LOCAL FAST REROUTE WITH FINE-GRAINED MONITORING AND PRIORITY-AWARE CONGESTION CONTROL

by

M A MOYEEN

Submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
August 2019

Table of Contents

List of Tables . . . . . **v**

List of Figures . . . . . **vi**

Abstract . . . . . **viii**

List of Abbreviations and Symbols Used . . . . . **ix**

Acknowledgements . . . . . **xi**

Chapter 1 Introduction **1**

1.1 Motivation 1

1.2 Research Objectives 3

1.3 Contributions 4

1.4 Thesis Outline 5

Chapter 2 Background **6**

2.1 Introduction to Software-Defined Networking (SDN) 6
2.1.1 Traditional Network 6
2.1.2 Software Defined Networking (SDN) 8
2.1.3 OpenFlow 10
2.1.4 OpenFlow Switch 11
2.1.5 OpenFlow Messages 15
2.1.6 Open vSwitch 16

2.2 Handling Failure in SDN 17
2.2.1 Failure Detection 17
2.2.2 Failure Recovery 18
2.2.3 Crankback (CB) Strategy 20
2.2.4 Crankback With Controller (CBC) 20

2.3 Compression Aware Fine-Grained Monitoring 21

2.4 Congestion Control for Priority Flows 22

Chapter 3 Related Work **24**

3.1 Failure Recovery in SDN 24
3.1.1 Restoration Recovery 24

3.1.2 Protection Recovery . . . 25
3.1.3 Hybrid Recovery . . . 27
3.1.4 Header Modification Based Recovery . . . 27

3.2 Network Monitoring . . . 29
3.2.1 Active Monitoring . . . 29
3.2.2 Passive Monitoring . . . 30
3.2.3 Hybrid Monitoring . . . 30

3.3 Congestion Control . . . 32
3.3.1 Centralized Schemes . . . 32
3.3.2 Distributed Schemes . . . 34

Chapter 4 SD-FAST: A Packet Rerouting Technique . . . 37

4.1 Architecture and Design . . . 37
4.1.1 Application and control plane modules . . . 37
4.1.2 Data plane modules . . . 39
4.1.3 Packet Rerouting Operation . . . 41

4.2 Evaluation Setup . . . 42

4.3 Performance evaluation . . . 45
4.3.1 Impact of Link Failure . . . 45
4.3.2 Impact of Topology . . . 47
4.3.3 Impact of Real Traffic . . . 48
4.3.4 Impact of Crankback backtracking . . . 49
4.3.5 Recovery Time of SD-FAST . . . 50

Chapter 5 Compression Aware Monitoring . . . 51

5.1 cMon Design and Implementation . . . 51
5.1.1 cMon Algorithm . . . 54

5.2 Evaluation Setup . . . 56

5.3 Performance evaluation . . . 57
5.3.1 Memory Usage . . . 57
5.3.2 Impact of Link Failure on cMon Accuracy . . . 58
5.3.3 Impact on Network Performance . . . 58

Chapter 6 Distributed Priority Aware Load Balancing . . . 60

6.1 DPAL Design and Architecture . . . 60
6.1.1 Management and Control Plane Modules . . . 60
6.1.2 Data Plane Modules . . . 61
6.1.3 Load Distribution Mechanism of DPAL . . . 62

6.2 Evaluation Setup 64

6.3 Performance Evaluation 65
6.3.1 Average Delay for Higher Priority Flow 65
6.3.2 Average Throughput for Higher Priority Flow 66
6.3.3 Average Hop Count for Higher Priority Flow 67
6.3.4 Impact of DPAL on The Network Performance 68

Chapter 7 Conclusions and Future Work 70

7.1 Conclusions 70

7.2 Future Work 71

Bibliography 72

List of Tables

Table 3.1 Summary of Different Failure Recovery Schemes 28
Table 3.2 Summary of Different Monitoring Techniques 31
Table 3.3 Summary of Different Congestion Control Techniques 36

List of Figures

Figure 2.1 Traditional Network Architecture. 7

Figure 2.2 SDN Architecture. 8

Figure 2.3 OpenFlow Switch Architecture. 10

Figure 2.4 Components of Flow Table Entry 11

Figure 2.5 Fast Failover Group Workflow. 13

Figure 2.6 Packet Processing Flow Chart. 14

Figure 2.7 Open vSwitch Architecture. 16

Figure 2.8 The operation of FFG, CB, and CBC. 18

Figure 2.9 Wild-card and Exact-Match Rule Examples. 21

Figure 2.10 Congestion Control Problem For Priority Flows. 22

Figure 4.1 The Architecture of SD-FAST [1]. 38

Figure 4.2 Different types of packet status in SD-FAST [1]. 39

Figure 4.3 24 node USNET Topology [1]. 42

Figure 4.4 28 node Darkstrand Topology [1]. 43

Figure 4.5 The average end-to-end delay in the presence of link failure in USNET Topology. 45

Figure 4.6 The average end-to-end delay in the presence of link failure in Darkstrand Topology [1]. 46

Figure 4.7 The average throughput in the presence of link failure in USNET Topology. 46

Figure 4.8 The average throughput in the presence of link failure in Darkstrand Topology [1]. 47

Figure 4.9 The average end-to-end delay in USNET and Darkstrand topology [1]. 48

Figure 4.10 The average throughput in USNET and Darkstrand topology [1]. 48

Figure 4.11 The average end-to-end delay while using real traffic [1]. . . . 49

Figure 4.12 The average throughput while using real traffic [1]. 49
Figure 4.13 The convergence time of backtracking in the Darkstrand topology [1]. 50
Figure 4.14 Average Recovery Time of SD-FAST. 50

Figure 5.1 The Architecture of cMon. 52
Figure 5.2 Impact On Memory Uses. 57
Figure 5.3 Impact of Link Failure on Per Flow Statistics. 58
Figure 5.4 Impact of cMon on Network Throughput. 59

Figure 6.1 The Architecture of DPAL. 61
Figure 6.2 Flowchart Showing DPAL Operation. 62
Figure 6.3 Average Delay For Higher Priority Flow in USNET Topology. 65
Figure 6.4 Average Delay For Higher Priority Flow in Darkstrand Topology. 66
Figure 6.5 Average Throughput For Higher Priority Flow in USNET Topology. 66
Figure 6.6 Average Throughput For Higher Priority Flow in Darkstrand Topology. 67
Figure 6.7 Average Hop Count For Higher Priority Flow. 68
Figure 6.8 Average Throughput of the Network. 68

Abstract

Communication links or nodes failure is common in Software-Defined Networking (SDN), which affect on-going communication and induces performance bottleneck. In SDN, protection-based recovery mechanisms pre-installs a backup route to offer local failure recovery. In those mechanisms, rule compression improves memory consumption. However, rule compression reduces network visibility and impedes fine-grained monitoring. When a topology has edge-disjoint routes, protection-based recovery uses OpenFlow Fast Failover Group (FFG); otherwise, it uses MPLS-crankback (CB) to reroute the affected traffic. But CB strategy suffers from continuous backtracking and induces extra latency overhead. Crankback with Controller (CBC) strategy can terminate such backtracking. CBC also suffers from the controller to switch communication overhead and impact the overall recovery time. Moreover, congestion in a network induces packet loss and impedes reliable communication. Priority flows can experience longer routes because of load distribution to alleviate congestion.

Thus, a failure recovery scheme must not impact fine-grained monitoring and priority flows while meeting the delay requirement (in the order of milliseconds) of applications. However, existing recovery schemes failed to meet these correlated requirements. This thesis fills that gap and proposes three data plane based techniques called SD-FAST, cMon, and DPAL. In particular, SD-FAST supports local failure recovery to terminate crankback backtracking, cMon offers fine-grained monitoring in the presence of compressed forwarding rules, and DPAL allows priority flows to quickly reach destination irrespective of network load. Extensive performance evaluation of the proposed solutions over real network topologies and traffic in Mininet emulator confirms that they significantly outperform corresponding standard solutions.

List of Abbreviations and Symbols Used

BFD Bidirectional Forwarding Detection

CB Crankback
CBC Crankback with Controller
CFM Connectivity Fault Management
cMon Compression Aware Monitoring

DPAL Distributed Priority-Aware Load Balancing

FFG Fast Failover Group

ICMP Internet Control Message Protocol
IoT Internet of Things
IP Internet Protocol

MPLS Multiprotocol Label Switching

NBI Northbound Interface
NOS Network Operating System

OVS Open vSwitch

SBI Southbound Interface
SDN Software Defined Network
SDN Ternary Content Addressable Memory
SLA Service Level Agreement

TCP Transmission Control Protocol

TLS Transport Layer Security

TTL Time To Live

UDP User Datagram Protocol

Acknowledgements

I am very grateful to my supervisor, Dr. Israat Haque for her continuous support throughout this thesis work. Thus, I want to thank her with full respect. Next, I want to thank Meysam, Dipon and Fangye for their constructive criticisms about this work. Finally, I want to thank my parents for their continuous support throughout my graduate studies.

Chapter 1

Introduction

In Software Defined Networking (SDN) [2, 3], the network control logic is separated from the operational data plane elements. Such separation makes the network programmable. Due to the programmable nature of the network, SDN boosts up the network innovation.

Reliable communication is highly desirable to get the full benefit of SDN applications. Link or node failure affects the reliable transfer of data. Also, congestion often has an impact on reliability. But congestion control mechanisms often lead the higher priority flow to a longer route. Also, fast reliable communication strategies exploit the ternary content addressable memory ($TCAM$). Thus, to reduce the TCAM usage, existing schemes often compress the forwarding rules. This kind of compression can reduce network visibility. Thus, in this thesis, we propose three algorithms that can offer 1) fast local failure recovery, 2) enable *fine-grained* network monitoring, and 3) construct appropriate paths for priority flow to distribute the load. At first, this chapter presents the motivation and objectives of the research. Then, it highlights the major contributions and outlines the content construction.

1.1 Motivation

In its programmable architecture, SDN uses *OpenFlow* [4] to enable communication between the network controller and data plane elements. Using this protocol, network managers easily impose policies or Service Level Agreements ($SLAs$) in the form of flow rules into the data plane switches. The flexible network control attracts SDN-based network design in data centers, cellular networks, cloud networks, Internet of Things (IoT), smart cities, and community networks [5–8]. However, service disruption can wipe out all the benefits of SDN and cause a huge amount of loss [9]. The interruption often happens due to faulty links. A link becomes faulty when it is damaged, or adjacent node fails.

To recover from link failure, one strategy in the SDN paradigm is the *restoration* approach [10]. In the restoration approaches, the controller installs alternative routes when a switch reports a failure. However, restoration approaches take a long time to recover from a link failure due to the high controller to switch communication overhead. Also, this kind of communication can not ensure a recovery time within the standard time of 50ms [11].

To recover from the link failure within 50ms, *protection* [11–13] schemes pre-installs both primary and backup rules into the data plane switches. They often use Fast Failover Group (*FFG*) table of OpenFlow protocol to provide local failure recovery. In this strategy, every traffic matches the data path flow table and moves to the FFG table to get the output path. In FFG, primary and backup paths are stored in the form of buckets. FFG selects the first bucket with live port status from the bucket list for packet forwarding. The failure detection protocols; such as-Bidirectional Forwarding Detection (*BFD*) [14] or Connectivity Fault Management (*CFM*) [15], gives the liveliness status of a port. In this strategy, every packet passes through the FFG to get the active path, which introduces extra processing overhead to the regular traffic as well as to the failure affected traffic. Moreover, FFG based strategies require *edge-disjoint* route at every node of a topology. A node is said to have edge-disjoint routes if paths from that node do not share common edges. This kind of property is often absent in many topologies.

When such edge-disjoint property is missing in topology, we can use crankback (*CB*) strategy [16]. In that case, traffic can move back towards the source and get the recovery route from a node having the edge-disjoint route. But in most of the CB strategy, both regular and affected traffic propagates up to the failed link and backtracks towards the node that has the recovery route. This continuous backtracking introduces significant delay and can reduce the overall network throughput.

To stop the continuous crankback backtracking, crankback with controller (*CBC*) strategy uses controller's assistance [17]. In this strategy, the controller installs rules in the node that has a recovery route to prevent the traffic movement towards the failed link. However, the controller often takes a long time to install such a rule due to its distant location or its load level. At that time, a large amount of traffic can experience the backtracking issue.

Moreover, in FFG, CB, and CBC, both the regular and failed traffic need to check the link failure status, which introduces a significant amount of processing overhead. Also, these strategies require pre-installed primary and backup routes. Thus they consume a substantial amount of limited TCAM space. There are techniques [18] that incorporate forwarding rule compression to prevent their TCAM overutilization. However, such compression reduces network visibility. Thus, fine-grained network monitoring becomes difficult when forwarding rules are compressed. The absence of fine-grained monitoring can introduce incorrect billing, inaccurate Malware detection, and enable DDoS attacks [18].

Finally, congestion in a network can induce data loss and disrupt ongoing network operation. Congestion control schemes [13, 19] often distribute loads to the less congested path after detecting congestion. However, such distribution usually leaves the priority flow with a longer route. To offer low route stretch [20] (which is the difference between the best possible path and the actual path of traffic flow) for the higher priority flow, we can distribute the low priority flow in the alternate route. However, such distribution often requires additional processing and induces extra overhead in the network. Thus, proper distribution of low priority rules is necessary.

1.2 Research Objectives

Resilient network design is highly desirable. Though FFG based schemes [13] can provide local failure recovery, they cannot survive when topology does not have nodes with edge-disjoint routes. CB approaches, on the other hand, suffer from backtracking delay and degrade the overall performance of the network. It is also noteworthy that, FFG, CB, and CBC schemes suffer from extra table processing overhead. Moreover, when these approaches try to save TCAM with forwarding rule compression, they reduce network visibility. Improper distribution of load can induce long route-stretch to the higher priority flows.

