Fast Recovery in Software-Defined Networks

Niels L. M. van Adrichem, Benjamin J. van Asten and Fernando A. Kuipers
Network Architectures and Services, Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands
{N.L.M.vanAdrichem@, B.J.vanAsten@student., F.A.Kuipers@}tudelft.nl

***Abstract*—Although Software-Defined Networking and its implementation OpenFlow facilitate managing networks and enable dynamic network configuration, recovering from network failures in a timely manner remains non-trivial. The process of (a) detecting the failure, (b) communicating it to the controller and (c) recomputing the new shortest paths may result in an unacceptably long recovery time. In this paper, we demonstrate that current solutions, employing both reactive restoration or proactive protection, indeed suffer long delays. We introduce a failover scheme with per-link Bidirectional Forwarding Detection sessions and preconfigured primary and secondary paths computed by an OpenFlow controller. Our implementation reduces the recovery time by an order of magnitude compared to related work, which is confirmed by experimental evaluation in a variety of topologies. Furthermore, the recovery time is shown to be constant irrespective of path length and network size.**

I. Introduction

Recently, Software-Defined Networking (SDN) has received much attention, because it allows networking devices to exclusively focus on data plane functions and not control plane functionality. Instead, a central entity, often referred to as the controller, performs the control plane functionality, i.e. it monitors the network, computes forwarding rules and configures the networking nodes' data planes.

One of the benefits of the central controller entity introduced in SDN is its possibility to monitor the network for performance and functionality and reprogram when necessary. Where the controller can monitor overall network health as granular as observing per-flow characteristics, such as throughput, delay and packet loss [1], the most basic task is to maintain end-to-end connectivity between nodes. Hence, when a link breaks, the controller needs to reconfigure the network to restore or maintain end-to-end connectivity for all paths. However, the time-to-restoration of a broken path, beside the detection time, includes delay introduced by the propagation time of notifying the event to the controller, path re-computation, and reconfiguration of the network by the controller. As a result, controller-initiated path restoration may take over 100 ms to complete, which is considered too long for provider networks where at most 50 ms is tolerable [2].

In this paper, we introduce a fast (sub 50 ms) failover scheme relying on link-failure detection by combining primary and backup paths configured by a central OpenFlow [3] controller and implementing per-link failure detection using Bidirectional Forwarding Detection (BFD) [4], a protocol that detects failures by detecting packet loss in frequent streams of control messages.

Our work is organized as follows. In section II, we discuss several failure detection mechanisms and analyze how network properties and BFD session configuration influence restoration time. In section III, we introduce and discuss the details of our proposed failover mechanism. Section IV presents our experiments, after which the results are discussed and analyzed to verify our failover mechanism. Related work is discussed in section V. Finally, section VI concludes this paper.

II. Failure detection mechanisms

In this section, we introduce different failure detection mechanisms and discuss their suitability for fast recovery. In subsection II-A we analyze and minimize the elements that contribute to failure detection, while subsection II-B introduces OpenFlow's Fast Failover functionality.

Before a switch can initiate path recovery, a failure must be detected. Depending on the network interface, requirements for link failure detection are defined for each network layer. Current OpenFlow implementations are mostly based on Ethernet networks. However, Ethernet was not designed with high requirements on availability and failure detection. In Ethernet, the physical layer sends heartbeats with a period of 16 ± 8 ms over the link when no session is active. If the interface does not receive a response on the heartbeats within a set interval of $50-150$ ms, the link is presumed disconnected [5]. Therefore, Ethernet cannot meet the sub 50 ms requirement and failure detection must be performed by higher network protocols.

On the data-link layer multiple failure detection protocols exist, such as the Spanning Tree Protocol (STP) or Rapid STP [6], which are designed to maintain the distribution tree in the network by updating port status in switches. These protocols, however, can be classified as slow, as detection windows are in the order of seconds. Instead, in our proposal and implementation we will use BFD [4], which has proven to be capable of detecting failures within the required sub 50 ms detection window.

