Fast Control Channel Recovery for Resilient In-band OpenFlow Networks

A. S. M. Asadujjaman*‡, Elisa Rojas†, Mohammad Shah Alam*, Suryadipta Majumdar§
*IICT, Bangladesh University of Engineering and Technology, Dhaka, Bangladesh
†Departamento de Automatica, University of Alcala, Alcala de Henares, Spain
‡Network Planning Department, Banglalink Digital Communications Ltd., Dhaka, Bangladesh
§Concordia University, Montreal, QC, Canada
Email: ‡aasadujjaman@banglalink.net, †elisa.rojas@uah.es, *smalam@iict.buet.ac.bd, §su_majum@encs.concordia.ca

***Abstract*—Software-Defined Networking (SDN) is a thriving paradigm in computer networks where control plane is portrayed as a logically centralized entity decoupled from the data plane. Forwarding decisions at the data plane rely on installation of rules by the control plane through the control channel. Therefore, control channel resilience represents a critical requirement for SDN-based networks. Moreover, control channel leverages the links in the data plane in case of in-band control. To protect the control traffic in this case, we propose a scheme that combines logical ring topology with source-routed forwarding to enable local recovery from failure. This recovery does not require intervention by the controller, while preserving the simplicity of data plane devices. Simulation results on a real network topology show that the proposed scheme can recover from multiple link failures within 50ms. In our approach, failure recovery time is unaffected by control channel latency and, more specifically, we show that a link failure approximately 4,920km away from the controller (control channel latency of approximately 16.4ms) can be recovered within 3.8ms.**

I. Introduction

Software-Defined Networking (SDN) is an emerging technology for service provider networks that decouples packet forwarding infrastructure from the logically centralized control layer. This elimination of complex control functionality from data plane makes forwarding switches more efficient, while enabling novel network management strategies. However, separation of control plane from data plane leads to a multitude of problems of its own. As data plane devices now have no mechanism to learn routes on their own, effective communication relies on installation of traffic forwarding instructions into the switches by the controller. This requires a control channel to be established between the controller and the switches, implemented either as out-of-band or in-band communication (as shown in Fig. 1).

Several limitations for both out-of-band and in-band controls exist. On one hand, out-of-band control is expensive and often impractical [1]–[3]. This is because extra physical interfaces and cabling are required to connect each data plane device with the controller. For example, in Fig. 1a, control channel connectivity for each switch requires a dedicated physical interface on both the controller and the switch. Moreover, data plane devices may be located at distant physical locations from the controller requiring large additional investment to build a dedicated network for out-of-band control. As in-band control does not require any dedicated link for the control channel, it is more cost effective and therefore preferred by service providers. On the other hand, in-band control brings the issue of control channel failure due to the data plane link and node outage. As there is no dedicated link between a data plane device and the controller, control channel is established through the data plane infrastructure generally encompassing multiple nodes and links. A failure in any of these intermediate nodes or links will result in a failure in the control channel. For example, in Fig. 1b, the control channel with switch S4 is established through the path C–S2–S3–S4 and failure in any of the links or switches in this path will cause the control channel with S4 to fail. Therefore, a recovery mechanism is required to ensure resilience of the control channel by re-establishing the control channel through an alternative path (e.g., C–S2–S1–S4 in Fig. 1b).

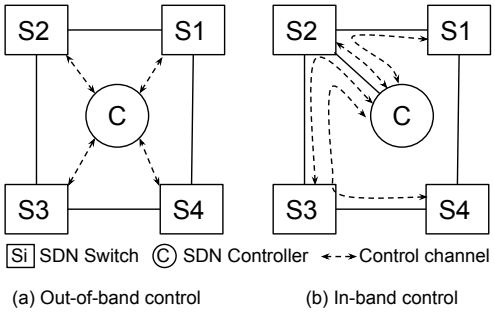


Fig. 1. Types of implementation of the SDN control channel

In this paper, we propose a fast control channel recovery mechanism for SDN considering in-band control. To this end, we first divide the control channel communication in two parts: switch-to-controller communication and controller-to-switch communication. We then use a ring based local recovery mechanism to ensure resilience of the switch-to-controller communication. As a resilient switch-to-controller communication channel enables the controller to be updated of the entire topology, we propose a source forwarding mechanism to ensure resilience of the controller-to-switch communication. A controller discovery process is also formulated as a part of

978-1-5386-4633-5/18/$31.00

this work with the help of the proposed controller-to-switch communication mechanism.

We developed a simulation tool in C++ to measure performance of the proposed mechanism. Using the simulation tool we evaluated our proposed protocol on a topology adapted from a large real network that spans approximately 4,920 km from Victoria to Montreal, Canada. Simulation results show that link failures can be recovered within a fraction of control channel latency. Moreover, we show that our protocol always recovers from multiple simultaneous failures within 50ms.

In summary, our main contributions are as follows:

1) We identify a set of design objectives from the literature for an effective control channel resiliency mechanism and compare existing works in light of these aspects.
2) We design a control channel recovery mechanism that satisfies the identified objectives.
3) We develop a simulation tool to evaluate the proposed mechanism on a topology adapted from a real network and show that our protocol can recover multiple simultaneous failures within 50ms.

The remainder of this paper is organized as follows. Firstly, we state the list of design objectives found in the literature in Section II. We then, analyze state-of-the-art approaches and map them to the previously defined objectives in Section III. Having done that, we propose a protocol that tries to accomplish the design criteria in Section IV. We describe our simulation environment in Section V, followed by the simulation results in Section VI and a brief discussion in Section VII. Finally, we conclude our paper in Section VIII.

II. Design objectives

We have identified several aspects that must be covered in designing an effective approach for control channel resilience.