In this thesis, the major objectives are to design local mechanisms in SDN environment, which does not affect regular traffic for extra comparison with the link status, can offer fast failure recovery, terminate backtracking locally, provide the best route for the higher priority flow while balance load, and enable fine-grained monitoring in a compression aware routing. The proposed scheme targets to provide the following

functions:

- **Fast Failure Detection:** Fast failure detection is necessary to recover from a link failure. This function monitors the link status at a periodic interval to quickly detect any link failure.
- **Local Failure Recovery:** This function provides the mechanism to recover from a link failure locally in a distributed fashion. In that case, it uses FFG when the topology has edge-disjoint routes; otherwise, it uses CB to recover from a failure. The main objective of this recovery scheme is to provide recovery passively without comparing each traffic with a link failure status.
- **Local Backtracking Termination:** This function ensures local termination of the crankback backtracking.
- **Fine-Grained Monitoring:** This allows fine-grained monitoring when forwarding rules are compressed.
- **Proper Path For Priority Flow:** This function allows a higher priority flow to achieve low route-stretch.

1.3 Contributions

In this thesis, we propose three distributed algorithms that work in the data plane to fulfill the above-mentioned research objectives. We implement the proposed algorithms at each switch in the data plane, where the BFD protocol provides the link status. In the case of a link failure, the proposed scheme replaces the affected rules with the backup rules. The proposed scheme stores backup rules in another table and uses that table only when a failure happens. In the case of regular traffic, packets follow the conventional flow tables and do not experience any comparison with the link status. With an appropriate tag, we differentiate between crankback and regular traffic. A switch prompts to send a CB traffic in a recovery route once it finds that route.

To offer fine-grained monitoring, we capture each traffic and process it in a parallel process. The per-packet monitoring prepares the necessary monitoring information

and pushes it to the monitoring controller. Moreover, the monitoring module survives in the presence of link failures and can preserve the monitoring data.

Finally, our load balancing algorithm only keeps track of the congestion status of a port. Based on that status, it distributes low priority flows in an alternative path to reduce the congestion level in the primary path. The algorithm replaces the minimum number of rules to reduce the processing overhead.

The overall contribution of this thesis can be summarized as follows:

- To provide local recovery, we propose a data plane technique SD-FAST that uses a distributed packet rerouting algorithm and runs at each data plane switches. The algorithm works passively and does not affect fail-free traffic.
- To terminate crankback backtracking locally, we use packet-status based routing decisions.
- In this thesis, we propose another algorithm called, cMon, which offers fine-grained monitoring when forwarding rules are compressed.
- We also, propose a distributed algorithm called, DPAL, which can offer a better route for higher priority flow when congestion happens in the primary route.
- We evaluate SD-FAST, cMon and DPAL in Mininet [21] using a Ryu controller [22] and a set of software switches (OVS [23]) over both real traffic and topologies. The evaluation result shows that SD-FAST can reduce a 73% backtracking delay and a 40% overall end-to-end delay. Also, cMon can provide fine-grained monitoring and improve the TCAM usage three times compared to FlowStat [18]. DPAL algorithm saves 49% average end-to-end delay for higher priority flows compared to its counterparts.

1.4 Thesis Outline

The remainder of this thesis is organized as follows: Chapter 2 introduces the necessary background to understand this thesis. We discuss relevant research works in Chapter 3. Chapter 4 presents SD-FAST design and its evaluation. In Chapter 5, we present the design and evaluation of cMon. We present DPAL in chapter 6. The thesis concludes in Chapter 7 with future research directions.

Chapter 2

Background

In this chapter, we present the necessary background to understand the thesis. At first, we introduce the architecture of the Software-Defined Network (*SDN*) in section 2.1 and highlight its advantages over the traditional network. In that section, we also present the OpenFlow protocol, OpenFlow Switch, OpenFlow messages, and an overview of the famous OpenvSwitch. We discuss the failure handling process of SDN in section 2.2. There we provide a brief discussion on failure detection protocols: BFD and CFM. We also discuss the failure recovery schemes: proactive, reactive, and hybrid techniques. At the end of that section, we discuss CB and CBC schemes. In section 2.3, we discuss the rule compression and fine-grained monitoring trade-off with proper examples. Finally, we conclude this chapter with the discussion on congestion control for priority flows.

2.1 Introduction to Software-Defined Networking (SDN)

2.1.1 Traditional Network

The traditional computer network comprises a different set of network devices such as switches, routers, firewalls, etc. These devices are autonomous system and mostly operates in a distributed manner. Most of the traditional routers or switches build their forwarding information base (*FIBs*) through the exchange of network information among its neighbors. With the *FIBs*, they can either forward or drop a packet.

Figure 2.1 presents the architecture of a traditional network. In this architecture, we can find that each switch has both hardware and software components. Among the software components, a switch may contain several network applications and network operating systems. Among the hardware components, a switch may contain the Central Processing Unit (*CPU*), Memory, and Application Specific Integrated Circuit (*ASIC*). The network applications usually communicate with the switch hardware

components through the network operating system.

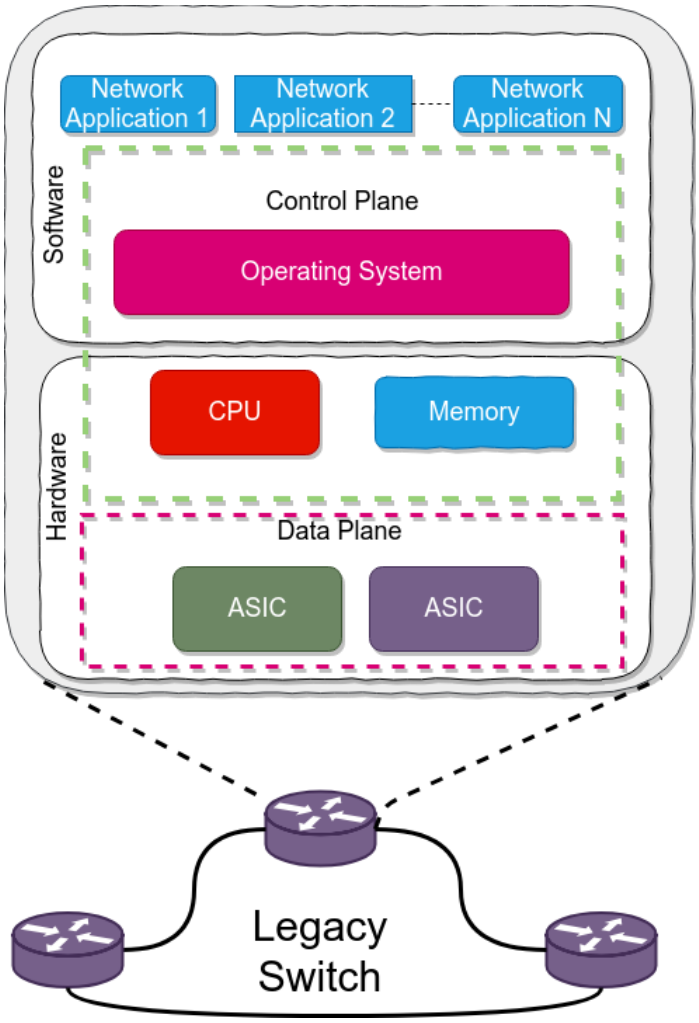


Figure 2.1: Traditional Network Architecture.

For most of the traditional network elements, both the hardware and software components are proprietary and act as the trade secret for manufacturers such as Cisco, Huawei, Dell, etc. The majority of the manufacturer releases its complex algorithm and network operating system as a closed source program. This kind of vendor dependent system has an interoperability issue and increases complexity in network management and maintenance. Thus, in a traditional network, innovation becomes slower because of time-consuming and error-prone network configuration and management.

2.1.2 Software Defined Networking (SDN)

To foster innovation, Software Defined Network (*SDN*), a new paradigm comes into the limelight in the early 2000s. It defines the way to design network systems with a combination of software and commodity network hardware. Unlike traditional network systems, where network control functions are embedded into the data plane devices, SDN separates network control functions from the data plane devices.

That's why the network administrator or engineer can reshape the traffic from the control plane without touching each switch. Figure 2.2 presents the SDN architecture that comprises of application, control, and infrastructure layer.

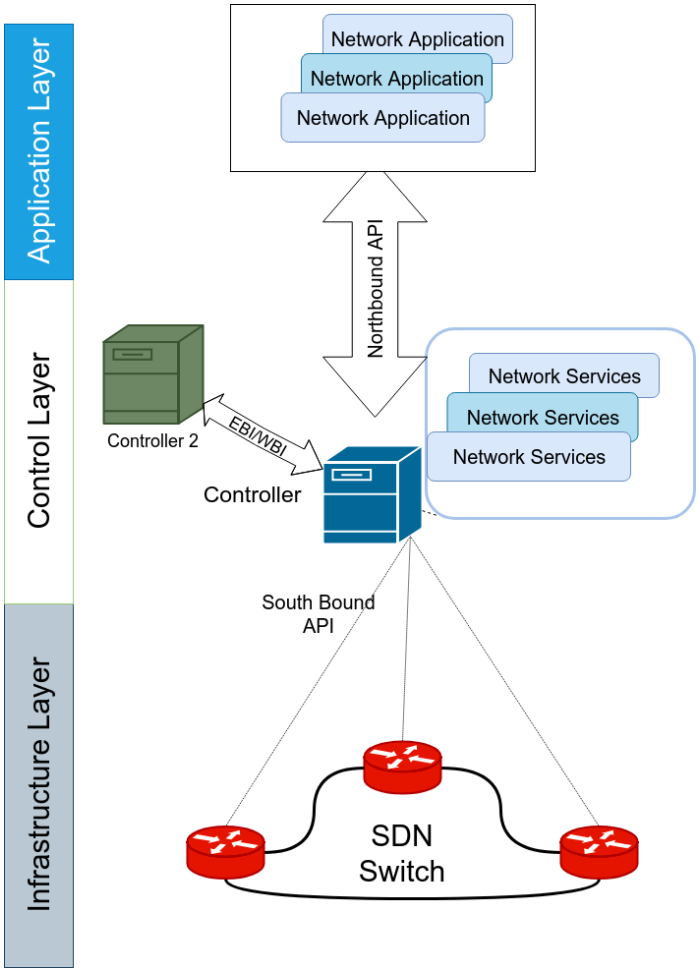


Figure 2.2: SDN Architecture.

In this architecture, the application layer communicates with the control layer

through Northbound API (*NBI*) and the control layer communicates with the Infrastructure layer by using Southbound API (*SBI*).

Application Layer: This is the top layer in an SDN architecture. SDN-based network applications reside in this layer. Some of the examples of SDN applications include network management, analytics, DDoS detection, etc. These applications communicate with the SDN controller through a Northbound API (*NBI*) interface. Network administrators or operators use this layer to manage and control the entire network.

Northbound Interface (*NBI*): *NBI* works as a communication interface between the control and application layers. Using this interface, network managers pass management decisions such as service level agreements to the SDN controller. The applications also get the network information from the SDN controller using this interface. REST API, ONIX API, JAVA API, etc. [24] are the most popular *NBI*.

Control Layer: The control layer or control plane of an SDN network contains the logically centralized network controller, which is often referred to as the Network Operating System (*NOS*). This layer provides an abstract view of the global network and hides the implementation details of it. An SDN network may contain one or more network controllers. Network controllers use East-Westbound APIs to communicate with each other. Some of the examples of the SDN controller includes- Ryu, ONOS, OpenDaylight, NOX, POX, and Floodlight [25]. The network controller communicates with network applications through *NBI*. It also communicates with data plane devices using the *SBI*. Using *SBI*, the controller senses the data plane events and gathers information about the data plane including network topology. For example, a controller installs route configuration rules in the data plane routers based on the collected network state information and the network topology, where the route configuration policy (e.g., shortest route) comes from the application layer.

Southbound Interface (*SBI*): This allows interactions between the control layer and the infrastructure layer. The interactions can be device information query, network event notification, installation of configuration, etc. OpenFlow, ForCES, PCEP are examples of some SBIs [24].

Infrastructure Layer: This is the bottom layer of an SDN architecture. In this layer, data plane network devices reside. These devices perform data processing

and forwarding functions. They also execute the policies provided by the network controller. When a new network event happens in the data plane, these devices communicate with the network controller to get appropriate network configuration rules.

2.1.3 OpenFlow

OpenFlow becomes a de facto protocol since its innovation during an academic experiment at Stanford University in 2008 [4]. It runs on top of transmission control protocol (*TCP*) and prescribes the use of transport layer security (*TLS*). It is the most popular protocol that fosters SDN innovation. With this protocol, different vendor dependent systems that support different scripting can communicate with the SDN controller using an open interface. Thus, using a single interface network users can manage different kinds of devices. Moreover, this protocol allows the SDN controller to gather overall network topology and events. With this information, the controller gets the global view of the network. Thus, the controller can plan the route properly and take complex routing and management decisions.

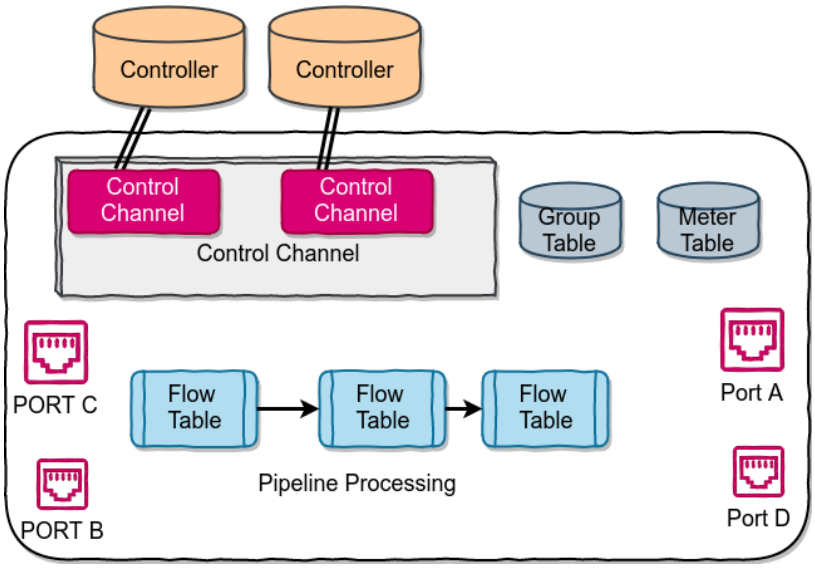


Figure 2.3: OpenFlow Switch Architecture.

