence on the controller and also reduces the latency in recovery. We use the legacy OpenState Pipeline and Multiprotocol Label Switching (MPLS) for our design inputs [19]. Our approach aims to minimize the packet loss and optimize the use of TCAM. We aim to achieve this by optimizing the flow tables, which in turn requires optimizing their forwarding behaviors for various events, such as, link failure and increased load. Various stages in our approach are as described below.

A. Boot Time

At the boot time, the controller supplies switches with a set of primary paths and backup paths. The backup paths avoid using all links towards the next nodes in case the failure occurs.

B. Failure Detection

Our approach uses Multiprotocol Label Switching (MPLS) to tag the flow packets in order to distinguish between the different purposes of the traffic flows. Also, we assume that the OpenFlow protocol can use a port as long as there is an incoming packet from the port during a given interval time. The reverse traffic helps the node to refresh its states to the zero-point, as a consequence, avoiding producing probe packets to check the communication to the next switch. However, our approach uses the reverse traffic to reduces the overhead to null or low signals and gets rid of the heartbeat procedure as well

as there is no overhead from heartbeats packets. Otherwise, it sends a "*probe*" packets to check the port and use the backup path to reach the packet to destination directly. If the probe reply does not come, the port is declared as DOWN.

C. Fast Reroute

When the port is declared DOWN, the switch uses the backup path or forwards the packet back to the previous node (using the same port where the packet was received) until finding a detour (backup path).

D. Path Probing

Our approach considers that the failure is temporary. Therefore, it sends another copy to check the situation (down port or node) periodically.

E. Flowlet-aware Failover

When the packets are forwarded to the node having a backup path, the packets may arrive out of order. This requires our approach to use a Flowlet-aware forwarding scheme [20] to solve this problem. This Flowlet-aware scheme requires the packets to be forwarded on the primary path (till the failure is detected) until a given idle timeout expires. Thereafter, the packets are sent over the backup path.

F. Flow Tables

In order to achieve the aforementioned processing, we use four tables as described below.

Table 1
Table 1 performs stateless operations. It processes packets based on the source. The connected host packets are processed to add the MPLS label that would be used for storing the tag. The connected switch packets are processed to read the input port. It then writes the input port in the meta-data field.

Table 2
Table 2 performs stateless operations. It manages the packets that are coming from Table 1. For connected host packets, it may add tag = 0 if the switch is not the last switch in the path, otherwise it removes the MPLS label for the last switch. It also passes the fault tag packet to the next switch in the backup path and gets the tag back to the normal tag if the switch acts as last node in the backup path. Table 2 sends the probe packets to the next switch in the primary path through the output port. It also drops the reply to probe packets. Finally, Table 2 updates the time and the status in Table 3 if it receives a probe packet. It also updates the time and the states in Table 4 if it receives any packet. The Tables 3 and 4 consist of a set of state values that are changed depending on the given time, or a certain action. SPIDER [5] approach propose to use 7 such intervals (timers D1 - D7) for various purposes. Using D7 in SPIDER to check the link situation leads to lost packet as well as it requires more TCAM utilization, hence installing more entries[5]. In our approach, we define 6 such intervals and achieve more efficiency by eliminating the interval D7. Our interval definitions are as following.

- **D1:** Idle state timeout that defines the duration *before* the flow starts using the backup path instead of the primary path. D1 is equal to the highest RTT on the bounce path for a specific request and fault.
- **D2:** Hard state timeout that defines the *longest* duration to accept using the primary path *before* changing the flow to the backup. D2 is always greater than D1.
- **D3:** Idle state timeout that defines the maximum time duration between the use of backup and primary paths. D3 is the idle time used to define the duration before the flow start using the primary instead of the backup path.
- **D4** Hard state timeout defines the *longest* duration to accept using the backup path before changing the flow to the primary. D4 is always greater than D3.
- **D5** Hard state timeout that defines an arbitrary duration to send probe packets to a certain point.
- **D6** Hard state timeout that defines the duration for packet reply before considering the link is broken. D6 is equal the Maximum RTT between two specific nodes.