1) In-band control: Building dedicated out-of-band control networks between controllers and switches is expensive [1]. Thus, in-band control is desirable [2].

2) Controller unaided recovery: Controller-based failure detection and recovery schemes put excessive burden on the controller as it should monitor every link, path or loop. Moreover, dependency on the controller to restore the control channel binds the recovery time to the control channel latency [4].

3) Local failure detection: Failure recovery time is directly dependent on the time required to detect a failure. Therefore failures should be ideally detected locally, by the adjacent nodes of a link or neighbors of a switch.

4) Maximum redundancy utilization: Recovery should be possible as long as there is a path between the switch and the controller in the underlying network graph.

5) Recovering from multiple failures: Service provider usually face multiple random failures. Therefore, a resilient design should withstand unpredicted multiple simultaneous failures.

6) Low data-plane complexity: One of the core advantages of SDN is the simplicity and cost-effectiveness of data plane hardware due to complete separation of the complex control layer [5]. This advantage should not be compromised.

7) Low memory consumption: Memory requirements should not limit scalability of the network. Protection mechanisms based on pre-installed backup paths rapidly increase memory requirements as the network grows and may exhaust switch memory [4], [6].

8) Supporting multiple SDN controllers: To enable redundancy in case of controller failure, the control channel resilience mechanism should support communication with multiple SDN controllers [7].

9) Auto-configuration: SDN has attracted attention of service providers because of its ability to manage networks with little human intervention. But, it still requires configuring controller settings and control channel. This should be improved by introducing auto-configuration capabilities [8]–[10].

III. Related Work

A comparative study of restoration and protection methods is presented in [1], [3], considering the protection methods essential to achieving carrier grade requirements. However, their protection approach requires monitoring every path by the switches plus pre-installed rules, which leads to high memory consumption. A different approach is shown in [2], which proposes using controller centric monitoring cycle to detect and identify fault location. Their main objective is to reduce controller load as compared to establishing Bidirectional Forwarding Detection (BFD) sessions with each switch. But, their work still involves the controller for each monitoring cycle. In addition, monitoring cycle cannot detect multiple failures and the controller-based restoration limits the speed of recovery.

Kotani et al. [11] use adaptive keep-alive messages and inter-controller communication to detect control channel failure but this approach is limited to avoiding the failed channel, which is not suitable for in-band control where all control channels may follow a common physical link. The problem of common physical path between different logical control channels have been recognized by [13] and solved by ensuring physically disjoint logical control channel connection to multiple controllers. But, even their failure recovery approach is limited to avoiding a failed control channel lacking the ability of local recovery. Moreover, this failed-channel-avoidance strategy relies on a set of pre-provisioned logical connections for each switch, which may put stress on a switch state to store all the backup flows in the network [14]. Watanabe et al. [8] propose a dedicated module which uses extensive communication with the controller for topology exchange. They implement their solution using a conventional link-state routing protocol which is complex and requires manual configuration.

The authors in [12] present a controller-rooted-tree based protection mechanism where failures are detected locally and control traffic is rerouted to neighbors of the affected node. Hu et al. [7] improve the limited coverage problem of [12] by adding reverse forwarding, but their work still cannot guarantee that all nodes will be protected. Moreover only a single link failure is studied in these works. A tree-based protection with precomputed backup route is also suggested

TABLE I
COMPARISON OF OPENFLOW CONTROL TRAFFIC FAILOVER SCHEMES

Requirement	Sharma [1], [3]	Lee [2]	Kotani [11]	Watanabe [8]	Beheshti [12]	Hu [7]	Proposed
In-band control	✓	✓		✓	✓	✓	✓
Controller unaided recovery	✓		✓	✓	✓	✓	✓
Local failure detection				✓	✓	✓	✓
Maximum redundancy utilization				✓			✓
Recover multiple failures	✓			✓			✓
Low data-plane complexity	✓	✓	✓		✓	✓	✓
Low memory consumption		✓	✓		✓	✓	✓
Support multiple controller			✓	✓		✓	✓
Auto-configuration							✓

by [15], where the authors propose calculation of the recovery path in the switches. They also propose reduction of flow table size by having switches only store route to neighbor. Their approach requires switches to have the capability of discovering the network topology, performing breadth first search and installing flow entry into other switches. Moreover, recovery time and performance under multiple failure has not been reported by the authors.

We compare the existing works and approach, in light of the aspects discussed in the previous section, in Table I.

IV. Proposed Protocol

In this section we present our solution for in-band (design objective 1) control channel recovery. The solution divides control channel communication in two parts based on the direction of communication, namely *Controller to Switch* (C2S) communication and *Switch to Controller* (S2C) communication. We then discuss our approach for recovery of each part of the communication separately. Our solution is based on a logical ring topology in S2C recovery and label-based forwarding for C2S channel. Initially, switches establish a control channel along the path via a unicast discovery mechanism. We have assumed that OpenFlow [16] is used as the southbound interface protocol (i.e., data plane to control plane communication).