2.1.4 OpenFlow Switch

An OpenFlow switch supports the OpenFlow protocol and can communicate with an OpenFlow supported controller on a dedicated port. The most important components of an OpenFlow supported switch are the flow tables and group tables. When a packet enters into an OpenFlow switch, it is matched against one or more flow tables in a pipeline and optionally with the group table. Figure 2.3 shows the architecture of an OpenFlow switch.

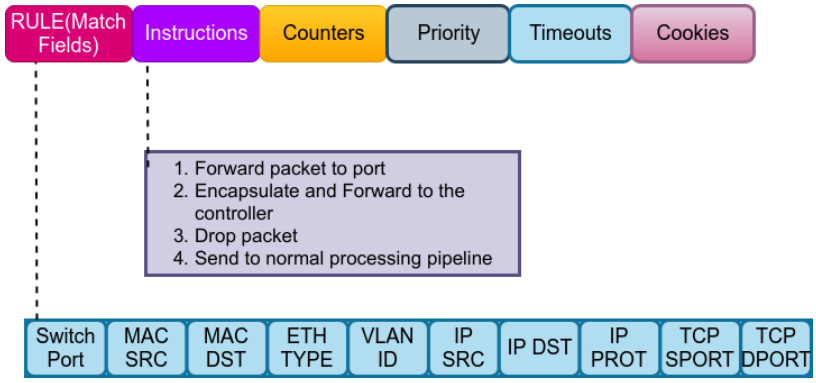


Figure 2.4: Components of Flow Table Entry

Flow Tables

Flow tables contain the packet forwarding logic in the form of flow rules and can take necessary action on a packet. A flow entry in an OpenFlow switch contains match field, priority, counters, instructions, cookie and timeouts. Figure 2.4 presents the components of a flow entry.

- **Match Field:** A match field contains the packet matching rule. It may contain ingress port, optional metadata, and packet header information; such as- source IP address, destination IP address, source MAC address, destination MAC address, Protocol information, etc. OpenFlow switch declares a packet match when the packet header information matches with all the match elements of a flow rule.

- **Priority:** A packet often matches with more than one flow rule. Priority defines the precedence of a flow rule. A switch always uses the highest priority matching

flow rule.

- **Counters:** Counters: They keep track of the usage statistics of a flow rule. It is updated when a packet match happens. Looking into these counters, we can easily tell about the number of the processed packet with a flow rule.

- **Instructions:** This component defines the action set for a packet. Among the possible actions set, a flow rule can forward a packet to a port, or it can send the packet to another flow table for further processing, or it may send the packet to a group table to execute group actions.

- **Timeouts:** There are two types of timeout in a flow rule. Among them, idle timeout expires the rule when it is unused for the specified time. Whereas, hard timeout clears a flow rule after the specified timeout interval irrespective of its usage.

- **Cookie:** Controller chooses these opaque data values to filter flow statistics or to execute flow modification and flow deletion.

Group Tables

Besides flow tables, OpenFlow group table allows grouping of different flows and executes common action on them. A group contains a unique ID and set of action buckets. An action bucket determines the possible set of actions that applies to a packet. A group table also contains counters for packet match and group types to separate different types of groups. There are four possible group types in an OpenFlow switch.

- **ALL:** This group executes all-action buckets. Thus, this is helpful for multicast forwarding.

- **SELECT:** This group executes only one action bucket. An algorithm selects the action buckets in a round-robin fashion. Thus, this is often used for load balancing purposes.

- **INDIRECT:** This group type allows multiple flow rules to point to a single bucket. That adds an indirection layer among the flow rules and allows the controller to modify a single rule for faster flow rule re-convergence.

- **FAST FAILOVER GROUP *FFG*:** This group type allows local failure recovery from a link failure. Thus, to recover from a failure, a switch does not need to contact the controller. It always takes the first live bucket for processing a packet. Failure detection protocols; such as- BFD or CFM determines the liveliness of a port. And the port status determines the liveliness of a bucket. When a switch does not find any live bucket in an FFG, it drops the packet. Figure 2.5 shows the workflow of an FFG table. At first, a packet enters through *Port C* and matches with the flow table. When the packet finds an action to go to the FFG table it looks into *bucket1* of FFG.

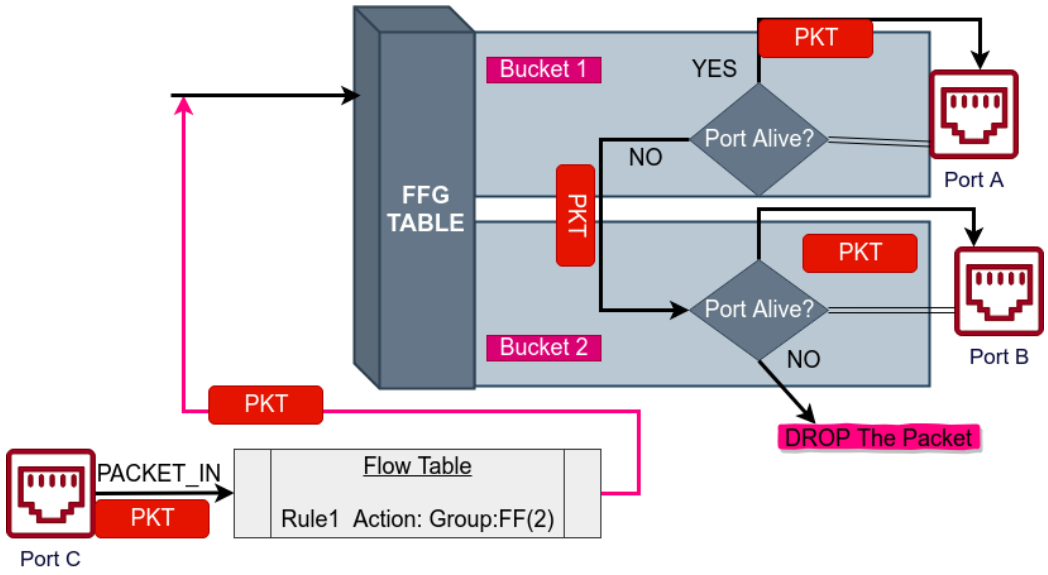


Figure 2.5: Fast Failover Group Workflow.

If *bucket1* senses *Port A* as live, packet forwards through *Port A*. If *Port A* is not live, then packet moves to the next bucket and that bucket again checks for the port status of *Port B*. If *Port B* is live, then the packet moves through the *Port B*. As there is no more bucket left in the FFG, as shown in Figure 2.5; thus, the failure of *Port B* results in the drop of the packet.

Packet Processing

When a packet enters into an OpenFlow switch, at first it receives a label with ingress port and holds an empty action set. After the labeling, the packet enters into the pipeline of flow tables and starts at table 0. An OpenFlow switch extracts the match fields from the packet and matches with the match fields of flow rules that are currently present in table 0. The rule matching technique follows matching with the highest priority rule first. If it finds a match, the corresponding rule will have proper instructions to process the packet. The instruction can send the packet to another table for further processing, or it may send it to the network controller. Also, it can send the packet to process by a group table, or the instructions can send the packet to an output port.

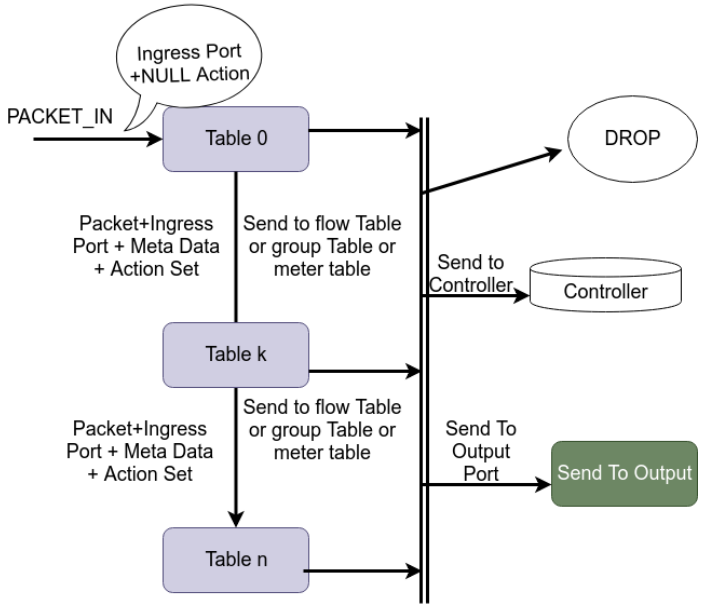


Figure 2.6: Packet Processing Flow Chart.

Finally, the instructions can instruct the switch to drop the packet. When an instruction sends a packet to another table, again the switch uses the match field to find corresponding flow rules. If instruction sends the packet to a group table, the group identifier allows the packet to match with the corresponding group table. When an instruction tells the switch to send the packet to the controller, switch encapsulates the packet and sends it to the controller on its reserved port. An output action allows

a switch to send the packet to the output port defined in the instruction. A switch drops the packet when the instruction asks it to do so, or there is no instruction to send it to the controller. Figure 2.6 presents the packet processing workflow of an OpenFlow switch.

2.1.5 OpenFlow Messages

To maintain a working communication between the SDN controller and OpenFlow enabled switches, this protocol supports three major communication message types.

Controller-to-Switch messages: Controller sends these messages to the switch and often demands a reply from the switch. FLOW_MOD, GROUP_MOD, and PACKET_OUT are the most common messages of this kind.

- **FLOW_MOD:** This message modifies the entries in a flow table. The entries added by this message determines whether to forward a packet or drop it. It also determines where to send the packet.
- **GROUP_MOD:** This message alters the group table entry.
- **PACKET_OUT:** With these messages, the controller instructs switches to send arbitrary data, e.g., sending Link Layer Discovery Protocol ($LLDP$) packets to one of its available ports.

Asynchronous messages: These messages are sent from the switch to the controller when a switch needs to pass some information to the controller. Among the most common asynchronous messages are PACKET_IN, FLOW_REM, and PORT_STATUS messages.

- **PACKET_IN:** This is the most important message that a switch sends to the controller. A switch sends this message when table-miss happens. A table-miss happens when a switch finds no matching flow rules for a packet other than sending it to the controller. It may generate PACKET_IN when the TTL value of a packet expires.
- **FLOW_REM:** When a flow removal happens in the flow table, switch notifies the controller about it.

- **PORT_STATUS:** A switch also notifies the controller about the addition, modification, or removal of a port using PORT_STATUS message. The controller can use this information to detect the change in topology and can reconfigure routes accordingly.

Symmetric messages: Either controller or switch sends these messages. Among the most common messages of this type are ECHO and HELLO.

- **HELLO:** Upon connection setup both switch and controller exchanges this message.
- **ECHO:** This message allows both switch and controller to detect their liveliness. A live controller or switch replies to the switch or controller, respectively with an echo-reply after receiving an ECHO message.

2.1.6 Open vSwitch

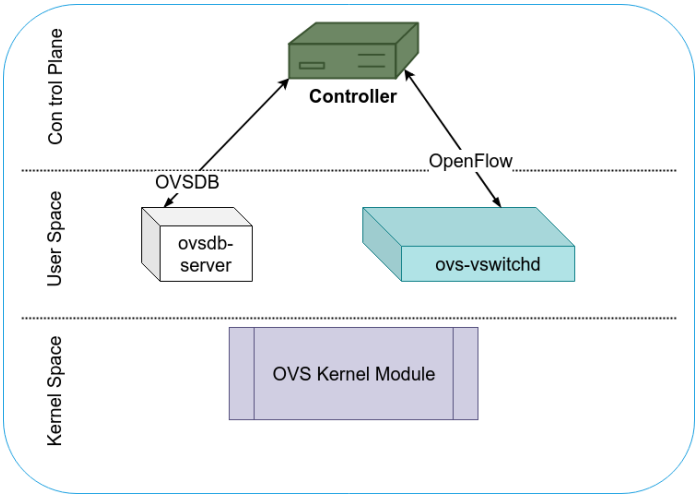


Figure 2.7: Open vSwitch Architecture.

OVS is the most widely adopted OpenFlow enabled, open-source, multi-layer software switch. We use this switch in all of our evaluations. In Figure 2.7, we present the architecture of *OVS*, where ovsdb-server acts as the database and holds the switch level configuration. The ovs-vswitchd is the core element of the *OVS* and acts as the communication hub between the kernel and userspace, or it can enable communication

between the switch and controller. *OVS* kernel module holds the flow rules for faster processing of incoming packets. If a packet does not match with those rules, it goes to the userspace. The ovs-vswitchd sends the packet to the controller when it does not get any matching rules in the userspace tables. In addition to OpenFlow, OVS uses the OVSDB protocol to receive switch level configuration from the SDN controller.

2.2 Handling Failure in SDN

To handle failure in the SDN environment, we need fast failure detection and fast recovery. In the following subsections, we present some background information on failure detection and failure recovery in SDN.

2.2.1 Failure Detection

A switch first needs to detect a link failure to recover from it. This is the first step while dealing with link failures. SDN switch can use LLDP packets or send continuous probe packet to determine the state of a link [26]. Among the failure detection protocols, BFD and CFM are the most popular failure detection protocols.