A. Bidirectional Forwarding Detection

The Bidirectional Forwarding Detection (BFD) protocol implements a control and echo message mechanism to detect liveliness of links or paths between preconfigured end-points. Each node transmits control messages with the current state of the monitored link or path. A node receiving a control message, replies with an echo message containing its respective session status. A session is built up with a three-way handshake, after which frequent control messages confirm

absence of a failure between the session end-points. The protocol is designed to be protocol agnostic, meaning that it can be used over any transport protocol to deliver or improve failure detection on that path. While it is technically possible to deploy BFD directly on Ethernet, MPLS or IP, Open vSwitch [7], a popular OpenFlow switch implementation also used in our experiments, implements BFD using a UDP/IP stream.

The failure detection time T_{det} of BFD depends on the transmit interval T_i and the detection time multiplier M, T_i defines the frequency of the control messages, while M defines the number of lost control packets before a session end-point is considered unreachable. Hence, the worst-case failure detection time equals $T_{det} = (M + 1) \cdot T_i$. Typically, a multiplier of $M = 3$ is considered appropriate to prevent small packet loss from triggering false positives. The transmit interval T_i is lower-bounded by the round-trip-time (RTT) of the link or path. Furthermore, BFD intentionally introduces a 0 to 25% time jitter to prevent packet synchronization with other systems on the network.

The minimal BFD transmit interval is given in equations (1) and (2), where $T_{i,min}$ is the minimal required BFD transmit interval, T_{RTT} is the round-trip time, T_{Trans} is the transmission delay, T_{Prop} is the time required to travel a link in which we include delay introduced by routing table look-up, L is the number of links in the path and T_{Proc} is the processing time consumed by the BFD end-points.

$$T_{i,min} = 1.25 \cdot T_{RTT} \tag{1}$$

$$T_{i,min} = 1.25 \cdot 2 \cdot (T_{Trans} + L \cdot T_{Prop} + T_{Proc}) \tag{2}$$

It is difficult to optimize for session RTT by improving T_{Trans} and T_{Proc} as those values are configuration independent. However, by using link monitoring $(L = 1)$, the interval time is minimized and smaller failure detection times are possible. A great improvement compared to per-path monitoring, where the number of links L is upper-bound by the diameter of the network.

An upper-bound for T_{RTT} can easily be determined with packet analysis, where we assume that the process of processing and forwarding is equal for each hop. On high link loads, the round-trip-time (RTT) can vary much and might cause BFD to produce false positives. During the development of TCP [8], a similar problem was identified in [9], where the retransmission interval of lost packets is computed by $\beta \cdot T_{RTT}$. The constant β accounts for the variation of inter-arrival times, which we implement in equation (3).

$$T_{i,min} = 1.25 \cdot \beta \cdot T_{RTT} \tag{3}$$

A fixed and conservative value $\beta = 2$ is recommended [10].

B. Liveliness monitoring with OpenFlow

From OpenFlow protocol version 1.1 onwards Group Table functionality is supported. Group Tables extend OpenFlow

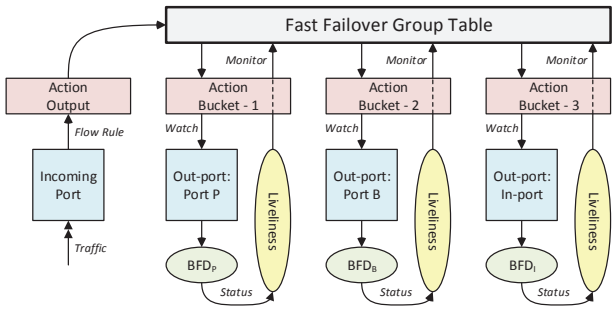


Fig. 1: OpenFlow Fast Failover Group Table.

