

FAILURE RECOVERY IN SDN: A SEGMENT ROUTING BASED LINK PROTECTION APPROACH

by

MIRJA SHAHRIAR ENAN

Bachelor of Science in Computer Science & Engineering, BRAC University
Dhaka, Bangladesh, August 2015

A thesis

presented to Ryerson University

in partial fulfillment of the
requirements for the degree of
Master of Applied Science
in the program of
Computer Networks

Toronto, Ontario, Canada, 2018

© Mirja Shahriar Enan, 2018

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my thesis may be made electronically available to the public.

Failure Recovery in SDN: A Segment Routing Based Link Protection Approach

Master of Applied Science, 2018

Mirja Shahriar Enan

Computer Networks

Ryerson University

Abstract

The present computer network has been evolved into a complex structure with a growing challenge to manage and scale modern day's requirements. A new approach to tackle these difficulties is SDN, which empowers network with programmability and is designed to perform fine grained traffic forwarding decisions. However, similar to the need of traditional networks, fault tolerance is necessary to achieve high availability. In this thesis, we propose a link protection method based on the Segment Routing (SR) for rapid failure recovery in OpenFlow based SDN. Our proposed scheme performs local recovery at the switch level without the controller intervention, thus significantly reducing the total recovery time. Additionally, it reduces initial load on the controller while proactively computing the backup paths by minimizing the algorithm complexity. Moreover, memory efficiency is achieved by using a per-link protection with aggregated flow rules instead of traditional per-flow based protection mechanism. In Segment Routing, we may encounter the limitation on the size of the label stack, known as Segment List Depth (SLD). Therefore, we also propose an efficient label encoding algorithm to mitigate the SLD impact.

Acknowledgments

This work was suggested by Dr. Ngok-Wah Ma, Computer Networks Dept., Ryerson University, as a master's thesis. This document has been prepared as an effort to compile the knowledge obtained by me during my master's study and produce a final thesis, which innovatively addresses one of the issues of research in computer networking field, link protection in SDN.

I would like to express my gratitude to Almighty Allah (SWT) who gave me the opportunity, determination, strength and intelligence to complete my thesis.

I would like to thank my supervisor, Dr. Ngok-Wah Ma sincerely for his consistent supervision, guidance and unflinching encouragement in accomplishing this work.

Special thanks to the Computer Networks Department and Yeates School of Graduate studies at Ryerson University, for giving me this great opportunity and their financial support throughout my studies.

Last, but not the least, I would like to thank my parents, who constantly supported and encouraged me throughout the course of my studies.

Table of Contents

Abstract.....	iii
List of Tables	vii
List of Figures	viii
Chapter 1 : Introduction	1
1.1 Problem Statement	1
1.2 Research Objectives and Contributions	3
1.3 Thesis Outline.....	3
Chapter 2 : Background And Literature Survey	4
2.1 Software Defined Networking.....	4
2.1.1 Traditional Network vs. SDN.....	4
2.1.2 SDN: Definition and Benefits	5
2.1.3 SDN Architecture	6
2.2 Segment Routing	8
2.2.1 Overview and Segment Routing Concepts	8
2.2.2 How Segment Routing Works	10
2.2.3 Benefits of Segment Routing.....	12
2.3 OpenFlow	13
2.4 Fault Tolerance in SDN.....	14
2.4.1 Failure Detection	14
2.4.2 Restoration vs. Protection Methodology	14
2.4.3 Path Protection	16
2.4.4 Link Protection	17
2.5 Related Works.....	18
2.6 Summary of Proposed Approach.....	20
Chapter 3 : Proposed Methodology	22
3.1 Link Protection with Segment Routing	22
3.2 Efficient Segment Encoding Algorithm for Segment Routing.....	39
3.3 Chapter Summary	45
Chapter 4 : Implementation And Result Analysis.....	46
4.1 System Environment.....	46
4.2 Implementation	47
4.3 Experimental Results and Analysis	51

4.3.1 Failure Recovery Time	54
4.3.2 Memory Requirement.....	64
4.3.3 Segment Encoding Algorithm Comparison	68
4.4 Concluding Remarks	70
Chapter 5 : Conclusion And Future Work	71
5.1 Conclusion	71
5.2 Future Work.....	72
References.....	73

List of Tables

Table 4.3.1: Recovery time simulation results for Test Case 1 (P1 <-> P2 link failure).....	56
Table 4.3.2: Summary of recovery time simulation results	60
Table 4.3.3: Simulation environment specifications in previous works	61
Table 4.3.4: Advantage and Disadvantages of proposed segment encoding algorithm	69
Table 4.3.5: Simulation Topology and Recovery Time Summary	70

List of Figures

Figure 1.1.1: Failures in SDN architecture	1
Figure 2.1.1.1: Traditional Network vs. SDN	5
Figure 2.1.3.1: SDN Architecture	7
Figure 2.2.1.1: Segment Identifier Classification	9
Figure 2.2.2.1: IGP based forwarding.....	10
Figure 2.2.2.2: Nodal forwarding.....	11
Figure 2.2.2.3: Adjacency forwarding.....	11
Figure 2.2.2.4: Mixed forwarding	12
Figure 2.3.1: OpenFlow Switch Components	13
Figure 2.3.2: OpenFlow Table Example	13
Figure 2.4.3.1: Path Protection	16
Figure 2.4.4.1: Link Protection.....	17
Figure 3.1.1: OpenFlow Group Table.....	23
Figure 3.1.2: Flowchart for local failure recovery scheme	24
Figure 3.1.3: Reference Topology	26
Figure 3.1.4: MPLS Packet Header	28
Figure 3.1.5: Flow Table 0 of P1.....	28
Figure 3.1.6: Flow Table 1 of P1.....	28
Figure 3.1.7: Flow Table 17 of P1	29
Figure 3.1.8: Group Table of P1.....	30
Figure 3.1.9: Link Protection in Literature	31
Figure 3.1.10: Proposed destination-based packet detouring	31
Figure 3.1.11: Additional Flow Tables of P1	32
Figure 3.1.12: Reference Ring Topology	35
Figure 3.1.13(a): Example 1- P2<->P4 link fails.....	36
Figure 3.1.13(b): Example 1- P2<->P4 link fails (cont.)	37
Figure 3.1.14(a): Example 2- P4<->P6 link fails.....	38
Figure 3.1.14(b): Example 2- P4<->P6 link fails (cont.)	39
Figure 3.2.1: Example topology for SLD constraint	40
Figure 3.2.2: Flow rule for gSID	41
Figure 4.2.1: Python script to create custom topology in Mininet.....	48
Figure 4.2.2: Method to get topology information	49
Figure 4.2.3: Method to create Network Graph	49
Figure 4.2.4: Method to create Flow Tables	50
Figure 4.2.5: Method to create Flow Table 1.....	50
Figure 4.2.6: Method to create Flow Table 17.....	51
Figure 4.2.7: Bash file to install the flow rules.....	51
Figure 4.3.1: Flow rules in P1's Table 0	52
Figure 4.3.2: Flow rules in P1's Table 1	52
Figure 4.3.3: Flow rules in P1's Group Table	52

Figure 4.3.4: Successful ping from H1 to H6.....	53
Figure 4.3.5(a): ICMP request packet on P1's port 2	53
Figure 4.3.5(b): ICMP reply packet on P1's port 2	54
Figure 4.3.6: no packets on P1's port 3	54
Figure 4.3.7: P1<->P2 link failure - Capture on P6's port 2.....	55
Figure 4.3.8: P1<->P2 link failure - Capture on P6's port 3.....	56
Figure 4.3.9: P2<->P4 link failure - Capture on P6's port 2	57
Figure 4.3.10: P2<->P4 link failure - Capture on P6's port 3	57
Figure 4.3.11(a): Crankback routing on P2<->P4 link failure.....	58
Figure 4.3.11(b): Crankback routing on P2<->P4 link failure (cont.).....	58
Figure 4.3.12: P4<->P6 link failure - Capture on P2's port 2	59
Figure 4.3.13: P4<->P6 link failure - Capture on P6's port 3.....	59
Figure 4.3.14: Occurrence graph for recovery times with N=6, H=4	60
Figure 4.3.15(a): Topology used in [26] (N=5, h=1)	62
Figure 4.3.15(b): Topology used in [38] (N=6, h=1)	62
Figure 4.3.16: Average recovery time with the proposed approach in different topology (h=1)	62
Figure 4.3.17: Average recovery time comparison between [39] and proposed method.....	63
Figure 4.3.18(a): Comparison graph for number of flow entries per switch (h=1).....	65
Figure 4.3.18(b): Comparison graph for number of flow entries per switch (h=2).....	65
Figure 4.3.18(c): Comparison graph for number of flow entries per switch (h=4)	66
Figure 4.3.19(a): Trendline graph for total flow entries in network (h=1)	66
Figure 4.3.19(b): Trendline graph for total flow entries in network (h=4).....	67
Figure 4.3.20: Comparison graph for flow entries per switch (N=6).....	67
Figure 4.3.21: Trendline graph for total flow entries between [39] and proposed method	68

1 Chapter 1

Introduction

1.1 Problem Statement

Carrier-grade networks are adopting Software Defined Networking (SDN) technology for providing better network management capabilities compared to traditional networks by transforming network infrastructure from configurable to programmable [1], [2]. However, from widespread legacy internet to newly emerging SDN, no matter what type of network architecture it is, network failures are inevitable. For an effective adoption of SDN, it must address the resiliency challenges since a failure can seriously influence the network performance and result in service inaccessibility. In traditional networking, routing protocols can help the automatic convergence of the whole network, whereas SDN controllers were not initially developed with the ability of fault tolerance.

Failures can be categorized into three areas in an SDN based network, (1) Data plane, where a switch or a link between two switches fail, (2) control plane, where link connecting controller and switch fails, (3) controller itself fails. Figure 1.1 illustrates these three types of failure.

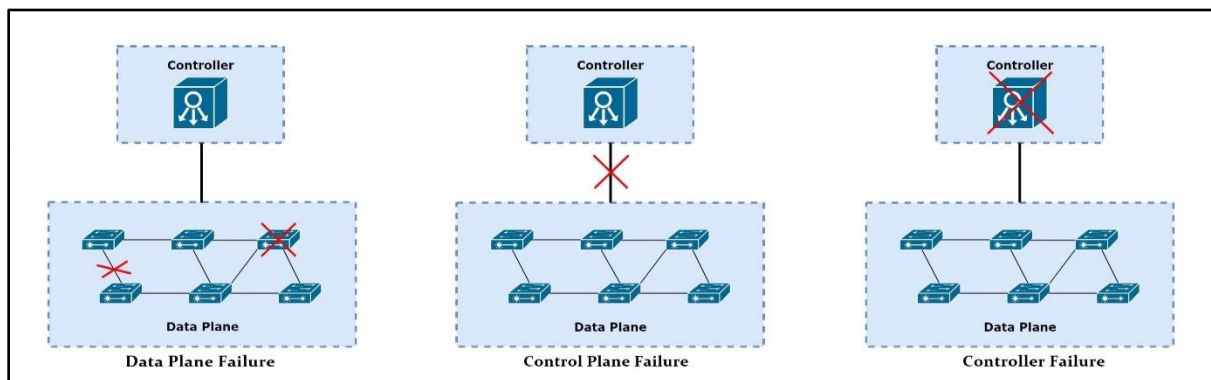


Figure 1.1.1: Failures in SDN architecture

For control plane and controller failure, there are mechanisms in place, such as redundant management link and backup/dual master-slave controller structure, respectively [3]. These are out of the scope of this thesis and we focus ourselves on the data plane recovery as it is possible to propagate data packets even though the control plane meets a failure by careful design of recovery mechanism.

As SDN architecture implies, the conventional method for failure recovery needs controller intervention. When a link fails, the switch must inform the controller and upon receiving the exception, the controller needs to reconfigure the network to restore end-to-end connectivity for all paths by modifying forwarding behaviors of relevant switches. In this method, a round trip delay between control and data plane is inevitable and when added to this the failure detection time, it is almost impossible to meet the carrier grade recovery time requirement, which is 50 ms [4]. A way around of this could be detouring of packets locally at the switch. However, to make this work, backup paths should be pre-configured into the switches. This poses another problem as the current switch hardware limits the TCAM memory to store flow entries. Segment routing provides a platform to address these problems. The use of segment routing, however, exposes to Segment List Depth (SLD) constraint. Current OpenFlow switches have a limitation on the number of labels they can push onto a packet header, thus reduces the number of paths potentially operable in the network, leading to poor network resource utilization.

So our research problem is twofold – main focus will be on designing an efficient local link failure recovery mechanism in terms of recovery time and memory usage, and a label encoding algorithm for segment routing SLD constraint.

1.2 Research Objectives and Contributions

- Implementing segment routing in an OpenFlow based SDN network.
- Design link protection algorithm for single link failure.
- Propose flow aggregation to minimize the number of flow entries.
- Design SLD mitigation scheme by fragmenting Segment Routing Path (SRP) using Group Segment Identifiers (gSIDs).

1.3 Thesis Outline

This report consists of 5 chapters with this being chapter one. The rest of this paper is organized as follows:

Chapter 2 introduces SDN and its architecture, followed by an overview of Segment Routing and OpenFlow. Various failure recovery methods in SDN are briefly described. The chapter ends with a summary of our proposed approach.

Chapter 3 describes the details of our link protection scheme with segment routing. It includes link failure scenarios, recovery algorithm, flow table structure, flow aggregation technique and algorithm for encoding label stack.

Chapter 4 presents the implementation of the proposed methods, with testbed setup, recovery time and total number of flow entries calculation, experimental results, analysis and comparison with existing approaches.

Chapter 5 concludes the thesis and introduces some future work.

2 Chapter 2

Background and Literature Survey

2.1 Software Defined Networking

Innovation in IT such as virtualization in computing, storage and networking has led to excessively complicated and inflexible networks, which can no longer meet the current business requirements. Software Defined Networking (SDN) [5] is swiftly becoming the new paradigm in the networking field with an expectation to overcome the limitations in traditional networking.

2.1.1 Traditional Network vs. SDN

Traditional networking consists of specific network devices that are used for certain tasks. Different networking vendors sold these network devices and often uses proprietary hardware. Primary configuration for most of these devices must be done manually through command line interface (CLI) or graphical user interface (GUI). Different types of network devices have different functions to perform. The network architecture consists of three main components or planes. They are:

- Control Plane
- Data Plane
- Management Plane

Control plane is responsible for tasks such as learning and build switch MAC address tables, create ARP table, running STP, running routing protocols and building routing table by exchanging routing information. **Data plane** is responsible to perform traffic forwarding based on the information that it gets from the control plane. Packet encapsulation, adding and removing header, matching IP and MAC address etc. are some of the complimentary tasks that are done at data plane. Specialized hardware like ASICs and TCAM tables are used to perform these tasks as fast as possible.

Management plane is used for access and management of network devices, such as accessing the device through Telnet, SSH or the console port.

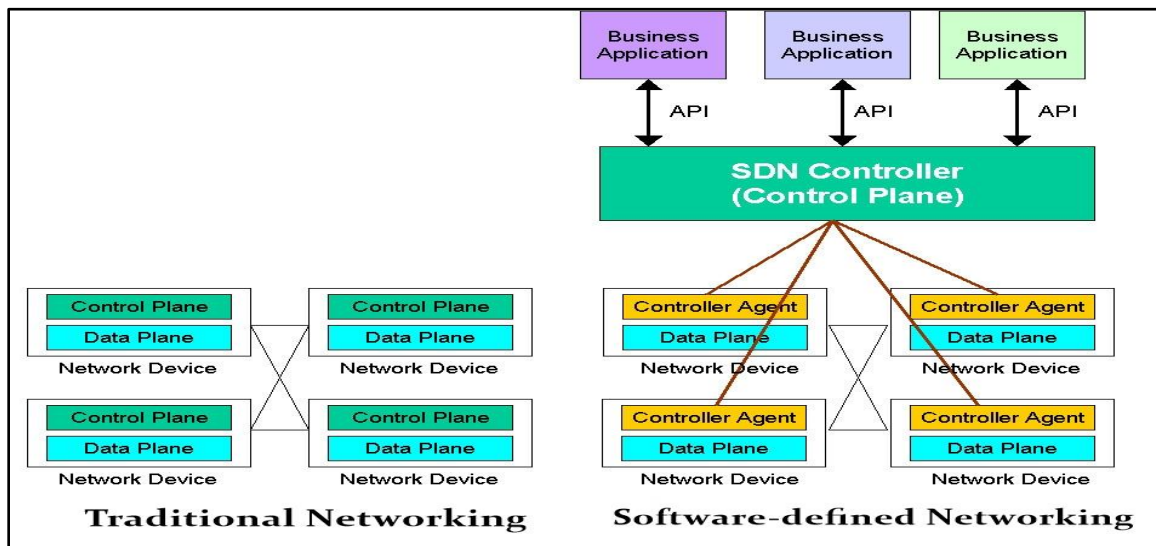


Figure 2.1.1.1: Traditional Network vs. SDN

Because of the distributed model for the control plane in traditional networks, protocols like STP, OSPF, BGP and others run separately on each network device. These network devices communicate with each other, but there is no central device that has a complete view of the entire network. With SDN, we use a central controller for the control plane, which has insight of all network devices in the network. The SDN controller could be a physical hardware device or a virtual machine. So, whereas the three planes are merely a logical concept in traditional network, SDN can separate them into actual devices.

2.1.2 SDN: Definition and Benefits

Open Networking Foundation defined SDN as - “An emerging network architecture where the control and data planes are decoupled, network intelligence and state are logically centralized in software-based SDN controllers, which maintain a global view of the network, and the underlying network infrastructure is abstracted from the applications” [6]. Being an architectural concept, not a specific product, SDN has originally thought of as to designing, building, and managing the networks that separates the network’s control and forwarding planes to virtualize data center

networks. This approach enables the network to be intelligent and centrally controlled or programmed, using software applications, thus helping the admins to manage the entire network consistently and reliably, regardless of the underlying network technology.

SDN enables network behavior to be controlled by the software that exists outside of the routers and switches. It increases **network programmability** and operators can tailor the behavior of their networks to support new services. As SDN is built on logically centralized network topologies, it enables **intelligent control and management** of network resources such as bandwidth management, restoration, security, and policies which can be highly intelligent and optimized. SDN also provides an **abstraction of the network**. Services and applications are abstracted from the underlying hardware and applications will interact with the network through APIs. **Openness** is another benefit of SDN, enabling multi-vendor interoperability as well as promoting a vendor neutral environment.

2.1.3 SDN Architecture

The splitting of the control and data forwarding functions can be denoted as desegregation, because these two can be obtained separately, rather than deployed as one integrated system. The three-stack architecture of SDN implies this with components such as:

Infrastructure layer is composed of various networking devices which forms underlying network to forward data traffic. It could be a set of physical or virtual devices and may be implemented in hardware or software. These devices are combination of ports, queues, memory, CPU etc. and upon receiving data packets on their ports and they may forward, discard or alter them according to the installed instructions. A network device contains a Forwarding, or Data Plane and Operational Plane. Data plane incorporates the resources that deal directly with customer traffic and executes forwarding decisions made in the controller plane. The Operational plane

deals with the operational state of a device such as the status of the ports, number of packets transmitted/received over an interface, memory utilization etc.

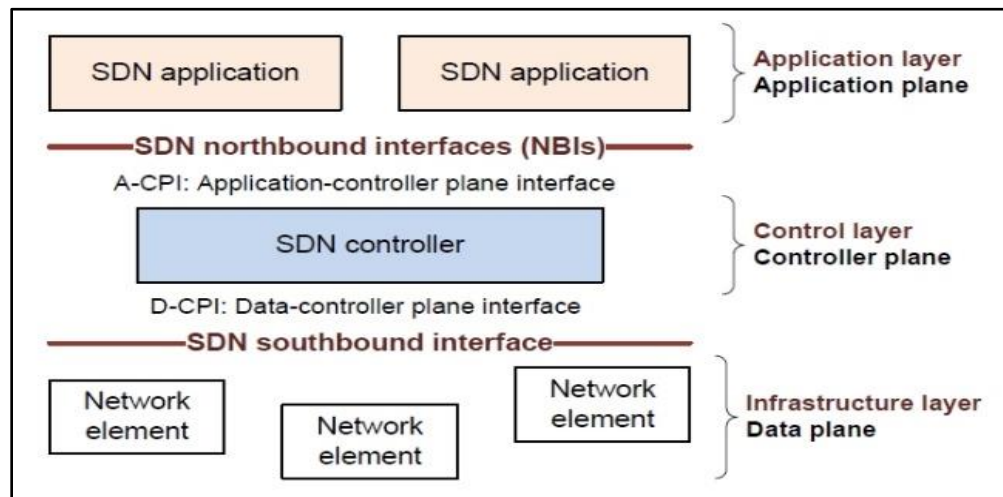


Figure 2.1.3.1: SDN Architecture

Control layer is where the SDN controller would reside to control network infrastructure. The SDN controller comprises of control plane and Management plane. The control plane works as a brain of the network, which provides various services such as inventory management, topology management, Flow programming, path selection etc. To communicate with the forwarding plane in the infrastructure layer, control plane uses a southbound interface with southbound protocols such as – OpenFlow. The Management plane is the central point to collect data from the operational plane of the network devices and provide various functionalities such as orchestration and fault management, using its own southbound interfaces such as - Netconf, Ovsdb etc.

SDN Applications, business Applications, cloud orchestration, etc. reside on the application layer. This is an open area to develop as much innovative application as possible by leveraging all the network information to solve today's business problems, such as - network automation, configuration, management, monitoring, troubleshooting etc. network policies and security. Control layer communicates with application layer using its northbound interfaces like RESTful APIs or RPC.

2.2 Segment Routing

Routing is the process of selecting the best path through the network for incoming data flows and destination-based routing is commonly used in traditional IP networks. In contrast of traditional destination-based routing, source routing is a method where ingress node specifies the route that a packet should take through the network. Segment routing is based on source routing technique as the IETF spring group stated it as “*Segment Routing (SR) leverages the source routing paradigm.*” [7]

2.2.1 Overview and Segment Routing Concepts

An adaptable and versatile method of doing source routing could be Segment Routing (SR). Route selection and encoding the selected path in the packet header as an ordered list of segments is done by the source router. Each segment is distinguished by a segment ID (SID) comprising of a flat unsigned 32-bit integer, and they are the identifier for any type of instruction. With segment routing, each intermediate node follows the forwarding instructions provided in the packet header instead of retaining a per-application or per-flow state. Lightweight extensions to routing protocols such as IGP, BGP and PCEP is sufficient to enable segment routing in a network. SR can operate with an MPLS or IPv6 architecture and it can be implemented without changing the forwarding plane in the existing MPLS architecture [7]. Each segment is treated as an MPLS label and list of the segments are equivalent to MPLS label stack. The currently processing segment is on the top of the stack and after the completion of the segment, the related label is popped from the stack and packets are forwarded accordingly. With a new extension in the routing header, SR can also on IPv6 architecture. Segment list is represented as an ordered list of IPv6 addresses in the header denote the list of segments of encoded path. Destination address indicates the currently active segment and the next active segment is specified by a pointer in the header. After the completion of a segment the pointer is incremented.

Segment: A segment resembles an instruction that a node executes on the inbound packet. There could be varying instructions such as forward the packet to

another node by IGP shortest path, forward through a specific port, or deliver the packet to a service [7]. Segments are identified by Segment Identifier (SID), which is a 32 bits label if applied in MPLS environment [8].

Segment Identifier (SID): Each segment in SR domain has an identifier that is distributed in the whole domain using the newly introduced extension of IGP such as OSPF and IS-IS. These extensions can work in both IP or IPv6 architecture. Global segments are those that every node in the domain is aware of and must be unique in the domain, whereas local segments are only known by a node itself. As IGP is used to distribute the segments, devices in a SR domain do not need any additional protocols such as LDP or RSVP-TE that is necessary in an MPLS network. Figure 2.2.1.1 illustrates the classification of segment identifiers:

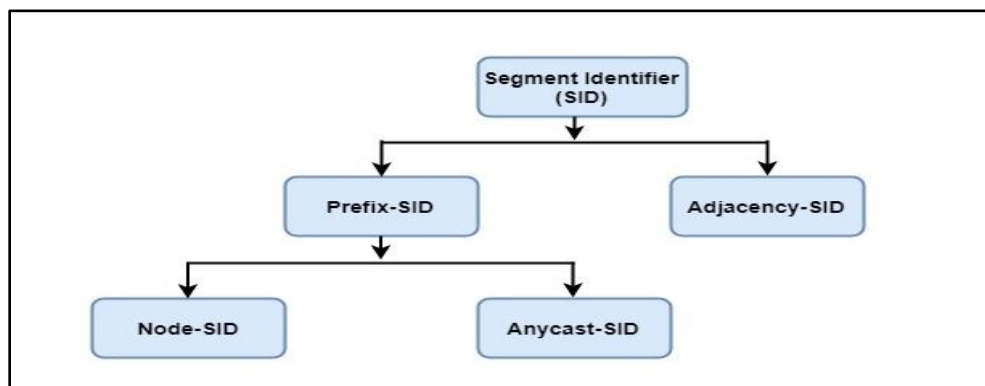


Figure 2.2.1.1: Segment Identifier Classification

A **Prefix-SID** is a unique identifier within the SR domain and it is a global identifier. A unique pool called the Segment Routing Global Block (SRGB) is used to assign the SID values. These SIDs denotes a multi-hop ECMP-aware shortest-path to the related prefix. A **Node-SID** is a commonly used sub type of Prefix-SID to identify a device, typically by using its loopback address. Another generally used type of Prefix-SID is **Anycast-SID**, which is used to specify a set of devices and represents the shortest path to go to the closest node of the anycast set.