BFD Protocol

BFD [14] can detect a failure between two forwarding elements. To detect a port failure, network admin or an automated program enables BFD in all OVS ports. BFD enabled ports connected via a link, at first establishes a BFD session and synchronizes their timer. After that, they exchange IGP hello packets, where each control packet is 24 bytes long. The exchange rate of control packets is defined by the network administrator. A switch declares a link failure when it misses a certain number of hello packets over a monitoring link.

CFM Protocol

Ethernet CFM [15] is an industry-standard protocol that can offer proactive connectivity monitoring, fault detection, fault recovery, and fault verification. CFM emits multicast heartbeat-like continuity check messages (*CCMs*) at a periodic interval.

These messages are unidirectional. With a reply to this message, other nodes confirm connectivity to the node from where they received CCM.

In this thesis, our initial investigation suggests the better performance of BFD over CFM. Thus, we consider BFD for link failure detection throughout this thesis.

2.2.2 Failure Recovery

After the detection of a link failure, a switch demands fast recovery. There are three different types of strategies to recover from a link failure; namely reactive, proactive and hybrid.

Restoration Recovery

In a restoration or reactive recovery strategy, a switch contacts the controller to get an alternative route after detecting link failure. The controller after receiving a request from the switch computes alternative routes and sends the alternative routes in the form of flow rules to the switch. After switch installs the flow rules provided by the controller, failure affected traffic can move through the alternative route. Often controller remains distant from the switch, or they become busy with other tasks. Thus, the configuration of alternative routes after failure can take a long time [11].

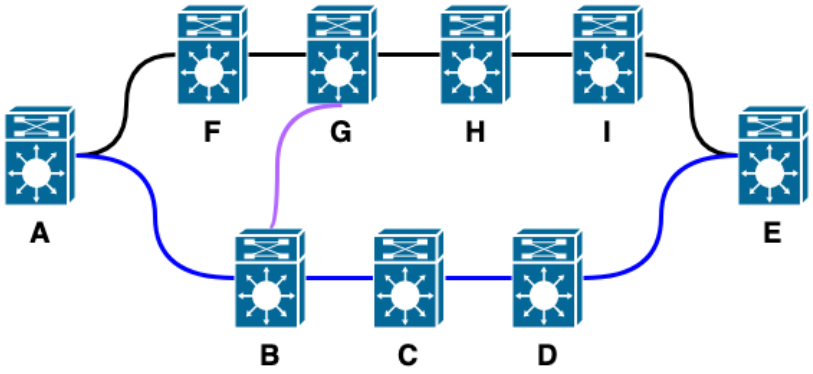


Figure 2.8: The operation of FFG, CB, and CBC.

Protection Recovery

To offer fast failure recovery, data plane based approaches can reduce controller communication delay. Thus, protection or proactive recovery pre-installs alternative route

in the switch flow table to offer data plane based local failure recovery. Most of these approaches use FFG and reroutes traffic promptly after the detection of a link failure. For example, in Figure 2.8, for a given source and destination pair (*A, E*) the primary route is *A-B-C-D-E*. If links between *B* and *C* fails, *B* uses FFG to reroute the affected traffic through *B-G-H-I-E*.

These kinds of recovery mechanisms install both primary and alternative routes and consume a lot of TCAM space. Moreover, these kinds of recovery always demand alternative routes.

Hybrid Recovery

Hybrid recovery [27] installs on-demand flow rules to ensure local failure recovery while efficiently utilize TCAM space. The controller proactively installs backup routes at the switches along the primary route for a given source-destination pair after receiving the initial flow installation request. Thus, the controller protects any single link failure along the primary route.

When an initial flow comes to the first switch, this technique installs flow rules along its primary path. In this technique, the controller does not install backup rules in the switches that appear in the backup path. Instead, it configures the FFG table of the primary path switches with the port number that points to the backup route. When failure happens in the primary path, traffic can move through the backup route using the FFG table. After moving through the FFG table the failure affected traffic appears in the next switch along the backup path. Upon receipt of such traffic, that switch informs the controller. In response, the controller follows the same process of the primary path establishment to install backup rules into the switches that appear in the backup route of the packet.

In Figure 2.8, for a given source and destination pair (*A, E*), *A-B-C-D-E* is the primary route. When the first packet enters into the switch *A*, the controller installs flow rules into the *A, B, C, D, and E* switch for that flow. It also adds the backup port number for the FFG table at *A* and *B*, because they have an alternative route. Thus, the link failure of *B-C* guides the affected traffic to the switch *G*. When the traffic arrives at **G**, the controller installs the flow rules at *G-H-I-E* for the affected flow. Note that, this approach cannot recover from the link failure between *C-D*

because of the unavailability of the edge-disjoint route.

From the above-mentioned discussion, we can easily find that hybrid recovery consumes less TCAM space and reduces the controller communication overhead. In this thesis, we use a hybrid recovery scheme because of their proper balance between TCAM usage and controller communication overhead.

2.2.3 Crankback (*CB*) Strategy

When link failure happens and a switch does not have a backup route then it can send the packet back to the switch from where it has received the packet. In Figure 2.8, for a given source and destination pair (*A, E*), if the link between *D and E* fails then *D* does not have an alternative route for the affected traffic. Thus, it can send the packets to *C*. Here *C* also does not have an alternative route for the affected traffic. Thus, it also sends the packets to *B*. As *B* has an alternative route, it can send those packets through the *B-G-H-I-E* path.

In most of the CB strategy, all the subsequent traffic between *A, E* follows the path *A-B-C-D-C-B-G-H-I-E* instead of *A-B-G-H-I-E*. Thus, they experience a significant amount of delay due to the traversal of the additional *C-D-C-B* hops. We name this delay as the *backtracking delay* [17].

2.2.4 Crankback With Controller (*CBC*)

To terminate the backtracking chain, CBC is a variant of CB. In CBC, the switch that experiences a failure informs the controller about that failure. After receiving that notification, the controller updates the flow rules in the switches that will be affected by that single link failure. Thus, subsequent traffic can move to the alternative route and they do not traverse towards the affected route. In Figure 2.8, for a given source and destination pair (*A, E*) assume the link between *D, E* fails. In that case, switch *D* notifies the controller about the failure and it sends the affected traffic to *C*. Thus, the controller can install new rules in *B*. Before the new rule installation in *B*, all the affected traffic follows the path *A-B-C-D-C-B-G-H-I-E*. When the controller installs new rule in *B*, the subsequent traffic can follow the *A-B-G-H-I-E* path. Often the controller requires some time to install the alternative routes. At that time traffic will experience the backtracking delay.

2.3 Compression Aware Fine-Grained Monitoring

Network monitoring gives insight about the various performance parameters such as delay, bandwidth, link loss, etc. of a network. Some of the monitoring technique provides approximate performance parameters. For example, 5 flows come to a switch and exchange 700 packets. This example cannot tell about the packet count for each flow that comes to that switch. To get such granular information fine-grained monitoring is necessary. The OVS switch maintains several counters to offer some monitoring information. Each flow rule has a corresponding counter to offer it's usage statistics. We can only separate a flow statistics from a flow rule only when each flow maintains separate flow rule. This kind of per-flow basis rules is called exact-match rules. These exact-match rules consume a significant TCAM space.

To improve the TCAM usage, we can compress the forwarding rule. Rule compression strategies aggregate the flow rule with the most common characteristics; such as common destination IP, a common output port, common tag, etc. This kind of aggregation forces multiple flows to share the same flow rule. Wild-card rule is the most common rule compression technique that is being used in the literature [28, 29]. Based on the common pattern such as common destination IP, port, etc., several flow rules compress into the wild card rule.

In Port	Source IP	Destination IP	Counter	Output
1	10.0.0.1	10.0.0.3	20	3
2	10.0.0.4	10.0.0.3	50	3
3	10.0.0.3	10.0.0.10	70	5
3	10.0.0.31	10.0.0.10	80	5

Exact-Match Rules

In Port	Source IP	Destination IP	Counter	Output
*	*.*.*.*	10.0.0.3	70	3
*	*.*.*.*	10.0.0.10	150	5

Wildcard Rules

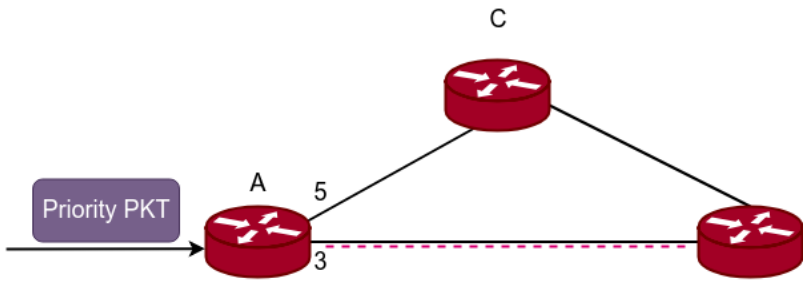
Figure 2.9: Wild-card and Exact-Match Rule Examples.

In Figure 2.9, we present both exact-match and wildcard rules. In the exact-match rules, we can see that for each source and destination pair there are separate flow rules installed in the flow table. But for wildcard rules, the rules that share common destination IP is compressed. Here, we can see *10.0.0.3* is the common destination for *10.0.0.1* and *10.0.0.4*. Also, *10.0.0.10* is the common destination for *10.0.0.3* and *10.0.0.31*. In the wildcard rule-based table, these 4 rules of the exact-match rule table are compressed into 2 rules based on the destination IP. This kind

of compression saves the flow rule storage overhead aka TCAM usage.

Though rule compression provides better TCAM usages, they reduce the visibility of the network [18]. To illustrate this, let's consider Figure 2.9. From the figure, we can see that for the exact-match rules we can get the packet count as 20 from *10.0.0.1* to *10.0.0.3*. And that is also true for the packets between *10.0.0.4* and *10.0.0.3* where the packet count is 50. But when the rule for these two flow aggregates into a wildcard rule, we cannot certainly tell which flow sends how many packets between them from the counted 70 packets. Thus, if we want to monitor the packet count from this compressed rule, we will not be able to find an exact rule count for each flow. Thus, we can conclude that rule compression reduces network visibility. In this thesis, we use the wildcard rule for cMon and provide fine-grained monitoring data.

2.4 Congestion Control for Priority Flows



In Port	Priority	Outport
1	1	3
2	2	3
4	10	*

Figure 2.10: Congestion Control Problem For Priority Flows.

A flow can be of different priorities based on service level agreements *(SLA)*. Network administrators often assign higher priority to flow when it strictly demands the shortest route. Congestion in the network often challenges the guaranteed shortest path for the flow.

A link can be congested due to the excessive arrival of packets that slowly exceeds its capacity. When congestion happens, we can reroute traffic in the alternative

congestion-free routes. In Figure 2.10, assume the link between A and B becomes congested. In that case, we can reroute the packets that use output port 3 to the output port 5, assume port 5 is not congested. Now assume if a higher priority flow comes at *In Port* 4, then we should make sure it passes through port 3 and moves through A-B path due to A-B being the shortest route.

That means if we eventually reroute the flows that come at *In Port* 1 and 2 to the outport 5 then, we can probably accommodate the higher priority flow in the path. If removal of low priority flow does not accommodate the higher priority flow then we can transmit that higher priority flow in the shortest alternative route to minimize the delay for that flow.

Chapter 3

Related Work

In this chapter, we compare and contrast relevant related works of this thesis. At first, we present the failure recovery schemes related to SD-FAST. After that, we present network monitoring strategies related to cMon. Finally, we present the congestion control strategies related to DPAL.

3.1 Failure Recovery in SDN

In this section, we present the works related to our SD-FAST. We categorize existing works into restoration, protection, hybrid, and header modification based recovery.

3.1.1 Restoration Recovery

In these schemes, after the detection of a link failure, switch contacts the controller to recover from it.

The works proposed in [30–32] use the controller to compute and restore the backup paths upon the detection of a failure. Muthumanikandan et al. [30] use heartbeat messages for failure detection and do not install extra flow rules in the switch for failure recovery. Sharma et al. [31, 32] delete the failure affected flow rules from the switches to ensure only correct rules reside in the SDN switches. Though, these approaches save TCAM space while failure recovery by not adding extra flow rules, they cannot meet the carrier-grade 50ms recovery time requirement due to the controller to switch communication overhead.

Cheng et al. [33] use *VLAN ID* to aggregate the flow rules of the post-recovery path to reduce memory usage. However, the traffic reroutes with the aggregate path can induce congestion in the post-recovery route. Moreover, the routing loop happens while using flow compression. Thus, they propose the CALFR algorithm, where they involve the controller decision to calculate the reroute path. With the use of VLAN ID, they reduce the restoration time. Still, their approach suffers from controller

communication overhead and can experience packet loss during the aggregate rule computation in the controller.

AFRO [34] generates a valid controller and switch state after a failure happens in the network. When link failure happens, AFRO emulates the actual network but removes the failure edge in the emulated topology. In this case, they use a shadow controller and replay the controller events that they stored from the actual network. After the replay of such events, the emulated network goes into a working state and AFRO transfers the switch configuration from the emulated network to the actual network. Though this approach can recover from link failure automatically, it adds up extra overhead due to the emulation of the actual network and replay of past events. Moreover, this approach cannot provide local failure recovery as it is a restoration approach and suffers from switch to controller communication delay.

3.1.2 Protection Recovery

The above-mentioned restoration based recovery schemes suffer from the controller to switch communication overhead and fail to recover within the carrier-grade 50ms time-bound. Proactive recovery schemes use pre-installed backup rules in the switch to recover from link failure locally.

To recover from link failure, fast failure detection and recovery is necessary. The authors in [11, 12, 35, 36] deploy BFD with the FFG to detect and recover from link failure locally. Besides the local failure recovery, Sharma et al. [11] highlight the pitfalls of reactive recovery. Ghannami et al. [12] generate a set of pre-computed rooted trees for the primary and backup routes and use FFG at each switch to redirect the traffic to the backup route. In [35], Xie et al. use geographic-based backup topologies generation and splicing to recover from disaster failure. They use multi-stage pipeline processing along with the FFG to locally recover from such failure. Along with the FFG based recovery, Van Adrichem et al. [36] deploy crankback routing for the topology that has a crankback path. Except for the work of Van Adrichem et al. [36], all the above-mentioned approaches do not suffer from link failure when the topology has a crankback route. Though the approach of Van Adrichem et al. [36] supports crankback routing, it still suffers from crankback backtracking delay.