A. Controller Discovery

When an OpenFlow switch is first installed in the network, it has no information about the controller and flow entries required for establishing a control channel. We propose a controller discovery mechanism with the help of two types of messages: 1) *Controller Discovery* (CoD) and 2) *Controller Information* (CoI), plus a cumulative cost. On startup, an OpenFlow switch will send CoD messages towards all of its live ports. On receiving the CoD message, the neighboring switch will send its cost to the controller. This cost is a cumulative value; the neighboring switch will report an incremented cost in the CoI. Only a controller will report a cost of zero in the CoI. On receiving a CoI from all ports, the new switch will compare and only store the CoI and associated interface with the lowest cost. The switch will then establish the OpenFlow channel with the controller, while the controller updates its

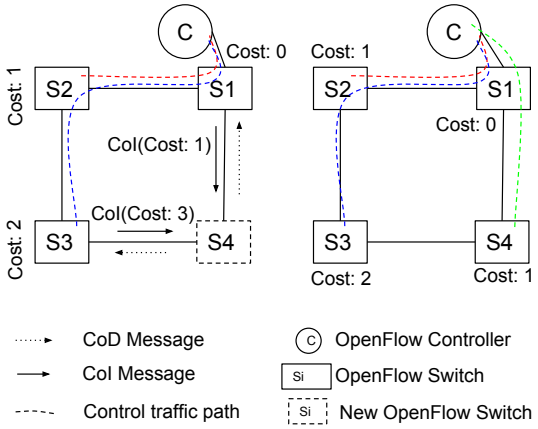


Fig. 2. Controller discovery via CoD and CoI messages

global topology view. This discovered path will be used until there is any ring assignment.

The controller discovery process is illustrated in Fig. 2. More specifically, the OpenFlow switch named S4 is added to a network with three switches (S1, S2 and S3), each of them already having established a control channel with controller C. According to the previously described procedure, the new switch sends CoD messages to S1 and S3. In response, S1 sends a CoI message to S4 with the cost of 1, while S3 sends it with a cost of 3. S4 decides that S1 has the lowest cost and installs a rule to forward future packets to controller towards S1.

B. Ring Design

In the proposed protocol, a logical ring topology is used to enable local failure detection which is our third design objective. The OpenFlow controller analyzes the topology to identify closed-rings and sub-rings. Closed-rings are closed loops, while sub-rings are segments with both their ends connected to a closed-ring or another sub-ring. The nodes where sub-rings attach to closed-rings are called interconnection nodes. The port where a sub-ring attaches to a closed-ring is called an interconnection port. In a closed-ring or sub-ring, all nodes will be configured with a *Ring ID* (RID) and the associated interfaces for the ring. One link will be provisioned

as the *Precluded Link* (PL), by marking the associated ports on its adjacent switches as PL ports.

We call the closed-ring that contains the switch connected to the controller the *central ring*. Accordingly, rings that are closer to the central ring are called *upstream rings*, while rings distant from the central ring are called *downstream rings*. Finally, one switch will be marked as the *Ring Breaker* (RB). The RB is only required for the central ring and when two rings meet at a single node. For the central ring, the controller-connected switch is the RB. For rings meeting at a single node, the common node is configured as RB for the downstream ring.

Although our ring design is similar to Ethernet Ring Protection Switching (ERPS) [17], [18], the underlying concept is different. The ERPS protocol runs on Ethernet switches but our protocol does not require it. Therefore, our protocol does not share the issues that are present in Ethernet (e.g., broadcast and MAC address table convergence).

In Fig. 5, S1-S2-S3-S4-S5-S1 is the central ring while S1-S6-S7-S2 is a sub-ring with interconnection nodes S1 and S2. Similarly, S7-S8-S3 is another sub-ring with interconnection nodes S7 and S3. In Fig. 5, the link S3-S4 is the PL for the main ring. Therefore, port 3 of S3 and port 1 of S4 are PL ports. Switch S5 is marked as the RB.

C. Ring and Sub-ring selection

In our design, the central ring is configured as a closed-ring. Remaining rings will be configured as sub-rings as long as they have a common link/segment with an upstream ring. A common link/segment between two rings will be part of the upstream ring.

D. Switch to controller communication (S2C)

We use three types of messages in the ring topology to provide resilience, namely: *Link Preclusion Message* (LPM), *Link Failure Message* (LFM) and *Link Restore Message* (LRM). LPM will be transmitted by both precluded link (PL) ports towards opposite direction of the PL. Every node in the ring or sub-ring will record the port from which it receives the LPM and forward it towards the opposite port. Only the ring breaker (RB) port and the interconnection port will drop the LPM. Finally, nodes will not forward packets that are destined to the controller to the port where an LPM has been received. This signaling with LPM creates a path from the switches to the controller and thereby establishes the S2C communication channel.

Similar to LPM signaling, upon link failure, link failure message (LFM) will be transmitted by the adjacent ports of the *Failed Link (FL)* towards opposite direction. Every node in the ring or sub-ring will record the port from which it receives the LFM and will forward the message towards the opposite port. Only the RB port and the interconnection port will drop the LFM. Finally, nodes will not forward packets that are destined to the controller to the port where an LFM has been received. The RB node and the interconnection node has the additional responsibility of reporting the LFM to the controller. When

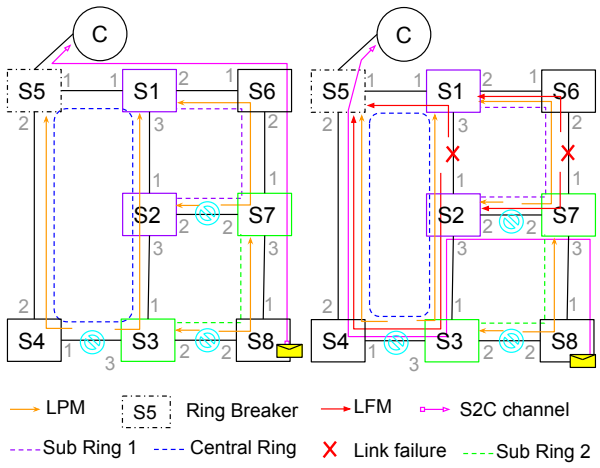


Fig. 3. S2C traffic forwarding

an LFM reaches these nodes, they will drop the message and report to the controller.