An **Adj-SID** is a segment that points to a link connecting to another device in the SR domain. A device maintains Adj-SID for its neighbors and it has local significance,

which means any other device can use the same values for its Adj-SID. The devices usually assign Adj-SID dynamically to all its interfaces and the value is chosen outside of the SRGB. This SID is especially useful to steer the packet through SR domain.

Actions: Three types of actions can be taken to forward a packet in SR domain. **Push** action is applied to push a segment on the top of the segment stack. **Next** action denotes that the active segment is completed and remove it from the stack. To indicate the current segment is not completed yet **Continue** operation is executed. We will look into the details of these operations in the next sub-section.

2.2.2 How Segment Routing Works

The following four figures illustrate an SR network applied in an eight routers MPLS architecture, while six of them (P1-P6) being core routers and two (PE1, PE2) are customer edge routers. A segment identifier range of 100 to 199 for P routers' node SIDs, 600-700 for PE routers, and 1000-1099 for adjacency SIDs has been assigned.

As the example shows in Figure 2.2.2.1, PE2 advertises a Node-SID of 650 to all other routers in SR domain. When PE1 wants to send a packet to PE2, it pushes the Node-SID {650} and forwards the packet using its IGP shortest-path towards PE2.

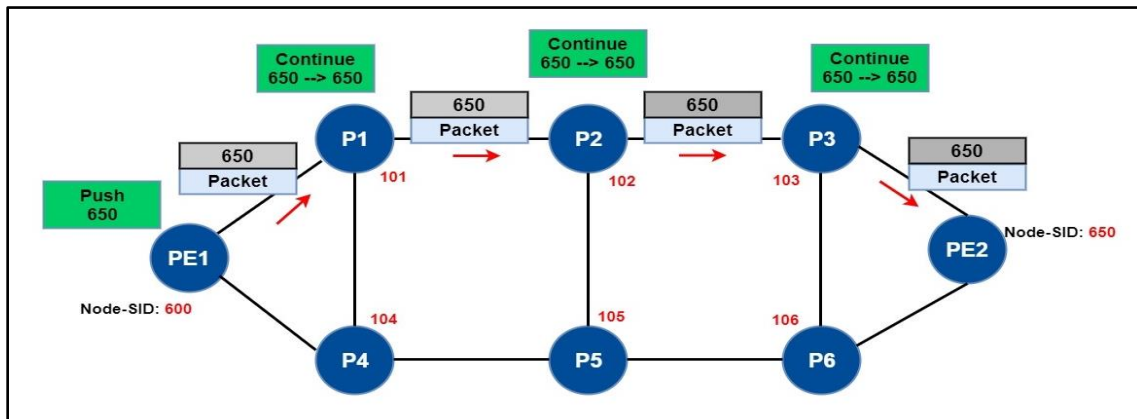


Figure 2.2.2.1: IGP based forwarding

Routers P1, P2 and P3 performs continue operation (no SID change) and forwards the packet to PE2. PE2 pops the SID and process the packet. Alternatively, PE2 can instruct P3 to perform 'next' operation instead of 'continue'. This means P3 will pop the SID as

it is the second last hop to PE2. This action of P3 is similar as penultimate hop popping (PHP) in MPLS. It's an optional behavior and we did not implement PHP in our work.

Figure 2.2.2.2 demonstrates a Nodal Forwarding, which is similar as loose source routing. PE1 pushes SIDs {105, 102, 650} and packet takes the shortest path to go to P5. Then P5 sends the packet to P2 because of the SID 102, though its shortest path to PE2 is through P6. Then from P2 packet again follows the shortest path towards PE2.

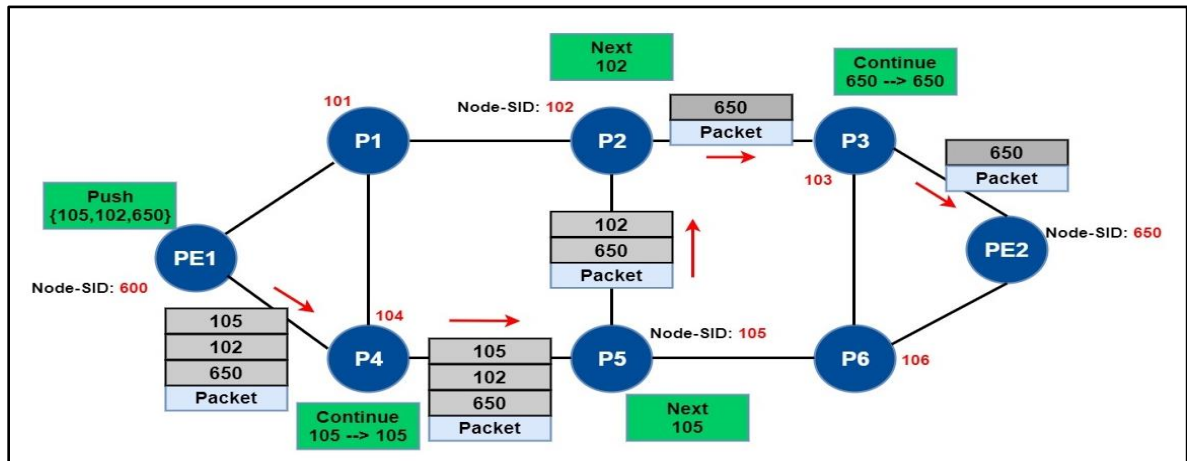


Figure 2.2.2.2: Nodal forwarding

Adjacency forwarding is shown in figure 2.2.2.3 where PE1 pushes all the adjacency SIDs to steer the packet. This is similar as strict source routing and packet follows exactly the path PE1→P1→P4→P5→P6→P3→PE2.

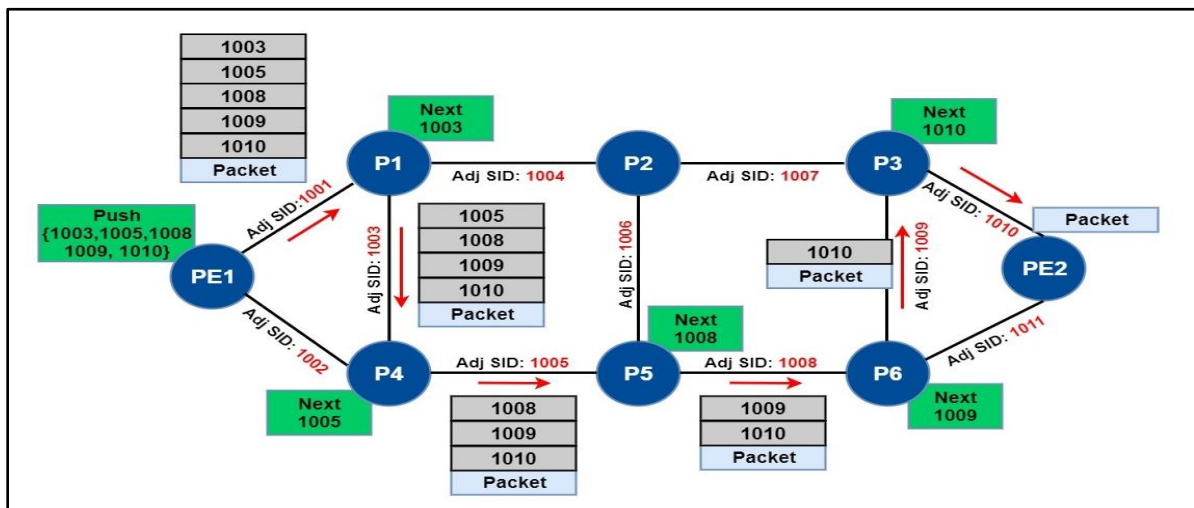


Figure 2.2.2.3: Adjacency forwarding

Nodal and adjacency forwarding can be mixed together to steer a packet through SR domain, which is depicted in figure 2.2.2.4. PE1 pushes SIDs {102,1006,700} so packet takes the IGP shortest path to P2. Then P2 is forced to send packet on its P2-P5 interface with the Adj-SID 1006 and from P5, packet again takes the IGP shortest path to PE2.

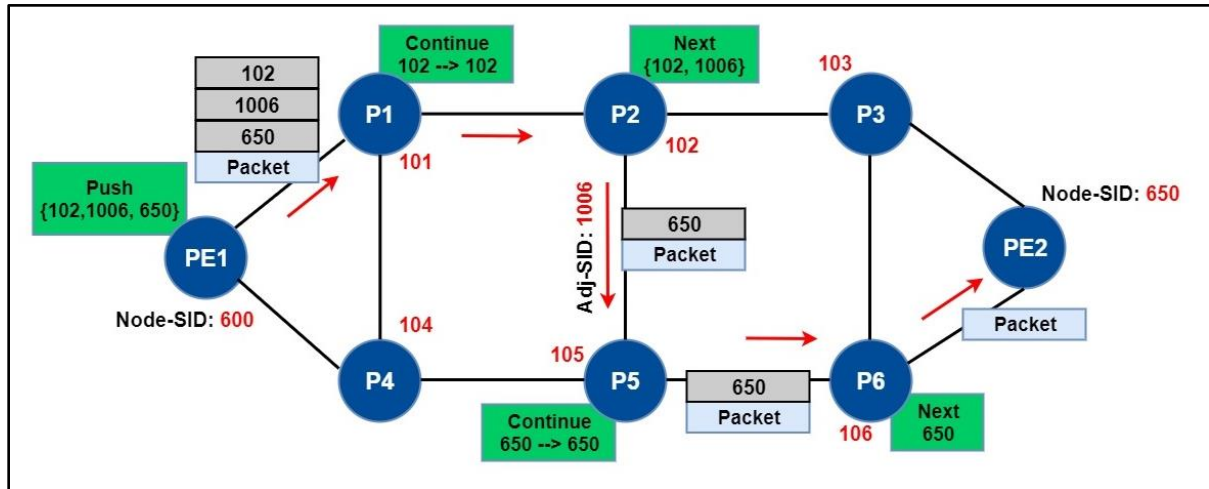


Figure 2.2.2.4: Mixed forwarding

2.2.3 Benefits of Segment Routing

- Segment Routing is developed to encompass SDN distributed intelligence, such as - link and node protection, traffic optimization etc.
- Simplifies MPLS architecture, providing the capability to tunnel MPLS services with only IGP.
- Separate label distribution protocol is not needed.
- Network infrastructures are better utilized.
- Provides TI-LFA (Topology Independent Loop Free Alternate) with 100% coverage.
- Scalable as it avoids thousands of labels in LDP database traffic engineering LSP's in the network.

2.3 OpenFlow

OpenFlow [9] is a communication protocol to communicate between the control and infrastructure layers of an SDN architecture [6]. It is the first open interface to be standardized by Open Networking Foundation (ONF), which allows the SDN controller to directly access and manipulate the forwarding plane of network devices. In OpenFlow, pre-defined match rules are used to match flows to categorize network traffic. These match rules are dynamically installed by the SDN controller onto the device and as it is programmed on a per-flow basis, a more granular control is obtained in an OpenFlow based SDN environment.

A fundamental component of the SDN infrastructure layer is OpenFlow switch. Initially they communicate with controller about their presence, thus SDN controller learn the whole topology and in case of a network change event, they also notify the controller using asynchronous messages. Figure 2.3.1 and 2.3.2 illustrate the components of an OpenFlow switch and the table respectively. Each entry in the flow table has some **Match fields** such as source and destination MAC and IP, VLAN ID, MPLS label etc., **Priority** - higher number means greater precedence over other flows, **Counters** – how many packets matches that flow, **Instructions** such as forward to a port/controller, drop etc., **Timeout** – after what time that flow is expired and deleted from the table.

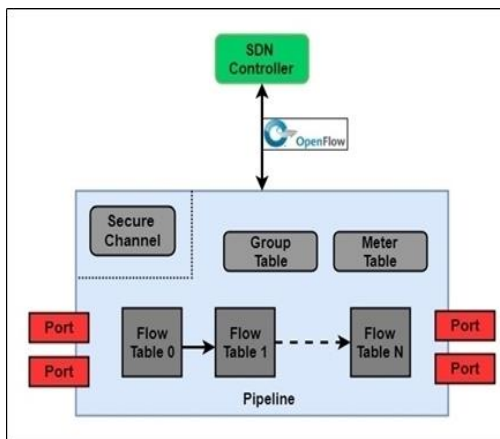


Figure 2.3.1: OF Switch Components

The diagram shows an 'SDN Controller' (green box) connected to a switch via an 'OpenFlow' protocol (blue box with a circular arrow). Below the connection is a table representing an OpenFlow table.

src MAC	dst MAC	src IP	dst IP	TCP dport	...	Action	Count
*	AA:FF:C1:51	*	*	*	*	port 1	150
*	*	*	10.0.2.2	*	*	port 2	320
*	*	*	*	25	*	drop	765
*	*	*	192.*	*	*	local	235
*	*	*	*	*	*	controller	55

Figure 2.3.2: OpenFlow Table Example

Upon receiving a packet, it tries to match the packet's header information against the entries in the flow tables, starting from flow table 0 to N, depending on how the SDN controller programmed it. If it finds a match, then the respective action is executed, otherwise they send the packet to the controller using OFPT_PACKET_IN message [10].

2.4 Fault Tolerance in SDN

General focus in current SDN approaches is on the controls and automation of the networks and there is a lack of design development of a reliable and fault-tolerant SDN environment. The following subsections discuss in details of SDN fault tolerance mechanisms in place, including detecting the failure and different recovery approaches.

2.4.1 Failure Detection

A failure must be detected before a switch can communicate with the controller for backup path and the controller initiates path recovery. For each network layer, constraints are in place to detect link failure depending on the network interface. Although the current OpenFlow operations are mostly based on Ethernet, but this is not designed for high failure detection requirements. When there is no session active, the physical layer sends periodic heartbeats (16 ± 8 ms) over the link and if the interface does not receive a response on the heartbeats within a set interval of 50–150 ms, the link is presumed disconnected [11]. Moving to the higher network protocols, there are multiple failure detection mechanism exists, such as STP or rapid STP [12], as they are designed to maintain the network distribution tree by updating port status in switches. However, their detection windows are in the order of the seconds and cannot meet the sub 50 ms requirement. Instead, we will use Bidirectional Forwarding Detection (BFD) [13] protocol, which has a proven capability of detecting failures within the sub 50 ms window.

2.4.2 Restoration vs. Protection Methodology

Two basic mechanisms to mitigate failures in SDN are restoration and protection [14]. In restoration, controller intervention is needed as after detecting a failure,

switches send notifications to the controller and upon receiving the notification the controller needs to modify the flow rules in switches to divert the packet into the new path. We cannot avoid the round trip between the controller and switches in restoration approach, thus making it difficult to meet the sub 50 ms requirement. The recovery time with restoration approach increases as the network size increases because the failure notification needs more time to reach the controller. Also, with a larger network, more flows are affected, and more time is required for the controller to find new paths and modify flow rules. Moreover, implementing restoration approach in an in-band network seems infeasible. Data and control messages share the same transmission channel in an in-band network. So when a link fails, the controller may not be able to reach the other end of the failed link and modify that node. Also, the control message to notify the controller about the failure must go through other network devices in a hop by hop manner. As a result, failure detection time increases because now it depends on the number of intermediate switches.

In protection scheme, the controller calculates both working and backup paths initially and install the flows into relevant switches. When the data plane meets a failure, switches can forward packets along the backup path automatically and locally without any controller intervention. This approach is suitable for any network scale and recovery time would be less than restoration approach by avoiding round trip time to the controller. However, the downside of this approach could be memory requirement. Forwarding rules in an OpenFlow switch are stored in TCAM which can only support a few thousand rules [15]. Flow rules in TCAM can be categorized for two types of paths, working path and backup path. The working path flow rules will be same for both restoration and protection, given that same path computation is used (e.g. - shortest path algorithm). Flow rules for the backup path is used in protection scheme only. For example, according to a simulation shown in [16], a network comprises of 100 firewalls and 100 IDS, when load balancing is enabled in the network, a switch in the center may need around 10,000 flow rules, which most of today's switch are not capable to install.

Besides, TCAMs are 400 times more expensive and consumes 100 times more power per Mbit than RAM based storage [17].

2.4.3 Path Protection

Path protection maintains two types of path for an LSP - a primary path, used in normal operations and a backup path, used when there is a failure anywhere in the primary path. With this approach, an end-to-end failure recovery can be ensured. Figure 2.4.3.1 illustrates this concept. The primary path PE1-P1-P2-PE2 is protected by backup path PE1-P3-P4-PE2. This backup path can be pre-sigaled, which requires more resources but faster recovery, or the backup path could be pre-configured, not pre-sigaled, in which case resources are not held up by backup path but recovery time will increase. In case of pre-sigaled path, it can be categorized into two groups [18]. A 1+1 path protection means both the primary and backup paths are used simultaneously and the selection is made at destination node. Upon a failure, destination node switchovers to reading data from backup path. With this scheme, source node needs to create copies of packets which reduce transmission rate and increase traffic in the network. Alternatively, the backup path could be ready for immediate use, but it remains idle as no data is sent on this path. This is known as 1:1 path protection. When a failure occurs, a notification is required to the ingress node so that it can switch the traffic to the backup path.

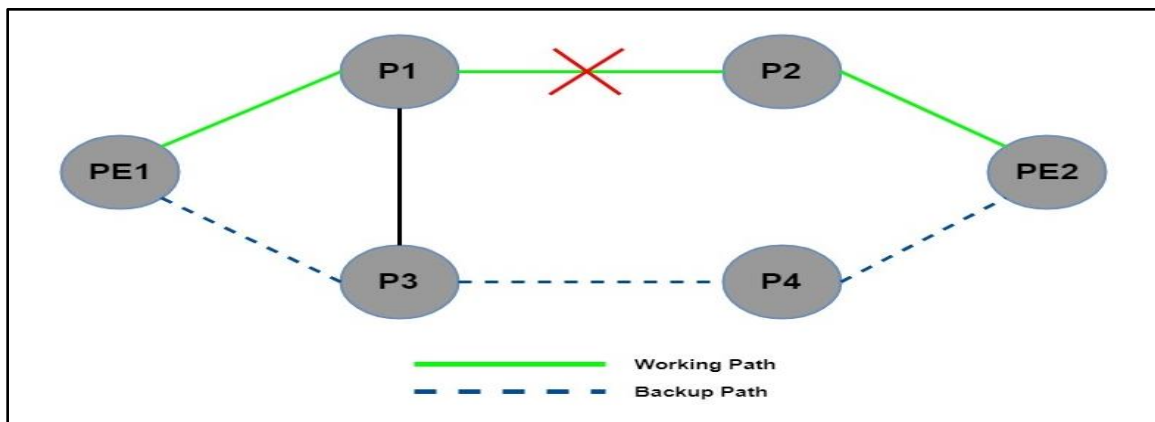


Figure 2.4.3.1: Path Protection

2.4.4 Link Protection

As the name suggests, link protection approach focuses on recovery from an adjacent link or node failure instead of end-to-end path protection. It ensures that the traffic flow is not disrupted while going to a neighboring node over a specific link when that link fails. A backup path to go to the same destination is created to detour the traffic in case of failure by using a different interface and path. The node associated with the failed link initiates the detouring in contrast to path protection where an ingress node needs to switch traffic to the backup path. Figure 2.4.4.1 shows an example of link protection. The link between P1 and P2 is protected by alternate path P1-P3-P4-PE2. In case of P1-P2 link or P2 node failure P1 can initiate the detouring and send traffic through P1-P3 interface.

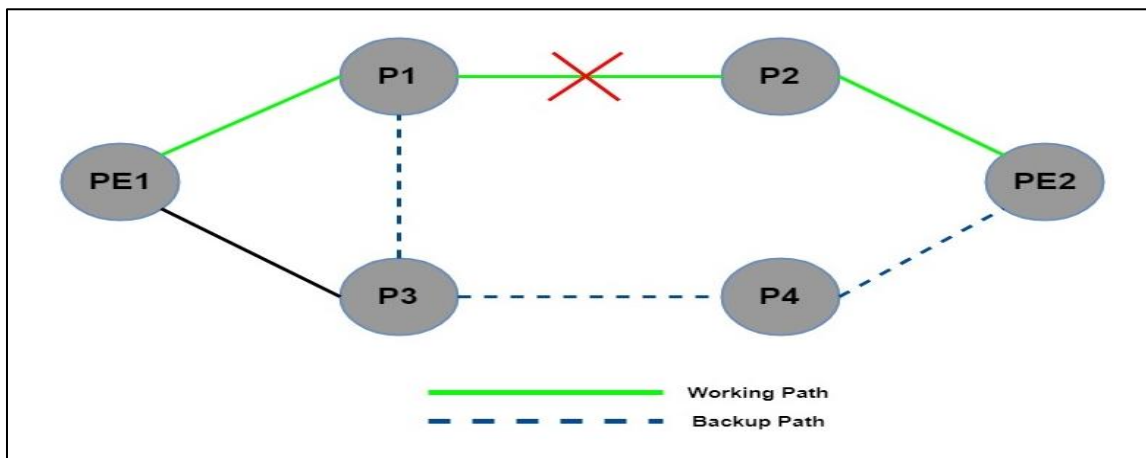


Figure 2.4.4.1: Link Protection

According to [19], there are two methods for local link recovery. First one is “*one-to-one backup*” method. In this method a backup path is created for each link in the primary path. From the figure 2.4.4.1, to protect 3 links in the primary path (PE1<->P1; P1<->P2; P2<->PE2), three backup paths will be created. Second one is the “*facility backup*” method where a single path serves multiple links of a primary path. It is called a **bypass tunnel**. From the example, the P3<->P4 link is a bypass tunnel as it can be shared by all the three links from the primary path in case of a link failure. So instead of creating three Label Switched Path (LSP) as per the first method, here only one LSP could serve the purpose.

2.5 Related Works

Currently there are different approaches in literature to mitigate data plane failure. A *path restoration* method is proposed in [20] and [21] which requires controller intervention and achieves recovery in approximately 200 ms. They improve the fast path restoration approach by introducing “predetermined restoration” where backup paths are pre-computed with priorities and the controller can choose one from those in case of failures. However, their approach suffers from the shortcomings of restoration method (described in section 2.4.2) and the authors reckon that the controller can be overburdened with recovery requests in a big network. The work in [22], [23] propose a liveliness monitoring by introducing logical group ports in transport networks based on OpenFlow. An interval of 1.25 ms between BFD liveliness packets is counted as an interruption of end-to-end BFD session and then failover is performed. They need to customize OpenFlow v1.0 for this and achieved a recovery time of 28.1 ± 0.5 ms. Again, the controller is involved here and there will be a packet overhead as frequent probe packets are sent to monitor each of the established flow. A later work by the same authors from [20] presented path-protection based restoration approach. They manage to improve the recovery time to 100 ms by replacing the controller-initiated recovery with a path failure detection using BFD, similar to the one proposed in [22]. Yet, the fast failure detection depends on switch implementation of BFD which is lower bounded by the protected path end points [24] and suffers from problems of path protection that we discussed in section 2.4.3.

Sgambelluri *et al.* [25] introduced ring topologies in ethernet networks and change to failover forwarding rules in the event of failure. Same authors perform segment protection in ethernet network by using auto-reject mechanism to remove flows [26]. Both the work needs enhancement in OpenFlow to automatically delete the flows associated with the failed link so that disrupted flows follow the low priority rules and their result shows an average recovery time of 32.74 ± 4.17 ms. Their enhancement to OpenFlow is not standardized which create problem to implement this approach seamlessly in current network environment.

In [27], the authors proposed a multipath rerouting method by computing a Mixed-Integer Linear Programming (MILP) formulation to pre-plan recovery paths based on QoS metrics. OpenState is used to monitor the port status of switches and when a port is down, reallocate flows to new paths. By planning multiple backup paths at every hop and splitting flows among them can significantly improve the segment protection and solve the link congestion problem. However, with this method, static planning is needed and there will be too many flow entries to build multiple backup paths. Also, the dependence of this approach on crank back routing may result in longer backup paths and redundant usage of links. Their follow-up work in [28] uses MPLS tags for failure rerouting mechanism. The proposed system relies on OpenState pipeline [29], which requires pipelining of four flow tables in each switch to achieve tailored failure detection and packet detouring. Thus, incompatibility issue with existing networks.

Robustness can be achieved using a centralized controller consecutively with a dispersed routing protocol. This is the method proposed in [30] as according to their proposal, initially all traffic is forwarded according to the controller's configuration and when a link is down, adjacent switches revert to the path determined by the traditional routing protocol. They overcome the problems of crank back routing like the previous work by using a custom local monitoring agent which can detect crankback paths. Downside of this approach is hardware customization is needed, as now switches need to run a routing protocol and implement a local agent along with connecting to the controller. Moreover, the convergence time of the routing protocol could significantly increase the post-failure recovery time, thus the approach may not be able to outperform pre-planned backup path.