Table 3
Table 3 performs stateful operations. It is used to save the order of the packets. When it receives the packets with the fault tag, it changes fault signal states for D1 or D2 (the shortest). These states aim to save the order of packets so that the packets with fault tag are sent through the backup path and the normal packets are sent to the primary path. When the fault signal state finishes, Table 3 changes to Detour Enabled state in order to send all the packets through the backup path. Every D5, Table 3 state becomes a Need Probe state that sends a probe packet in order to check the primary path. Then, the state reverts back to Detour Enabled state. However, Table 3 transitions to Fault Resolved state if it receives a reply to the probe packet from the second table. Fault Resolved state then sends any incoming packets through the backup path to save the packet order for D3 or D4 (the shortest), thereby returning Table 3 to the normal state.

Table 4
Table 4 performs stateful operations. It manages the validation of the communication between the adjacent switches. It assumes that the communication is available as long as there is incoming traffic from the adjacent switch. When there is no incoming traffic for D6, Table 4 transitions to the state Need probe instead of Up:wait state. In this state, it duplicates an incoming packet. It sends one of the duplicated packets with a probe tag to the next switch. It then sends the second packet with a fault tag to the backup path. The goal of this procedure is to confirm the link availability to the next switch and save the packets. After that, Table 4 state changes to Down Probe and waits for the interval D5. After the interval D5 is over, Table 4 considers the link is unavailable and changes its state to Down Need Probe state.

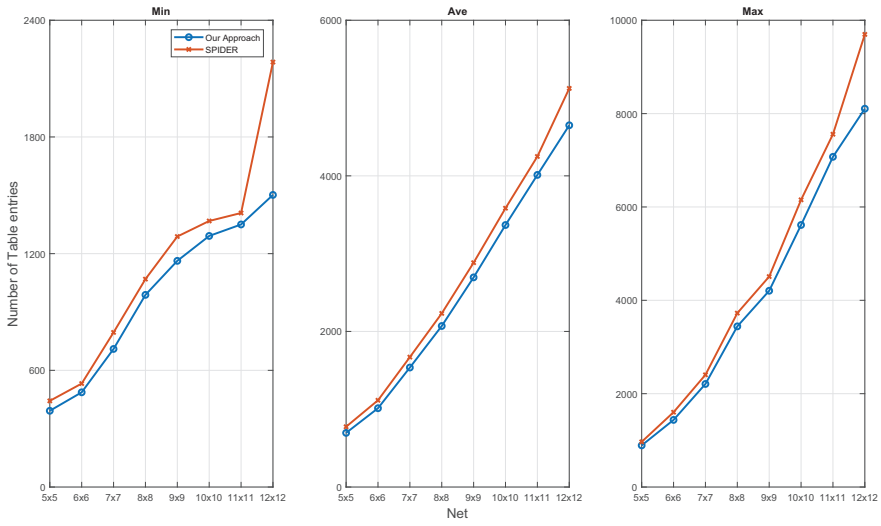

Fig. 1. TCAM utilization

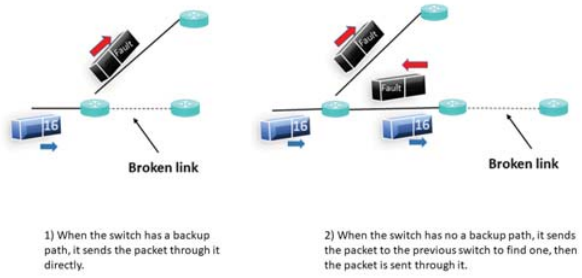

Fig. 2. Search backup paths

In this state, it repeatedly duplicates all incoming packets. It then marks one copy of the packet with the Probe tag and sends it through the primary path. It marks the second copy with the fault tag and sends it through the backup path. However, in case Table 1 receives any packet from the destination switch, the state returns to Up:wait state. In case of a second or further failure, our approach calls for the controller intervention to reconfigure the network topology. Moreover, in case there is no backup path in the switch that detects the failure, the switch tags and forwards the packets back to the previous switch. Consequently, the previous switch knows that the tagged packets need to be forwarded on a different path. Therefore, the previous switch re-forwards the packets on its backup path. Otherwise, the packets are sent back again to the previous switch over the primary path. Fig. 2 depicts an example of this mechanism.

Initially, the packets are forwarded through the primary path. But if there is no incoming traffic during D6 interval, the switch assumes the possibility of link failure. It then begins to duplicate the packets on the primary and the backup paths.