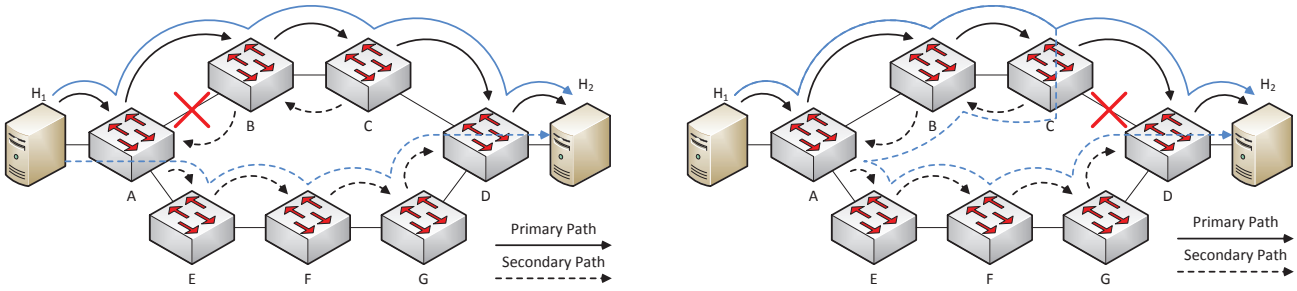
configuration rules allowing advanced forwarding and monitoring at switch level. In particular, the Fast Failover Group Table can be configured to monitor the status of ports and interfaces and to switch forwarding actions accordingly, independent of the controller. Open vSwitch implements the Fast Failover Group Table where the status of the ports is determined by the link status of the corresponding interfaces. Ideally, the status of BFD should be interpreted by the Group Table as suggested in figure 1 and implemented by us in section (IV).

III. Proposal

We propose to divide the recovery process into two steps. The first step consists of a fast switch-initiated recovery based on preconfigured forwarding rules guaranteeing end-to-end connectivity. The second step involves the controller calculating and configuring new optimal paths. Switches initiate their backup path after detecting a link failure, meaning that each switch receives a preconfigured backup path in terms of Fast Failover rules. Link loss is detected by configuring per-link - instead of per-path - BFD sessions. Using per-link BFD sessions introduces several advantages:

1) A lower detection time due to a decreased session round-trip-time (RTT) and thus lower transmit interval.
2) Decreased message complexity and thus network overhead as the number of running BFD sessions is limited to 1 per link, instead of the multiplication of all end-to-end sessions and their intermediate links.
3) Removal of false positives. As each session spans a single link, false positives due to network congestion can be easily removed by prioritizing the small stream of control packets.

In situations where a switch has no feasible backup path, it will return packets to the previous switch by crankback routing. As the incoming-port is part of the packet-matching filter of OpenFlow, preceding switches have separate rules to process returned packets and forward them across their backup path. This implies a recursive returning of packets to preceding switches until they can be forwarded over a feasible link- or node-disjoint path. Figure 2 shows an example network with primary and backup paths, as well as two failover situations.



(a) Topology with broken link resulting in usage of a secondary path.

(b) Topology with broken link resulting in usage of a backup path by crankback.

Fig. 2: A topology showing its primary and secondary paths including two backup scenarios in case of specific link-failures. Where the first scenario (a) uses a disjoint backup path, the second scenario (b) relies on crankback routing.

By instructing all switches up front with a failover scenario, switches can initiate a failover scenario independent of the SDN controller or connectivity polling techniques. The OpenFlow controller computes primary and secondary paths from every intermediate switch to destination to supply the necessary redundancy and preconfigures switches accordingly. Although the preconfigured backup-path may not be optimal at the time of activation, as in subfigure 2b, it is a functional path that is applied with low delay. Additionally, once the controller is informed of the malfunction, it can reconfigure the network to replace the current backup path by a more suitable path without traffic interruption as performed in [11].

The transmit interval of BFD is upper-bounded by the RTT between the session endpoints. Since we are configuring per-link sessions, the transmit interval decreases greatly. For example, in our experimental testbeds we have a RTT below 0.5 ms, thus allowing a BFD transmit interval of 1 ms. Although this might appear as a large overhead, per-link sessions limit the number of traversing BFD sessions to 1 per link. In comparison, per-path sessions imply $O(N \times N)$ shortest paths to travel each link. Even though most of them may be forwarded to their endpoint without inspection, each node has to maintain $O(N)$ active sessions.