Stephens *et al.* [31], [32] were able to dynamically merge the flow entries for new incoming flow with the existing ones by developing a forwarding table compression algorithm. Such dynamic procedure may increase the burden on the controller to configure the primary as well as the alternate path for every incoming flow. As the number of incoming flows in the network increases, the number of alternate path flow

rules also increase. Therefore, the amount of TCAM resource will be used cannot be predicted, as the compression method depends on the number of incoming flows.

Loop-Free Alternates (LFA) concept from IP networks is adopted to SDN in the method proposed in [33], where single-link backup rules are pre-installed on the nodes avoiding loops. Although full protection using this approach requires topological adaptations, meaning it is not topology independent. Link layer discovery protocol (LLDP) messages are used in CORONET to detect link failures [34]. However, to monitor the network continuously using LLDP messages may result in increase of failure detection time and overload the controller. Raeisi *et al.* [35] proposed a method to detect link failures by polling packets using a centralized polling mechanism and showed how the interval between polling packets affected the recovery time. However, a centralized failure detection mechanism will congest the network with probe packets and interfere with data and control traffic.

Authors of [36] came up with a novel design to reduce the number of flow entries required while building the backup paths. They abstract the SDN network into working plane, where flows are transmitted in normal network state, and transient plane consists of sharable links. As a result, number of flow entries to build backup paths is reduced. But it is prone to congestion during rerouting process and needs a smooth switchover between working and transient plane. An approach based on source routing called SlickFlow is proposed in [37]. The packet header contains the information for both primary and backup paths. This approach overall increases the packet header size so requires more bandwidth. Also, this header size is topology dependent and increase in primary path length results in header size increase.

2.6 Summary of Proposed Approach

Initially our goal was to come up with an efficient recovery scheme that satisfies standard requirements as well as address the limitations of previous works on failure recovery discussed above. To settle our research objective, we start by choosing the best

options from the literature, such as a protection approach is chosen over a restoration approach. Narrowing it down further, local link protection was our choice instead of end to end path protection for a faster recovery. Our proposed approach is a Segment Routing (SR) based local link failure recovery mechanism that addresses those limitations. For example, standard OpenFlow Fast-Failover group type is used for packet switching to avoid hardware customization or introducing new packet types. Computational burden on the controller is reduced by taking a per-link approach, not for every flow going through that link. As SR is a source routing paradigm, the intermediary switches do not have to maintain all the states to forward a packet. Packets were detoured locally from the adjacent switch so only a single path is encoded and pushed by the ingress node. Thus, packet header size does not increase. Details of our proposed approach is discussed in chapter 3.

3 Chapter 3

Proposed Methodology

The objective of this thesis is to design a link failure recovery scheme in an SDN network to detour the traffic locally using segment routing. This chapter introduces and discusses the failure recovery algorithm as well as a label encoding algorithm to meet the Segment List Depth (SLD) constraint of segment routing. The first part of this chapter therefore explains our proposed recovery method using OpenFlow group table and the second part focuses on minimizing label stack depth by grouping multiple labels into one.

3.1 Link Protection with Segment Routing

Network traffic must be detoured locally at the switch in case of a link failure to avoid a round-trip delay to the controller, thus improving the recovery time. For local traffic detour, the adjacent node must sense the broken link and react upon that. To achieve this, we apply the group table concept that is introduced in OpenFlow v1.1 [10] in our proposed approach.

With the introduction of group table, OpenFlow switches now have additional forwarding capabilities. With this, now a flow entry from the regular flow table can point to a group table entry to execute a set of actions depending on the group type. A group entry in a group table is associated with multiple action buckets, where each action bucket contains a set of actions to execute. A group entry consists of a group identifier, a group type, a counter field and several action buckets.

Fast Failover (FF) group type is used in our work. FF group enables the switch to locally detour the disrupted flows without any intervention from the controller. There are multiple action buckets in an FF group and each action bucket is associated with a specific port. The buckets are evaluated orderly and the first bucket which is associated

with a live port is selected. Packets are dropped if no buckets are alive, means none of the associated ports are up. An OpenFlow switch checks the liveness of an action bucket by monitoring the up/down status of its output port. It relies on the port configuration bit *OFPPC_PORT_DOWN* value to check the status of the ports. If the bit indicates that the port is down, then the group executes the next available action bucket.

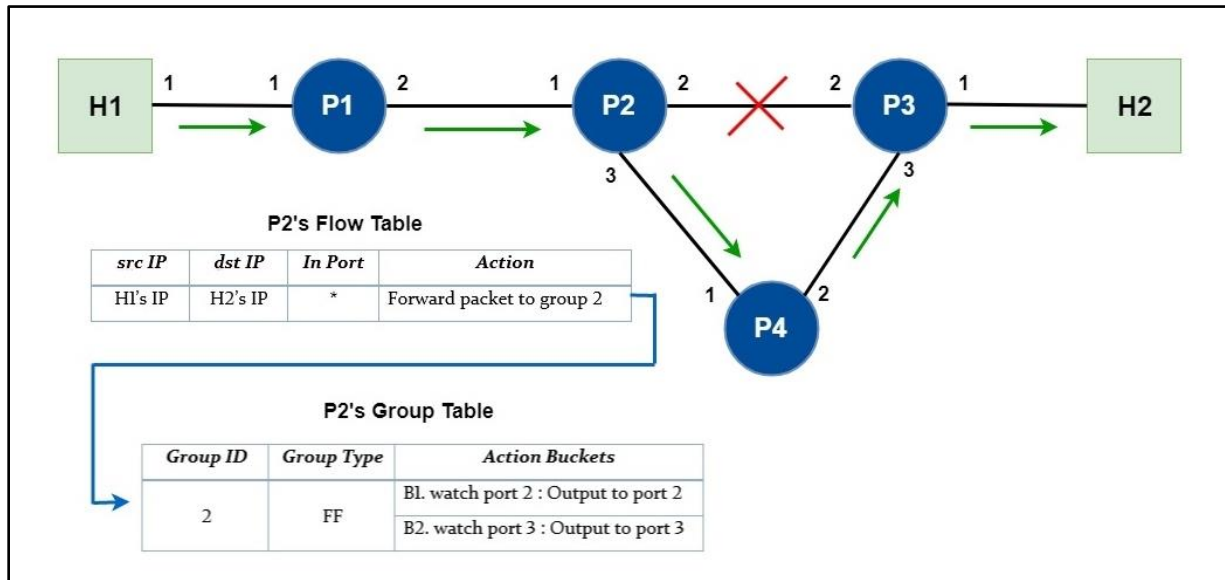


Figure 3.1.1: OpenFlow Group Table

Figure 3.1.1 illustrates the functionality of FF group. Let's say we want to protect the link between P2 and P3. When a packet arrives at P2, it is matched against the flow rules in the flow table. The packet is matched with the flow as H1 is sending a packet to H2, then the packet is forwarded to the group table using group ID 2, as per the action in the flow table. On receiving a packet from the flow table, the group table executes the action bucket 1, which is to check if port 2 is alive and then send packet on port 2. In case of P2-P3 link failure, this bucket will not be executed. Then the next action bucket becomes active and as port 3 is alive, the packet will be forwarded to port 3.

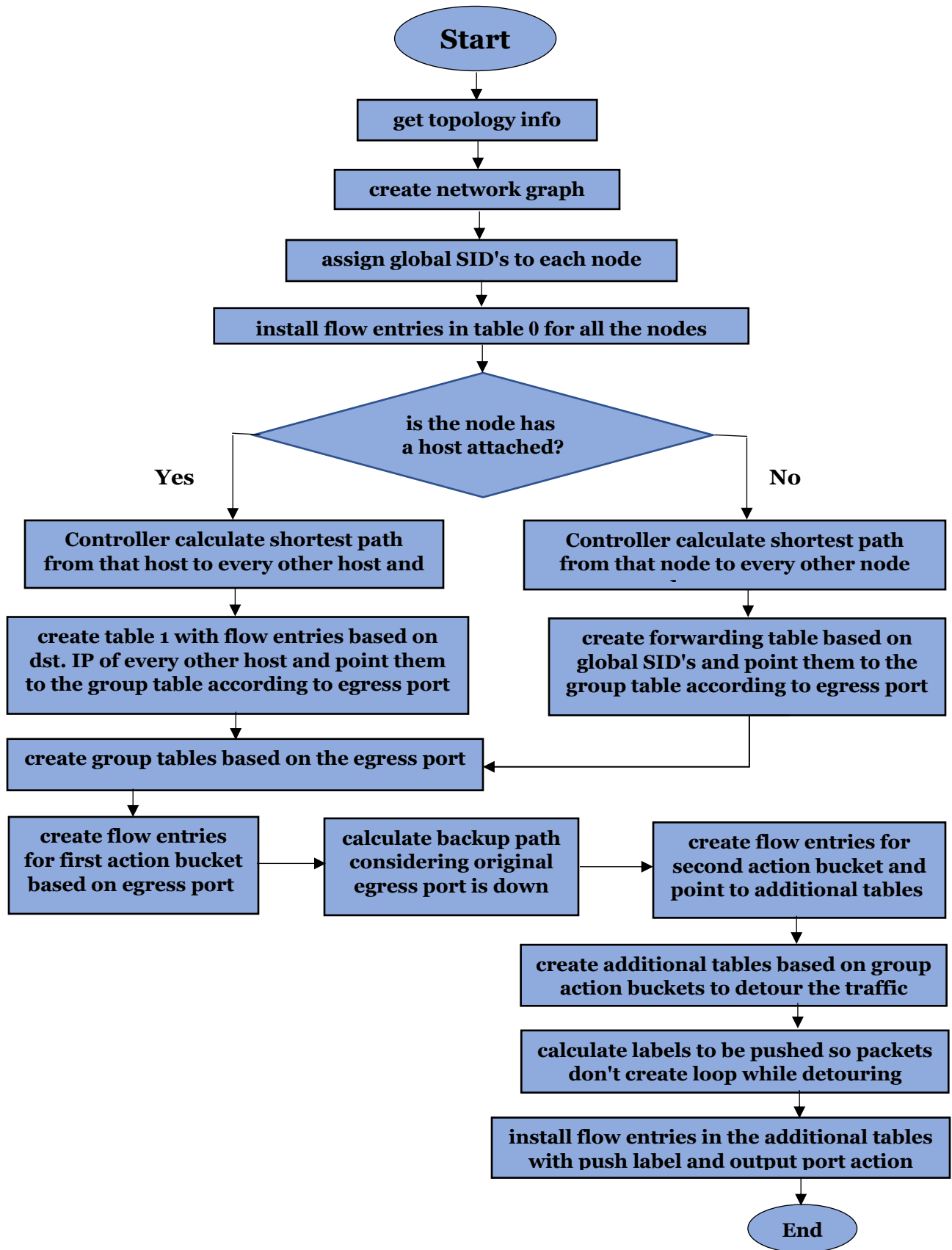


Figure 3.1.2: Flowchart for local failure recovery scheme

Figure 3.1.2 illustrates the general workflow of our failure recovery scheme. All these steps are done in the controller at the beginning, as we are using a link protection approach, not restoration method. Controller calculates both primary and backup paths to protect each link in the network. Backup paths are calculated assuming a link is broken, although initially there is actually no failure. After installing flow rules for both primary and backup paths, we do not need any controller intervention in our approach. The controller first gathers the following information from the topology:

$$H = \{ H_1, H_2, H_3, \dots, H_n \}$$

$$N = \{ N_1, N_2, N_3, \dots, N_n \}$$

$$L = \{ L_{1N_x \leftrightarrow N_y}^{s \leftrightarrow d}, L_{2N_x \leftrightarrow N_y}^{s \leftrightarrow d}, L_{3N_x \leftrightarrow N_y}^{s \leftrightarrow d}, \dots, L_{nN_x \leftrightarrow N_y}^{s \leftrightarrow d} \}$$

Where H , N and L is the set of hosts, switches and links in the domain respectively. $L_{nN_x \leftrightarrow N_y}^{s \leftrightarrow d}$ is the ID of the link, where s and d are source and destination ports, and N_x and N_y are source and destination switches. Using this information, a network graph is created to run the path calculation, considering each host and switch as graph vertices, their global SID as node attribute and links as graph edges.

Then we start building the flow entries in the switches for both normal and detoured traffic. OpenFlow pipeline processing starts from table 0 by default, so we aim to keep the flow entries in table 0 as minimum as possible for a faster flow searching. With our proposed design, in table 0 we will have maximum 4 flow entries, regardless of the topology and number of nodes. Depending on whether a host is attached to that switch or not, table 1 is created, assuming every switch has only a single host attached with it on its port 1. We treat every switch as one of these three states – source/ingress node, intermediary node or destination node. If a host is attached, then that node can be all three of the states and if that host send a packet to any other host, that switch should be able to create the packet with appropriate destination node SID. That is why a separate table 1 is needed if a host is attached. For an intermediary only node, means

that node doesn't have any host attached with it and only used as a transit node, table 1 is not necessary.

A table 17 is created which works similar as a Label Information Base (LIB) in an MPLS scenario, only that this table points to a particular group entry depending on the labels instead of an output port. We chose table id as 17 assuming a standard 16 port switch and our design requires to create some additional flow tables based on switch ports. After that group tables entries are created. First action bucket outputs to a port same as group ID, which is basically the egress port for the shortest path to the destination node. Second action bucket refers to another flow table with table ID same as the output port for backup path.

Algorithm 3.1.1. – 3.1.3 describes our recovery design in detail, with calculation and used match and actions to build the flow rules. We will use figure 3.1.3 to illustrate these pseudocodes graphically.

In the topology shown in figure 3.1.3 we have 6 switches, 4 hosts and 11 links in total. So $H = \{H_1, H_2, H_3, H_4\}$, $N = \{N_1, N_2, \dots, N_6\}$ and $L = \{L_1, L_2, \dots, L_{11}\}$. Global ID of 101-106 has been assigned to switches 1-6 respectively. We are implementing segment routing in MPLS architecture, so segment identifiers and MPLS labels are used interchangeably onwards. Hosts has an IP assigned in format 10.1.1.*i* where *i* is the host number.

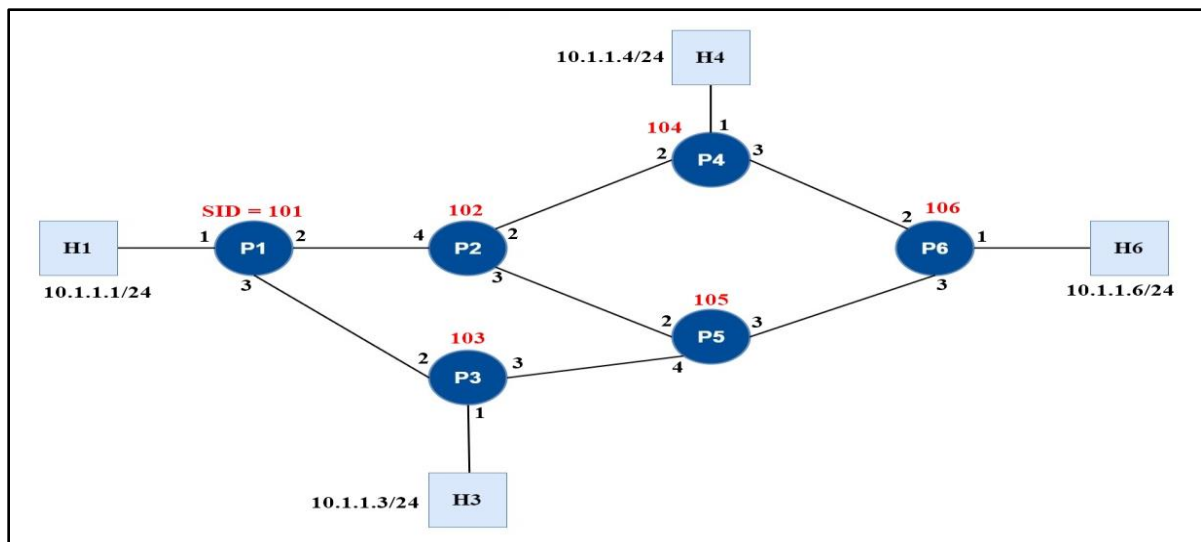


Figure 3.1.3: Reference Topology

Algorithm 3.1.1 - Pseudocode for Failure Recovery

Require: Set of switches, hosts and links in the topology

Input: Graph $G(V, E)$; switch $v \in V$, link $e \in E$

Output: local detouring of disrupted traffic to an alternate path

```
1:  for each node  $v \in V$  do
2:      assign a global SID to that node
3:      if this node has a host attached then
4:          generate a flow entry in table 0
5:          Match: In_Port; Action: go to table 1
6:          Match: own SID, mpls_bos = 1; Action: pop label, output port 1
7:          for every other host  $H$  in topology do
8:              calculate shortest path from that host to every other host
9:               $d \leftarrow$  destination node
10:              $p \leftarrow$  egress port
11:             generate flow entries in table 1
12:             Match: destination IP
13:             Action: set label  $\leftarrow$  global SID of  $d$ ; forward to group, group ID  $\leftarrow p$ 
14:         end for
15:      end if
16:      generate flow entries in table 0
17:      Match: own SID, mpls_bos = 0; Action: pop label, go to table 17
18:      Match: * (wildcard); Action: go to table 17
19:      for every other node  $N$  in  $V$  do
20:          calculate shortest path from this node  $v$  to node  $N$ 
21:           $p \leftarrow$  egress port
22:          generate flow entries in table 17
23:          Match: global SID of  $N$ 
24:          Action: forward to group, group ID  $\leftarrow p$ 
25:          Call create_group_entries ( $v, N, p$ )
26:      end for
27:  end for
```

We start by building flow tables in P1. It has host H1 attached with it, so lines from 3-15 in algorithm 3.1.1 will be executed for this node. This will create a flow entry in P1's table 0, matching In_port and send it to table 1, assuming a host is always attached with a switch on its port 1. In table 1 three entries will be created as we have three other hosts except H1, matching each of their IP as the destination IP address. We calculate the shortest path from the node that H1 is attached to, in this case which is P1 to every other node that has a host attached. The end node and egress port for next hop in the path is recorded in variable d and p respectively. Then action rules are installed, such as “push global SID of node d and forward to group # p ”.

For example, from H1 to H6 the chosen shortest path is H1 → P1 → P2 → P4 → P6 → H2. So we will push the SID of P6, which is 106 and forward it to group 2 (egress port for P1→P2).

20	3	1	8
label	Traffic Control (TC)	Bottom of Stack (bos)	TTL

Figure 3.1.4: MPLS packet header

Line 16-19 creates another three flow entries in P1's table 0. One of them matches the label if it's the SID of P1 itself and mpls_bos bit is 1, then pop the label and forward the packet to port 1. This flow rule considers as if P1 is the end node (a packet from H6 to H1). The *mpls_bos* bit indicates the bottom of the stack. It is a 1-bit field in the MPLS packet header (figure 3.1.4), where 1 means the current label is the last one in the stack and 0 indicates that another MPLS label will follow this one. The third and fourth flow in table 0 is if P1 is a transit node. Final table 0 and table 1 of P1 is given below:

<i>Flow ID</i>	<i>Match field</i>	<i>Instruction</i>
1	InPort → 1	Go to table 1
2	Label → 101 mpls_bos → 1	pop label forward to port 1
3	Label → 101 mpls_bos → 0	pop label forward to table 17
4	*	Go to table 17

Figure 3.1.5: Table 0 of P1

<i>Flow ID</i>	<i>Match field</i>	<i>Instruction</i>
1	Dest. IP → 10.1.1.3	Set label → 103 forward to group #3
2	Dest. IP → 10.1.1.4	Set label → 104 forward to group #2
3	Dest. IP → 10.1.1.6	Set label → 106 forward to group #2

Figure 3.1.6: Table 1 of P1

Line 20-27 are executed to create table 17, as well as group tables by calling algorithm 3.1.2 in line 26. Table 17 is used to forward a packet based on labels. It has global SID of all other nodes as match condition and forward to group entries based on egress port.

<i>Flow ID</i>	<i>Match field</i>	<i>Instruction</i>
1	Label → 102	Forward to group #2
2	Label → 103	Forward to group #3
3	Label → 104	Forward to group #2
4	Label → 105	Forward to group #3
5	Label → 106	Forward to group #2

Figure 3.1.7: Table 17 of P1

Algorithm 3.1.2 is used to create group tables and additional flow tables. Groups are created based on the egress port for a destination node and port number becomes the group ID. From table 1 and 17 we can see that we are forwarding packets to either group #2 or #3. So two groups will be created in P1's group table with ID 2 and 3 respectively.

The first action bucket is just output to the port same as group ID, which is basically the egress port that we calculated previously according to the shortest path from source node to destination node. We don't need to push any label here because this bucket will be executed in normal condition (no link failure) and packets will eventually follow the shortest path to the destination node.

Second action bucket will be monitoring the port that indicates our backup path. In case our primary link is broken, meaning first action bucket is inactive, then packets will be forwarded using the second action bucket. There could be multiple backup paths which is not optimal, and some calculation is needed to optimize this as much as possible. For example, if we think about node P2 and want to protect the P2 <-> P4 link, our primary bucket will monitor port 2 of P2, as this is the egress port for P2 <-> P4. Now for the second action bucket, when we calculate backup path from P2 to P4, there are two paths:

I) $P2 \rightarrow P5 \rightarrow P6 \rightarrow P4$ with egress port 3

II) $P2 \rightarrow P1 \rightarrow P3 \rightarrow P5 \rightarrow P6 \rightarrow P4$ with egress port 4.

Our algorithm will choose the optimal one here and port 3 will be monitored in the second action bucket. Now as the port to be monitored in second action bucket has been decided, action for this rule is to forward the packet to an additional flow table with table ID same as the port number.

<i>Group ID</i>	<i>Group Type</i>	<i>Action Buckets</i>
2	FF	B1. watch port 2 : Output to port 2
		B2. watch port 3 : go to table 3
3	FF	B1. watch port 3 : Output to port 3
		B2. watch port 2 : go to table 2

Figure 3.1.8: Group Table of P1

We designed additional flow tables here because backup paths are not the shortest paths, they are sub-optimal. If we forward a packet in backup path only with the original label, which is the global SID of the destination node, it may create a forwarding loop in the topology. To avoid this and steer the packet accordingly through the backup path we will push additional labels, and this is done in these additional tables.

With this design we are leveraging the segment routing architecture and gained a significant advantage over the previous works that is discussed in section 2.5. All the previous link protection method focuses on to reach the other end of the broken link, no matter what the destination is, and then follow the shortest path from that node to the destination node. In our proposed design, we consider a destination-based approach for this scenario.

Figure 3.1.9 illustrates link protection method done in previous works. A packet is being sent from H1 to H2 with the shortest path $H1 \rightarrow P1 \rightarrow P2 \rightarrow P4 \rightarrow P6 \rightarrow H2$. Now if the link between $P1 \rightarrow P2$ fails, their method reroutes the packet up to P2 (other end of the broken link) and from there, packet again follows the shortest path (shown in green

solid arrow). Instead of this, the packet could be forwarded to P6 from P5 directly while detouring (shown with blue dotted arrow).

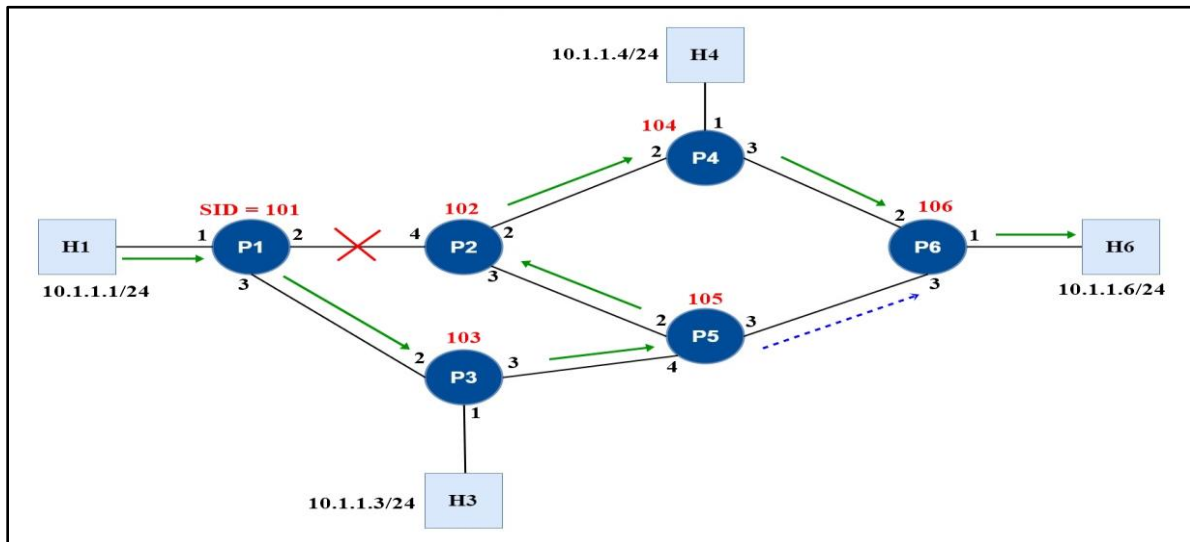


Figure 3.1.9: Link Protection in literature

Figure 3.1.10 shows packet detouring with our proposed recovery scheme. P1 will execute its secondary action bucket and send the packet to table 3. In table 3 action is output to port 3, without adding any additional labels. Eventually P3 and P5 receives the packet and forwards the packet using their table 17. The label stack in the header will be {106}, so both P3 and P5 will forward the packet to P6 without any popping. Thus, avoiding taking the longer path discussed before.