If a switch port receives an LFM, it will save any previously received LPM and use the LFM for forwarding decisions until the LFM is cleared. This means that, if an LFM is received for a ring, the LPM will be ignored for that ring. When a failed link is recovered, the adjacent nodes will transmit LRM messages in opposite direction of the recovered link. Upon receiving the LRM, any LFM will be cleared and previously received LPM will become effective.

The process is illustrated in Fig. 3 (left). Here, the S3-S4 link is the PL for the central ring and therefore the LPM flows from S3 (port 3) to S4 up to S1. LPM also flows from S4 (port 1) to S5. All other LPMs are shown as dashed lines in Fig. 3. When the LPM reaches S5, it will be dropped, since switch is the ring breaker (RB). Similarly, S1 will drop the LPM generated from port 2 of S7 (sub-ring 1) because S1 is an interconnection node for sub-ring 1.

Let us now examine how a message generated by S8 reaches the controller using the LPM based S2C signaling. There are two ports to forward the message toward in S8 and one of the ports (port 2) has an incoming LPM. Therefore S8 forwards the packet to port 1. Now, S7 sees that it has an incoming LPM at ports 2. This instructs S7 to forward the packet to port 1. Thus the packet takes the path of S8-S7-S6-S1-S5-C.

Finally, if there are two simultaneous failures, for example, link S6-S7 and link S1-S2, LFM travels as shown in Fig. 3 (right). When the message arrives at S7, it has an incoming LFM at port 1 and an incoming LPM at port 2. As the LFM has a higher priority, the LPM will be ignored and the message will be forwarded to port 2. Thus the message will be successfully delivered to the controller C through the new path S8-S7-S2-S3-S4-S5-C despite having two failures simultaneously in the path. The controller is not involved here in the S2C communication channel recovery process, and thus design objective 2 (controller unaided recovery) is met.

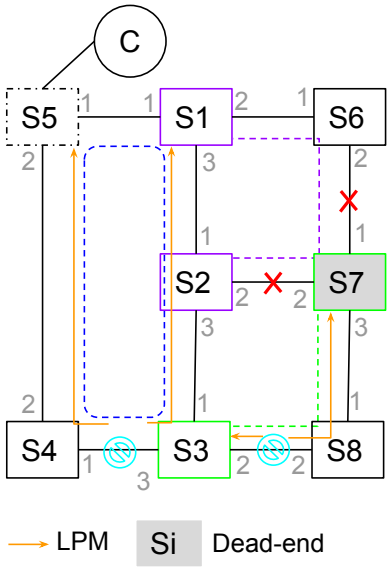


Fig. 4. Dead-end in a network

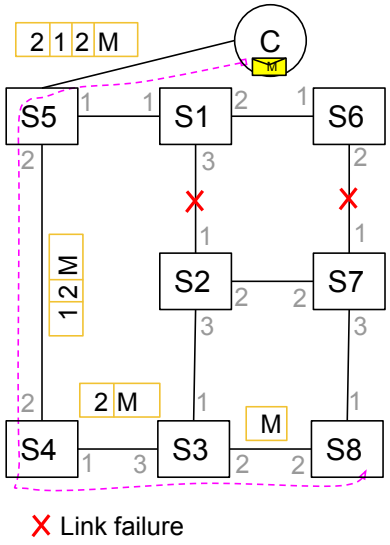


Fig. 5. C2S traffic forwarding using stacked label header

E. Dead-end recovery

It is possible that, after one or more adjacent links of a node fails, the node has only one adjacent link in the network. This transforms a formerly ring node into a spur node in the topology under failure. Therefore, this node can't forward any packet towards the controller. We call this node a *Dead-end*. Now, other nodes in the network having no mechanism to be notified of the *Dead-end* will continue to forward packets towards it, resulting in packet loss.

To solve the problem created by the *Dead-end*, we propose the following process. After a failure, when a switch detects that it has only one link in the network, it will send a *Dead-End Notification (DEN)* message in that link. Any switch receiving the *DEN* at an interface will stop forwarding packet in that interface. The switch receiving the *DEN* will forward the *DEN* if it also has only one interface in the network after excluding the dead-end direction.

For example, in the two simultaneous failures discussed previously, if the failed links are S6-S7 and S2-S7, S7 becomes a dead-end. As shown in Fig. 4, S7 has only one active link, S7-S8, in the network. Now, according to the proposed *Dead-end recovery* process, S7 will send *DEN* in the S7-S8 link and S8 will forward this in the S8-S3 link. S8 will consider S7-S8 link as failed and forward future packets towards S3. At S3, *DEN* will be dropped because there are still two usable links.

The S2C traffic forwarding process accompanied by the *DEN* messages will ensure failure recovery as long as there is a redundant path. This realizes our design objective 4 (maximum redundancy utilization).

F. Controller to switch communication (C2S)

The LPM or LFM guided S2C path can ensure switch to controller communication. However, when a message is destined towards a switch, this path cannot suggest which port to forward the packet toward. This is because switches do not store forwarding information for other switches.

We exploit the fact that, in the OpenFlow channel, controllers are the only senders that send packets toward the switches. In other words, switches do not need to establish OpenFlow channels among themselves. Because the S2C path enables the controller to have real-time updates of the entire topology, the controller can now guide the switches to route its messages. To realize the C2S channel, we use a source routed forwarding approach similar to [19] with the important difference that only the SDN controller is the source for the C2S channel. Therefore, in the proposed protocol, the SDN controller is not required to send state information to any switch and switches are not required to have the capability of receiving state information from the controller. Another difference is that, our work does not use the *reverse path calculation* and *link failure recovery* process described in [19].