A BFD control packet consists of at most 24 bytes with authentication, encapsulation in a UDP, IPv4 and Ethernet datagram results in $24 + 8 + 20 + 38 = 90$ bytes $= 720$ bits. Sent once every 1 ms, this results in an overhead of 0.067 % and 0.0067 % in, respectively, 1 and 10 Gbps connections.

IV. Experimental Evaluation

In this section, we will first discuss our experimental setup followed by the used measurement techniques, the different experiments and finally the results.

A. Testbed environments

We have performed our experiments on two types of physical testbeds, being a testbed of *software* switches and a testbed of *hardware* switches.

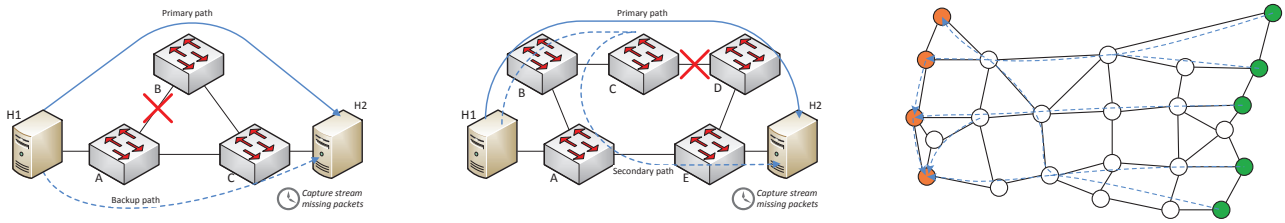
Our software switch based testbed consists of 24 physical, general-purpose, servers enhanced with multiple network interfaces and software to run as networking nodes. Each server contains a 64 bit Quad-Core Intel Xeon CPU running at 3.00GHz with 4.00 GB of main memory and has 6 independent 1 Gbps networking interfaces installed and can hence perform the role of a 6-degree networking node. Links between nodes are realized using physical Ethernet connections, hence deleting any measurement inaccuracies introduced by possible intermediate switches or virtual overlays. OpenFlow switch capability is realized using the Open vSwitch [7] software implementation, installed on the Ubuntu 13.10 operating system running GNU/Linux kernel version 3.11.0-12-generic.

The hardware switch based testbed we used was graciously made available to us by SURFnet, the NREN of the Netherlands. The testbed consists of 5 Pica8 P3920 switches, running firmware release 2.0.4. The switches run in Open vSwitch mode to deliver OpenFlow functionality, meaning that they implement the same interfaces as defined by Open vSwitch.

B. Recovery and measurement techniques

We use two complementary techniques to implement our proposal: (1) We use OpenFlow's Fast Failover Group Tables to quickly select a preconfigured backup path in case of link-failure. The Fast Failover Group Tables continuously monitor a set of ports, while incoming packets destined for a specific Group Table are forwarded to the first active and alive port from its set of monitored ports. Therefore, when link functionality of the primary output port fails, the Group Table will automatically turn to the secondary set of output actions. After restoring link functionality, the Group Tables revert to the primary path. (2) Link-failure itself is detected using the BFD protocol. We chose a BFD transmit interval conform equation (3), with $\beta = 2$ to account for irregularities by queuing. Although the RTT allowed smaller transmit intervals, the implementation of BFD forced a minimum window of 1 ms. We use a detection multiplier of $M = 3$ lost control messages to prevent false positives.

At the time of writing, both BFD and Group Table functionality are implemented in the most recent snapshots from Open vSwitch' development repository [12]. Although both BFD status reports and Fast Failover functionality operate



(a) Simple Network Topology. (b) Ring Network Topology. (c) USNET Topology.