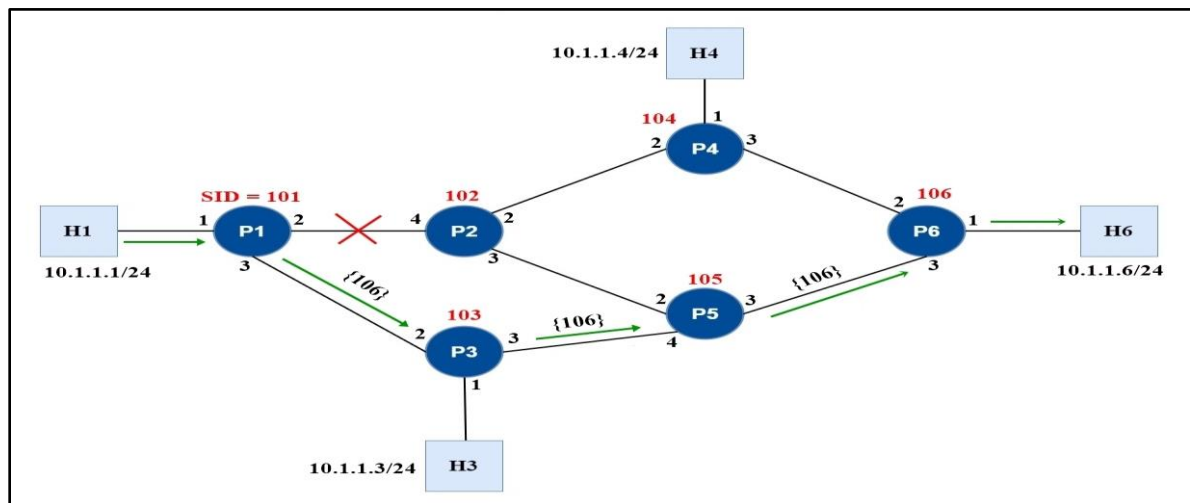


Figure 3.1.10: Proposed destination-based detouring

Additional flow tables have match condition of global SID of other nodes, but not every other node is included in every additional flow table. A node should not be in the table same as its primary group (also egress port), because then the action would be just to forward it to that port, which is already done in the first action bucket.

For example, let's look at table 17 and group table of P1 in figure 3.1.7 and 3.1.8 respectively. For label 102 the action in table 17 is to go to group #2, whereas for label 103 it's group #3. Now from the group table, we can see that we need to create two additional tables, table 2 and table 3. It is not necessary to include both label 102 and 103 in both tables. This design allows us to limit the total flow entries in a switch, because regardless of the number of additional tables we need to create, labels are distributed among them. So total number of flows in additional tables are always $N-1$, where N is the total number of nodes in the topology.

Table 2			Table 3		
Flow ID	Match field	Instruction	Flow ID	Match field	Instruction
1	Label → 103	Push label → 105 Output port 2	1	Label → 102	Push label → 105 Output port 3
2	Label → 105	Output port 2	2	Label → 104	Push label → 105 Output port 3
			3	Label → 106	Output port 3

Figure 3.1.II: Additional Flow Tables of P1

Here some of the entries have an additional label push action, whereas some of them only forward to a port. This checking is done in line 4-8 of *additional_tables* pseudocode in algorithm 3.1.2, where we check if the backup path without proper additional label(s) will create forwarding loop. If so, then we push additional label to steer the traffic. Which label to push is calculated by calling the *compute_segment* in algorithm 3.1.3. If it does not create a forwarding loop, then we keep the original label, and packet will follow the shortest path from there, which we showed in figure 3.1.10.

Algorithm 3.1.2 - Pseudocode for creating group tables and additional flow tables

create_group_entries (node v , node N , port p)

```
1:   $d \leftarrow$  adjacent node attached with  $p$ 
2:   $group\_id \leftarrow p$ 
3:   $L \leftarrow$  list of all the paths from node  $v$  to node  $d$ 
4:  for every path  $l$  in  $L$  do
5:      if  $l$  contains node  $v$  and node  $d$  in it then
6:          remove that path from  $L$ 
7:      end if
8:  end for
9:   $B_p \leftarrow$  choose shortest path from  $L$ 
10:  $W \leftarrow$  egress port for next node in  $B_p$ 
11: if group table of node  $v$  already has an entry with ID =  $group\_id$  then
12:     Call additional_tables ( $v$ ,  $N$ ,  $W$ )
13: else
14:     create a new group table entry with  $group\_id$ 
15:     create first action bucket
16:     watch_port:  $p$ ; Action: output to port  $p$ 
17:     create second action bucket
18:     watch_port:  $W$ ; Action: go to table  $W$ 
19:     Call additional_tables ( $v$ ,  $N$ ,  $W$ )
20: end if
```

additional_tables (node src , node N , table_id T)

```
1:  port  $p \leftarrow T$ 
2:   $d \leftarrow$  adjacent node attached with  $p$ 
3:   $L \leftarrow$  list of shortest paths from node  $d$  to node  $N$ 
4:  if there is any path in  $L$  that contains node  $src$  then
5:       $S \leftarrow$  Call compute_segment ( $src$ ,  $d$ ,  $N$ )
6:      create flow entry in table  $T$ 
7:      Match: global SID of node  $N$ 
8:      Action: push segment  $S$ , output to port  $p$ 
9:  else
10:     create flow entry in table  $T$ 
11:     Match: global SID of node  $N$ 
12:     Action: output to port  $p$ 
13:  end if
```

For instance, for a packet going to P1 to P2 with label 102 (global Sid of P2), if $P1 \leftrightarrow P2$ link is broken, P1 will send the packet to port 3 according to its group #2 rule. Now if we do not push any additional label, P3 will see the label 102 and try to reach P2 by either $P3 \rightarrow P1 \rightarrow P2$ or $P3 \rightarrow P5 \rightarrow P2$. These are equal cost paths and the shortest path

algorithm can choose any one of these two. That is why we need to push an additional label 105 and label stack becomes {105,102}. So P3 sees 105 and forward it to P5, and then from P5 packet goes to P2. This is one of the advantages of segment routing to have more control on steering the packet. Similarly, for label 106 we did not push any additional label. This is because from P3 to P6 the shortest path is $P3 \rightarrow P5 \rightarrow P6$ and this path does not include source node (P1), so there will be no forwarding loop and we can use just the original label.

Algorithm 3.1.3 - Pseudocode for Segment Label Calculation

```

compute_segment (node src, node d, node dst)
1:  M  $\leftarrow$  node to be found that does not create forwarding loop
2:  path  $\leftarrow$  select a path from node src to node dst that contains node d
3:  for every node M in the path do
4:      len1  $\leftarrow$  path length from node src to node M
5:      len2  $\leftarrow$  path length from node M to node dst
6:      if len1 + 1 > len2 do
7:          return global SID of node M
8:      else
9:          continue
10:     end if
11: end for

```

Algorithm 3.1.3 demonstrates the pseudocode for label computation. A node should be selected on the backup path in such a way that packets do not go back to the source node, thus avoiding any forwarding loop. A downside of local recovery is only the nodes adjacent to the failed link knows about this immediately. Other nodes have no idea about it and they will try to follow the usual shortest path to forward any packet. To illustrate this problem, we will use a classic ring topology in figure 3.1.12.

In figure 3.1.12, we have nine nodes P1-P9, connected in a ring network. In case of $P2 \rightarrow P3$ link failure, the only other way to go to P2 from P3 is all the way around. Now if we forward the packet with original label, 102 (assuming global SID's are assigned from 101-109) to P4, then P4 will send it back to P3 seeing the label 102, which is termed as Crankback Routing. This is because for P4, the shortest path to P2 is through P3 and it

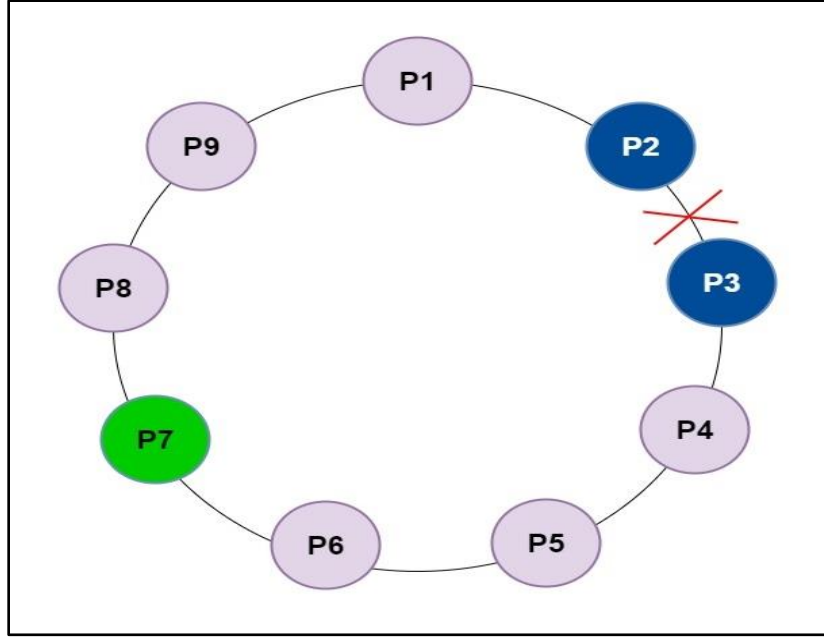


Figure 3.1.12: Reference Ring Topology

has no way to know about the P2 \leftrightarrow P3 link failure. The same thing will happen for P5 and P6 also. So we need to iterate through the backup path for a node that satisfies the equation (1).

$$\text{len}(S \rightarrow M) + 1 > \text{len}(M \rightarrow D) \quad (1)$$

Here, $S \rightarrow$ source node, P3 in our example

$D \rightarrow$ destination node, P2 in our example

$M \rightarrow$ intermediary node to be determined

In the above figure, P7 satisfies equation (1) :

$$\begin{aligned} & \text{len}(P3 \rightarrow P7) + 1 > \text{len}(P7 \rightarrow P2) \\ \Rightarrow & 3 + 1 > 3 \\ \Rightarrow & 4 > 3 \end{aligned}$$

Pseudocode in 3.1.3 does this in line 6-10 and returns the global SID of the node that satisfies the condition, which is then used in additional flow tables as a push action.

We will try to illustrate our proposed recovery scheme visually by using two examples in the subsequent sections below. Only relevant tables are shown in these figures for each switch due to space constraint and used flow entries are marked with green rectangle.

Example 1: failure of P2<->P4 link

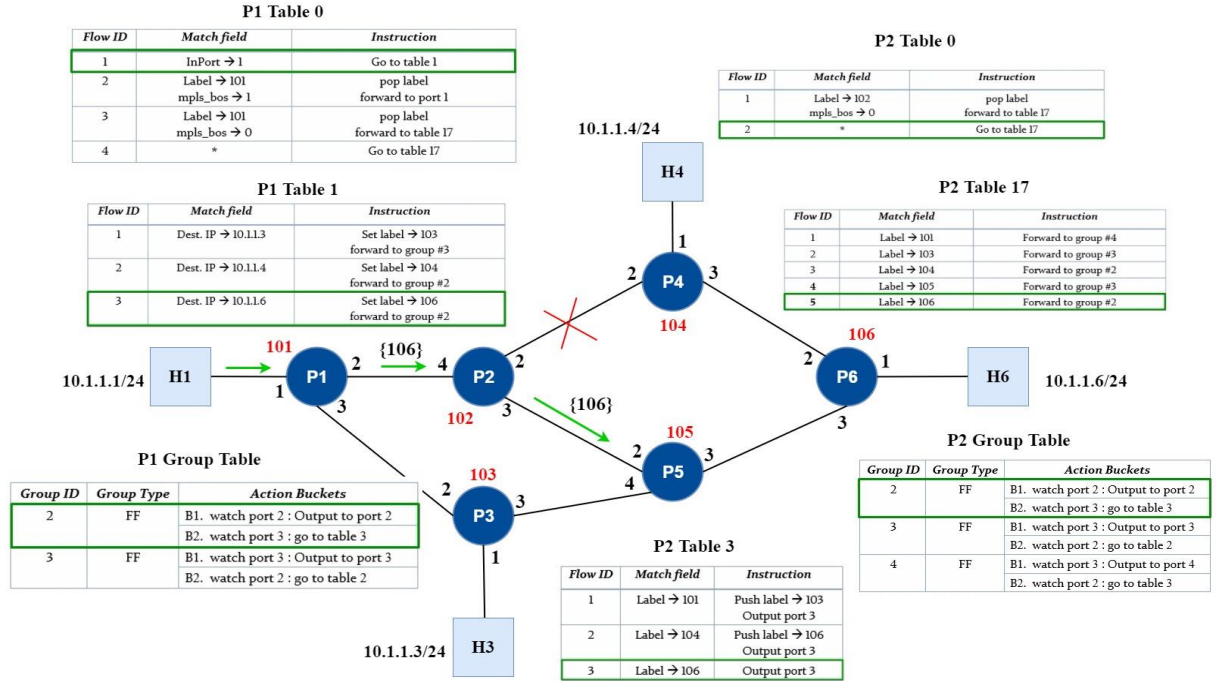


Figure 3.1.13 (a): Example 1 - P2<->P4 link fails

A packet will be sent from H1 to H2. H1 creates the packet with src.IP = 10.1.1.1 and dst.IP = 10.1.1.6 and forward it to P1. This packet matches with the first entry on P1's table 0 and as per the instruction, the packet is forwarded to table 1. In table 1, destination IP 10.1.1.6 is matched with the third flow. P1 pushes the label 106 and forward the packet to group 2. Group 2 monitors the status of port 2 in primary action bucket and as this port is live, packet is forwarded through this port to P2.

Upon receiving, this packet matches with the second flow in P2's table 0 where the instruction says go to table 17. In table 17, label 106 is matched and the packet is forwarded to group 2. Now the primary action bucket of P2's group 2 will not be executed because port 2 is not alive. So, according to the second action bucket, the packet is forwarded to table 3. Label 106 is matched again and forwarded to port 3, without pushing any additional label. This is because P5 is directly connected to P6, so obviously this is the shortest path and there is no way in normal condition that P5 will send that packet back to P2 to go to P6. Here we are also assuming single link failure at a time.

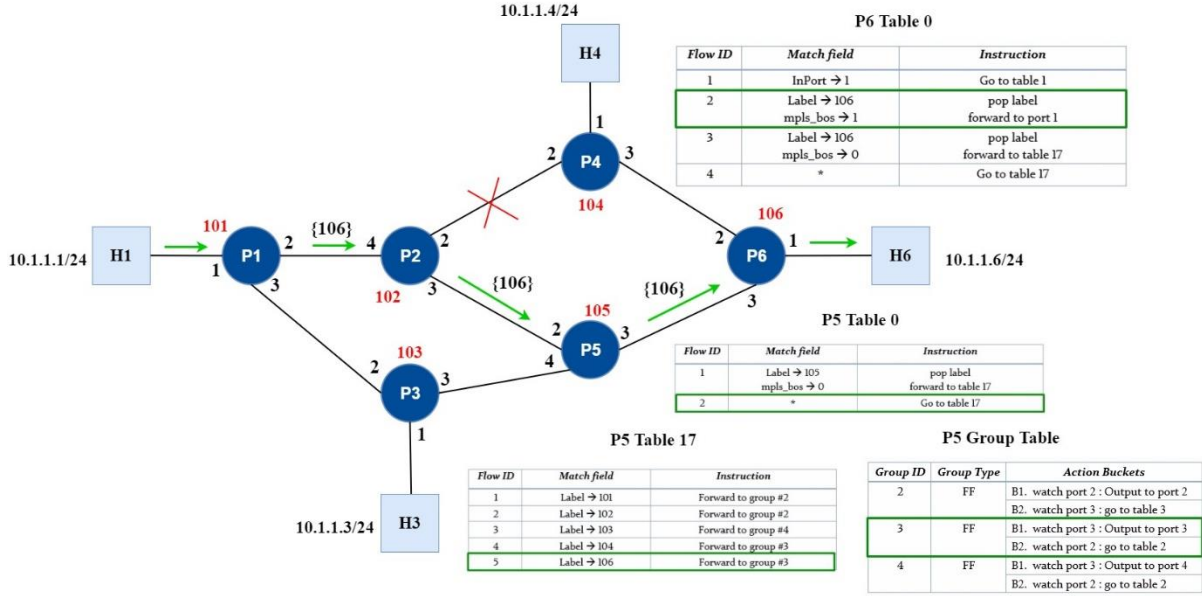


Figure 3.1.13 (b): Example 1 - P2<->P4 link fails (cont.)

P5 receives the packet and forward the packet to port 3 using its group 3 rule, where the primary action bucket is to monitor the port 3 and as port 3 is alive, the packet is forwarded accordingly. This packet matches with the second flow rule in P6's table 0. It has a label {106} which is SID for P6 itself and mpls_bos bit is 1, means there is no other label followed by this one. Then as per the instruction, P6 pops the label and forward the packet to its port 1, and eventually H2 gets the packet.

Example 2: failure of P4<->P6 link

In this example, we will consider the P4<->P6 link is broken and will also observe how our recovery design handles crankback routing, illustrated in figure 3.1.14 below. A packet will be sent from H1 to H2 and the processing at P1 and P2 is same as before (figure 3.1.13-a). Only difference is now P2 will send the packet on its port 2 because this link is now working. P4 receives the packet and matches it with the fourth flow rule in its table 0. Then the packet is forwarded to table 17, label 106 is matched and forwarded to group 3 as per the instruction. Primary action bucket for group 3 is to monitor the status of port 3, which is down for this example. So second action bucket is executed and the packet is forwarded to an additional table, table 2.

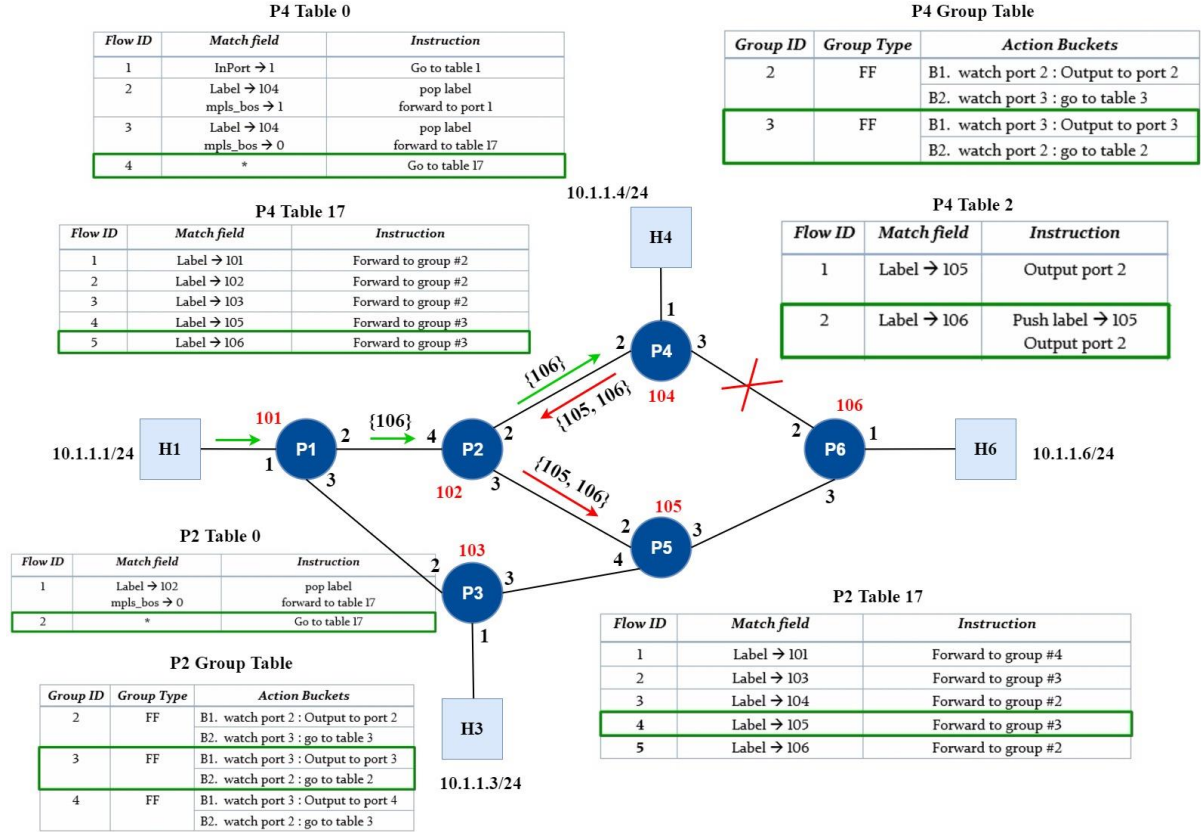


Figure 3.1.14 (a): Example 2 – P4 \leftrightarrow P6 link fails

Now if the packet is forwarded with original label on port 2 of P4, there is a chance of forwarding loop. P2 will see label 106 and to go to P6, it has two equal cost paths – P2 \rightarrow P4 \rightarrow P6 or P2 \rightarrow P5 \rightarrow P6. As we do not control which ECMP to choose in our algorithm, any one of these two can be installed in P2's flow table. If the first one is chosen, then P2 will send any packet to go to P6 on its port 2 if that port is alive. So we will have a forwarding loop between P2 and P4.

As previously discussed, pseudocode 3.1.2 and 3.1.3 tackles this scenario. For instance, we can see that for label 106, P4's table 2 has an additional instruction to push label 105. This ensures that P2 forwards the packet to P5. In other words, P4 is forcefully making P2 to choose the path P2 \rightarrow P5 \rightarrow P6. Now the packet has a label stack of {105, 106}.

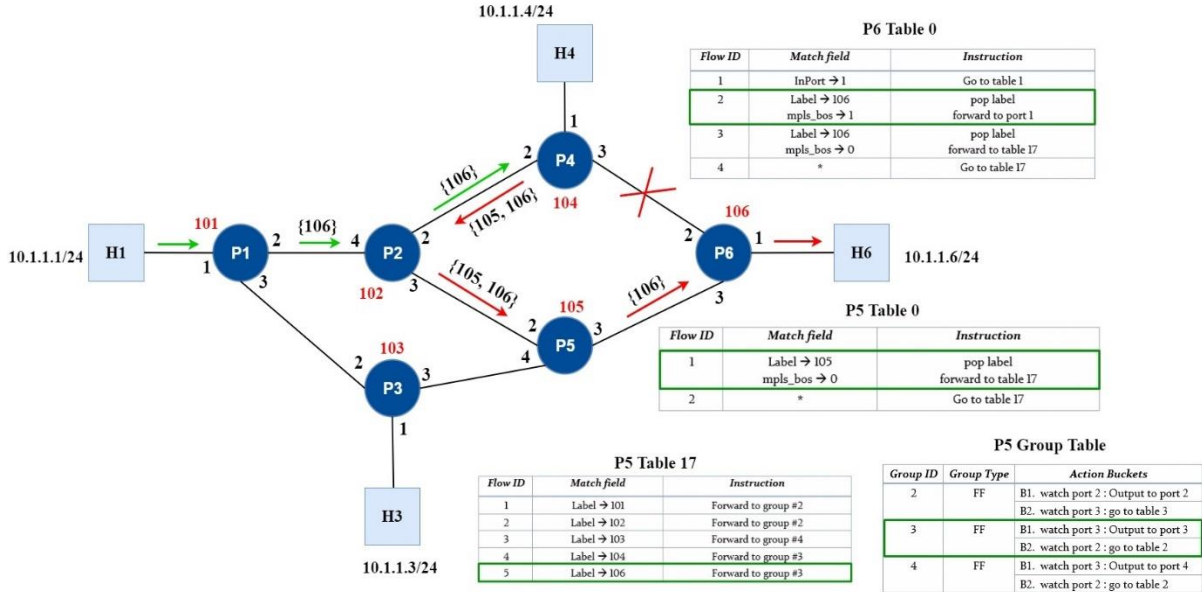


Figure 3.1.14 (b): Example 2 – P4 \leftrightarrow P6 link fails (cont.)

P5 receives the packet and it matches with the flow rule “label \rightarrow 105, mpls_bos \rightarrow 0” because the outer label is now 105 and there is another label followed by it, so mpls_bos bit will be 0. Then P5 pops the label 105 and forward the packet to table 17. There it is matched with label 106 and forwarded to group 3. As port 3 of P5 is alive, packet is eventually forwarded on this port to P6. P6 process the packet same as before by popping the label {106} and forward the packet to its port 1, and eventually H2 gets the packet.