Sgambelluri et al. [37,38] use the controller to pre-install both primary and backup

routes in the switch. They use the flow priority concept of OpenFlow to offer local failure recovery, where they store primary routes with higher priority and backup routes with lower priority. When a failure happens, the switch uses the auto-reject mechanism to remove primary routes from the flow table. Thus, affected traffic can use the remaining backup routes from the flow table. However, this approach is TCAM hungry due to the installation of both primary and backup rules in the primary flow table. Moreover, this approach will not sustain when the topology contains crankback routes.

Liao et al. [39] propose local fast failover mechanism with hierarchical disjoint paths and ensure connectivity as long as the underlying topology has the connectivity. They use hierarchical disjoint paths to simulate distributed Depth First Search (DFS) to find the backup path. However, they mention that their approach will not work on the topology that causes a routing loop. Moreover, their implementation suggests that this approach will suffer from backtracking delay.

To prevent routing loops, the work in [40, 41] use loop-free alternatives (LFA) to configure the primary and backup routes. Moreover, Braun et al. [41] encode visited node information into the packet header to provide another layer for routing loop prevention. Merling et al. [40] generate tree-based topology and tunnel the failure affected traffic to the remote neighbor to offer a loop-free path for the affected traffic. Though these approaches use loop-free paths, they do not offer crankback routing.

In another protection mechanism, Zhu et al. [42] use enhanced Breadth-First Search (BFS) based proactive backup path installation scheme to offer failure recovery. This approach aggregates backup rules with 2-stage aggregation and removes the routing ambiguity. Though this approach saves TCAM space with rule compression, it cannot recover from link failure when the topology has a crankback route.

Stephens et al. [43] propose a compression-aware routing mechanism along with the forwarding table compression algorithm. They use PLINKO [44] forwarding table compression and add traversed path info into the packet header. In this approach rule compression process adds extra processing delay in the packet forwarding.

SPIDER [45] uses OpenState [46] to provide fast failure recovery. In this architecture, a packet at first matches with the state table and based on the state of the packet they match with the flow rules in the OpenFlow table. In this approach, the

controller pre-computes the backup path using their previous work [16] and store a state in the state table. When a node experiences a link failure then they change the state of the incoming packet that follows the affected link. Based on the state, the packet reroutes to the backup path. Though this approach supports crankback, they introduce extra state table matching overhead and cannot terminate crankback backtracking.

3.1.3 Hybrid Recovery

The restoration approaches require too much time to recover from link failure and proactive approaches require more memory to store backup rules. Thus, Revive [27] introduces the first hybrid recovery scheme that pre-installs backup rules only for the communicating pairs along their primary route. It uses the benefit of the spanning structures and reroutes the failure affected traffic in the shortest path using FFG. Moreover, it can utilize better TCAM space by tweaking between different spanning structures. This kind of hybrid recovery is effective when the topology has edge-disjoint routes. However, this mechanism does not survive when topology requires crankback routing.

Another hybrid recovery mechanism called ReMon [17] ensures resiliency in flow monitoring. They also terminate cranback backtracking with the help of the controller. However, in this approach, failure affected traffic crankbacks until the controller terminates the backtracking. As the controller to switch communication introduces a longer delay, this approach still suffers a significant amount of backtracking delay.

3.1.4 Header Modification Based Recovery

Some techniques use packet headers to carry route and failure-related information. This kind of recovery does not involve a controller for failure recovery.

Liaoruo et al. [47] embed the packet route in the packet header and plan the backup path using a dynamic backup path planning algorithm. When a failure happens, the packet follows the backup route using the header information. This approach consumes less memory but packet carries extra overhead. Also, dynamic backup path planning often takes a long time to converge.

SWIFT [48] can reduce Border Gateway Protocol (BGP) convergence time when remote failure happens. A swift router runs the inference algorithm to localize a failure and predicts the affected BGP prefix using Root Cause Analysis (RCA). It reroutes the affected prefixes in the alternative route and uses a data plane encoding mechanism to match and reroute all the prefixes affected by the remote failure. The accuracy of this approach relies on the inference algorithm and its two-stage forwarding with the SDN switch increases more latency.

Table 3.1: Summary of Different Failure Recovery Schemes

Failure Detection Techniques	**Mechanism Type**	**CB Support**	**Backtracking Termination**	**Recovery Technique Affects Regular Data?**
Liao et al. [39]	Proactive	Yes	No	Yes
Sharma et al. [31, 32]	Reactive	NA	NA	No
Sharma et al. [11]	Proactive	No	No	Yes
Sgambelluri et al. [37, 38]	Proactive	No	No	No
Revive [27]	Semi Proactive	No	No	Yes
LFA [40]	Proactive	No	No	Yes
CALFR [33]	Reactive	NA	NA	No
Xie et al. [35]	Proactive	No	No	Yes
Van Adrichem et al. [36]	Proactive	Yes	No	Yes
SPIDER [45]	Proactive	Yes	No	Yes
AFRO [34]	Reactive	NA	NA	No
Liaoruo et al. [47]	Proactive	NA	NA	Yes
SWIFT [48]	Proactive	NA	NA	Yes
Blink [49]	Proactive	NA	NA	No
FFG [12]	Proactive	No	No	Yes
CB [16]	Proactive	Yes	No	Yes
CBC [17]	Proactive	Yes	Yes Controller Based	Yes
SD-FAST	Proactive	Yes	Yes Distributed	No

Blink [49] identifies that the first BGP update message propagation on SWIFT [48] takes several minutes. Thus, it uses TCP re-transmission statistics and infers failure based on such statistics. While rerouting, Blink sends few flows to the several

alternative routes and from those flows, it infers the best alternative route. Though, it shows better performance on real Tofino Wedge 100BF-32X switch [50], failure detection in Blink is dependent on TCP traffic.

The above-mentioned techniques mostly do not support crankback. If they do, they cannot terminate crankback backtracking effectively. Also, most of them affect regular traffic for failure protection. In table 3.1, we present a summary of some relevant works and show a comparison with SD-FAST.

3.2 Network Monitoring

In this section, we ve, passive or hybrid techniques. present works related to cMon. We categorize the existing monitoring approaches into acti

3.2.1 Active Monitoring

Active monitoring techniques insert probe packets to learn the network state.

OpenNetMon [51] continuously queries the OpenFlow switch counters and gathers the statistics; such as bytes sent, duration of each flow, etc. They measure the end-to-end delay in the network by injecting probe packets from the controller. This approach relies on switch counters thus, it cannot provide per-flow statistics when flow rules are compressed.

The works proposed in [18, 52] collects per-flow statistics using few exact-match rules. Flowstat [18] uses an ILP to determine the optimal number of switches to place exact-match flow rules. While liteFlow [52] places exact-match rules into few switches in a load-balanced manner and ensures nonredundancy. Both of the approaches use the controller to place exact-match rules and can lose monitoring data if link failure happens. Also, these approaches cannot fully use wildcard rules and consume extra memory for exact-match rules.

SwitchPointer [53] uses the end-host programmability and visibility feature of the switch to monitor the network. It uses the switch as a directory service to point to the end host where it stores network telemetry data. It collects monitor data with PathDump [54] and embeds into the packet headers for transmission to the end-host. Even though the experiment on real-world testbed shows the efficacy of SwitchPointer,

failure driven monitoring data loss can happen in transition. Moreover, embedding data on the packet header creates extra overhead.

3.2.2 Passive Monitoring

While active monitoring techniques introduce extra packets for monitoring the network, passive monitoring techniques do not insert any probe packet for monitoring the network statistics. With *FLOW_REM* message or after the periodic interval, these strategies report the monitoring data to the collector node.

In OpenTM [55], switch reports network traffic information at a fixed polling interval. With this information, it computes a network-wide traffic matrix and aggregates monitoring data to produce final statistics. For the monitoring data, they rely on the statistics that are generated by the OpenFlow switch. Thus, for wildcard rules, it will fail to provide per-flow statistics.

Instead of on-demand active polling, FlowSense [56] analyzes the control messages that exchange between the switch and the controller. It uses the *PACKET_IN* and *FLOW_REM* messages for gathering the monitoring data. This kind of monitoring works on the discrete point of time and cannot provide per-flow statistics in a flow rule compression environment.

3.2.3 Hybrid Monitoring

These kinds of techniques combine both active and passive monitoring techniques. In these techniques, switches either export the data at a periodic interval to a remote collector or collector query the switch to obtain monitoring data.

The work proposed in [57, 58], samples the packet at each switch to gather flow statistics. These approaches do not require any additional flow rule placement and work in a flow rule compression scenario. The accuracy of these approaches depends on the choice of the sampling rate. Though NetFlow [57] allows per-flow monitoring, it relies on lossy UDP to transfer flow statistics. On the other hand, OpenSample [58] always samples packet at a fixed interval and losses monitoring data. Moreover, when link failure happens, both of these approaches lose some monitoring data.

FlowCover [59] uses an optimal polling mechanism to get per-flow-rule statistics in the network and reduces communication overhead for gathering monitoring data.

They use a global view of the topology and active flow to formulate a weighted set cover to optimize the communication cost.

Table 3.2: Summary of Different Monitoring Techniques

Monitoring Techniques	**Per flow statistics support**	**Compression Support**	**Link Failure Affects Accuracy**	**Evaluation**
SwitchPointer [53]	Yes	Can be applied	Yes	Real Hardware
OpenNetMon [51]	No	No	Yes	Real Testbed with Custom Topology
NetFlow [57]	No	Can be applied	Yes	Real Testbed
FlowCover [59]	No	No	Yes	Simulation with Erdos-R enyi and Waxman graph
FlowSense [56]	No	No	Yes	Small OpenFlow Testbed
OpenTM [55]	No	Can be applied	Yes	Real Testbed but Small Toplology
FlowRadar [60]	Only per flow counter data	No	Yes	Emulation with FatTree topology
LiteFlow [52]	Yes	Partial	Yes	Mininet with Custom Topology
OpenSample [58]	No	No	Yes	Mininet and Real Testbed
FlowStat [18]	Yes	Partial	Yes	Mininet on AttMpls and Goodnet Topology
Exact-Match [36]	Yes	No	No	Mininet on AttMpls and Goodnet Topology
cMon	Yes	Yes	No	Mininet with Real Topology

Though their evaluation reduces 50% monitoring cost, still this approach will fail to provide per-flow statistics when wild card compression is in use.

FlowRadar [60] encodes and decodes the measurement counter with invertible bloom filter lookup table (IBLT). In this work, they introduce low memory overhead while storing the counters. It decouples the statistics computation labor to the monitoring collector node because switches have resource constraints. This work is best suited for gathering per-flow counter data. However, this approach fails to identify more granular monitoring data.

Most of the related work that we presented so far either suffers from accuracy or large resource consumption. In table 3.2.3, we summarize their contributions with limitations and highlights the novelty of our proposed cMon algorithm.

3.3 Congestion Control

In this section, we discuss the congestion control schemes related to our DPAL algorithm. They can be categorized either into centralized or distributed schemes. In centralized schemes, the controller gathers resource utilization information from SDN switch with OpenFlow messages and detects congestion. While in the distributed schemes, congestion control happens solely in the data plane. At first, we present centralized schemes and later we mention about the distributed schemes. At the end of this section, we summarize the existing literature and highlight their problems.

3.3.1 Centralized Schemes

In these schemes, controller senses about the congestion and plans the route accordingly to reroute the affected traffic in the best alternative path.

In [13], Lin et al. use the controller to monitor the load status on each port of the switch. When it senses the congestion, it iteratively switches the flow in the backup path. In this work, they also use FFG to recover from link failure locally. Though this approach switches the flow in the backup path, they did not mention about the path preservation of higher-priority flow.

In a Machine-to-Machine network (M2M), traffic is more frequent but has a smaller payload. Chen et al. [61] use the SDN controller to monitor the load status of the network service capability layer (NSCL) and delay requirements on the packet header.

With such information, it distributes the load in the best path. However, this work does not preserve the primary path for a higher-priority flow.

In Hedera [62], the controller monitors the flow rate at edge switches to detect large flows. When a flow has a rate larger than 10% of the link capacity, it considers that flow as an elephant flow. Using Global First Fit and Simulated Annealing, Hedera schedules elephant flows with the help of the controller. Though the scheduling of elephant flow improves congestion in Hedera, the large flow detection method of Hedera suffers from accuracy because not all large flow reaches to the rate limit.

FDALB [63] uses sent-byte count as the metrics for long flow detection. The threshold value for the declaration of long flow is adaptive based on the box plot method. It transmits short flows using ECMP [64] and schedules long flow using the centralized controller. The sent byte count as a threshold value for large flow detection cannot truly predict the remaining bytes of a flow. Thus, FDALB can suffer from unnecessary switching of flow rules to offer better congestion control.

MicroTE [65] collects sent or received byte count from the datacenter servers. A routing module of the controller uses these statistics to determine the appropriate path. After that, the network controller installs these predictable traffic paths in the SDN switches. That path distributes the predictable traffic in the noncongested path. In this strategy, nonpredictable traffic uses static weighted ECMP to distribute the load. Though MicroTE outperforms ECMP, it shows poor performance on less predictable traffic such as- university data center traffic. Moreover, this strategy has significant computation overhead for predictable pattern identification.

Fastpass [66] achieves zero-queuing by controlling path and sending time of each packet. It exchanges traffic metrics and allocation decisions between controller and end-host using Fastpass Control Protocol (FCP). Its timeslot based mechanism determines the transmission time of the packet to achieve better network utilization. Also, It's path selection algorithm assigns each timeslot to the path having zero-queue. As it does the path selection and timeslot selection with the help of the controller, it's performance is correlated to the controller capacity. Also, traffic scheduling from the end-host is not feasible when lots of hosts try to exchange data.