The C2S forwarding process is illustrated in Fig. 5. In the figure, the controller knows the topology and about the failure of links S1-S2, S6-S7. So, it selects the path S5-S4-S3-S8 to deliver its message to switch S8. The controller then appends the label stack "`2,1,2`" as the message header, "`M`" being the message content. The first switch S5 pops the top label `2` and forwards the message to port 2. Similarly, switches S4 and S3 find labels 1 and 2, respectively, and forward the message accordingly.

V. Simulation Environment

In this section we describe the framework, topology and methodology used to perform the simulation.

A. Simulation framework and topology

To evaluate the performance of our proposed approach, we have developed our own simulation model in C++ using the OMNeT++ 5.0 simulation framework [21].

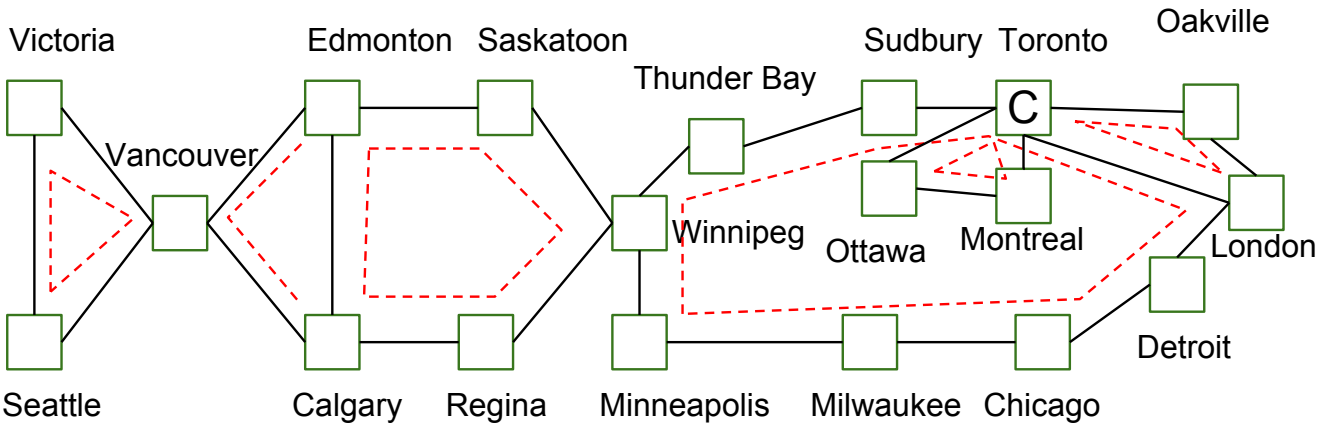


Fig. 6. Simulation Topology: Adapted from Rogers fiber backbone network [20]

We use a real network topology, as shown in Fig. 6, adapted from Rogers fiber backbone map [20] to perform the simulation. In this topology, we have taken 19 nodes connected by 24 links spanning approximately 4,920 km from Victoria to Montreal, Canada. The total fiber length of the 24 simulated links is approximately 11,183 km. The distances between the nodes (cities) were found using an online distance calculator [22]. We have considered the actual distance through highways to estimate the fiber lengths, as fiber is usually laid along highways.

B. Simulation methodology

We have created an OMNeT++ module using the NED (NEtwork Description) language called *ofswitch* to implement our protocol. This module, given in Listing 1, represents an OpenFlow switch. A C++ class is used to implement this module. It also represents the OpenFlow controller with modifications in the C++ class. We have also created a traffic generator module called *Pktgen*.

Listing 1. OMNET++ NED Module for the OpenFlow Switch

```
1  simple ofswitch
2  {
3      parameters:
4          string portstatus = default("");
5          string isRingBreakerFor = default("");
6          double failureDetectionTime=default(1);
7          bool shouldRecordStat=default(false);
8      gates:
9          inout port[8] @loose;
10 }
```

The *ofswitch* NED module has four parameters and one gate array. The parameters are 1) `portstatus` 2) `isRingBreaker` 3) `failureDetectionTime`, and 4) `shouldRecordStat`.

The `portStatus` parameter stores information on all ports of an OpenFlow switch in a single string with the format: `{port,ring,PL,FL};{..}`. The reason behind introducing these parameters as a string is that OMNeT++ 5.0 does not support parameter arrays. This string is later parsed in the C++ code to generate a C++ array accordingly. The `ring` field defines the ring/sub-ring that the `port` belongs to, and the `PL` field is used to indicate if the port is attached to a precluded link. Finally, the `FL` field is used to specify link failure time.

The `isRingBreakerFor` parameter is used to define the ring breaker (RB) node. A switch will act as an RB for the rings configured by this parameter.

The `failureDetectionTime` is used to simulate the delay between a failure and its detection by the adjacent NE.

The `shouldRecordStat` parameter specifies if the switch should generate statistics file in the hard drive. This helps to minimize generation of large statistics file with unnecessary information. It is only set to true for the controller node to measure failure recovery time.

The C++ class for the *ofswitch* module performs six major operations:

1) Transmit LPM messages
2) Forward LFM messages
3) Transmit LFM messages
4) Forward LFM messages
5) Forward S2C messages
6) Simulate failure detection delay

Operations 1 to 5 are implemented according to section II, whereas failure detection time is simulated using self-messages. In the beginning a node schedules a self-message at time-of-failure. When this message is received, the port starts to drop any further packets and schedules another self-message at failure detection time. When failure detection time is reached, LFM process is initiated. We have estimated the failure detection time as $3 * \text{delay}_{\text{roundtrip}} * 1.25$ where, $\text{delay}_{\text{roundtrip}}$ is defined as $2 * \text{delay}$ for our simulation assuming that BFD (bidirectional forwarding detection) [23] is used for per-link failure detection. We have only considered propagation delay in our simulation to calculate link delay from geographical distance between nodes located at different cities. Delay is obtained from the expression d/c where, d is the distance in meters and c is the velocity of light $(3 * 10^8 ms^{-1})$.