If communication to the next switch is possible (no fault), the next switch receives the packet with Probe tag. It then duplicates this packet and changes its tag value to 16 (16 refers to the normal situation). This duplicated packet (with tag value 16) is forwarded to the next switch on the primary path. The original packet with tag value Probe is sent to the previous switch in order to inform it that the communication is available and there is no fault. The switch then changes its state to normal operation and refreshes the timers. On the other hand, if the switch does not send the Probe packet to the previous switch, it assumes that the link is broken and uses the backup path. The switch sends a Probe packet on the primary path every D6 to check if the fault is repaired.

IV. Experimental Implementation and Performance Evaluation

We implemented and validated our approach using the Mininet emulator. Mininet provides an implementation of the OpenState pipeline. This encompasses an OpenFlow Ryu controller and CPqD openFlow softswitch [21, 22].

We use number of TCAM entries and lost packets as parameters to evaluate the performance of our solution.

A. TCAM

Controller sends a set of instructions as part of forwarding table entries to switches which are stored inside the switch's TCAM. As our aim is to minimize the TCAM utilization,

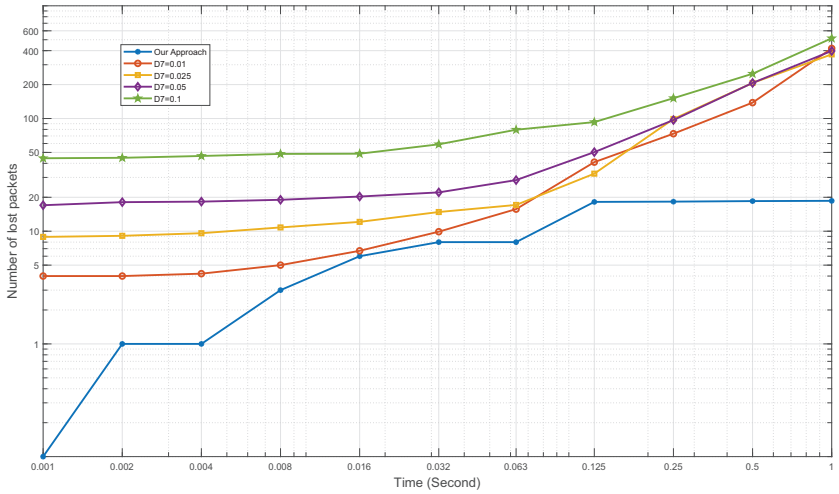

Fig. 3. Packet Loss

we measure the number of entries created in the TCAM. We evaluate our approach by comparing our TCAM utilization with that of SPIDER. Fig. 1 shows the comparison of SPIDER and our approach in terms of the summations of the entry number inside grid networks (N x N), where each edges node has one host and the hosts have to communicate to each other in the network. Fig. 1 shows three different metrics of TCAM entries namely Min, Max, & Average. Min shows the minimum number of TCAM entries among the network switches. The Max line signifies the maximum number of TCAM entries and the average shows the average number of TCAM entries in the switches across the whole networks. Fig. 1 shows that our approach performs better in terms of TCAM entries when compared to the SPIDER approach.

B. Packet Loss

We measure the performance of our approach in terms of lost packets by comparing it to the SPIDER approach. We send 20000 ping packets in one direction from the source to the destination 20 times. We also consider a link with traffic of 1000 packets/sec between two switches and toggle its status between ON & OFF. We vary the value of the D6 interval between 0.001-1s. We measure the number of lost packets under these conditions for our approach and the SPIDER approach. We also measure the number of lost packets for the SPIDER approach for various D7 values. Figure 3 shows the comparison of the number of lost packets using our approach and SPIDER. We can observe that the number of lost packets increases with D6. However, we can see that our approach performs better than all variations of the SPIDER approach. With our approach, the network suffers from a maximum packet loss ratio of 0.1% which is an improvement over the SPIDER approach (2.0%). In our analysis, we found that our approach requires to send more Probe packets compared to the SPIDER approach. However, as our approach does not use any Heartbeat packets, the overhead caused by Probe packets is less than that of Heartbeat packets.

V. Conclusion

Multiple mechanisms have been proposed for SDN failure recovery. However, due to its reliance on the data plane for decision-making, the Openstate pipeline already shows improvement in failure recovery. In this paper, we proposed an approach to further enhance the link failure recovery based on the Openstate pipeline. Our approach significantly reduces the overhead by predicting the link failure situation. By preemptively sending packets on the backup path, we significantly improve the packet loss. Our results also show that utilization of TCAM can be improved with our approach.