3.3.2 Distributed Schemes

In these schemes, the data plane switch senses the congestion and reroutes the affected traffic in the alternative route without contacting to the controller.

The work proposed in [67, 68] uses VXLAN [69] header to carry congestion information. Modularized Load-Aware Balancing (MLAB) [67] divides the higher-tier network into multiple routing domains and maintains a remote load matrix to gather global congestion view. This technique uses a flowlet scheduler to split traffic to provide max link utilization. CONGA [68] also maintains a local congestion table and updates it when feedback information reaches the leaf node of a 2-tier Leaf-Spine topology. When the inter-arrival time of a packet exceeds the maximal latency of all path then CONGA chooses a congestion-free path for that packet from the congestion table. Though these approaches show better performance compared to their counterparts, they require a significant amount of time to gather global congestion information. Moreover, they cannot offer the shortest path guarantee for higher-priority flow.

CLOVE [70] uses in-band network telemetry to calculate path utilization. It also relies on explicit congestion notification to calculate the weight of the path. Using these weight it balances the load in a round-robin fashion. However, this approach uses continuous probe packet among switches to discover the path. Such continuous exchange can create a performance bottleneck in the data traffic.

Hwang et al. [71] propose a scalable congestion control protocol, called SCCP to control in cast network congestion. It monitors each switch output port and considers the number of flow, link capacity, RTT, etc. to calculate fair_share value. Such value is feed to the sender so the sender adjusts data size to transmit. To hold the fair_share value SCCP utilizes the TCP advertisement window field. fair_share value modification happens in the packet only when the current value supersedes the previous one. In this strategy, fair_share propagation back to the sender might be lossy and time-consuming. In that time, network congestion can reach to the peak point.

A global congestion-aware load balancing technique called HULA [72] highlights the large memory requirement for the storage of link utilization information of every Top-of-Rack (ToR) pairs. Thus, it stores the best next hop in the switch to save

memory. To offer congestion-free routing it broadcasts small-sized probe packets from the leaf switch to other neighbor switches. HULA schedules flow based on the flowlet and maintains an updated time table to handle link failure. It replaces the best next-hop switch in its routing table when it does not receive probe packets for a certain period from that switch. Though HULA claims 8x better performance than ECMP, it suffers from the overhead introduced in the periodic exchange of probe packets.

In FlowBender [73], authors use Explicit Congestion Notification (ECN) to sense for congestion. After detection of congestion, it modifies the packet header such that, switches triggers to setup a new hash value for the packet to reroute it in the new path. Though it shows better performance than ECMP and does not require hardware modification like DeTail [74] and RPS [75], it may require several attempts to find the non-congested path.

An active queue management technique, called Random Early Detection (*RED*) probabilistically drops the arriving packet to prevent congestion. Based on the estimated time-averaged queue length, RED decides whether or not to drop an incoming packet. It drops a packet if the queue size is above a threshold. Otherwise, it does not drop the packet. However, this approach provides congestion avoidance in the cost of packet loss. Moreover, this approach does not consider the behavior of higher priority flows.

Congestion control techniques presented in this section mostly base their work on effective load distribution to maximize link utilization. However, none of them guarantees the shortest path for higher priority flow like our DPAL algorithm. Table 3.3 presents the summary of related works of DPAL with some shortcomings.

Table 3.3: Summary of Different Congestion Control Techniques

Congestion Control Techniques	**Mechanism Type**	**Provides Best Path For Priority Flow**	**Overhead Inducing Points**
Chen et al. [61]	Centralized	No	Transfers Load Status in Packet Header
Hedera [62]	Centralized	No	Large Flow Detection
FDALB [63]	Centralized	No	Large Flow Detection
MicroTE [65]	Centralized	No	Monitoring Module Collects Sent/ Received Traffic Count
Fastpass [66]	Centralized	No	zero-queuing path Finding
MLAB [67]	Distributed	No	Transfers Load Status in Packet Header
SCCP [71]	Distributed	No	fair share computation
HULA [72]	Distributed	No	Probe Packets
FlowBender [73]	Distributed	No	Explicit Congestion Notification
CONGA [19]	Distributed	No	VxLAN Header Processing
CLOVE [70]	Distributed	No	Continuous Probe Packet
Fast Switchover [13]	Centralized	No	Controller collects port status
DPAL	Distributed	Yes	Per Packet Processing

Chapter 4

SD-FAST: A Packet Rerouting Technique

In this chapter, we present a novel packet rerouting mechanism, called SD-FAST, that works in a distributed fashion and offers a best-effort failure recovery. We present the architecture and design of SD-FAST in section 4.1. After that, we describe the experimental setup in 4.2. In section 4.3, we present the evaluation results to show the performance of SD-FAST compared to its counterparts.

4.1 Architecture and Design

In this section, we present the design and architecture of SD-FAST. Figure 4.1 shows the architecture of SD-FAST, where different network functions are decomposed into modules. In the application layer, the *route planner* module communicates with the controller over the NBI. *topology control*, *route configuration*, and *statistic collection* are the main functional modules of the control-plane in the SD-FAST architecture. The data plane elements of SD-FAST architecture are OVS switches. The key functional modules of SD-FAST includes the *link and flow monitors*, *rule changer*, and *rule grabber* that reside in the data plane. The communication between the data and control plane occurs over OpenFlow 1.3 SBI.

4.1.1 Application and control plane modules

With the help of network events (e.g., a link failure or adding a switch), *topology control* module constructs and maintains the network topology. The *route planner* module takes the topology from the *topology control* module and constructs a graph $G(V, E)$, where V and E are the set of nodes and vertices, respectively. From this graph, it generates a subgraph according to the algorithm presented in [27]. From the subgraph, it computes the primary and backup route using the shortest path tree (SPT) or minimum spanning tree (MST) depending on an application's demand. The subgraph generation algorithm includes crankback edges in the backup path

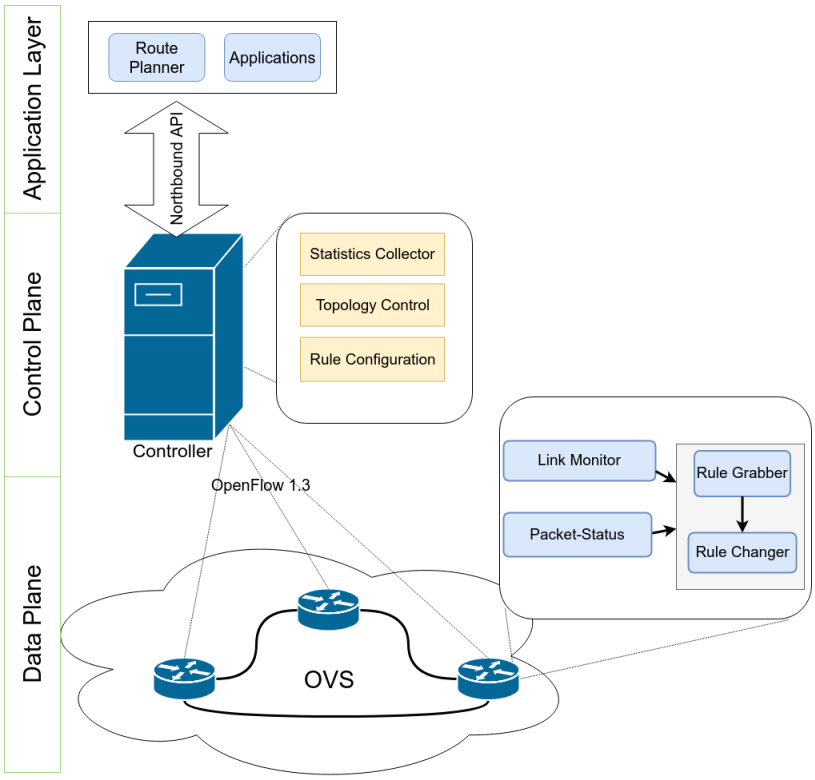


Figure 4.1: The Architecture of SD-FAST [1].

when topology lacks edge-disjoint route. The *route configuration* module uses the routes obtained from the *route planner* module to determine the output ports for a given source-destination pair, (s, d). It begins with the primary route between (s, d) and sets two output ports for the primary and backup path at every node until it reaches the destination. When the intermediate nodes between (s, d) do not have an alternative route to the destination we can redirect the traffic to *IN_PORT*.

After obtaining the necessary pair of output ports, the route configuration module prepares and installs *only* the primary route (port) in the flow table at every switch between (s, d). It keeps the backup port in a *backup table* inside OVSDB. Switch fetches and installs these rules in the flow table only when there is a link failure. Thus, SD-FAST does not need to traverse a chain of tables, whereas in FFG, CB, and CBC each packet experiences the pipeline processing overhead.

4.1.2 Data plane modules

SD-FAST is a local failure recovery mechanism. Thus, the major functional modules of SD-FAST resides in the data plane. Among these modules, the link monitor module provides the link status information by using the BFD protocol. To decide on the link status, BFD continuously exchanges control packets between two neighbor nodes and declares a link failure when certain number of them are missing or a packet is missing for a certain amount of time. The data packet that comes to a switch can be in one of the three status: *regular*, *crankback*, or *recovered*. The packet-status module inspects and checks the status of each packet. When a packet moves through the crankback route, its associated status is *crankback*. The status changes to the *recovered* as soon as the packet finds an alternative route. We use the recovered status at the destination switch to update the reverse-path for a given source-destination pair.

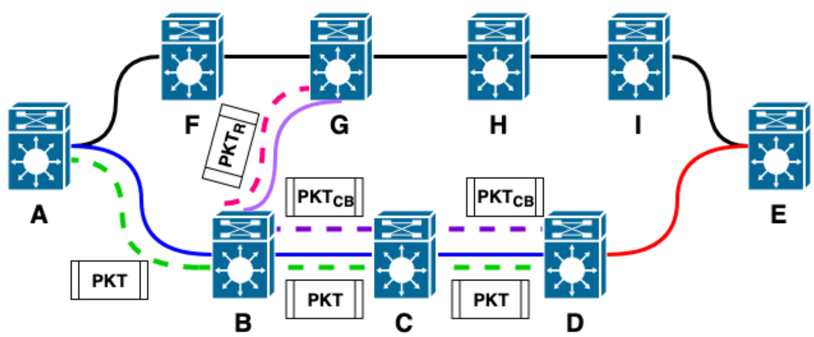


Figure 4.2: Different types of packet status in SD-FAST [1].

In Figure 4.2, we present different types of packet status. For example, PKT, PKT_{CB}, and PKT_R represent regular, crankback, and recovered status, respectively. Assume $A - B - C - D - E$ is the primary route for a given source-destination pair (A, E). If the link between D and E fails, then SD-FAST changes the status of the packet from PKT to PKT_{CB} at D. Once that packet, reaches to the B, it gets recovery path. Thus, its status again changes to PKT_R.

To recover from link failure a switch requires to install the backup rules. The *rule grabber* module of SD-FAST fetches the relevant backup rules from the user-space table. To do so, it filters the rules that are using that failed link based on the destination IP address. The *rule changer* module uses the fetched backup rule from

the *rule grabber* module and replaces the current forwarding rules with the grabbed backup ones. Using the newly installed backup rules, affected traffic moves through the backup route and receives recovered or crankback status. We present the detailed operation of the rule changer module in Algorithm 1.

Algorithm 1 Rule Changer [1]

Input:

Selected Backup Rules: r_b where, $r_b \in R$ Destination Status: dst

Output:

Rule Change Status: s_r

```
1: for each rule ∈ r_b do
2:   if !dst then
3:     if in_port(rule)! = out_port(rule) then
4:       pushRcInst(rule)
5:     else
6:       pushCbInst(rule)
7:     end if
8:   end if
9:   push rule to data path table
10: end for
11: update s_r
```

To modify and install the selected backup rules, r_b from the set of backup rules, R, we use Algorithm 1. For each rule $rule$ of r_b at first, we check whether a packet P arrives at the destination node or not. If the current node is not the destination of the packet then we leave the rule as it is and install in the data path table. But if the current node does not belong to the destination of the packet, then we further check whether the rule has the same input and output port. If it has the same input and output port we need to modify the rule with crankback instruction. Otherwise, we modify the rule with recovery instruction.

If Algorithm 1 needs to change m flow rules for a particular IP and rule insertion takes place in a hash structure then the time complexity for the Algorithm 1 is $\mathcal{O}(m \log p)$. Here, $\log p$ is the time taken to insert a rule into a hash structure. We

consider the time complexity to push an instruction into the rules as negligible.

4.1.3 Packet Rerouting Operation

Algorithm 2 Packet Rerouting Algorithm [1]

Input:

Packet: P

Link Status: L_s

Output:

Rule Change Status: S

1: $S \leftarrow 0$
2: $dst \leftarrow 0$
3: $IP \leftarrow Extract(P)$
4: **if** $isDown(L_s)$ and $isEmpty(S[IP])$ **then**
5: $R \leftarrow RuleGrabber()$
6: $S[IP] \leftarrow RuleChanger(R, dst)$
7: **else if** ($isCbtag(P)$ or $isRctag(P)$) and $isEmpty(S[IP])$ **then**
8: $R \leftarrow RuleGrabber()$
9: **if** $isRctag(P)$ and $isDest(P)$ **then**
10: $dst \leftarrow 1$
11: $S[IP] \leftarrow RuleChanger(R, dst)$
12: **else if** $isCbtag(P)$ **then**
13: $S[IP] \leftarrow RuleChanger(R, dst)$
14: **end if**
15: **end if**

When a packet comes to an interface of a switch, the OVS switch process it with the defined flow rules in its primary flow table. While doing so, we also pick the packet, P and feed into the algorithm 2. In addition to the packet, we also feed the output link status, L_s into that algorithm. At line number 1 and 2 of the algorithm, we reset the rule change status, S and destination node status, dst. After that, we grab the destination IP address from the packet. If we find the L_s is down and the rule is not changed for the destination IP, we grab the backup rule with the

RuleGrabber module and change them in the primary table using the *RuleChanger* module. At line number 8 to 15, we take care of remote failure by checking different packet status. If the packet has crankback or recovered tag and the rule is not already changed for the packet's destination, the algorithm grabs the backup rules using the *RuleGrabber* module. If the packet is in destination, this algorithm notifies the *RuleChanger* algorithm to change destination switch specific rules. Otherwise, it asks *RuleChanger* algorithm to change backup rule such that, it adds crankback tag to the subsequent packets.