In the simulations, we have used the *Pktgen* module to generate traffic from Victoria (see Fig. 6) whereas the SDN controller is located at Toronto. In all of them, packets are generated at random delays between 0.01-0.1ms.

C. Simulations performed

In the first set of simulations, we verify the robustness of our protocol by randomly selecting a failed link to see if traffic is restored properly. Here, we set the parameters: 1) number of packets to send, and 2) failure time of a link in milliseconds. We then observe traffic in the failed link, traffic in the recovery link and traffic received by the controller. Only a single link failure is considered in this first set of simulations.

In the second set of simulations, we observe the recovery time for a single failure. The objective is to find if the recovery time is bound to: 1) distance of the failed link from the controller 2) number of links in the ring or 3) delay of the failed link. In our topology, we have a total of four rings between Victoria and Toronto. Therefore, in this set, we perform four simulations where each simulation has a single failed link from one of the four rings. We then compare the failure recovery time with respect to distance from the controller, size of the ring in terms of number of nodes and delay of the links in the ring. To determine the recovery time we note the failure time (i.e. time when traffic falls to zero at controller) and restoration time (i.e. time when traffic rises above zero at controller). We subtract failure time from restoration time to find the recovery time.

In the third set of simulations, we initiate multiple simultaneous failures to observe the behavior of the control traffic. Here, recovery time due to multiple failures along the path is compared with recovery time from single failure. The links are selected as follows. For the first simulation, one link from the first ring is selected. Here, there is no multiple simultaneous failure because we consider only one link for the first simulation. For the second simulation, one link from the second ring is selected for individual failure. In this simulation, one ring from the first ring and one ring from the second ring is selected for multiple simultaneous failure. Similarly, for the third simulation, individual failure corresponds to one failed link from the third ring, whereas multiple simultaneous failure corresponds to one failed link from each of first three rings. Finally, for the fourth simulation, individual failure refers to one failed link from the fourth ring and simultaneous failure is realized by considering one failed ring from all four rings. We also record the failure time and restore time to see how an individual link failure affects the recovery time of simultaneous failure.

VI. Simulation Results

In this section, we show the results from the three sets of simulations described in the previous section. Fig. 7 depicts the result of the first set of simulations where we verify the effectiveness of the proposed protocol. We can see that: (a) the link failure caused drop in traffic at 10.4ms, (b) restored by the recovery link at 14.4ms and (c) there is no traffic at the controller until 16.4ms because the first packet from Victoria didn't reach the controller at Toronto until that time. The controller traffic drops at 26ms due to the failure and recovered at 29.8ms. Here recovery time is 3.8ms. It is notable from this experiment that, although the control channel latency is close to 16.4ms, recovery time is only 3.8ms. Thus, recovery time is not dependent on the control channel latency. If restoration based on controller were used, the failure recovery time would be multiple of 16.4ms.

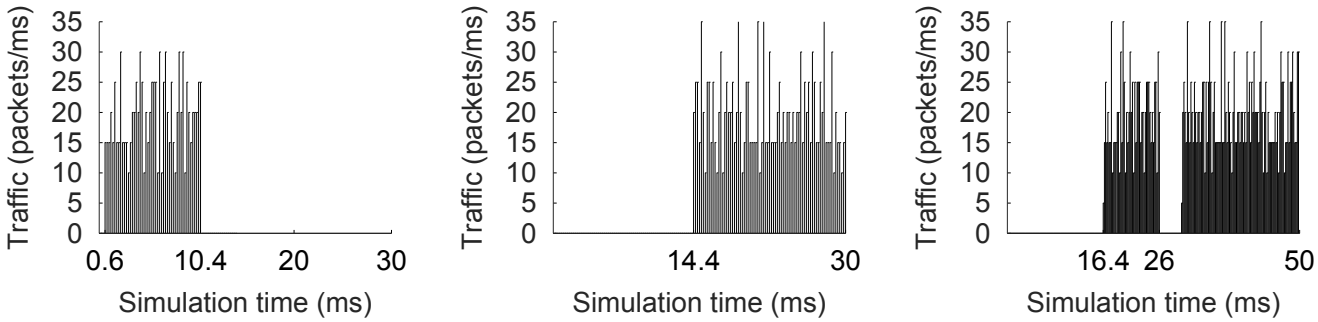
TABLE II
Failure and restore time

Link	Time (ms)			
	Simultaneous		Individual	
	Failure	Restore	Failure	Restore
1	55.9	59.8	55.9	59.8
2	55.6	79.5	55.6	79.4
3	49.6	80.5	49.6	73.7
4	43.3	80.6	43.3	54.5

The results from the second set of simulations, where we observe recovery time from single link failure by varying the failed link, is shown in Fig. 8. Here, we can see from Fig. 8a that the recovery time is increased with distance from the controller for the first three links, but decreased for the fourth link. Therefore, failure recovery time is not bound to the distance of the link from the controller. This is also clear from the first set of simulations. In Fig. 8b, we can see that failure recovery time is less for the ring with eight links than the ring with five links. Therefore, the recovery time can be lower for a ring with higher number of links. Thus, recovery time is not directly related to the number of links in the ring. However, in Fig. 8c, we can see that increased link delay (latency) always results in higher recovery time. This is because the latency of the link affects failure detection time.