For future work, we plan to investigate the work on recovery from multiple faults and implement the Openstate pipeline approach for multiple fault recovery mechanism. In addition, we plan to realize our approach with P4 which could make it more realistic in real networks.

References

[1] (2017) Open networking foundation, url https://www.opennetworking.org. [Online]. Available: URL https://www.opennetworking.org

[2] P. Thorat, S. Jeon, and H. Choo, "Enhanced local detouring mechanisms for rapid and lightweight failure recovery in openflow networks," *Computer Communications*, vol. 108, pp. 78–93, 2017.

[3] M. Sun, Z. Shi, S. Chen, Z. Zhou, and Y. Duan, "Energy-efficient composition of configurable internet of things services," *IEEE Access*, vol. 5, pp. 25 609–25 622, 2017.

[4] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "Openflow: Meeting carrier-grade recovery requirements," *Computer Communications*, vol. 36, no. 6, pp. 656–665, 2013.

[5] C. Cascone, L. Pollini, D. Sanvito, A. Capone, and B. Sanso, "Spider: Fault resilient sdn pipeline with recovery delay guarantees," in *NetSoft Conference and Workshops (NetSoft), 2016 IEEE*. IEEE, 2016, pp. 296–302.

[6] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "Openstate: programming platform-independent stateful openflow applications inside the switch," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 44–51, 2014.

[7] (2017) Open flow protocol, url http://www.openflow.org/wp/learnmore. [Online]. Available: URL http://www.openflow.org/wp/learnMore

[8] W. John, A. Kern, M. Kind, P. Skoldstrom, D. Staessens, and H. Woesner, "Splitarchitecture: Sdn for the carrier domain," *IEEE Communications Magazine*, vol. 52, no. 10, pp. 146–152, 2014.

[9] 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, 2008.

[10] 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, 2013.

[11] N. L. Van Adrichem, B. J. Van Asten, and F. A. Kuipers, "Fast recovery in software-defined networks," in *Software Defined Networks (EWSDN), 2014 Third European Workshop on*. IEEE, 2014, pp. 61–66.

[12] A. Capone, C. Cascone, A. Q. Nguyen, and B. Sanso, "Detour planning for fast and reliable failure recovery in sdn with openstate," in *Design of Reliable Communication Networks (DRCN), 2015 11th International Conference on the*. IEEE, 2015, pp. 25–32.

[13] J.-P. Vasseur, M. Pickavet, and P. Demeester, *Network recovery: Protection and Restoration of Optical, SONET-SDH, IP, and MPLS*. Elsevier, 2004.

[14] E. Mannie and D. Papadimitriou, "Recovery (protection and restoration) terminology for generalized multi-protocol label switching (gmpls)," 2006.

[15] R. Ahmed, E. Alfaki, and M. Nawari, "Fast failure detection and recovery mechanism for dynamic networks using software-defined networking," in *Basic Sciences and Engineering Studies (SGCAC), 2016 Conference of*. IEEE, 2016, pp. 167–170.

[16] M. S. M. Zahid, B. Isyaku, and F. A. Fadzil, "Recovery of software defined network from multiple failures: Openstate vs openflow," in *Computer Systems and Applications (AICCSA), 2017 IEEE/ACS 14th International Conference on*. IEEE, 2017, pp. 1178–1183.

[17] S. Zhang, Y. Wang, Q. He, J. Yu, and S. Guo, "Backup-resource based failure recovery approach in sdn data plane," in *Network Operations and Management Symposium (APNOMS), 2016 18th Asia-Pacific*. IEEE, 2016, pp. 1–6.

[18] P. Thorat, S. Raza, D. S. Kim, and H. Choo, "Rapid recovery from link failures in software-defined networks," *Journal of Communications and Networks*, vol. 19, no. 6, pp. 648–665, 2017.

[19] B. Niven-Jenkins, D. Brungard, and M. Betts, "sprecher, n., ueno, s.," mpls-tp requirements," RFC 5654, September, Tech. Rep., 2009.

[20] S. Kandula, D. Katabi, S. Sinha, and A. Berger, "Dynamic load balancing without packet reordering," *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 2, pp. 51–62, 2007.

[21] (2017) Ryu openflow controller, http://osrg.github.io/ryu/. [Online]. Available: URL http://osrg.github.io/ryu/

[22] (2017) Cpqd openflow 1.3 software switch, url http://cpqd.github.io/ofsoftswitch13/. [Online]. Available: URL http://cpqd.github.io/ofsoftswitch13/