4.2 Evaluation Setup

In this section, we present and describe the emulation environment for the evaluation of SD-FAST. We evaluate SD-FAST using Mininet 2.3 emulator, Open vSwitch version 2.9.1 and Ryu controller version 4.30 on a server that consists of 2.66GHz 12 core CPU, 44GB RAM, and Ubuntu 16.04.3 LTS operating system. The data plane OVS switches communicate with the Ryu controller using the OpenFlow 1.3 protocol. For each link, we fix the propagation delay at 5ms. Moreover, we assume zero percent link loss [76] to hold on the consistency in the outcome.

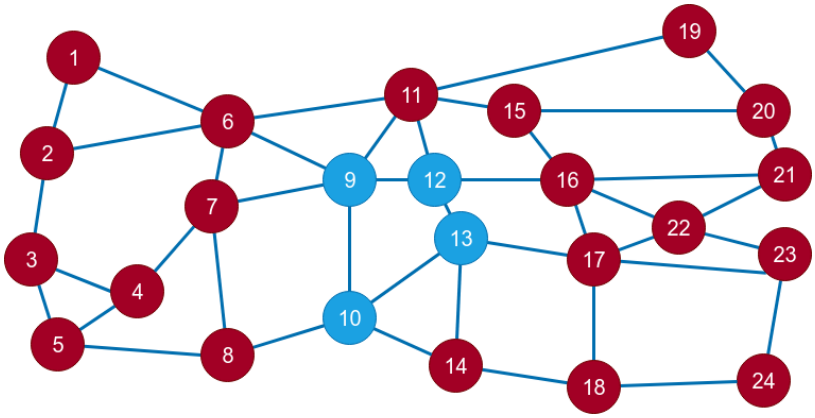


Figure 4.3: 24 node USNET Topology [1].

To evaluate the performance of SD-FAST, we choose 24 nodes USNET topology (Figure 4.3) and 28 node backbone topology called Darkstrand (Figure 4.4). In USNET topology, every node has an edge-disjoint route whereas the Darkstrand topology contains crankback route. Thus, with the USNET topology, we can evaluate the performance of SD-FAST and FFG, whereas we can evaluate the performance of

SD-FAST, CB, and CBC in the Darkstrand topology. In all experiments, we randomly

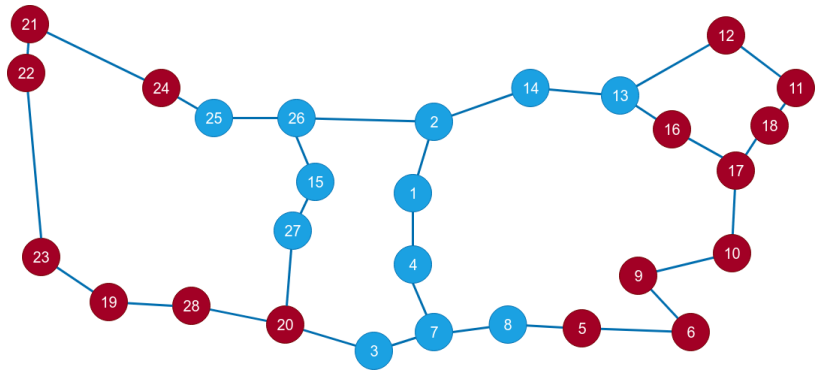


Figure 4.4: 28 node Darkstrand Topology [1].

choose 60 source-destination pairs from the USNET and Darkstrand topology and make sure that they have at least 6 hop primary routes in USNET topology and 10 hop primary routes in Darkstrand topology. We use this 60 source-destination pair to exchange data between them. Due to resource constraints, we use 12 pairs at a time from the 60 source-destination pair for concurrent communication and make sure that all 60 source-destination pair gets the flavor of concurrent data exchange. In Figure 4.3 and 4.4, the red colored nodes from the USNET and Darkstrand topology are the source and destination for the evaluation of this thesis.

In our first set of evaluations, we observe the impact of a link failure on the performance of SD-FAST, FFG, CB, and CBC. Thus, we measure the average end-to-end delay and average throughput in the presence of link failure in both USNET and Darkstrand topology. We exchange 50KB of iPerf [77] and ICMP [78] data between each pair and consider 60 source-destination pairs. We randomly fail different percentage of links from those pairs. During this evaluation, we make sure that, link fails concurrently for each percentage of failure. For every percentage of a link failure, we measure end-to-end delay among the 60 source-destination pair and take the average to get the average end-to-end delay. We follow the same procedure to measure the average throughput.

Next, we investigate the impact of topology. During this evaluation, we consider the same 60 source-destination pairs and emulate data exchange between them in both USNET and Darkstrand topology. We randomly fail a single link between each communicating pairs and measure end-to-end delay and throughput for each of them.

We use that end-to-end delay and throughput for calculation of average end-to-end delay and average throughput. For this evaluation, we also maintain the 50KB as the data size for ICMP and iPerf traffic.

After the observation of the impact of topology, we evaluate the impact of real traffic on SD-FAST, FFG, CB, and CBC. During this evaluation we use wget [79] to transfer MP4 files between 60 sources and destination pair, where 12 pairs concurrently exchange the file. For this evaluation, we use Darkstrand topology and vary the size of the MP4 file between 200MB to 350MB. For each pair of the 60 source-destination pair, we exchange MP4 files and measure end-to-end transfer delay and throughput. Finally, we calculate average delay and average throughput by taking the average of the end-to-end delay and throughput data of 60 source-destination pairs. During this evaluation, we fail a single link along the path of each communicating pair.

After that, we evaluate the impact crankback backtracking delay on the performance of SD-FAST, FFG, CB, and CBC. During this experiment, we use Darkstrand topology and choose the same set of source and destination pairs that we choose for the previous evaluation. To measure the crankback backtracking delay we timestamp at the failure moment in the affected node and observe the recovery node for crankback termination. We again timestamp in the recovery node when we find crankback traffic moves to the recovery path and do not follow the affected path. We take the difference between the initial timestamp and final timestamp and considers this as the crankback backtracking delay.

Finally, we measure the average recovery time of SD-FAST to show whether it meets the carrier-grade 50ms recovery time-bound or not. During the experiment, we exchange iPerf based UDP traffic among 60 source-destination pairs in USNET topology and break 20% link concurrently. We timestamp the affected packet at the failure moment and timestamp again when the packet moves through recovery route. We take the difference between two timestamps to measure the recovery time. Finally, we take the average of recovery time of each failure to measure the average recovery time.

4.3 Performance evaluation

In this section, we present our evaluation results and compare SD-FAST with FFG [12], CBC [17] and CB [16] techniques. At first, we present the results that show the impact of link failure. After that, we present the results to see the impact of topology. Next, we present the results to see the impact of real traffic. After that, we present the results to show the impact of crankback backtracking. Finally, we present the result that shows the average recovery time of SD-FAST.

4.3.1 Impact of Link Failure

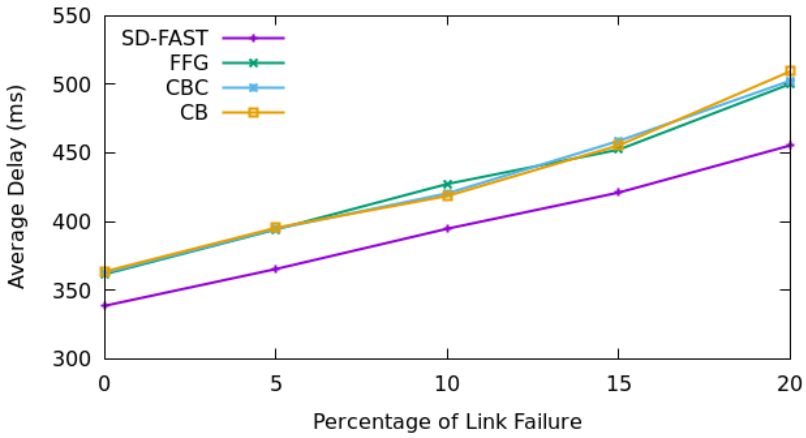


Figure 4.5: The average end-to-end delay in the presence of link failure in USNET Topology.

Figure 4.5 and 4.6 presents the results that show the impact of a link failure on average delay. As FFG cannot survive in crankback route, in Darkstrand evaluation we find an empty result for it at different failure moment. In USNET topology, CBC and CB use the FFG table to recover from link failure. Thus, FFG, CBC, and CB show similar performance in USNET topology. But, in Darkstrand topology, packet experiences backtracking delay. In FFG, CBC, and CB, the packet experience another type of delay due to their continuous comparison of data traffic with a link failure status. Such, type of delay adds up to the end-to-end delay at each switch.

The increase of path length or data size increases those types of delay. That's why SD-FAST improve up-to 60% and 28% average delay compared to CB and CBC

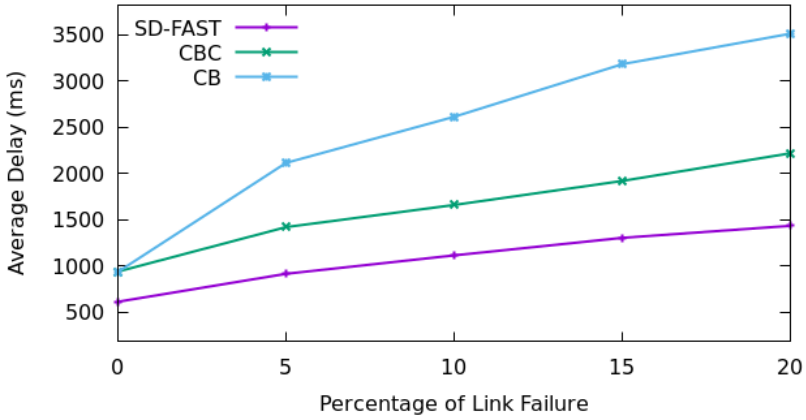


Figure 4.6: The average end-to-end delay in the presence of link failure in Darkstrand Topology [1].

respectively. Due to controller-based termination of backtracking, CBC shows better performance than CB. In the USNET evaluation result of Figure 4.5, SD-FAST improves around 14% average delay compared to FFG, CBC, and CB. Because in SD-FAST, data traffic does not experience that failure status comparison, whereas they do in FFG, CBC, and CB.

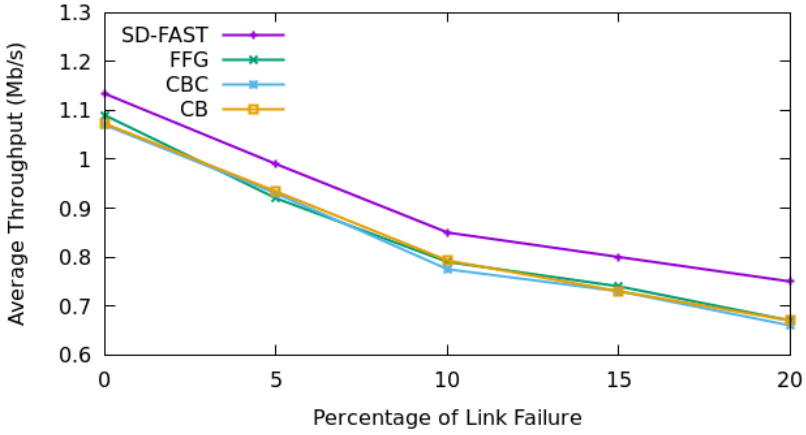


Figure 4.7: The average throughput in the presence of link failure in USNET Topology.

We present average throughput results in Figure 4.7 and 4.8 for USNET and Darkstrand topology respectively. We observe up-to 57% and 27% average throughput improvement of SD-FAST over CB and CBC respectively. Because CB and CBC experiences backtracking delay and failure status comparison delay. Recall that, CBC terminates backtracking with the help of controller and CB cannot terminate

it. Thus, CBC shows nearly 50% better performance than CB in terms of throughput.

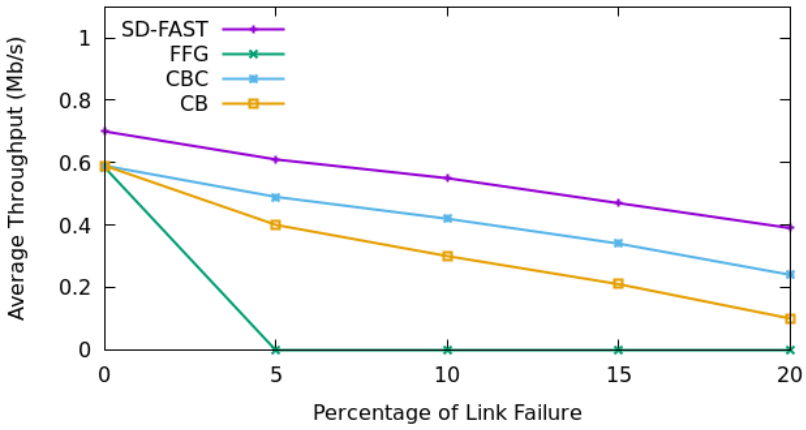


Figure 4.8: The average throughput in the presence of link failure in Darkstrand Topology [1].

In USNET topology, we can find around 12% better throughput in SD-FAST than FFG, CB, and CBC. As there is no backtracking delay in USNET, this difference in throughput is due to the failure status comparison delay.