Fig. 3: Topologies used in our experiments, including functional and failover scenarios in case of specific link-failure.

correctly, we found the Fast Failover functionality did not include the reported BFD status to initiate the backup sequence and only acts on (administrative) link down events. To resolve the former problem, we wrote a small patch for Open vSwitch to include the reported interface BFD status [13]. Furthermore, the patch includes minor changes to allow BFD transmit intervals smaller than 100 ms.

In order to simulate traffic on the network we use the *pktgen* [14] packet generator, which can generate packets with a fixed delay and sequence numbers. We use the missing sequence numbers of captured packets to determine the start and recovery time of the failure. *Pktgen* operates from kernel space, therefore high timing accuracy is expected and was confirmed at an interval accuracy of 0.005 ms. In order to get 0.1 ms timing accuracy, *pktgen* is configured to transmit packets at a 0.05 ms interval.

C. Experiments

This subsection describes the performed experiments. We run baseline experiments using link-failure detection by regular Loss-of-Signal on both testbeds. Additionally, we run experiments using link-failure detection by per-link BFD sessions on the software switch testbed to show improvement. In general, links are disconnected using the Linux *ifdown* command. To prevent the administrative interface change to influence the experiment, the *ifdown* command is issued from the switch opposite to the switch performing recovery.

The experiments are executed as follows. We start our packet generator and capturer at $t = 0$. At $t_{failure}$, a failure is introduced in the primary path. The packet capture program records missing sequence numbers, while the failover mechanism autonomously detects the failure and converts to the backup path. At $t_{recovery}$, connectivity is restored, which is detected by a continuation of arriving packets, the recording of missing packets is stopped and the recovery time is computed.

Basic functionality: In our first experiment we measure and compare the time needed to regain connectivity in a simple network to prove basic functionality. The network is depicted in figure 3a and consists of two hosts named H_1 and H_2, which connect via paths $A \rightarrow B \rightarrow C$ (primary) and $A \rightarrow C$ (backup). In this experiment, we stream data from H_1 to H_2 and deliberately break the primary path by disconnecting link $A \leftrightarrow B$ and measure the time needed to regain connectivity.

Crankback: After confirming restoration functionality we need to confirm crankback functionality. To do so, we introduce a slightly more complicated ring topology in figure 3b, in which the primary path is set as $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E$. We break link $C \leftrightarrow D$ enforcing the largest crankback path to activate, resulting in the path $A \rightarrow B \rightarrow C \rightarrow B \rightarrow A \rightarrow E$.

Extended experiments: To test scalability, we perform additional experiments in which we simulate a real-world network scenario. To do so, we configure our software-switch testbed in the USNET topology shown in figure 3c. We set up connections from all East-coast routers to all West-coast routers, thereby exploiting 20 shortest paths. We configure each switch to initially forward packets along the shortest path to destination, as well as a backup path from each intermediate switch to destination omitting the protected link. At each iteration we uniformly select a link from the set of links participating in one or more shortest paths and break it.

D. Results and analysis

Our first set of results, displayed in figure 4, shows baseline measurements without BFD performed for the simple topology on both testbeds. This experiment shows link-failure detection based on Loss-of-Signal implies an infeasibly long recovery time in both our software and hardware switch testbeds. The Open vSwitch testbed shows an average recovery time of 4759 ms. Although the hardware switch testbed performs better with an average recovery time of 1697 ms, the duration of both recoveries are unacceptable in carrier-grade networks.

In order to determine the time consumed by administrative processing additional to actual failure detection, we repeated previous experiments with the exception that we administratively brought down the interface at the switch performing the recovery. By doing this, we only trigger and measure the time consumed by the administrative processes managing link status. We found that Open vSwitch on average needs 50.9 ms to restore connectivity after detection occurs. Due to this high value, our patch omits the administrative link status update and directly checks BFD status instead.