3.2 Efficient Segment Encoding Algorithm for Segment Routing

Using only a single label at the ingress node and follow the shortest path throughout the topology does not entirely utilize the segment routing capability. With node and adjacency SID's, SR provides much more granular control to steer the traffic based on different scenarios, such as:

- QoS requirement - a path that satisfies a specific requirement may not be the shortest path. So we need more than one label to divert the traffic along this path.

- ii) While implementing segment routing, the service provider may implement strict encoding instead of loose encoding, which means every node in the path must be encoded in the label stack.
- iii) Depending on the statistical data, some of the links on the shortest path can be marked as the most error prone, means those links have failed frequently. So a path that avoids those links can be encoded at the source node, thus minimizing the failure recovery overhead. Again, to encode this path, a stack of multiple labels is needed.

This third scenario can think of as an extension of our current proposed recovery scheme, which we would like to address in our future work. Figure 3.2.1 illustrates this:

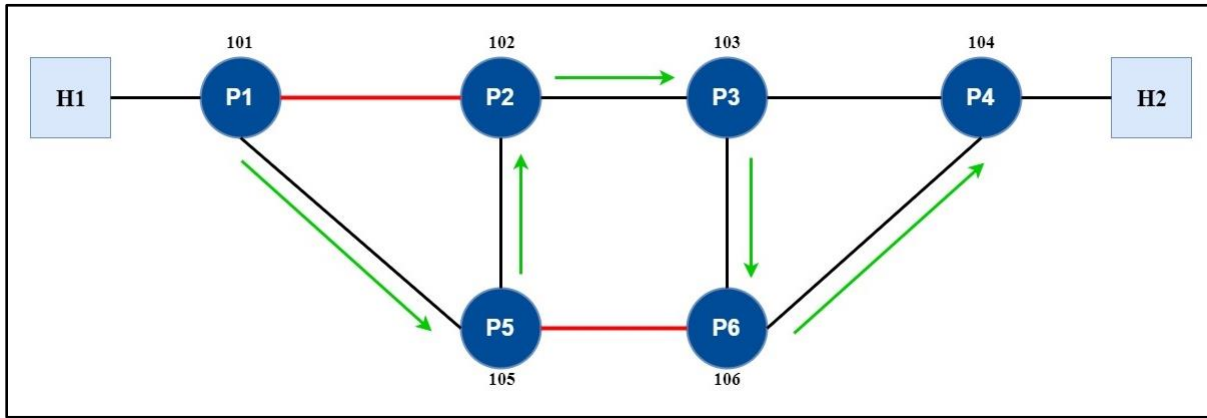


Figure 3.2.1: example topology for SLD constraint

For example, to go from H1 to H2, we have two equal cost paths: $P1 \rightarrow P2 \rightarrow P3 \rightarrow P4$ and $P1 \rightarrow P5 \rightarrow P6 \rightarrow P4$. From statistical data, we analyzed that the $P1 \leftrightarrow P2$ and $P5 \leftrightarrow P6$ link is most prone to failure. Considering this, initially when we are installing flows for normal condition, it is better to choose the path $P1 \rightarrow P5 \rightarrow P2 \rightarrow P3 \rightarrow P6 \rightarrow P4$, thus we reduce the possibility of using the recovery scheme. Being a source routing paradigm, SR architecture requires that this path should be encoded by P1 when it receives the packet from H1 to go to H2. As this is not a shortest path, multiple labels are needed to steer the packet accordingly, which is {105, 102, 103, 106, 104}.

According to [10], Open vSwitch currently supports up to three labels. In segment routing with MPLS plane, this is known as Segment List Depth (SLD) constraint. An

ingress node cannot push more than three segments. In our example above, the label stack is comprised of five segments. So P1 will not be able to encode the whole path in the packet header.

In this section, we are proposing a label encoding algorithm to meet this constraint by slicing the segment list and grouping multiple segments into one segment as necessary. We will define a new segment type called group segment identifier (gSID). This will replace multiple labels in the original stack and will be treated as one single label, thus reducing the stack depth. For example, if we group labels {103, 106, 104} as gSID1 then our original stack to be pushed at P1 will be {105, 102, gSID1}. Now there are three labels in total and P1 can push them into the packet header. This group SIDs will be local to a node and declared outside of the segment routing global block (SRGB), so they don't clash while assigning the node SIDs. SDN controller will group the labels as necessary, assign them to a gSID and install required flow rule on the relevant nodes. With our gSID example, now we need an additional flow rule in P2's table 17 (as per our design in section 3.1) –

<i>Flow ID</i>	<i>Match field</i>	<i>Instruction</i>
1	Label \rightarrow gSID1	pop label \rightarrow gSID1 push label \rightarrow {103, 106, 104} Forward to interface P2 \leftrightarrow P3

Figure 3.2.2: flow rule for gSID

The packet will be processed as follows: P1 pushes the label stack as {105, 102, gSID1} and send to P5. P5 pops its own label and send it to P2 because of the label 102. P2 receives the packet and matches the flow table entry as above. So P2 will pop the gSID label, and push three new labels {103, 106, 104} and send it to P3. From there packet will follow the regular pop action and reaches to H2. As we can see there is not more than three labels at any time, thus satisfying the SLD constraint.

Algorithm 3.2.1 - Pseudocode for Optimized Label Encoding

Input: a segment list > 4

Output: segment list with gSIDs ≤ 3

```
1:  $L \leftarrow$  list of candidate gSIDs,  $2 \leq \text{gSID} \leq 3$ 
2:  $S \leftarrow$  list of possible solutions generated from L using equation (2)
3:  $\text{weightDB} \leftarrow$  an array that holds the weight for each solution
4: for sol in S do
5:   if sol requires creation of new gSIDs then
6:      $w \leftarrow$  number of new gSIDs required
7:      $\text{weightDB}(\text{sol}) \leftarrow w$ 
8:   else
9:      $\text{weightDB}(\text{sol}) \leftarrow 0$ 
10:  end if
11: end for
12:  $\text{bestSolution} \leftarrow \min(S, \text{weightDB})$ 
13: for gs in bestSolution do
14:    $i \leftarrow$  index for replaces original segments with gs
15:    $N \leftarrow (i-1)\text{th}$  segment for the node to install flow rules
16:   install flows in the node having N global SID
17:   Match: label  $\leftarrow$  gs
18:   Action: pop label  $\leftarrow$  gs; push label  $\leftarrow$  segments in gs
19:   stack  $\leftarrow$  replace segment list with gs
20: end for
21: return stack
```

Given a segment list of more than three labels, algorithm 3.2.1 returns a new stack satisfying SLD and installs the necessary flows into the relevant switches. We start by generating a set of candidate gSIDs where each of them has a size of at least two labels and a maximum of size of SLD, which is three in this case. A candidate gSID cannot be more than the size of SLD because in that case when it pushes all the labels it is comprised of, it will exceed the SLD. From this candidate set, we then generate possible solution sets where a candidate gSID satisfies the equation 2 below:

$$S_p - \sum_{g \in L} (S_g - 1) \leq SLD \quad (2)$$

where, S_p = size of the original segment list

L = list of candidate gSIDs

S_g = size of a gSID

For instance, in our previous example the original segment stack is {105, 102, 103, 106, 104}. From this if we generate the list of candidate gSIDs with length $2 \leq \text{gSID} \leq 3$ -

$$L = \{(105,102), (102,103), (103,106), (106,104), (105,102,103), (102,103,106), (103,106,104)\}$$

This is not a set of all possible combinations, because we need to maintain the label order. Now from this set, we need to find out possible solution set that satisfies equation (2). gSIDs with length 3 will be tried first and solution with single gSID will have higher priority instead of composing multiple gSIDs. All the gSID with length 3 will satisfy the equation (2) because:

$$\begin{aligned} S_p - \sum_{g \in L} (Sg - 1) &\leq SLD \\ \rightarrow 5 - (3 - 1) &\leq 3 \\ \rightarrow 5 - 2 &\leq 3 \\ \rightarrow 3 &\leq 3 \end{aligned}$$

gSIDs with length 2 will not satisfy the equation (2) and will be discarded. Two or three gSID with length 2 will satisfy the equation, but as we are giving higher preference to the single gSID, only the length 3 gSIDs will be in our possible solution set. Depending on the topology and path length, sometimes it is inevitable to use multiple gSID. For example, with an original segment stack of 6, use of a single length-2 or 3 gSID will not satisfy equation (2). In that case, we will consider a combination of multiple gSIDs. To illustrate this, let's increase our original segment stack by one - {105, 102, 103, 106, 104, 107}. A combination of three length-2 gSID, or two length-3 gSID or a combination of both lengths satisfies equation (2).

<u>All length-2</u>	<u>All length-3</u>	<u>Both</u>
$S_p - \sum_{g \in L} (Sg - 1) \leq SLD$	$S_p - \sum_{g \in L} (Sg - 1) \leq SLD$	$S_p - \sum_{g \in L} (Sg - 1) \leq SLD$
$\rightarrow 6 - [(2 - 1) + (2 - 1) + (2 - 1)] \leq 3$	$\rightarrow 6 - [(3 - 1) + (3 - 1)] \leq 3$	$\rightarrow 6 - [(3 - 1) + (2 - 1)] \leq 3$
$\rightarrow 6 - [1 + 1 + 1] \leq 3$	$\rightarrow 6 - [2 + 2] \leq 3$	$\rightarrow 6 - [2 + 1] \leq 3$
$\rightarrow 6 - 3 \leq 3$	$\rightarrow 6 - 4 \leq 3$	$\rightarrow 6 - 3 \leq 3$
$\rightarrow 3 \leq 3$	$\rightarrow 2 \leq 3$	$\rightarrow 3 \leq 3$

While choosing this combination, we need to enforce a constraint that same label should not occur in multiple gSIDs. For this will create a forwarding loop. We will use equation (3) to enforce this.

$$\sum_{g \in L} \alpha_{lg} \leq 1 \quad (3)$$

where, α_{lg} = a binary variable which is equal to 1 if the label l is used in gSID g , or 0 otherwise.

With the above 6-segment stack, we will get the candidate list as below:

$$L = \{(105,102), (102,103), (103,106), (106,104), (104,107), (105,102,103), (102,103,106), (103,106,104), (106,104,107)\}$$

A combination of $\{(105,102) \& (102,103)\}$ or $\{(102,103,106) \& (106,104,107)\}$ cannot be chosen because label 102 is repeated in the first combination and 106 is in the second one. So if we choose any of these combinations, it will not satisfy equation (3), as label 102 and 106 will be added two times, making equation (3) as $2 \leq 1$, thus these two combinations will not be in our solution set.

After generating the possible solution set, we assign a weight to each of the solutions (lines 4-11). We have a gSID database to keep track of all the used gSIDs. If a solution does not require to create any new gSID, it has been assigned a weight of 0. Otherwise the number of required gSIDs to be created is assigned as that solutions weight. By doing this we minimize the number of total gSIDs and preferring the reutilization of already created gSIDs. Then the solution with minimum weight is chosen as the best solution.

For example, let's say two of the possible solutions from our 6-segment stack is {gSID1, 103, gSID2} and {gSID3, gSID4}; where gSID1 = {105,102}, gSID2 = {106,104,107}, gSID3 = {105,102,103}, gSID4 = {106,104,107}. We search in the gSID database and found that gSID1 is already created for some previous encoding. So the first solution needs to create

one new gSID (gSID2) and has been assigned a weight of 1 (0 for gSID1, and 1 for gSID2 $\rightarrow 0+1 \rightarrow 1$). The second solution needs to create both gSID3 and gSID4 so it has been assigned a weight of 2. Now from this two, we will choose the first solution as our best solution because it has a lower weight.

After choosing the best solution, we then install the necessary flows in the respective switches. In this case, we will install a flow rule for gSID1 in P1 and a flow rule for gSID2 in P3 (figure 3.2.2). Then we replace the labels in our original stack with gSIDs and return the new stack. The source node will now push this stack into the packet header.

3.3 Chapter Summary

In this chapter, we discussed our proposed local link failure recovery scheme and a segment encoding algorithm to mitigate SLD constraint. The proposed recovery approach employs OpenFlow FF group to detect failure and detour traffic by introducing Segment Routing (SR) in an SDN environment. Disrupted traffics are detoured locally at the switch and forwarded along the backup path based on segment identifiers (SR) and multiple distributed flow tables. Segment encoding algorithm dynamically groups multiple labels to reduce the stack size by introducing a new type of segment identifier, called gSID.

In the next chapter, the results of the proposed failure recovery approach will be examined and compared with other previous works.

4 Chapter 4

Implementation and Result Analysis

In this chapter, we will evaluate our proposed failure recovery and label stack optimization algorithm from the previous chapter. System environment and components used will be described, followed by performance assessment of our recovery scheme with some previous work in terms of recovery time and storage (total number of flow entries).

4.1 System Environment

For our testbed, we have used a laptop with Intel® Core™ i5-6200U CPU, 2.40 GHz with 8.00 GB RAM, running a 64-bit windows 10 as its operating system. VMware Workstation is used to run an Ubuntu 14.04.5 OS as a VM. OpenDaylight is used as the SDN controller and to simulate the OpenFlow network, we use mininet. Both are installed in the Ubuntu VM.

OpenDaylight is a collaborative open source project by The Linux Foundation, aiming to provide the functionality of an SDN controller. It is a modular open platform for customizing and automating networks of any size and scale. It can be deployed in a variety of production network environments and can provide support for other SDN standards and upcoming protocols. OpenDaylight is implemented solely in software and is kept within its own Java Virtual Machine (JVM), which means it can be deployed on any hardware and operating system platforms that support Java. The core of the OpenDaylight platform is the Model-Driven Service Abstraction Layer (MD-SAL). In OpenDaylight, underlying network devices and network applications are all represented as objects, or models, whose interactions are processed within the SAL. The various in-built modules utilizing the services provided by the SAL offer various network services. In our experiment, we used OpenDaylight version 0.5.2 (Boron SR2).

Mininet is a well-known network emulator for SDN research problems. It is a software-based network emulator that can emulate network topology with components such as switches, hosts, links in a virtual environment running on a single Linux Kernel. Custom topologies can be created in mininet by using python scripts. We created various topologies for simulation purpose, including our reference topology in figure 3.1.3. Mininet deploys open vSwitch as part of the network. . Open vSwitch is an open source virtual switch that is used extensively in network virtualization under production environment and supports OpenFlow protocol to communicate with the SDN controller. In our simulation, we have used mininet version 2.3.0dl and upgraded the open vSwitch to version 2.5.5 that support up to OpenFlow version 1.4.

4.2 Implementation

We started by installing OpenDaylight and Mininet in our Ubuntu VM. OpenDaylight requires Java version 1.7 or later to work. So first we need to install and set the Java environment variable. Maven is required as OpenDaylight uses this to handle builds and dependencies. Then we download the OpenDaylight Boron SR2 distribution zip file and run karaf to start the controller. OpenDaylight will start on VM's eth0 IP, which is 192.168.2.130 in our case.

```
# sudo apt-get install openjdk-8-jdk openjdk-8-jre
# sudo apt-get update
# export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
# sudo apt-get install maven
# wget https://nexus.opendaylight.org/distribution-karaf/0.5.2-Boron-SR2/
# unzip distribution-karaf-0.5.2-Boron-SR2.zip
# ./distribution-karaf-0.5.2-Boron-SR2/bin/karaf
```

Necessary features were installed using the *feature:install* command in the OpenDaylight CLI:

```
> feature:install odl-dlux-all odl-l2switch-switch-ui odl-openflowplugin-flow-services-ui
odl-bgpcep-pcep odl-restconf
```

To install mininet, we ran below commands:

```
# git clone git://github.com/mininet/mininet
# mininet/util/install.sh -a
```

We used python scripts to create different custom topologies in mininet. Figure 4.2.1 illustrates such a script to create our reference topology from figure 3.1.3 -

```
1. from mininet.topo import Topo
2.
3. class MyTopo( Topo ):
4.     "Simple topology example."
5.
6.     def __init__( self ):
7.         "Create custom topo."
8.         Topo.__init__( self )           # Initialize topology
9.
10.    # Add hosts and switches
11.    H1 = self.addHost( 'h1', ip='10.0.0.1/24', mac='00:00:00:00:00:01' )
12.    H3 = self.addHost( 'h3', ip='10.0.0.3/24', mac='00:00:00:00:00:03' )
13.    H4 = self.addHost( 'h4', ip='10.0.0.4/24', mac='00:00:00:00:00:04' )
14.    H6 = self.addHost( 'h6', ip='10.0.0.6/24', mac='00:00:00:00:00:06' )
15.
16.    S1 = self.addSwitch( 's1' )
17.    S2 = self.addSwitch( 's2' )
18.    S3 = self.addSwitch( 's3' )
19.    S4 = self.addSwitch( 's4' )
20.    S5 = self.addSwitch( 's5' )
21.    S6 = self.addSwitch( 's6' )
22.
23.    # Add links
24.    self.addLink( H1, S1 )
25.    self.addLink( H3, S3 )
26.    self.addLink( H4, S4 )
27.    self.addLink( H6, S6 )
28.    self.addLink( S1, S2 )
29.    self.addLink( S1, S3 )
30.    self.addLink( S3, S5 )
31.    self.addLink( S2, S4 )
32.    self.addLink( S2, S5 )
33.    self.addLink( S4, S6 )
34.    self.addLink( S5, S6 )
35.
36.    topos = { 'mytopo': ( lambda: MyTopo() ) }
```

Figure 4.2.1: python script to create a custom topology in mininet

Our main python program will invoke algorithms 3.1.1 - 3.1.3 to implement our recovery scheme and installs the flow rules into the switches accordingly. To get the topology information from the controller, we used RESTful API and then parse the JSON reply to extract the topology data. Figure 4.2.2 illustrates this method.

```
def get_topology():
    topology_url = base_url + "operational/network-topology:network-topology/topology/flow:1"
    response = requests.get(topology_url, auth=HTTPBasicAuth('admin', 'admin'))
    data = json.loads(response.content)
    return data
```

Figure 4.2.2: method to get topology information

Figure 4.2.3 illustrates our method to create the network graph. We used a python library called *networkX* (release 2.0). From the topology data, hosts and switches are inserted as graph vertices and links are inserted as graph edges. Hosts and switches are added into the graph with their respective attributes, such as – hosts has IP, MAC, host id, and attached port (port that it is connected with a switch) and switches has node id. This node id attribute represents the global SID for that node. This id is built by appending the switch number after “10”. In that sense, our segment routing global block (SRGB) starts from 100. For example, switch 4 will have a node ID of 104. Adjacency SIDs are added as link attribute while adding the links into the graph. We declared a range of [6000, 7000] for this. So each switch will have a link id (adjacency SID) within this range.

```
def create_graph(topo):
    print("Creating Network Graph... \n")
    odlNodes = topo["topology"][0]["node"] # Parsing All Nodes
    global odlLinks
    odlLinks = topo["topology"][0]["link"] # Parsing All Links
    segment_id = segment_id_range[0]

    for node in odlNodes:
        n = node['node-id']
        if "host" in n:
            host_list.append(n.encode('ascii', 'ignore'))
            host_id = "H" + n.split(":")[6][1]
            attached_port = node['host-tracker-service:attachment-points'][0]['tp-id']
            sw = "openflow:" + attached_port[9]
            sw_w_h_list.append(sw.encode('ascii', 'ignore'))
            ip = node['host-tracker-service:addresses'][0]['ip']
            mac = node['host-tracker-service:addresses'][0]['mac']
            graph.add_node(n.encode('ascii', 'ignore'), ip=ip, mac=mac, attached_port=attached_port, host_id=host_id)
        else:
            switch_list.append(n.encode('ascii', 'ignore')) # List of all switches in the network
            node_id = "10" + n.split(":")[1]
            graph.add_node(n.encode('ascii', 'ignore'), node_id=node_id)

    for link in odlLinks:
        source_node = link['source']['source-node']
        dest_node = link['destination']['dest-node']
        graph.add_edge(source_node, dest_node, seg_id=segment_id)
        segment_id += 1
```

Figure 4.2.3: method to create network graph

```

def create_table():
    print"Installing Flow Rules... \n"
    for sw in switch_list:
        label = graph.node[sw]["node_id"]
        if sw in sw_w_h_list:
            table0_with_host(sw, label)
            table_group(sw)
            table1(sw)
        else:
            table0_without_host(sw, label)
            table_group(sw)
            table17(sw)
            additional_tables(sw)
    print"Done! \n"

```

Figure 4.2.4: method to create flow tables

As depicted in figure 4.2.4, *create_table* method builds all the necessary flow tables by calling other methods from it. For example, first we check if a switch has a host attached to it. If so, then a ***table0_with_host*** method is called along with ***table1*** method. Otherwise, method ***table0_without_host*** is used and ***table1*** method is not called for these switches. Methods to create table 17, additional tables and group tables are called for each of the switches. Figure 4.2.5 and 4.2.6 give a glimpse of our ***table1*** and ***table17*** method respectively.

```

def table1(sw):
    id = sw[9]
    script.write("#S" + id + ' - Table 1 \n')
    for nodes in sw_w_h_list:
        if nodes==sw:
            continue
        else:
            path_list = list(nx.shortest_path(graph, source=sw, target=nodes))
            label = graph.node[nodes]["node_id"]
            gid = get_egress_port(sw, path_list[1])
            mac = get_host(nodes)
            script.write("sudo ovs-ofctl -O OpenFlow13 add-flow s" + id + " 'ta

```

Figure 4.2.5: method to create table 1

```

def table17(sw):
    id = sw[9]
    script.write("#S" + id + ' - Table 17 \n')
    for nodes in switch_list:
        if nodes==sw:
            continue
        else:
            path_list = list(nx.shortest_path(graph, source=sw, target=nodes))
            label = graph.node[nodes]["node_id"]
            gid = get_egress_port(sw, path_list[1])
            values = [sw, label, gid, nodes]
            table_17_dict_list.append(dict(zip(table_17_keys, values)))
            script.write("sudo ovs-ofctl -O OpenFlow13 add-flow s" + id + " 'ta

```

Figure 4.2.6: method to create table 17

After running our main python script, it generates necessary flow rules for the whole network and writes them into a bash script, as illustrated in figure 4.2.7. Then it runs the bash script to install the flow rules into the switches accordingly. Flow tables of the switches, general connectivity and link failure test results are shown in the next section.

```

#S1 - Table 0
sudo ovs-ofctl -O OpenFlow13 add-flow s1 'table=0,in_port=1,eth_type=0x800,actions=goto_table:1'
sudo ovs-ofctl -O OpenFlow13 add-flow s1 'table=0,eth_type=0x8847,mpls_label=101,mpls_bos=1,actions=pop_mpls:0x800,output:1'
sudo ovs-ofctl -O OpenFlow13 add-flow s1 'table=0,eth_type=0x8847,mpls_label=101,mpls_bos=0,actions=pop_mpls:0x8847,goto_table:17'
sudo ovs-ofctl -O OpenFlow13 add-flow s1 'table=0,eth_type=0x8847,actions=goto_table:17'
#S1 - Group Table
sudo ovs-ofctl -O OpenFlow13 add-group s1 'group_id=2,type=ff,bucket=watch_port:2,output:2,bucket=watch_port:3,resubmit(,3)'
sudo ovs-ofctl -O OpenFlow13 add-group s1 'group_id=3,type=ff,bucket=watch_port:3,output:3,bucket=watch_port:2,resubmit(,2)'
#S1 - Table 1
sudo ovs-ofctl -O OpenFlow13 add-flow s1 'table=1,dl_dst=00:00:00:00:00:03,actions=push_mpls:0x8847,set_field:103->mpls_label,group:3'
sudo ovs-ofctl -O OpenFlow13 add-flow s1 'table=1,dl_dst=00:00:00:00:00:04,actions=push_mpls:0x8847,set_field:104->mpls_label,group:2'
sudo ovs-ofctl -O OpenFlow13 add-flow s1 'table=1,dl_dst=00:00:00:00:00:06,actions=push_mpls:0x8847,set_field:106->mpls_label,group:2'
#S1 - Table 17
sudo ovs-ofctl -O OpenFlow13 add-flow s1 'table=17,eth_type=0x8847,mpls_label=102,actions=group:2'
sudo ovs-ofctl -O OpenFlow13 add-flow s1 'table=17,eth_type=0x8847,mpls_label=103,actions=group:3'
sudo ovs-ofctl -O OpenFlow13 add-flow s1 'table=17,eth_type=0x8847,mpls_label=104,actions=group:2'