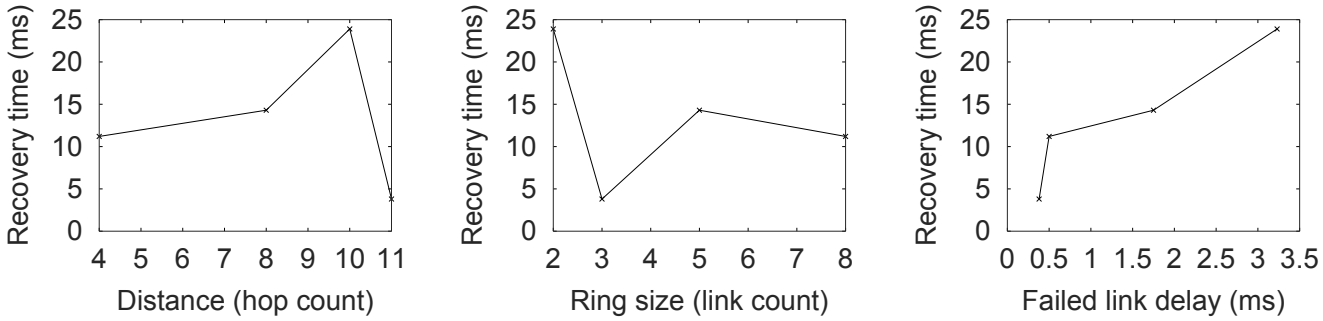
The results from the third set of simulations is given in Fig. 9, where we observe recovery time from multiple simultaneous link failures. Here we plot two lines. The first one shows recovery time due to multiple simultaneous link failure against number of failed links, whereas the second line shows recovery time for an individual single link. As we choose one link from each ring as the failed link, the link numbers correspond to the ring numbers. We can see that, the recovery time for three simultaneous failure is more than the recovery time when only the link from ring 3 is failed but the multiple failure recovery time is still less than 50ms. Thus, although recovery time is always higher for multiple failure than for individual failure our protocol can restore traffic within 50ms.

We list the failure time and restore time for both individual link failure and simultaneous link failure in Table II. It should be noted that, the actual failure time is same for all links. Table II lists the failure times as observed from the controller. We can see from Table II that, the time of control traffic failure gradually decreases as number of failed links increase. On the other hand, the time when control traffic is restored gradually increase increasing the gap between failure and restoration. Thus increased number of failed links result in overall rise in



(a) failed link (b) recovery link (c) controller

Fig. 7. Link failure at Victoria-Vancouver link at 10ms



(a) Distance from controller (b) Number of links in ring (c) Delay of failed link

Fig. 8. Failure recovery time evaluation

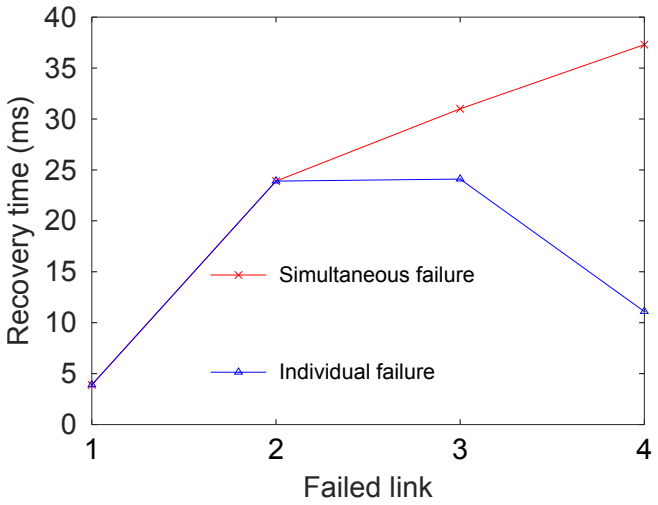


Fig. 9. Recovery time under multiple failure

recovery time.

VII. Discussion

A. Data plane simplicity and multiple failure recovery

In the proposed protocol, switches only forward packets based either on status of the port (e.g., LPM, LFM), or on label in the packet header. Therefore, complexity of traditional routing protocols (e.g., exchange of protocol messages, storage of network topology, calculation of spanning tree) is avoided. Our protocol also does not require any flow entry to forward control packets. Thus design objectives 6-7 (i.e., low data-plane complexity and low memory consumption) are achieved. The memory requirement of the proposed protocol for a single controller topology is as follows-

1) One bit Link Preclusion Message (LPM) status for each port
2) One bit Link Failure Message (LFM) status for each port
3) One bit Dead-End Notification (DEN) status for each port
4) One byte Ring Breaker (RB) status for each ring

We detect failure locally within the ring which is our design objective 3 (local failure detection). Recovery from multiple failures, as stipulated by design objective 5, is demonstrated through simulation.

B. Multi controller support

Our design is flexible enough to support multiple controllers. Firstly, controllers can use the label based forwarding to communicate with each other. Secondly, each controller can assign its own ring topology in the data plane for S2C communication. These ring topologies will operate independently of each other to enable switches to communicate with respective

controllers. Thus design objective 8 (supporting multiple SDN controllers) is realized.

C. SDN bootstrapping problem

Our approach completely eliminates the SDN bootstrapping problem mentioned by [8]. New switches added to the network will automatically discover the controller using CoD messages without requiring any manual intervention. Once the controller is aware of the switch, it first establishes the OpenFlow channel with the switch. Contrary to [10], we do not require DHCP (Dynamic Host Configuration Protocol) in the bootstrapping process because our C2S forwarding process can deliver control messages to switches without any IP address. After establishing the C2S channel, the controller is able to install further configuration and flow instructions into the switch. This realizes our design objective 9 (auto-configuration).

In a multi-controller network, only one controller will participate in the Controller Discovery (CoD) process, based on statically configured priority. Thus CoD messages do not need to distinguish controllers and auto-configuration process will work transparently to the switches.