Currently, the firmware of the hardware switches does not support BFD, therefore we only verified recovery using link-failure detection by BFD on the software switch testbed. Figure 5 shows 100 samples of measured recovery delay for the simple and ring topologies using three different BFD transmit intervals, namely 15 ms, 5 ms and 1 ms. For the

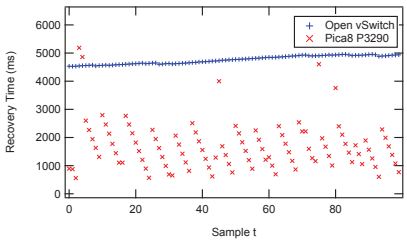


Fig. 4: Baseline measurements using Loss-of-Signal failure detection.

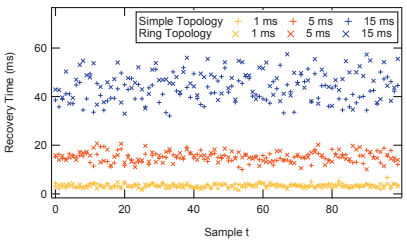


Fig. 5: Recovery times for selected BFD transmit intervals and topologies.

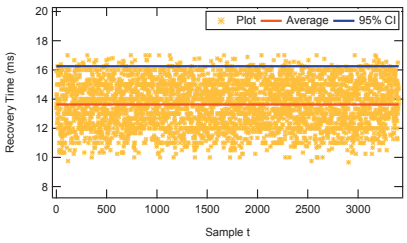


Fig. 6: Recovery times for 5 ms transmit interval in the extended topology.

simple topology, we measure a recovery time that was a factor between 2.5 and 4.1 ms larger than the configured BFD transmit interval due to the detection multiplier of $M = 3$. As shown in figure 7, for each BFD transmit interval we measure average and 95% confidence interval recovery times of 3.4 ± 0.7 ms, 15.0 ± 2.2 ms and 42.1 ± 5.0 ms. The results show that, especially with links having low RTTs, a great decrease in recovery time can be reached. At a BFD transmit interval of 1 ms, we reach a maximal recovery time of 6.7 ms.

For the ring topology, after 100 samples of measuring the recovery time for three different BFD transmit intervals, figure 7 shows averages and 95% confidence intervals of respectively 3.3 ± 0.8 ms, 15.4 ± 2.7 ms and 46.2 ± 5.1 ms for the different selected transmit intervals. At a BFD transmit interval of 1 ms, we reach a maximal recovery time of 4.8 ms. Even though the ring topology is almost *twice as large* as the simple topology, recovery times remain constant due to the per-link discovery of failures and the failover crankback route.

In order to verify the scalability of our implementation, we repeated the experiments by configuring our software-switch testbed with the USNET topology, hence simulating a real-world network. For each iteration, we average the recovery times experienced by the affected destinations. Figure 6 shows over 3400 samples taken at a BFD transmit interval of 5 ms, resulting in an average and 95% confidence interval of 13.6 ± 2.6 also shown in figure 7. The high frequency of arriving BFD and probe packets congested the software implementation of Open vSwitch, showing the necessity for hardware line card support of BFD sessions. As a consequence, we were unable to further decrease the polling frequency and had to decrease the frequency of probe packets to 1 per ms, slightly reducing the accuracy of this set of measurements. Although we experience software system integration issues showing the need to optimize switches for degree and traffic throughput, *recovery times remain constant independent of path length and network size*, showing our implementation also scales to larger real-world networks.

Prior to implementing BFD in the Group Tables, the Fast Failover Group Tables showed a 2 second packet loss when reverting to the primary path after repair of a failure. With BFD, switch-over is delayed until link status is confirmed to be restored and no packet loss occurs, resulting in a higher stability of the network.

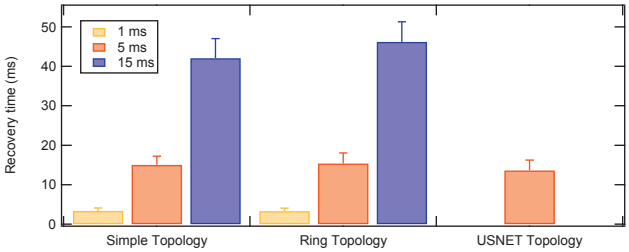


Fig. 7: Bar diagram summarizing average recovery times and 95% confidence interval deviation on our Open vSwitch testbed using BFD at selected transmit intervals.