```

Figure 4.2.7: bash file to install the flow rules

4.3 Experimental Results and Analysis

We are going to use our reference topology in figure 3.1.3 to demonstrate our recovery scheme. Figure 4.3.1 illustrates the flow rules in table 0 of switch *openflow:1 (PI)*. As it has a host attached to it, there would be 4 flow rules according to our design. There are some other flow rules that the controller installed by default into every switch. These rules are there to facilitate ARP packets for initial ping purpose. When

we start the mininet topology and connect it to the controller, the controller does not know about the hosts. So we have to issue a *pingall* command in mininet so that the controller gets the hosts in its topology inventory. If the controller does not install those ARP flows by default, the *pingall* command would fail.

```
mirja@ubuntu:~$ sudo ovs-ofctl -O OpenFlow13 dump-flows s1 table:0
OFPST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x0, duration=869.252s, table=0, n_packets=3, n_bytes=294, ip,in_port=1 actions=goto table:1
cookie=0x0, duration=869.235s, table=0, n_packets=3, n_bytes=306, mpls,mpls_label=101,mpls_bos=1 actions=pop_mpls:0x0800,output:1
cookie=0x0, duration=869.222s, table=0, n_packets=0, n_bytes=0, mpls,mpls_label=101,mpls_bos=0 actions=pop_mpls:0x8847,goto table:17
cookie=0x2b00000000000004, duration=1007.739s, table=0, n_packets=403, n_bytes=34255, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
cookie=0x0, duration=869.212s, table=0, n_packets=0, n_bytes=0, mpls actions=goto table:17
cookie=0x2b00000000000008, duration=1004.576s, table=0, n_packets=2713, n_bytes=725700, priority=2,in_port=3 actions=output:1,output:2,CONTROLLER:65535
cookie=0x2b00000000000016, duration=1001.074s, table=0, n_packets=14, n_bytes=1078, priority=2,in_port=1 actions=output:2,CONTROLLER:65535
cookie=0x2b00000000000017, duration=1001.074s, table=0, n_packets=4965, n_bytes=1182517, priority=2,in_port=2 actions=output:1
cookie=0x2b00000000000004, duration=1007.739s, table=0, n_packets=15, n_bytes=3168, priority=0 actions=drop
mirja@ubuntu:~$
```

Figure 4.3.1: Flow rules in PI's table 0

PI's table 1 (figure 4.3.2) has three flow rule entries, as there are three other hosts (H3, H4 and H6). The action is to push appropriate SID of the destination node, which is 103, 104 and 106 respectively because these hosts are attached with P3, P4 and P6. Then forward them to appropriate group entries. If we look at PI's group table, we will see two entries, as in table 1 flows are forwarded to either group 2 or group 3 (figure 4.3.3). Each of them has a primary action bucket that outputs to a port (marked red) and a secondary bucket that forward the packet to an additional table (marked green.)

```
mirja@ubuntu:~$
mirja@ubuntu:~$ sudo ovs-ofctl -O OpenFlow13 dump-flows s1 table:1
OFPST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x0, duration=510.873s, table=1, n_packets=0, n_bytes=0, dl_dst=00:00:00:00:00:03 actions=push_mpls:0x8847,set_field:103->mpls_label,group:3
cookie=0x0, duration=510.859s, table=1, n_packets=0, n_bytes=0, dl_dst=00:00:00:00:00:04 actions=push_mpls:0x8847,set_field:104->mpls_label,group:2
cookie=0x0, duration=510.848s, table=1, n_packets=0, n_bytes=0, dl_dst=00:00:00:00:00:06 actions=push_mpls:0x8847,set_field:106->mpls_label,group:2
mirja@ubuntu:~$
```

Figure 4.3.2: Flow rules in PI's table 1

```
mirja@ubuntu:~$
mirja@ubuntu:~$ sudo ovs-ofctl -O OpenFlow13 dump-groups s1
OFPST_GROUP_DESC reply (OF1.3) (xid=0x2):
group_id=2,type=ff,bucket=watch_port:2,actions=output:2,bucket=watch_port:3,actions=resubmit(,3)
group_id=3,type=ff,bucket=watch_port:3,actions=output:3,bucket=watch_port:2,actions=resubmit(,2)
mirja@ubuntu:~$
mirja@ubuntu:~$
```

Figure 4.3.3: Flow rules in PI's group table

To test reachability under normal condition, we generated ping packets from H1 to H6 and used Wireshark to capture these packets. For this purpose, we started Wireshark on PI's both interface 2 and 3. According to the PI's table above, to go to H6, packets should be sent through port 2. Hence, we should not see any packets on port 3.

```
mininet>
mininet> h1 ping -c 2 h6
PING 10.0.0.6 (10.0.0.6) 56(84) bytes of data.
64 bytes from 10.0.0.6: icmp_seq=1 ttl=64 time=1.00 ms
64 bytes from 10.0.0.6: icmp_seq=2 ttl=64 time=0.689 ms

--- 10.0.0.6 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1000ms
rtt min/avg/max/mdev = 0.689/0.846/1.004/0.160 ms
mininet>
```

Figure 4.3.4: successful ping from H1 to H6

We transmitted two ping packets from H1 to H6 and the ping is successful. Figure 4.3.5 shows the Wireshark capture on PI's port 2. There are in total 4 ICMP packets because of two pings. In the ICMP request packet, we see the label 106 in the MPLS header. Similarly, for an ICMP reply packet, there is the label 101 as P6 pushes this label to reach P1. And there is no ICMP packet captured on PI's port 3 (figure 4.3.6).

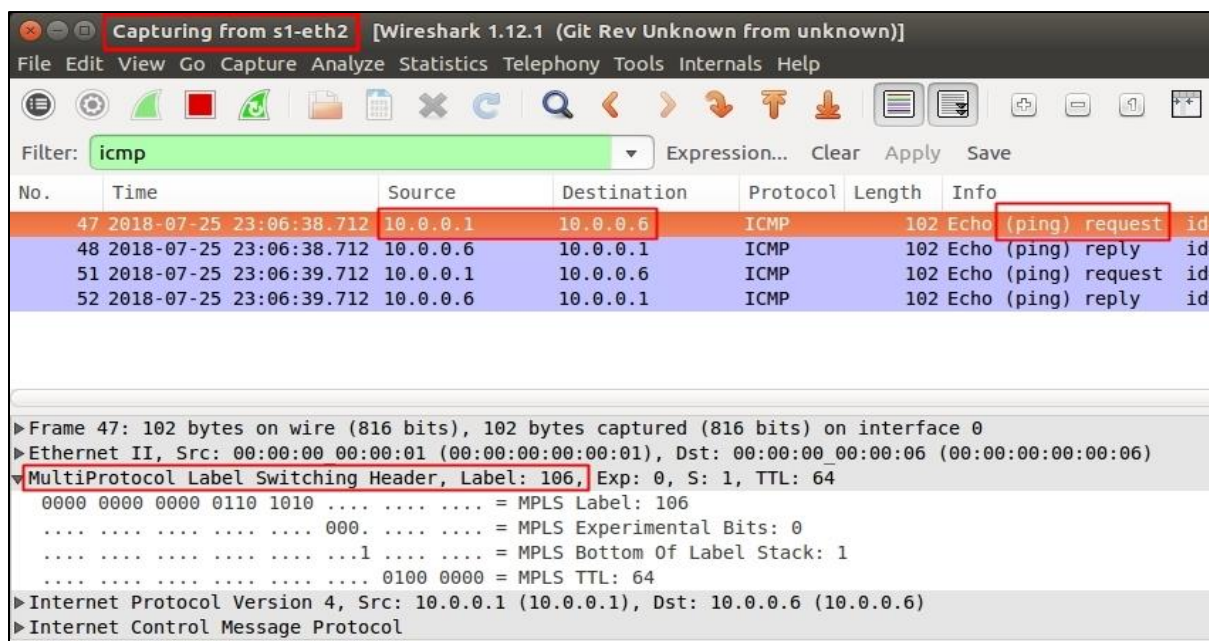


Figure 4.3.5(a): ICMP request packet on PI's port 2

Filter: icmp

No.	Time	Source	Destination	Protocol	Length	Info
47	2018-07-25 23:06:38.712	10.0.0.1	10.0.0.6	ICMP	102	Echo (ping) request
48	2018-07-25 23:06:38.712	10.0.0.6	10.0.0.1	ICMP	102	Echo (ping) reply
51	2018-07-25 23:06:39.712	10.0.0.1	10.0.0.6	ICMP	102	Echo (ping) request
52	2018-07-25 23:06:39.712	10.0.0.6	10.0.0.1	ICMP	102	Echo (ping) reply

Frame 48: 102 bytes on wire (816 bits), 102 bytes captured (816 bits) on interface 0
 Ethernet II, Src: 00:00:00 00:00:06 (00:00:00:00:00:06), Dst: 00:00:00 00:00:01 (00:00:00:00:00:01)
 MultiProtocol Label Switching Header, Label: 101, Exp: 0, S: 1, TTL: 64
 0000 0000 0000 0110 0101 = MPLS Label: 101
 000. = MPLS Experimental Bits: 0
 1 = MPLS Bottom Of Label Stack: 1
 0100 0000 = MPLS TTL: 64
 Internet Protocol Version 4, Src: 10.0.0.6 (10.0.0.6), Dst: 10.0.0.1 (10.0.0.1)
 Internet Control Message Protocol

Figure 4.3.5(b): ICMP reply packet on P1's port 2

Filter: icmp

No.	Time	Source	Destination	Protocol	Length	Info
-----	------	--------	-------------	----------	--------	------

Figure 4.3.6: no packets on P1's port 3

4.3.1 Failure Recovery Time

Now to test the recovery from link failure, we captured packets on port 2 and port 3 of P6. Continuous ping packets was generated from H1 to H6 with an interval of 1 ms. We did three test cases as there are three switch links along the primary path from H1 to H6 (H1 → P1 → P2 → P4 → P6 → H2). In the first test case, we failed the P1-P2 link, the second test case P2-P4 link, and the P4-P6 link for the third test case. We simulated each test case 10 times, so in total we have 30 simulation results to calculate the average recovery time.

We calculated the recovery time using equation (4) below:

$$(T_F - T_L) - (F - L) \quad (4)$$

Where L = the sequence number of the last packet arrived before the link fails

F = the sequence number of the first packet arrived after the link recovery

T_L = the arrival timestamp of packet L

T_F = the arrival timestamp of packet F

Test Case 1: P1<->P2 link failure

Figure 4.3.7 and 4.3.8 illustrate the Wireshark capture on P6's port 2 and 3 respectively. As we can see from figure 4.3.7, there is normal traffic (an ICMP request packet followed by a reply packet) up to packet sequence 526. Then there are only reply packets on this interface. This is because P6 has no idea about the P1<->P2 link failure. So when sending the reply, it chooses the normal path and send the packet to P4. P4 forwards the packet to P2 and as P2 is aware about the failure, so it sends the packet to P5 and make sure it takes the path through P3 by pushing label 103 on top of original label 101. This can be shown by capturing packets on P5's port 2.

No.	Time	Source	Destination	Protocol	Length	Info
2919	2018-07-26 18:58:51.818	10.0.0.6	10.0.0.1	ICMP	102	Echo (ping) reply id=0x148f, seq=522/2562, ttl=64
2920	2018-07-26 18:58:51.819	10.0.0.1	10.0.0.6	ICMP	102	Echo (ping) request id=0x148f, seq=523/2818, ttl=64
2921	2018-07-26 18:58:51.819	10.0.0.6	10.0.0.1	ICMP	102	Echo (ping) reply id=0x148f, seq=523/2818, ttl=64
2922	2018-07-26 18:58:51.820	10.0.0.1	10.0.0.6	ICMP	102	Echo (ping) request id=0x148f, seq=524/3074, ttl=64
2923	2018-07-26 18:58:51.820	10.0.0.6	10.0.0.1	ICMP	102	Echo (ping) reply id=0x148f, seq=524/3074, ttl=64
2924	2018-07-26 18:58:51.823	10.0.0.1	10.0.0.6	ICMP	102	Echo (ping) request id=0x148f, seq=525/3330, ttl=64
2925	2018-07-26 18:58:51.824	10.0.0.6	10.0.0.1	ICMP	102	Echo (ping) reply id=0x148f, seq=525/3330, ttl=64
2926	2018-07-26 18:58:51.826	10.0.0.1	10.0.0.6	ICMP	102	Echo (ping) request id=0x148f, seq=526/3586, ttl=64
2927	2018-07-26 18:58:51.826	10.0.0.6	10.0.0.1	ICMP	102	Echo (ping) reply id=0x148f, seq=526/3586, ttl=64
2928	2018-07-26 18:58:51.854	10.0.0.6	10.0.0.1	ICMP	102	Echo (ping) reply id=0x148f, seq=528/4098, ttl=64
2929	2018-07-26 18:58:51.860	10.0.0.6	10.0.0.1	ICMP	102	Echo (ping) reply id=0x148f, seq=529/4354, ttl=64
2930	2018-07-26 18:58:51.861	10.0.0.6	10.0.0.1	ICMP	102	Echo (ping) reply id=0x148f, seq=530/4610, ttl=64
2931	2018-07-26 18:58:51.862	10.0.0.6	10.0.0.1	ICMP	102	Echo (ping) reply id=0x148f, seq=531/4866, ttl=64
2932	2018-07-26 18:58:51.864	10.0.0.6	10.0.0.1	ICMP	102	Echo (ping) reply id=0x148f, seq=532/5122, ttl=64
2933	2018-07-26 18:58:51.865	10.0.0.6	10.0.0.1	ICMP	102	Echo (ping) reply id=0x148f, seq=533/5378, ttl=64

Frame 2909: 102 bytes on wire (816 bits), 102 bytes captured (816 bits) on interface 0
 Ethernet II, Src: 00:00:00:00:00:06 (00:00:00:00:00:06), Dst: 00:00:00:00:00:01 (00:00:00:00:00:01)
 MultiProtocol Label Switching Header, Label: 101, Exp: 0, S: 1, TTL: 64
 0000 0000 0110 0101 = MPLS Label: 101
 = MPLS Experimental Bits: 0
 = MPLS Bottom Of Label Stack: 1
 0100 0000 = MPLS TTL: 64

Figure 4.3.7: P1 <-> P2 link failure – Capture on P6's port 2

As we can see from figure 4.3.8, the first captured packet on P6's port 3 is with sequence number 528. So packets up to seq. 526 was coming from H1 through normal path, one packet lost (seq 527) and from seq. 528 we start receiving packet from H1 on P6's port 3. As we can see, there are only request packets on this interface because P6 is sending replies to port 2. If we look at the timestamps, P6 receives the request packets with seq. 526 and 528 on 18:58:51.826 (in hh:mm:ss.ms format) and 18:58:51.852 respectively.

No.	Time	Source	Destination	Protocol	Info
2735	2018-07-26 18:58:51.852	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x148f, seq=528/4098, ttl=64
2736	2018-07-26 18:58:51.859	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x148f, seq=529/4354, ttl=64
2737	2018-07-26 18:58:51.861	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x148f, seq=530/4610, ttl=64
2738	2018-07-26 18:58:51.862	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x148f, seq=531/4866, ttl=64
2739	2018-07-26 18:58:51.863	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x148f, seq=532/5122, ttl=64
2740	2018-07-26 18:58:51.865	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x148f, seq=533/5378, ttl=64
2741	2018-07-26 18:58:51.866	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x148f, seq=534/5634, ttl=64
2742	2018-07-26 18:58:51.868	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x148f, seq=535/5890, ttl=64
2743	2018-07-26 18:58:51.869	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x148f, seq=536/6146, ttl=64
2744	2018-07-26 18:58:51.871	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x148f, seq=537/6402, ttl=64

Frame 2735: 102 bytes on wire (816 bits), 102 bytes captured (816 bits) on interface 0
 Ethernet II, Src: 00:00:00:00:00:01 (00:00:00:00:00:01), Dst: 00:00:00:00:00:06 (00:00:00:00:00:06)
 MultiProtocol Label Switching Header, Label: 106, Exp: 0, S: 1, TTL: 64
 0000 0000 0000 0110 1010 = MPLS Label: 106
 = MPLS Experimental Bits: 0
 = MPLS Bottom Of Label Stack: 1
 = MPLS TTL: 64
 Internet Protocol Version 4, Src: 10.0.0.1 (10.0.0.1), Dst: 10.0.0.6 (10.0.0.6)
 Internet Control Message Protocol

Figure 4.3.8: P1 <-> P2 link failure – Capture on P6's port 3

From these timestamps, we can calculate the time difference using eq. (4), which is 24 ms. This is our recovery time for test case 1 simulation 01. Similarly, we did nine other simulations for this link and recorded the details in table 4.3.1.

	Packet 1		Packet 2		Recovery Time (ms)
	Seq.	Timestamp	Seq.	Timestamp	
Sim-01	526	18.58.51.826	528	18.58.51.852	24
Sim-02	388	19.00.42.912	390	19.00.42.927	13
Sim-03	436	19.02.23.528	437	19.02.23.549	20
Sim-04	415	19.07.13.210	416	19.07.13.228	17
Sim-05	387	19.08.48.422	388	19.08.48.438	15
Sim-06	351	19.26.10.333	353	19.26.10.355	20
Sim-07	325	19.28.40.485	326	19.28.40.499	13
Sim-08	289	19.34.51.630	292	19.34.51.653	20
Sim-09	328	19.37.03.394	330	19.37.03.419	23
Sim-10	281	19.38.55.566	283	19.38.55.590	22
Test Case 1 - Average Recovery Time					18.7

Table 4.3.1: Recovery time simulation results for P1 <-> P2 link failure

Test Case 2: P2<->P4 link failure

This test case also shows crankback routing in place. Similar as previous test case, we identified two packets. One is with seq. 342 on P6's port 2. This is the last packet on normal condition. Another packet is with seq. 344 on P6's port 3, which is the first packet captured on this interface. From their timestamp, we get the recovery time 25 ms. All simulation results are recorded in table 2.

No.	Time	Source	Destination	Protocol	Info
733	2018-07-26 23:56:36.952	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x4376, seq=339/21249, ttl=64
734	2018-07-26 23:56:36.953	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=339/21249, ttl=64
735	2018-07-26 23:56:36.954	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x4376, seq=340/21505, ttl=64
736	2018-07-26 23:56:36.955	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=340/21505, ttl=64
737	2018-07-26 23:56:36.958	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x4376, seq=341/21761, ttl=64
738	2018-07-26 23:56:36.958	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=341/21761, ttl=64
739	2018-07-26 23:56:36.964	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x4376, seq=342/22017, ttl=64
740	2018-07-26 23:56:36.965	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=342/22017, ttl=64
741	2018-07-26 23:56:36.991	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=344/22529, ttl=64
742	2018-07-26 23:56:36.992	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=344/22529, ttl=64
743	2018-07-26 23:56:37.000	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=345/22785, ttl=64
744	2018-07-26 23:56:37.001	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=345/22785, ttl=64
745	2018-07-26 23:56:37.004	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=346/23041, ttl=64
746	2018-07-26 23:56:37.004	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=346/23041, ttl=64

Figure 4.3.9: P2 <-> P4 link failure – Capture on P6's port 2

No.	Time	Source	Destination	Protocol	Info
77	2018-07-26 23:56:36.991	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x4376, seq=344/22529, ttl=64
78	2018-07-26 23:56:36.992	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=344/22529, ttl=64
79	2018-07-26 23:56:37.000	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x4376, seq=345/22785, ttl=64
80	2018-07-26 23:56:37.001	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=345/22785, ttl=64
81	2018-07-26 23:56:37.004	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x4376, seq=346/23041, ttl=64
82	2018-07-26 23:56:37.004	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=346/23041, ttl=64
83	2018-07-26 23:56:37.005	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x4376, seq=347/23297, ttl=64
84	2018-07-26 23:56:37.006	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=347/23297, ttl=64
85	2018-07-26 23:56:37.007	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x4376, seq=348/23553, ttl=64
86	2018-07-26 23:56:37.008	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=348/23553, ttl=64
87	2018-07-26 23:56:37.008	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x4376, seq=349/23809, ttl=64
88	2018-07-26 23:56:37.009	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=349/23809, ttl=64
89	2018-07-26 23:56:37.015	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x4376, seq=350/24065, ttl=64
90	2018-07-26 23:56:37.016	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=350/24065, ttl=64
91	2018-07-26 23:56:37.017	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x4376, seq=351/24321, ttl=64

Figure 4.3.10: P2 <-> P4 link failure – Capture on P6's port 3

If we analyze the packets on port 2, we can see that there are two reply packets for the same sequence number after the link failure occurs (from seq. 344). This is because P6 forward the reply packets on its port 2, but then P4 cannot send that packet to P2 because of the failed link. So it pushes a new label to avoid forwarding loop and send

the reply packet back to P6 on the incoming port. That's why on the captured image we are seeing two reply packets. Figure 4.3.II(a) and 4.3.II(b) illustrates this.

Two reply packet pair (seq. 351) is highlighted. As we can see, one of them has only one label (101), which is the original reply forwarded by the P6. The next one has two labels, 105 and 101, which is the crankbacked reply packet forwarded by P4. Then P6 forwards the packet to P5 by seeing the top label 105 and from there it follows the shortest path to P1 according to P5's flow table.

Filter: icmp Expression... Clear Apply Save

No.	Time	Source	Destination	Protoc	Info
750	2018-07-26 23:56:37.00	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=348/23553, ttl=64
751	2018-07-26 23:56:37.00	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=349/23809, ttl=64
752	2018-07-26 23:56:37.00	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=349/23809, ttl=64
753	2018-07-26 23:56:37.01	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=350/24065, ttl=64
754	2018-07-26 23:56:37.01	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=350/24065, ttl=64
755	2018-07-26 23:56:37.01	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=351/24321, ttl=64
756	2018-07-26 23:56:37.01	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=351/24321, ttl=64
757	2018-07-26 23:56:37.02	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=352/24577, ttl=64
758	2018-07-26 23:56:37.02	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=352/24577, ttl=64
759	2018-07-26 23:56:37.02	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=353/24833, ttl=64

Frame 755: 102 bytes on wire (816 bits), 102 bytes captured (816 bits) on interface 0
 Ethernet II, Src: 00:00:00 00:00:06 (00:00:00:00:00:06), Dst: 00:00:00 00:00:01 (00:00:00:00:00:01)
 MultiProtocol Label Switching Header, Label: 101, Exp: 0, S: 1, TTL: 64
 0000 0000 0000 0110 0101 = MPLS Label: 101
 000. = MPLS Experimental Bits: 0
 1 = MPLS Bottom Of Label Stack: 1
 0100 0000 = MPLS TTL: 64
 Internet Protocol Version 4, Src: 10.0.0.6 (10.0.0.6), Dst: 10.0.0.1 (10.0.0.1)
 Internet Control Message Protocol

Figure 4.3.II(a): Crankback routing on P2 <-> P4 link failure

Filter: icmp Expression... Clear Apply Save

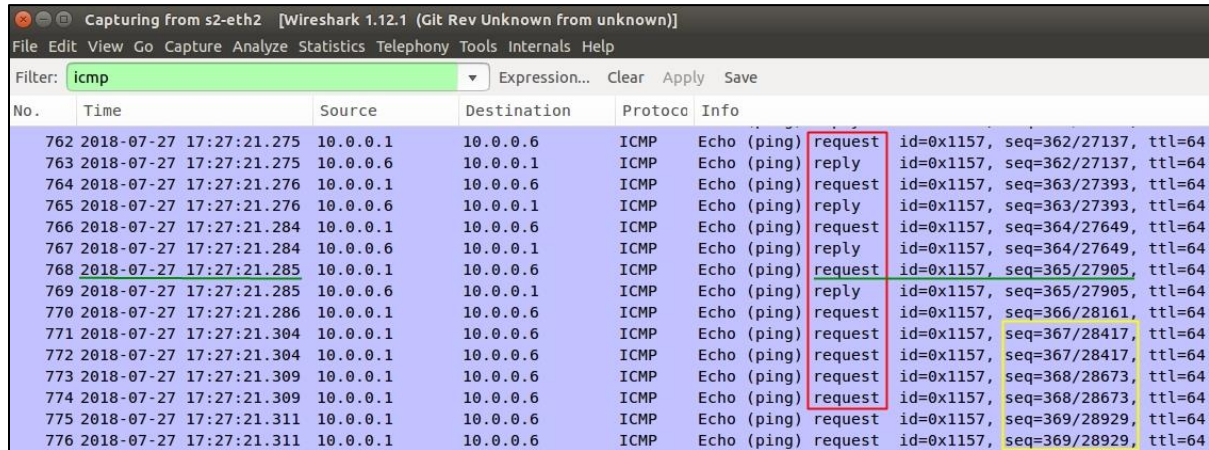
No.	Time	Source	Destination	Protoc	Info
750	2018-07-26 23:56:37.00	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=348/23553, ttl=64
751	2018-07-26 23:56:37.00	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=349/23809, ttl=64
752	2018-07-26 23:56:37.00	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=349/23809, ttl=64
753	2018-07-26 23:56:37.01	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=350/24065, ttl=64
754	2018-07-26 23:56:37.01	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=350/24065, ttl=64
755	2018-07-26 23:56:37.01	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=351/24321, ttl=64
756	2018-07-26 23:56:37.01	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=351/24321, ttl=64
757	2018-07-26 23:56:37.02	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=352/24577, ttl=64
758	2018-07-26 23:56:37.02	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=352/24577, ttl=64
759	2018-07-26 23:56:37.02	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x4376, seq=353/24833, ttl=64

Ethernet II, Src: 00:00:00 00:00:06 (00:00:00:00:00:06), Dst: 00:00:00 00:00:01 (00:00:00:00:00:01)
 MultiProtocol Label Switching Header, Label: 105, Exp: 0, S: 0, TTL: 64
 0000 0000 0000 0110 1001 = MPLS Label: 105
 000. = MPLS Experimental Bits: 0
 0 = MPLS Bottom Of Label Stack: 0
 0100 0000 = MPLS TTL: 64
 MultiProtocol Label Switching Header, Label: 101, Exp: 0, S: 1, TTL: 64
 0000 0000 0000 0110 0101 = MPLS Label: 101
 000. = MPLS Experimental Bits: 0
 1 = MPLS Bottom Of Label Stack: 1
 0100 0000 = MPLS TTL: 64

Figure 4.3.II(b): Crankback routing on P2 <-> P4 link failure (cont.)