D. SDN domain splitting problem

The SDN domain splitting problem, reported in [8], cannot be solved by our work itself. However, we believe that the controller placement problem should deal with this issue. With appropriate multiple controller placement, SDN domain splitting problem can be minimized. Moreover, an isolated domain should not be allowed to exist and networks should be designed to avoid such scenario.

VIII. Conclusion

In this paper, we have identified nine requirements for an effective control channel resilience protocol. Afterwards, we have proposed a protocol designed to meet those nine objectives and have performed simulations for a large real network topology to verify its efficacy. Simulation results show that our protocol is able to recover from multiple simultaneous failures. We also demonstrate through simulation that failure recovery times in our protocol are independent of the distance from the controller. Indeed, failure recovery times mainly depend on the packet delay of the failed link, for the proposed protocol. Finally, we find that multiple failure recovery times are greater compared to single failures, but overall recovery times are still always within 50ms for the simulated large real network topology.

In this paper, we did not simulate the label-based C2S communication and dead-end notification, which we envision as future work. We also plan to compare the performance of our method with other existing techniques in the extended version of this paper. Moreover, we intend to consider different network topologies, evaluate message overhead, evaluate processing delay and explore all combinations of possible link failures to verify the robustness of our protocol.

References

[1] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "Fast failure recovery for in-band OpenFlow networks," *Proc. IEEE International Conference on the Design of Reliable Communication Networks (DRCN)*, pp. 52–59, 2013.

[2] S. S. Lee, K.-Y. Li, K. Y. Chan, G.-H. Lai, and Y. C. Chung, "Software-based fast failure recovery for resilient OpenFlow networks," *Proc. IEEE International Workshop on Reliable Networks Design and Modeling (RNDM)*, pp. 194–200, 2015.

[3] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "In-band control, queuing, and failure recovery functionalities for OpenFlow," *IEEE Network*, vol. 30, no. 1, pp. 106–112, 2016.

[4] P. Fonseca and E. Mota, "A Survey on Fault Management in Software-Defined Networks," *IEEE Communications Surveys & Tutorials*, 2017.

[5] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[6] 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.

[7] Y. Hu, W. Wendong, G. Xiangyang, C. H. Liu, X. Que, and S. Cheng, "Control traffic protection in Software-Defined Networks," *Proc. IEEE Global Communications Conference (GLOBECOM)*, pp. 1878–1883, 2014.

[8] T. Watanabe, T. Omizo, T. Akiyama, and K. Iida, "Resilientflow: Deployments of distributed control channel maintenance modules to recover SDN from unexpected failures," *Proc. IEEE International Conference on the Design of Reliable Communication Networks (DRCN)*, pp. 211–218, 2015.

[9] P. Heise, F. Geyer, and R. Obermaisser, "Self-configuring deterministic network with in-band configuration channel," *Proc. International Conference on Software Defined Systems (SDS)*, pp. 162–167, 2017.

[10] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "Automatic bootstrapping of OpenFlow networks," *Proc. IEEE Workshop on Local & Metropolitan Area Networks (LANMAN)*, pp. 1–6, 2013.

[11] D. Kotani and Y. Okabe, "Fast Failure Detection of OpenFlow Channels," *Proc. of the ACM Asian Internet Engineering Conference*, pp. 32–39, 2015.

[12] N. Beheshti and Y. Zhang, "Fast failover for control traffic in Software-Defined Networks," *Proc. IEEE Global Communications Conference (GLOBECOM)*, pp. 2665–2670, 2012.

[13] S. Song, H. Park, B.-Y. Choi, T. Choi, and H. Zhu, "Control Path Management Framework for Enhancing Software-Defined Network (SDN) Reliability," *IEEE Transactions on Network and Service Management*, 2017.

[14] P. Thorat, R. Challa, S. M. Raza, D. S. Kim, and H. Choo, "Proactive failure recovery scheme for data traffic in software defined networks," *Proc. IEEE NetSoft Conference and Workshops (NetSoft)*, pp. 219–225, 2016.

[15] P. Goltsmann, M. Zitterbart, A. Hecker, and R. Bless, "Towards a Resilient In-Band SDN Control Channel," *Universität Tübingen*, 2017.

[16] B. Pfaff, B. Heller, D. Talayco, D. Erickson, G. Gibb, G. Appenzeller, J. Tourrilhes, J. Pettit, K. Yap, M. Casado *et al.*, "OpenFlow Switch Specification," 2009.

[17] J.-d. Ryoo, H. Long, Y. Yang, M. Holness, Z. Ahmad, and J. K. Rhee, "Ethernet ring protection for carrier ethernet networks," *IEEE Communications Magazine*, vol. 46, no. 9, 2008.

[18] C. Assi, M. Nurujjaman, S. Sebbah, and A. Khalil, "Optimal and Efficient Design of Ring Instances in Metro Ethernet Networks," *Journal of Lightwave Technology*, vol. 32, no. 22, pp. 4445–4455, 2014.

[19] M. Soliman, B. Nandy, I. Lambadaris, and P. Ashwood-Smith, "Source routed forwarding with software defined control, considerations and implications," *Proc. ACM conference on CoNEXT student workshop*, pp. 43–44, 2012.

[20] "Rogers fiber backbone map," *[Online] Available: http://www.rogers.com/enterprise/wholesale/*.

[21] A. Varga, "OMNeT++ simulation manual," *URL: https://omnetpp.org/doc/omnetpp/manual/*, 2016.

[22] "Distance calculator," *[Online] Available: https://www.distancecalculator.net/*.

[23] D. Katz and D. Ward, "Bidirectional Forwarding Detection (BFD)," *RFC-5880*, 2010.