V. Related Work

Sharma et al. [15] show that a controller-initiated recovery needs approximately 100 ms to regain network connectivity. They propose to replace controller-initiated recovery with a path-failure detection using BFD and preconfigured backup paths. The switches executing BFD detect path failure and revert to previously programmed backup paths without controller intervention resulting in a reaction time between 42 and 48 ms. However, the correctness and speed of detecting path failure depends highly on the configuration and switch implementation of BFD. According to [4], a detection time of 50 ms can be achieved by using an "aggressive session", using a transmit interval of 16.7 ms and window multiplier of 3. However, the authors of [15] do not provide details on their configuration of BFD. The BFD transmit interval is lower-bounded by the propagation delay between the end-points of the BFD sessions, therefore we claim a path-failure detection under-performs compared to a per-link failure detection and protection scheme as presented in this paper.

Kempf et al. [16][17] propose an alternative OpenFlow switch design that allows integrated operations, administration and management (OAM) execution, foremost connectivity monitoring, in MPLS networks by introducing logical group ports. Introducing ring topologies in Ethernet-enabled networks and applying link-failure detection results to trigger failover forwarding rules, results in an average failover time of 101.5 ms [18] with few peaks between 140 ms and 200 ms. However, the introduced logical group ports remain unstandardized and are hence not part of shipped OpenFlow switches.

TABLE I: Summary of results and most important differences compared to related work.

Reference	*Avg recovery time* $\pm 95\%$ *CI*	*Network size* (N, L)	*Detection Mechanism*	*Recovery Mechanism*
[15]	$42 - 48$ms	$(28, 40)$	Per-path BFD	OpenFlow Fast Failover Group Tables using virtual ports
[16][17]	28.2 ± 0.5 ms	N.A.	Per LSP OAM + BFD	Custom extension of OpenFlow
[18][19]	32.74 ± 4.17 ms	$(7, 7)$	Undocumented	Custom auto-reject mechanism
This Paper	3.3 ± 0.8 ms	$(24, 43)$	Per-link BFD	Commodity OpenFlow Fast Failover Group Tables

Finally, Sgambelluri et al. [19] perform segment protection in Ethernet networks depending on OpenFlow's auto-reject function to remove flows of failed interfaces. The largest part of their experimental work involves Mininet emulations, which is considered inaccurate in terms of timing due to its high level of virtualization [20]. The experiment performed on a physical testbed shows a switch-over time of at most 64 ms, however, the authors do not describe the method of link-failure detection, which make the results difficult to reproduce.

Table I shows the main differences between the aforementioned proposals and our work. Where the previous best recovery time was close to 30 ms, section IV showed that our configuration regains connectivity within a mere 3.3 ms.

VI. Conclusion

Key to supporting high availability services in a network is the network's ability to quickly recover from failures. In this paper we have proposed a combination of protection in OpenFlow Software-Defined Networks through preconfigured backup paths and fast link failure detection. Primary and secondary path pairs are configured via OpenFlow's Fast Failover Group Tables enabling path protection in case of link failure, deploying crankback routing when necessary. We configure per-link, in contrast to the regular per-path, Bidirectional Forwarding Detection (BFD) sessions to quickly detect link failures. This limits the number of traversing BFD sessions to be linear to the network size and minimizes session RTT, enabling us to further decrease the BFD sessions' window intervals to detect link failures faster. By integrating each interface's link status into Open vSwitch' Fast Failover Group Table implementation, we further optimize the recovery time.