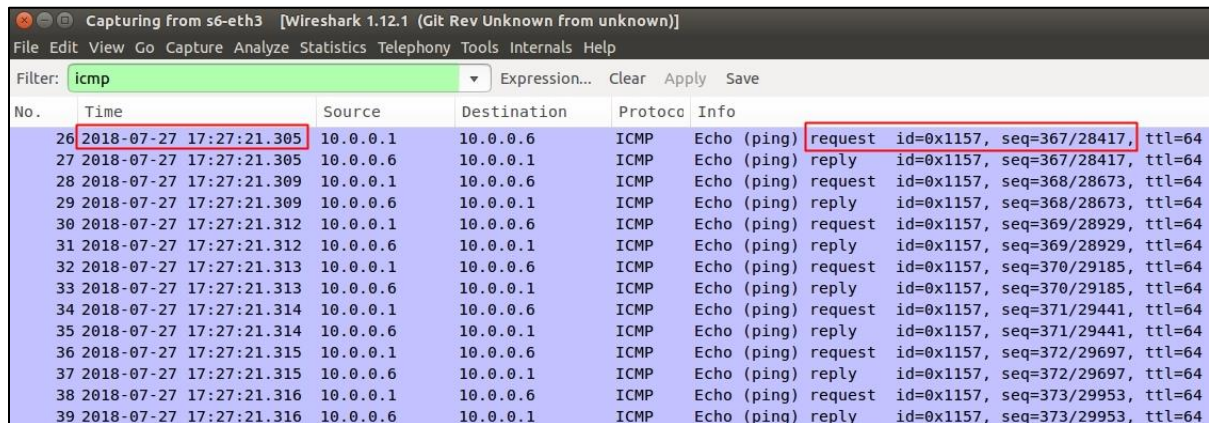
Test Case 3: P4<->P6 link failure

For this test, we cannot capture packets on P6's port 2 because as soon as we shut down the link, Wireshark stops at that port showing error message. Instead, for this test case we captured packets on P2's port 2 and P6's port 3. So we will see request packets going through P2's port 2 and then crankbacked by P4 and finally received on P6's port 3.



No.	Time	Source	Destination	Protocol	Info
762	2018-07-27 17:27:21.275	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x1157, seq=362/27137, ttl=64
763	2018-07-27 17:27:21.275	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x1157, seq=362/27137, ttl=64
764	2018-07-27 17:27:21.276	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x1157, seq=363/27393, ttl=64
765	2018-07-27 17:27:21.276	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x1157, seq=363/27393, ttl=64
766	2018-07-27 17:27:21.284	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x1157, seq=364/27649, ttl=64
767	2018-07-27 17:27:21.284	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x1157, seq=364/27649, ttl=64
768	2018-07-27 17:27:21.285	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x1157, seq=365/27905, ttl=64
769	2018-07-27 17:27:21.285	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x1157, seq=365/27905, ttl=64
770	2018-07-27 17:27:21.286	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x1157, seq=366/28161, ttl=64
771	2018-07-27 17:27:21.304	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x1157, seq=367/28417, ttl=64
772	2018-07-27 17:27:21.304	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x1157, seq=367/28417, ttl=64
773	2018-07-27 17:27:21.309	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x1157, seq=368/28673, ttl=64
774	2018-07-27 17:27:21.309	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x1157, seq=368/28673, ttl=64
775	2018-07-27 17:27:21.311	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x1157, seq=369/28929, ttl=64
776	2018-07-27 17:27:21.311	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x1157, seq=369/28929, ttl=64

Figure 4.3.12: P4 <-> P6 link failure – Capture on P2's port 2



No.	Time	Source	Destination	Protocol	Info
26	2018-07-27 17:27:21.305	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x1157, seq=367/28417, ttl=64
27	2018-07-27 17:27:21.305	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x1157, seq=367/28417, ttl=64
28	2018-07-27 17:27:21.309	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x1157, seq=368/28673, ttl=64
29	2018-07-27 17:27:21.309	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x1157, seq=368/28673, ttl=64
30	2018-07-27 17:27:21.312	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x1157, seq=369/28929, ttl=64
31	2018-07-27 17:27:21.312	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x1157, seq=369/28929, ttl=64
32	2018-07-27 17:27:21.313	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x1157, seq=370/29185, ttl=64
33	2018-07-27 17:27:21.313	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x1157, seq=370/29185, ttl=64
34	2018-07-27 17:27:21.314	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x1157, seq=371/29441, ttl=64
35	2018-07-27 17:27:21.314	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x1157, seq=371/29441, ttl=64
36	2018-07-27 17:27:21.315	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x1157, seq=372/29697, ttl=64
37	2018-07-27 17:27:21.315	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x1157, seq=372/29697, ttl=64
38	2018-07-27 17:27:21.316	10.0.0.1	10.0.0.6	ICMP	Echo (ping) request id=0x1157, seq=373/29953, ttl=64
39	2018-07-27 17:27:21.316	10.0.0.6	10.0.0.1	ICMP	Echo (ping) reply id=0x1157, seq=373/29953, ttl=64

Figure 4.3.13: P4 <-> P6 link failure – Capture on P6's port 3

As we can see in the figure 4.3.12, a request packet (seq. 365) go through P2's port 2 with the timestamp 17:27:21.285 and get a reply. This was the last packet before failure. The request packet with seq. 366 is sent by P2 but we didn't see any reply packet for this on neither interface. So we lost one packet here. Also, we can see after that there are a series of two request packets with same seq (yellow marked) because of the

crankback routing. On P6's port 3 capture (figure 4.3.13), first packet is the request packet with seq 367 and from the timestamp we can calculate our recovery time as 18 ms. Other simulation result for this test and other two test cases are summarized in table 4.3.2.

	S-01	S-02	S-03	S-04	S-05	S-06	S-07	S-08	S-09	S-10	Avg.
P1-P2	24	13	20	17	15	20	13	20	23	22	18.7
P2-P4	25	20	22	23	15	20	14	14	16	13	18.2
P4-P6	17	19	16	20	17	21	18	21	16	16	18.1
Average Recovery Time											18.3

Table 4.3.2: Summary of recovery time simulation results

Figure 4.3.14 shows the occurrence graph for these recovery times. As we can see, most of our simulation results are confined within 16-20 ms range (15 out of 30). 20 ms is most frequent with 6 times, followed by 16 ms (4 times). No particular trendline is observed as these recovery times vary randomly.

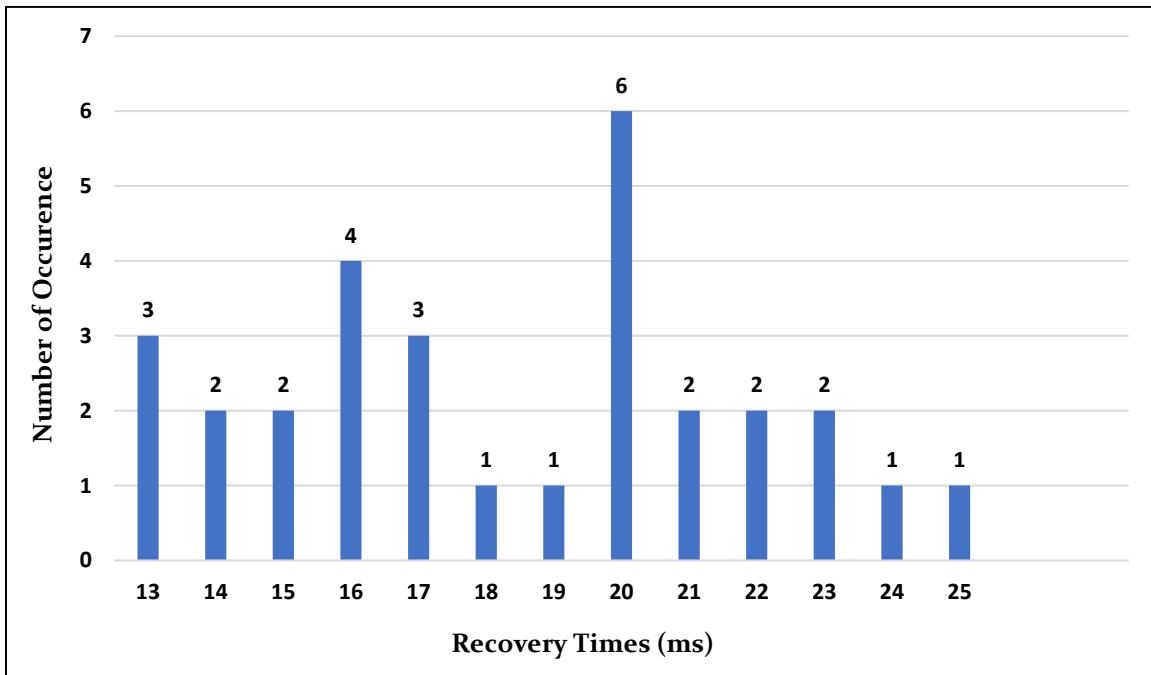


Figure 4.3.14: Occurrence graph of recovery times with N=6, H=4

Comparison:

In this section, we compare the performance of our proposed recovery approach with the work done in [26], [38] and [39]. Reason for choosing these is because they have described their simulation environment, topology used, generating traffic and failing links elaborately so it is possible to simulate and compare their work with our proposed approach under same environment and parameters. Moreover, all these approaches is based on link protection, which is also same as our approach. Simulation environment specifications are given in table 4.3.3 –

<i>Previous Works</i>	<i>Specifications</i>
In [26]	Intel Core i5-2400 CPU 3.10 GHz, 4 GB RAM, Ubuntu 11.04
In [38]	Intel Core i3-2120 CPU 3.30GHz, Ubuntu 12.04
In [39]	Nothing particularly specified
This paper	Core i5-6200U CPU 2.40 GHz, 8.00 GB RAM, Ubuntu 14.04.5

Table 4.3.3: Simulation environment specifications in previous works

Although in [39] no particular environment specification is given, they stated that a VM environment is used, not a production environment. Also, recovery time comparison with this paper [39] is close to our result. The clear advantage of our work over theirs is in terms of memory requirement, which we will discuss in section 4.3.2. All these works used a Linux VM, Mininet and controller inside it, which is same as ours. As the controller, NOX is used in [26], Ryu in the other two, and we used OpenDaylight. We created the same topologies used by them, generated ping between the same hosts and failed the same links to make the results comparable.

In [26], authors proposed an auto-reject mechanism for segment protection in ethernet network. They had to introduce a new packet type (0x88dd) to remove the flows. As a result, an enhancement in current OpenFlow standard is required and their recovery time increases. Packet forwarding is based on priority-based flow rules and every switch in the network had to maintain flow rules for every flow, resulting in a huge number of flow entries (discussed in section 4.3.2). Authors of [38] proposed a fast failover method

incorporating a congestion aware switchover mechanism, thus increases recovery time. Two protection methods are described in [39], such as “*pro flow*” – protection for each link of each flow and “*pro vlan*” – protection using vlan id. Although their recovery time is close to us, both the methods focus on reaching the other end of the failed link, which results in unnecessary traversing of packets and recovery time will be longer in certain topologies. This scenario is described in section 3.1 (figure 3.1.9).

Figure 4.3.15 illustrates topologies used in [26] and [38] respectively. In [26], they have used two ring topologies with 5 and 13 switches ($N=5, 13$) and one host per switch ($h=1$). Topology in [38] has six switches with one host each. We created all three topologies in our environment, installed required flow entries using our python script and run the simulation 10 times to get an average recovery time. In each case, H1 pings H3 and link $P2 \leftrightarrow P3$ fails. Figure 4.3.16 shows the comparison graph of average recovery time.

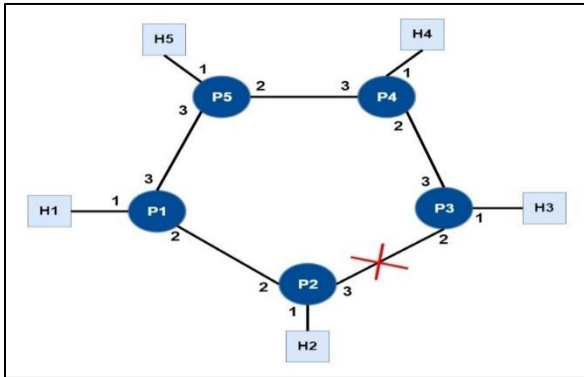


Figure 4.3.15(a): Topology used in [26]
($N = 5, h = 1$)

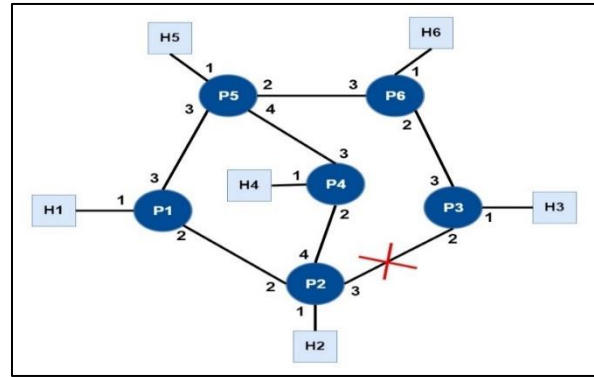


Figure 4.3.15(b): Topology used in [38]
($N = 6, h = 1$)

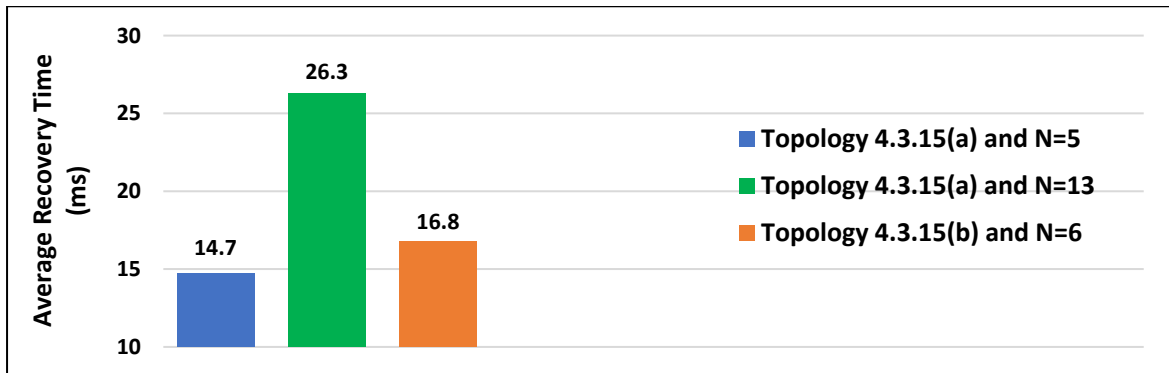


Figure 4.3.16: Average recovery time with the proposed approach in different topology ($h=1$)

As we can see, we have been able to achieve an average recovery time of 14.7ms and 26.3 ms in a ring topology with 5 and 13 nodes respectively. However, the recovery times are 79.4 ms and 80.9 ms with the approach proposed in [26]. The higher recovery time with their approach is caused by the removal of flow entries before new backup flow rules can be activated. We get an average recovery time of 16.8 ms with the topology used in [38], whereas their result shows average recovery time of 31.7 ms. Their switchover mechanism checks for link congestion which results in a higher recovery time.

In [39], they have used a grid network (8x8) of 64 switches with each has a single host attached to it. Host pair and failed links are chosen randomly for each simulation. We did the same and run 10 simulations to get an average recovery time of 20.3 ms. In their paper they compared the result of two proposed methods with the restoration approach (recovery with controller intervention). So in figure 4.3.17, we showed comparison for all four of them (restoration, pro-flow, pro-vlan and proposed approach).

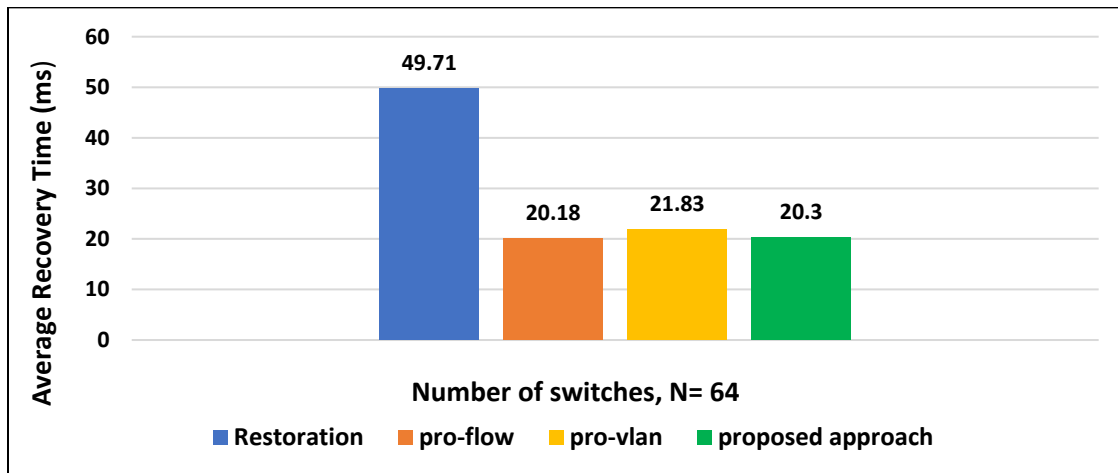


Figure 4.3.17: Average recovery time comparison between [39] and proposed method

Recovery time with the restoration approach is noticeably higher as it needs controller intervention to recover. However, their pro-flow method recovery time is slightly less than our result and the other method, pro-vlan is also close. In this case a significant advantage of our approach is in terms of memory requirement, means the number of flow entries required to implement the recovery scheme. This comparison is shown in the next section.

4.3.2 Memory Requirement

In an efficient failure recovery scheme, use of switch's memory resource is as important as reducing the recovery time. If a fast recovery method needs thousands of extra flow entries in each switch, then it is not efficient in terms of time-space consideration. This leads us to the memory requirement computation by our proposed approach and compare it with previous works mentioned in section 4.3.1.

We can calculate the total flow entries (T_f) required by our scheme using eq. (5). A point to mention here is in our proposed design (Chapter 3) we showed single host attached to the switches, but our design can accommodate multiple hosts per switch with a little modification (discussed in section 5.2). Equation (5) gives us the number of total flow entries even for a multiple host scenario.

$$T_f = \{N * (4 + H)\} + (M * 2) + \{2 * (N + M) * (N + M - 1)\} + 2 * (L - H) \quad (5)$$

Where, N = number of switches with host attached (PE nodes)

M = number of switches without any host (P nodes)

H = total number of hosts

L = total number of links in the network

The derivation of this equation is as follows: first two terms calculates all the entries in table 0 and 1. Each PE has in total $(4+H)$ flow entries to be able to forward the traffic to every host. It has 4 basic flow entries in table 0 and H number of entries which can be found in table 0 or table 1. The second term indicates that each P switch has 2 basic entries in table 0. A switch, regarding it is a PE or not, needs $(N+M-1)$ flow entries to reach every other switch in the network (table 17). In addition, it needs additional $(N+M-1)$ flow entries as a backup for link protection (additional tables). Therefore, the total number of entries used to reach among switches for all the switches is $2*(N+M)*(N+M-1)$. Last term gives us the total number of group entries. As these are created based on the interface, so a switch will have the same number of group entries as its ports connected with other switches. We need to exclude the host link from here. So $(L-H)$ gives us the links only between the switches. It is multiplied by 2 because each

link is connected with two switches and each of them will have an entry for that link. So every link is counted twice.

In [26], number of flow entries per switch has been shown with varying number of switches and attached hosts in the ring topology. They have used $N=\{5,7,9,11,13\}$ and $h=\{1,2,4\}$. So we also calculate the total flow entries required by our method for same numbers of N using eq. (5). For example, when $N=7$ and $h=2$ (every switch has two hosts attached), $M=0$, $H=14$ and total links, L will be: 7 (between switches) + 14 (for hosts) = 21. Then from eq. (5) we get $T_f = 224$. So the average flow rules per switch would be $\frac{T_f}{N+M} = 32$. Figure 4.3.18(a), 4.3.18(b) and 4.3.18(c) shows the comparison with $h=1, 2$ and 4 respectively.

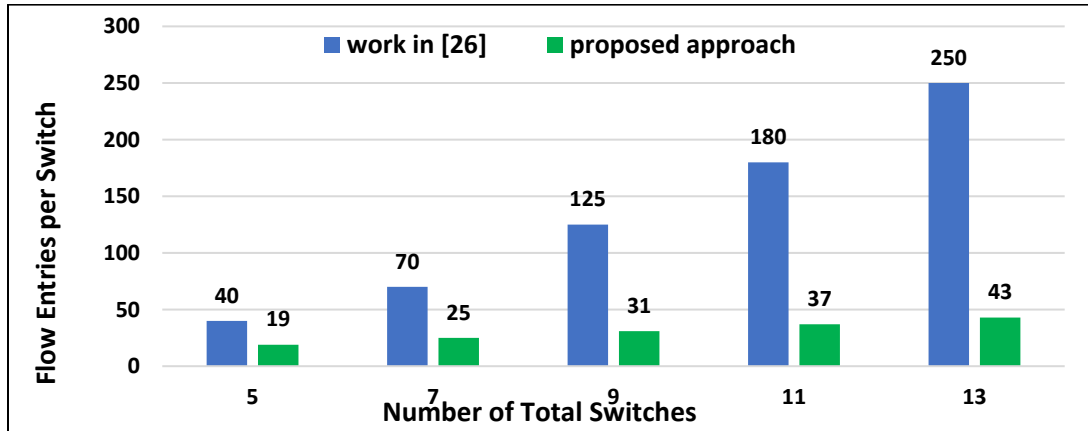


Figure 4.3.18(a): Comparison graph for flow entries per switch, $h = 1$

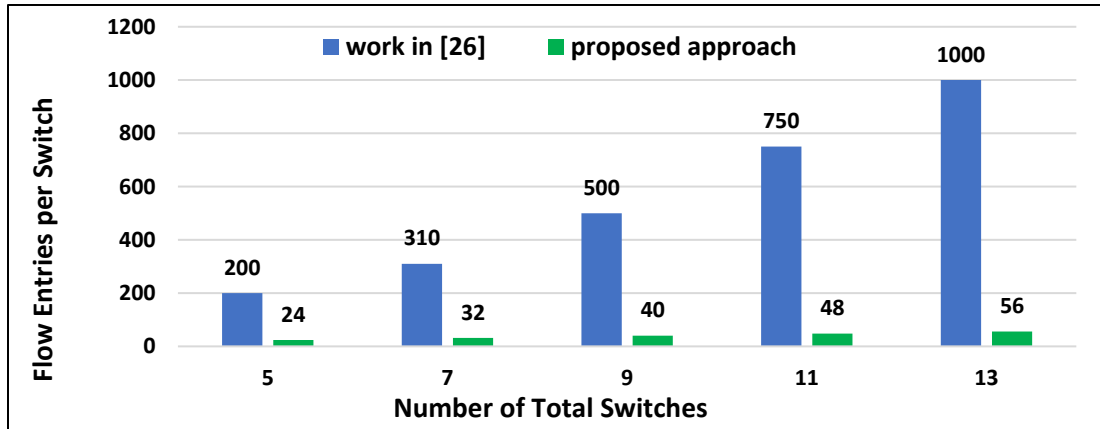


Figure 4.3.18(b): Comparison graph for flow entries per switch, $h = 2$

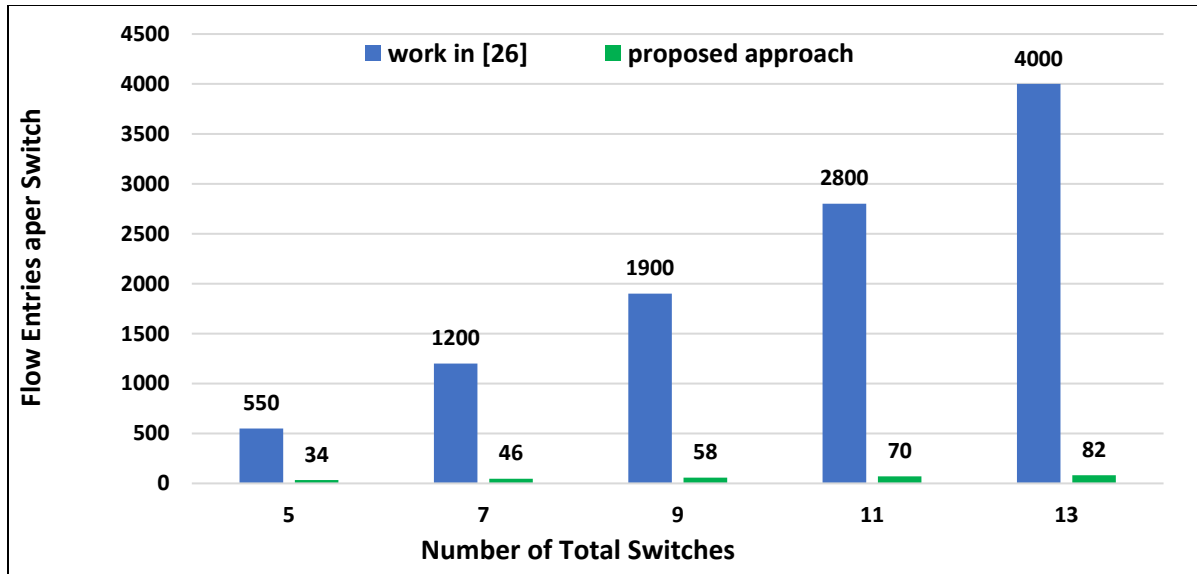


Figure 4.3.18(c): Comparison graph for flow entries per switch, $h = 4$

As we can see from the graph, in previous work, number of flow entries per switch grows rapidly with the increasing switch number and hosts attached per switch, whereas in our proposed approach it's almost linear. The trendline graphs shown in figure 4.3.19(a) and 4.3.19(b) validates our inference as we can see the total number of entries increasing exponentially with the method in [26], but it's a linear increase with our proposed approach.