After performing baseline measurements of link failure detection by Loss-of-Signal on both a software and hardware switch testbed, we have performed experiments using BFD on our adapted software switch testbed. In our experiments we have evaluated recovery times by counting lost segments in a data stream. Our measurements show we already reach a recovery time considered necessary for carrier-grade performance at a 15 ms BFD transmit interval, being sub 50 ms. By further decreasing the BFD interval to 1 ms we reach an even faster recovery time of 3.3 ms, which we consider essential as network demands proceed to increase. Compared to average recovery times varying from 28.2 to 48 ms in previous work, we show an enormous improvement.

Since we perform per-link failure detection and preconfigure each node on a path with a backup path, the achieved recovery time is independent of path length and network size, which is confirmed by experimental evaluation.

Acknowledgment

We thank SURFnet, and in particular Ronald van der Pol, for allowing us to conduct experiments on their hardware switch based OpenFlow testbed.

This research has been partly supported by the EU FP7 Network of Excellence in Internet Science EINS (project no. 288021).

References

[1] N. L. M. van Adrichem, C. Doerr, and F. A. Kuipers, "Opennetmon: Network monitoring in openflow software-defined networks," in *Network Operations and Management Symposium (NOMS), 2014 IEEE*.

[2] B. Niven-Jenkins, D. Brungard, M. Betts, N. Sprecher, and S. Ueno, "Requirements of an MPLS Transport Profile," RFC 5654 (Proposed Standard), Internet Engineering Task Force, Sep. 2009.

[3] 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, 2008.

[4] D. Katz and D. Ward, "Bidirectional Forwarding Detection (BFD)," RFC 5880 (Proposed Standard), Internet Engineering Task Force, Jun. 2010.

[5] L. Wang, R.-F. Chang, E. Lin, and J. C.-s. Yik, "Apparatus for link failure detection on high availability ethernet backplane," Aug. 21 2007, uS Patent 7,260,066.

[6] D. Levi and D. Harrington, "Definitions of Managed Objects for Bridges with Rapid Spanning Tree Protocol," RFC 4318 (Proposed Standard), Internet Engineering Task Force, Dec. 2005.

[7] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker, "Extending networking into the virtualization layer." in *Hotnets*, 2009.

[8] J. Postel, "Transmission Control Protocol," RFC 793 (INTERNET STANDARD), Internet Engineering Task Force, Sep. 1981.

[9] V. Jacobson, "Congestion avoidance and control," in *ACM SIGCOMM Computer Communication Review*, vol. 18, no. 4. ACM, 1988.

[10] D. D. Clark, "Window and acknowledgement strategy in tcp," 1982.

[11] M. Reitblatt, N. Foster, J. Rexford, and D. Walker, "Consistent updates for software-defined networks: Change you can believe in!" in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, 2011.

[12] "Open vSwitch GIT Web Front-End," http://git.openvswitch.org, 2014.

[13] N. L. M. van Adrichem, B. J. van Asten, and F. A. Kuipers, https://github.com/TUDelftNAS/SDN-OpenFlowRecovery, Jul. 2014.

[14] R. Olsson, "pktgen the linux packet generator," in *Proceedings of the Linux Symposium, Ottawa, Canada*, 2005.

[15] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "Openflow: meeting carrier-grade recovery requirements," *Computer Communications*, 2012.

[16] J. Kempf, E. Bellagamba, A. Kern, D. Jocha, A. Takacs, and P. Skoldstrom, "Scalable fault management for openflow," in *Communications (ICC), 2012 IEEE International Conference on*, 2012.

[17] E. Bellagamba, J. Kempf, and P. Skoldstrom, "Link failure detection and traffic redirection in an openflow network," May 19 2011, uS Patent App. 13/111,609.

[18] A. Sgambelluri, A. Giorgetti, F. Cugini, F. Paolucci, and P. Castoldi, "Effective flow protection in open-flow rings," in *Optical Fiber Communication Conference*. Optical Society of America, 2013.

[19] ——, "Openflow-based segment protection in ethernet networks," *Optical Communications and Networking, IEEE/OSA Journal of*, vol. 5, no. 9, 2013.

[20] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, 2010.