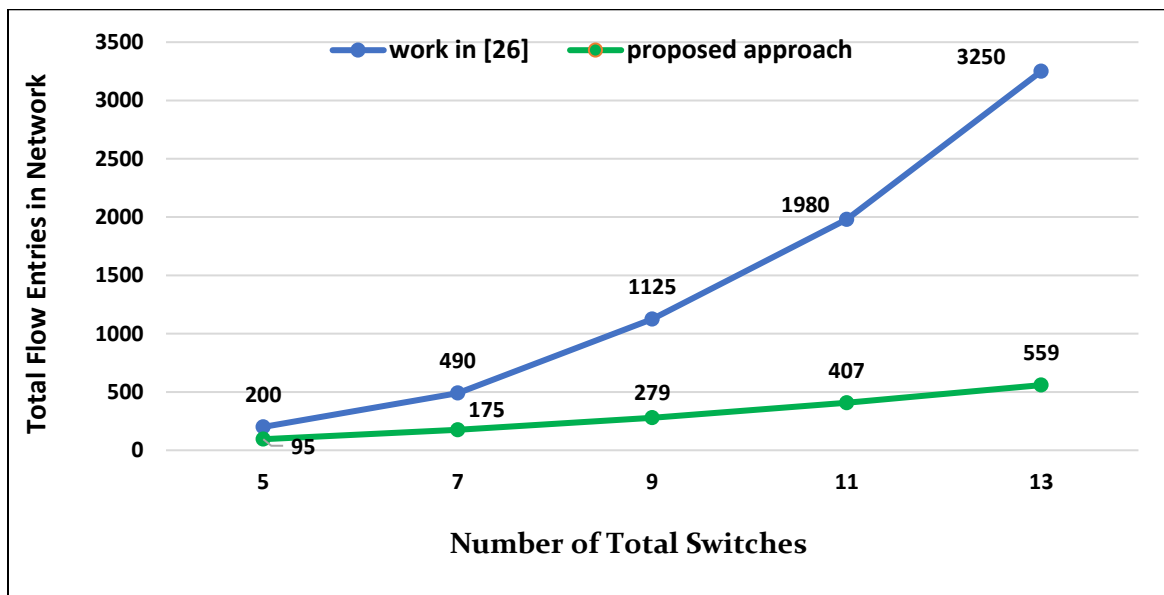


Figure 4.3.19(a): trendline for total flow entries in network, $h=1$

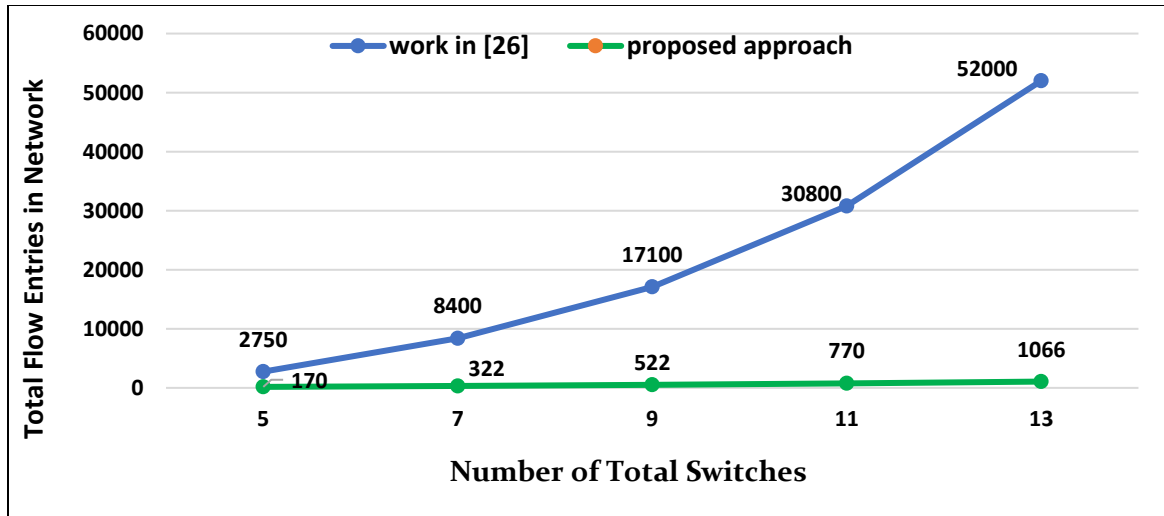


Figure 4.3.19(b): trendline for total flow entries in network, $h=4$

In [38], they have used the same topology shown in figure 4.3.15(b), with a varying number of hosts ($h = 1, 2, 4, 8, 16$) attached to each switch. Figure 4.3.20 shows the flow entry comparison per switch. As we can see, with their approach the number of flow entries per switch grows exponentially and with 16 hosts attached per switch, each switch has 6416 flow entries on an average. Whereas in the same scenario, our method has the average number of flow entries of 112.

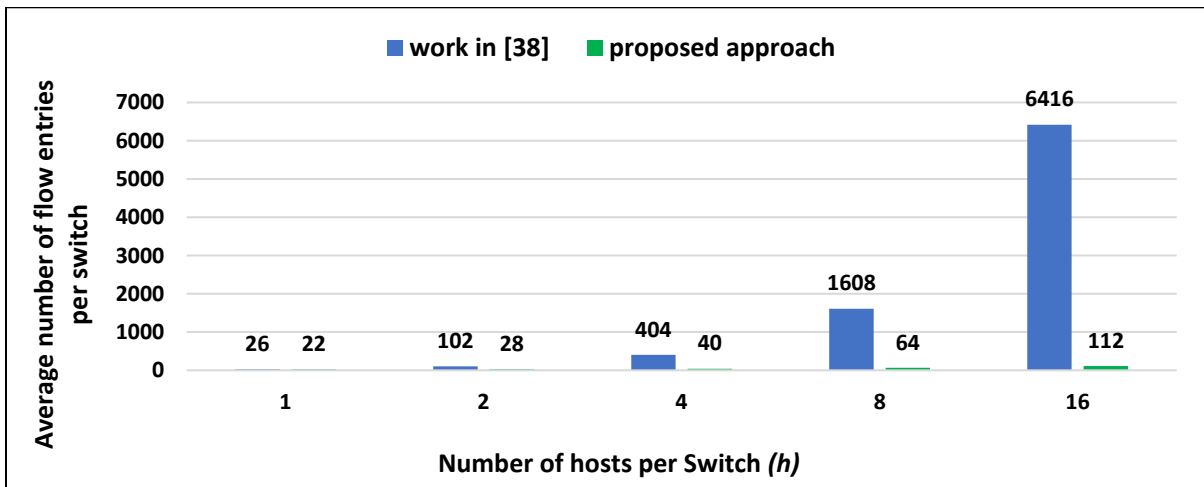


Figure 4.3.20: Comparison graph for flow entries per switch, $N = 6$

In [39], same grid topology is used to calculate the total number of flows, but this time with varying number of switches. Initial grid is composed of 4 rows and 5 columns, so in total 20 switches. Then every time they add one new row up to in total 65 switches

(13x5). A single host is attached with every switch. We calculate flow entries using eq. (4) again. For a grid network of 20 switches, there would be in total 51 links (31 links between switches and 20 for the hosts) and will increase by 14 for each new row.

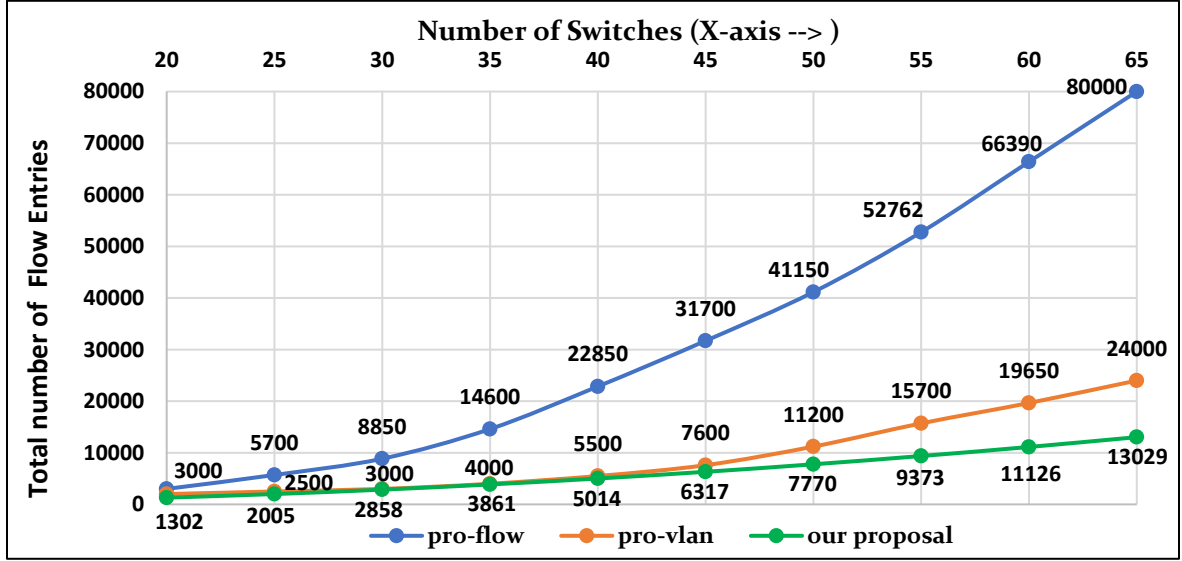


Figure 4.3.21: trendline for total flow entries in network between [39] and proposed method

Figure 4.3.21 illustrates the trendline comparison graph between two methods in [39], *pro-flow* and *pro-vlan*, and our proposed approach (X-axis is shown on top to accommodate data values). As discussed previously, while “*pro-flow*” method having slightly better recovery time, it needs way more flow entries and number of flows grow exponentially with the number of switches. Their second method “*pro-vlan*” started close to our approach, with around 1700 more flow entries (N=20) but ended up with a difference of almost 11,000 in a 65-switch grid network.

4.3.3 Segment Encoding Algorithm Comparison

Several algorithms have been proposed in the literature to encode and optimize the segment list for Segment Routing Path (SRP). Although they address and slacken the SLD problem, none of them totally solves the issue. All the proposed methodologies use a combination of different SID types (Node-SID and Adj-SID) and loose path routing. Which means if an SRP or any part of SRP follows the original shortest path

computed by the IGP computation, it is not required to express that path in detail segments. Scenarios may occur where this will not be the case, especially when QoS or traffic engineering are in place.

Giorgetti *et al.* [40] proposed two algorithms that outputs two label stack with the same size, one is for forward direction (src→dst) and second one the reverse path (dst→src). Their assumption is that the direct link between two nodes is the shortest path. This assumption is acceptable in normal condition, but problem arises when a higher cost is assigned to that link or it is too much congested. In [41], optimized segment list is calculated by creating an augmented network graph by inserting virtual links for every pair of nodes in addition to the actual links in the topology. With this approach, a 1000 node graph would result in a new graph of approximately half million links and a significant computing load to the controller. Authors of [42], [43] proposed algorithm to slice the SRP into multiple sub-paths and then if any sub-path contains more than three nodes, all of them is replaced by the tail nodes SID and if a sub-path has only two nodes in it, the adj-SID of the link between those two nodes are inserted. Although this approach can minimize the total number of labels used to express an SRP, it does not guarantee to meet the requirement of limited list depth. With a bigger topology, this problem may worsen. With our approach, it is possible to optimize the label stack for any given number of maximum depth size. Table 4.3.4 highlights the improvement and shortcomings of our proposed algorithm.

Advantages	Disadvantages
<ul style="list-style-type: none"> • Can always produce a fixed size label stack within a given depth, unlike previous works where output stack size depends on the path length. • No need of creating an auxiliary graph with virtual links, such as in [41]. 	<ul style="list-style-type: none"> • New SIDs are created that introduces overhead. Though the weighing mechanism can reduce the creation of gSIDs. • Had to maintain a gSID database for weighing mechanism to work.

Advantages	Disadvantages
<ul style="list-style-type: none"> • Can work with any given SRP which may not be preferred by IGP, meaning integrating QoS and TE is easier. 	<ul style="list-style-type: none"> • New flow rules had to be installed for gSIDs apart from the regular SIDs, which requires memory.

Table 4.3.4: Advantages and Disadvantages of proposed segment encoding algorithm

4.4 Concluding Remarks

In this chapter, we show the implementation and results of our proposed approach for a SR based link protection mechanism. The results demonstrate that the implementation is successful. We simulated our scheme in different topologies and were able to achieve a standard sub 50 ms recovery time with. Table 4.3.5 summarizes the topology details and average recovery time with our proposed approach.

<i>Topology Type</i>	<i>(Switches, Links)</i>	<i>Average Recovery Time (ms)</i>
Ring	(5, 5)	14.7
Ring	(13, 13)	26.3
Random (Figure 3.1.3)	(6, 7)	18.3
Random (Figure 4.3.15b)	(6, 7)	16.8
Grid	(64, 112)	20.3

Table 4.3.5: Simulation Topologies and Recovery Time Summary

Memory requirements are also evaluated, which shows that our approach requires much less flow entries with varying number of switches and hosts. Also, comparison with some previous works in literature in terms of both time and memory shows that the proposed approach is a viable method for local link failure recovery using the segment routing capabilities.

5 Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this thesis, we presented a Segment Routing (SR) based local link failure recovery mechanism for a resilient network. The objective of this thesis is to integrate Segment Routing in an SDN OpenFlow based network and come up with a failure recovery scheme that leverages SR capabilities and meet the standard recovery requirements. We proposed algorithms for segment identifier (SID) based forwarding and backup path computation, with modules to generate flow rules in different tables as necessary and use of OpenFlow Fast-Failover (FF) group for local switchover of traffic. This separation of flow tables results in a faster index searching, which accumulates in our total recovery time. Use of SR segment identifier ensures a minimal path even in case of a link failure and traffics had to detour the original shortest path. Then we also propose an algorithm to mitigate the SLD constraint so that the limitation in OpenFlow switch hardware implementation does not interfere with flow rules pushing multiple labels.

The results show that implementation is successful, and the proposed approach performs well under different network topologies and can meet the sub 50 ms requirement specified in RFC 5654 [4]. SDN controller intervention is not needed after initial computation as failure can be recovered locally and independently at the switches. Comparison is shown with other recovery methods such as restoration, priority-based link protection and recovery using VLAN-tag and results confirm that our proposed scheme performs better in terms of recovery time, average number of flows per switch and total flow entries required.

5.2 Future Work

Below are some of the recommendations to improve our current proposed approach and we would like to address them in future:

- Current implementation is done considering a single host is attached to each switch. This can be extended for multiple attached host by slight modification in our existing table design. A new table will be generated for those hosts and the table 0 flow rule to forward packet into port 1 will now forward to this table.
- We considered a single link failure at any given time. Although our current proposal can handle multiple link failure, but this is under restricted condition that path from each switch to the destination should be disjoint. Our algorithm does not calculate this disjoints and can be extended to fully accommodate multiple simultaneous link failure.
- While detouring the traffic, normal condition of topology is assumed, and link congestion or bandwidth is not considered. QoS and Traffic Engineering (TE) requirements can also be integrated while computing the backup path and detouring the traffic.
- Test the segment encoding algorithm into more complex topologies and get quantitative measurements to evaluate its performance and integrate it with our extended recovery scheme with QoS and Traffic Engineering (TE).

References

- [1] W. John, A. Kern, M. Kind, P. Skoldstrom, D. Staessens and H. Woesner, "Splitarchitecture: SDN for the carrier domain," *IEEE Communications Magazine*, vol. 52, pp. 146-152, 10 2014.
- [2] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow and G. Parulkar, "ONOS: Towards an Open, Distributed SDN OS," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, New York, NY, USA, 2014.
- [3] L. Sidki, Y. Ben-Shimol and A. Sadoski, "Fault tolerant mechanisms for SDN controllers," in *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2016.
- [4] 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 (IETF), 2009.
- [5] E. Haleplidis, K. Pentikousis, S. Denazis, J. H. Salim, D. Meyer and O. Koufopavlou, "Software-Defined Networking (SDN): Layers and Architecture Terminology," RFC 7426, Internet Research Task Force (IRTF), 2015.
- [6] ONF, "Software-Defined Networking: The New Norm for Networks," 2012.
- [7] C. Filsfils, S. Previdi, B. Decraene, S. Litkowski and R. Shakir, "Segment Routing Architecture," IETF draft-ietf-spring-segment-routing-13, 2015.
- [8] C. Filsfils, S. Previdi, A. Bashandy, B. Decraene, S. Litkowski and R. Shakir, "Segment Routing with MPLS data plane," IETF draft-ietf-spring-segment-routing-mpls-11, 2017.
- [9] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 69-74, 3 2008.
- [10] T. O. N. Foundation, "OpenFlow Switch Specification," 2015.
- [11] L. Wang, . R.-F. Chang, E. Lin and . J. . C.-S. Yik, "Apparatus for link failure detection on high availability Ethernet backplane". USA Patent 7260066, 21 8 2007.
- [12] D. Levi and D. Harrington, "Definitions of Managed Objects for Bridges with Rapid Spanning Tree Protocol," RFC Editor, 2005.
- [13] D. Katz and D. Ward, "Bidirectional Forwarding Detection (BFD)," RFC 5880, Internet Engineering Task Force (IETF), 2010.

- [14] J.-P. Vasseur, M. Pickavet and J.-P. Demeester, *Network recovery: protection and restoration of optical, SONET-SDH, IP, and MPLS*, Elsevier, 2004.
- [15] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma and S. Banerjee, "DevoFlow: Scaling Flow Management for High-performance Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 41, pp. 254-265, 8 2011.
- [16] V. Sekar, S. Ratnasamy, M. K. Reiter, N. Egi and G. Shi, "The Middlebox Manifesto: Enabling Innovation in Middlebox Deployment," in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, New York, NY, USA, 2011.
- [17] H. Huang, S. Guo, P. Li, W. Liang and A. Y. Zomaya, "Cost Minimization for Rule Caching in Software Defined Networking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, pp. 1007-1016, 4 2016.
- [18] E. Harrison and B. Miller, *Protection And Restoration In MPLS Networks*, 2001.
- [19] P. Pan, G. Swallow and A. Atlas, "Fast Reroute Extensions to RSVP-TE for LSP Tunnels," RFC Editor, 2005.
- [20] S. Sharma, D. Staessens, D. Colle, M. Pickavet and P. Demeester, "Enabling fast failure recovery in OpenFlow networks," in *2011 8th International Workshop on the Design of Reliable Communication Networks (DRCN)*, 2011.
- [21] D. Staessens, S. Sharma, D. Colle, M. Pickavet and P. Demeester, "Software defined networking: Meeting carrier grade requirements," in *2011 18th IEEE Workshop on Local Metropolitan Area Networks (LANMAN)*, 2011.
- [22] E. Bellagamba, J. Kempf and P. Skoldstrom, "Link Failure Detection and Traffic Redirection in an Openflow Network". USA Patent 8665699B2, 5 2011.
- [23] J. Kempf, E. Bellagamba, A. Kern, D. Jocha, A. Takacs and P. Sköldström, "Scalable fault management for OpenFlow," in *2012 IEEE International Conference on Communications (ICC)*, 2012.
- [24] S. Sharma, D. Staessens, D. Colle, M. Pickavet and P. Demeester, "OpenFlow: Meeting carrier-grade recovery requirements," *Computer Communications*, vol. 36, pp. 656-665, 2013.
- [25] A. Sgambelluri, A. Giorgetti, F. Cugini, F. Paolucci and P. Castoldi, "Effective flow protection in OpenFlow rings," in *2013 Optical Fiber Communication Conference and Exposition and the National Fiber Optic Engineers Conference (OFC/NFOEC)*, 2013.
- [26] A. Sgambelluri, A. Giorgetti, F. Cugini, F. Paolucci and P. Castoldi, "OpenFlow-based segment protection in Ethernet networks," *IEEE/OSA Journal of Optical Communications and Networking*, vol. 5, pp. 1066-1075, 9 2013.

- [27] A. Capone, C. Cascone, A. Q. T. Nguyen and B. Sansò, "Detour planning for fast and reliable failure recovery in SDN with OpenState," in *2015 11th International Conference on the Design of Reliable Communication Networks (DRCN)*, 2015.
- [28] C. Cascone, L. Pollini, D. Sanvito, A. Capone and B. Sansó, "SPIDER: Fault resilient SDN pipeline with recovery delay guarantees," in *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, 2016.
- [29] G. Bianchi, M. Bonola, A. Capone and C. Cascone, "OpenState: Programming Platform-independent Stateful Openflow Applications Inside the Switch," *SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 44-51, 4 2014.
- [30] O. Tilmans and S. Vissicchio, "IGP-as-a-backup for robust SDN networks," in *10th International Conference on Network and Service Management (CNSM) and Workshop*, 2014.
- [31] B. Stephens, A. L. Cox and S. Rixner, "Scalable Multi-Failure Fast Failover via Forwarding Table Compression," in *Proceedings of the Symposium on SDN Research*, New York, NY, USA, 2016.
- [32] B. Stephens, A. L. Cox and S. Rixner, "Plinko: Building Provably Resilient Forwarding Tables," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, New York, NY, USA, 2013.
- [33] W. Braun and M. Menth, "Loop-Free Alternates with Loop Detection for Fast Reroute in Software-Defined Carrier and Data Center Networks," *J. Netw. Syst. Manage.*, vol. 24, pp. 470-490, 7 2016.
- [34] H. Kim, M. Schlansker, J. R. Santos, J. Tourrilhes, Y. Turner and N. Feamster, "CORONET: Fault tolerance for Software Defined Networks," in *2012 20th IEEE International Conference on Network Protocols (ICNP)*, 2012.
- [35] B. Raeisi and A. Giorgetti, "Software-based fast failure recovery in load balanced SDN-based datacenter networks," in *2016 6th International Conference on Information Communication and Management (ICICM)*, 2016.
- [36] N. Kitsuwon, D. B. Payne and M. Ruffini, "A novel protection design for OpenFlow-based networks," in *2014 16th International Conference on Transparent Optical Networks (ICTON)*, 2014.
- [37] R. M. Ramos, M. Martinello and C. E. Rothenberg, "SlickFlow: Resilient source routing in Data Center Networks unlocked by OpenFlow," in *38th Annual IEEE Conference on Local Computer Networks*, 2013.

- [38] Y. Lin, H. Teng, C. Hsu, C. Liao and Y. Lai, "Fast failover and switchover for link failures and congestion in software defined networks," in *2016 IEEE International Conference on Communications (ICC)*, 2016.
- [39] J. Chen, J. Chen, J. Ling and W. Zhang, "Failure recovery using vlan-tag in SDN: High speed with low memory requirement," in *2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC)*, 2016.
- [40] A. Giorgetti, P. Castoldi, F. Cugini, J. Nijhof, F. Lazzeri and G. Bruno, "Path Encoding in Segment Routing," in *2015 IEEE Global Communications Conference (GLOBECOM)*, 2015.
- [41] F. Lazzeri, G. Bruno, J. Nijhof, A. Giorgetti and P. Castoldi, "Efficient label encoding in segment-routing enabled optical networks," in *2015 International Conference on Optical Network Design and Modeling (ONDM)*, 2015.
- [42] R. Guedrez, O. Dugeon, S. Lahoud and G. Texier, "Label encoding algorithm for MPLS Segment Routing," in *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)*, 2016.
- [43] O. Dugeon, R. Guedrez, S. Lahoud and G. Texier, "Demonstration of Segment Routing with SDN based label stack optimization," in *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*, 2017.