4.3.2 Impact of Topology

In Figure 4.9 and 4.10, we present the average end-to-end delay and throughput results in the presence of link failure to show the impact of different topology on the performance of SD-FAST, FFG, CB, and CBC. From the results we can see that, SD-FAST experiences almost 3 times more end-to-end delay in Darkstrand topology compared to the USNET topology due to the longer path length of Darkstrand topology than the USNET topology. The situation is worse for CB, as it experiences almost 7 times more end-to-end delay in Darkstrand topology compared to the USNET topology. This difference comes not only for the increased path length but also for the increased amount of crankback route of Darkstrand topology. Whereas in USNET topology there is no crankback route, thus FFG, CB, and CBC experience similar end-to-end delay and average throughput. Due to the longer route length and crankback route, we observe 2.5, 4 and 6 times less throughput in Darkstrand topology for SD-FAST, CBC and CB respectively. As FFG cannot survive in crankback

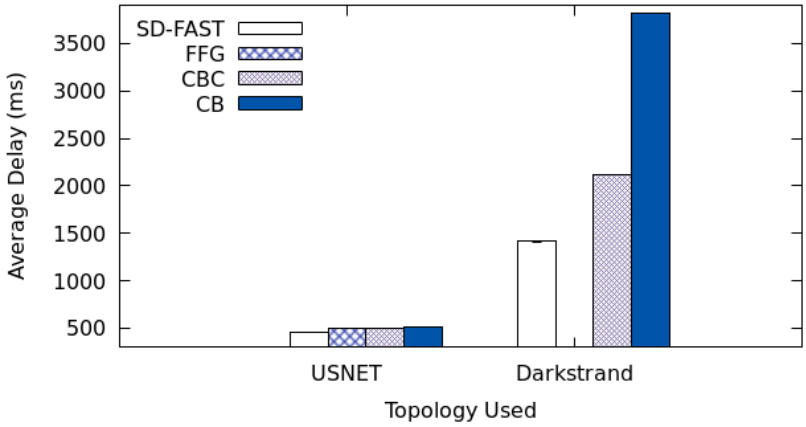


Figure 4.9: The average end-to-end delay in USNET and Darkstrand topology [1].

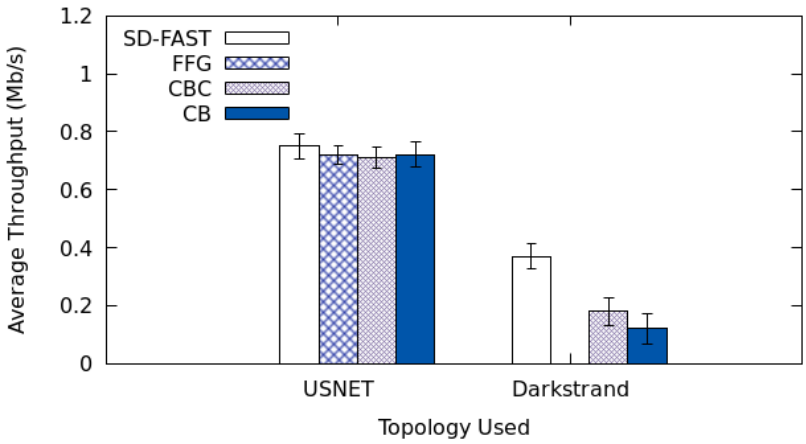


Figure 4.10: The average throughput in USNET and Darkstrand topology [1].

route we observe empty result for it in Darkstrand topology. But in USNET topology FFG survives and provides similar results to CB and CBC.

4.3.3 Impact of Real Traffic

Based on the previous evaluation we already know that link failure can introduce a longer recovery path and crankback path. If real large-sized data travels through such a path for a long time it could suffer from performance bottleneck. Thus, we evaluate the impact of real traffic in the Darkstrand topology with variable sized MP4 data. Figure 4.11 and 4.12 presents the average end-to-end delay and average throughput results for MP4 data of different size. The evaluation results suggest that the SD-FAST is up-to 64% and 40% faster than CB and CBC respectively in terms of

average end-to-end delay. In the case of throughput, SD-FAST offers almost 62% and 34% better throughput than CB and CBC respectively. As large-sized data moves over the crankback path for a long time, SD-FAST shows this improvement. Also, it terminates backtracking locally and does not affect regular traffic for failure recovery.

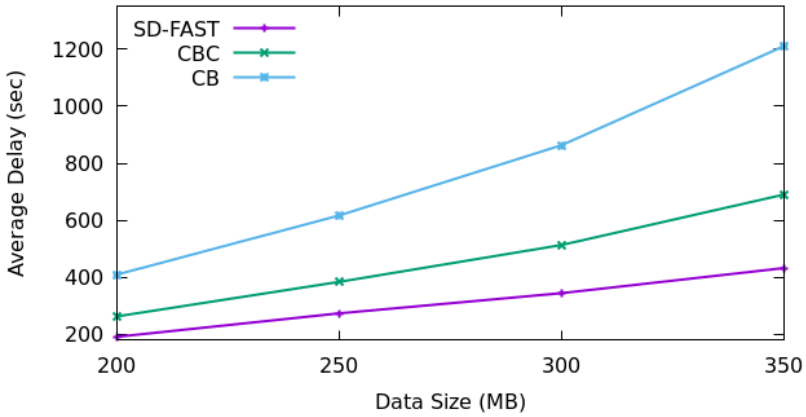


Figure 4.11: The average end-to-end delay while using real traffic [1].

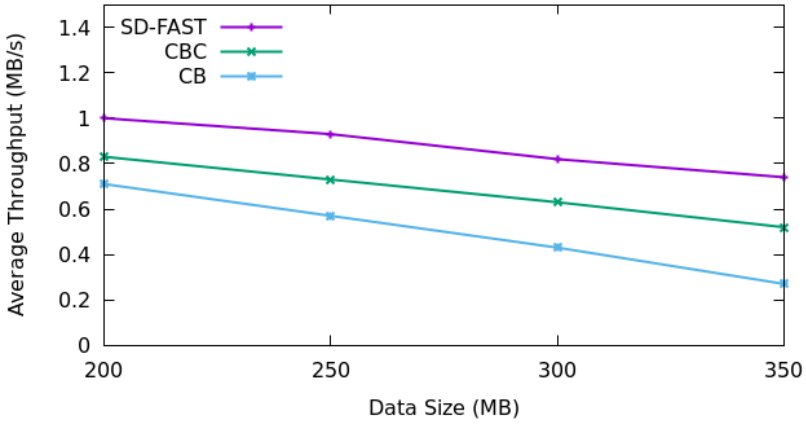


Figure 4.12: The average throughput while using real traffic [1].

4.3.4 Impact of Crankback backtracking

In Figure 4.13, we show the impact of the backtracking path. From the result, we can see SD-FAST can save nearly 73% crankback backtracking than CBC. In CBC, the controller configures the node having a recovery route to prevent the backtracking. In that time subsequent traffic suffers from backtracking delay. Whereas, SD-FAST terminates backtracking locally and shows better performance than CBC. As FFG

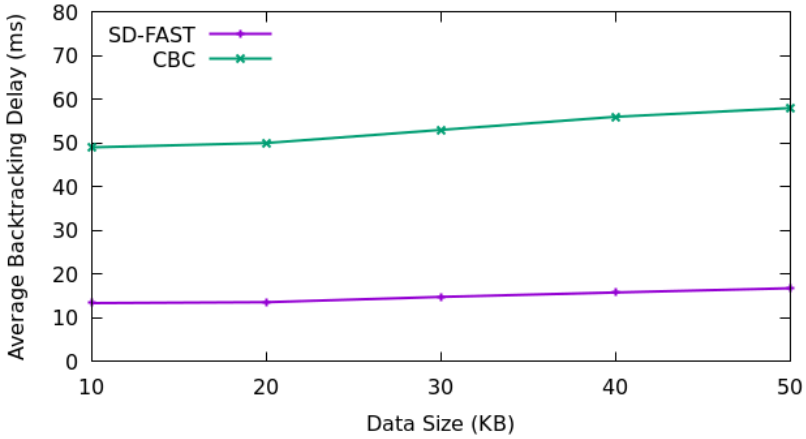


Figure 4.13: The convergence time of backtracking in the Darkstrand topology [1].

does not support crankback and CB never terminates backtracking, we omit their results from the Figure.

4.3.5 Recovery Time of SD-FAST

In Figure 4.14, we present the average recovery time of SD-FAST. During this experiment, we concurrently failed 20% link and varied the data size. From the result, we can see SD-FAST recovers in nearly half of the carrier-grade requirements of 50ms time.

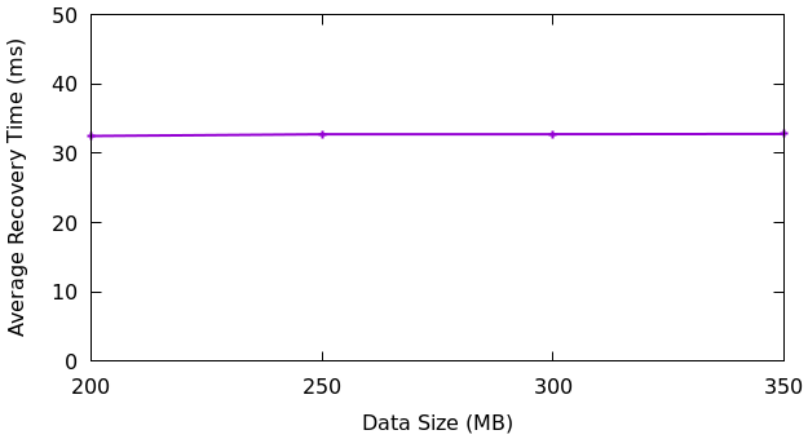


Figure 4.14: Average Recovery Time of SD-FAST.

Chapter 5

Compression Aware Monitoring

In this chapter, we present and describe our compression aware monitoring strategy called cMon. In section 5.1 we present the design and implementation of cMon and in section 5.2 we present the evaluation setup for cMon evaluation. We evaluate and discuss the performance of cMon in section 5.3.

5.1 cMon Design and Implementation

In this section we present the architecture and algorithm of cMon. The Figure 5.1 presents the cMon architecture. The architecture of cMon employs several data plane modules; such as- *Packet Collector*, *Per Flow Stats Manager*, *Link Failure Handler*, *Resource Limit Handler*, and *Stats Exporter*. It also employs another module called *stats collector* that we can place in either control plane or management plane based on the requirement.

Packet Collector: this module runs in a parallel process and collects a copy of the packet at each interface of the switch. The packet collector stores those packet in a queue. We maintain this structure because the packet processing is a slow operation than the packet arrival rate. This kind of storage in a queue structure allows faster packet storage. *packet collector* shares this queue with the *Per flow stats manager*.

Link Failure Handler: This module reacts when it senses a link failure in the packet path. Failure can happen either in the adjacent link of a switch or in a remote switch. For the adjacent link, this module senses the failure using the BFD status. But for the remote failure, it uses the packet tag to detect such failure. For a failure affected packet, there can be four different scenarios.

Firstly, the current switch monitors the packet and failure affects the route of the flow. Also, the subsequent packet for that flow will not come to this switch. In this scenario, the current switch stops monitoring the flow of the current packet and tags the packet so that the next capable switch on the path of the packet can monitor

the packet. In this case, we notify the collector to inform about the failure and the change of location for monitoring the flow.

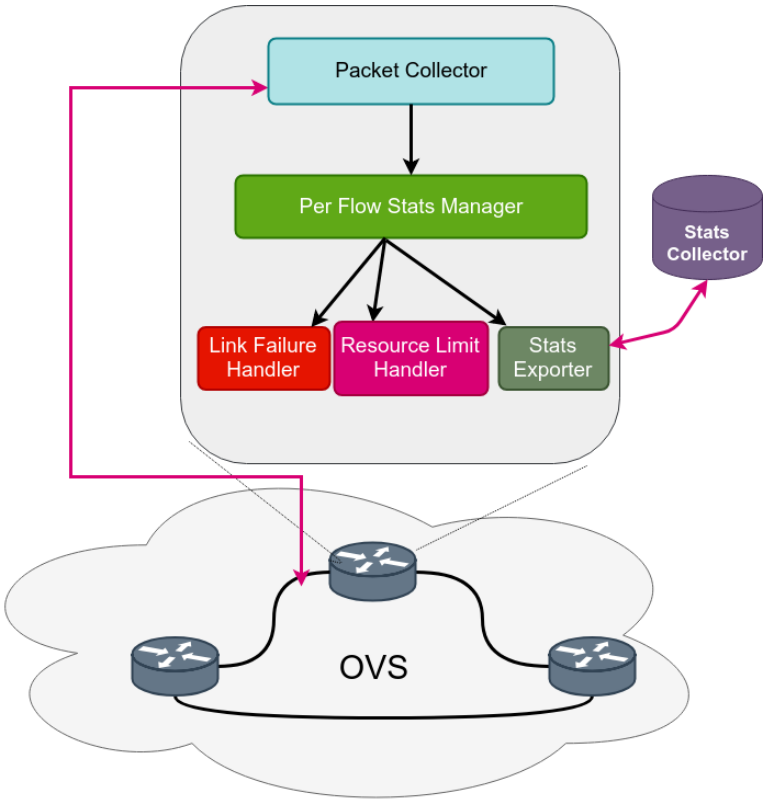


Figure 5.1: The Architecture of cMon.

Secondly, the current packet is new to the current switch and the packet has an alternative route to the destination from the current switch. In this case, we can ask the next available switch on the alternative route to monitor this packet. We do this so that the current switch does not reach the resource limit issue due to the failure shock. A failure shock is a phenomenon that requires the switch to service lots of failure affected traffic. In failure shock, a switch experiences lots of processing overhead and eventually looses its available resource. In this case, we also notify the collector so that it remains informed about the failure and the packet reroute.

In the third scenario, the current switch does not monitor the packet and failure forces the packet to move to the crankback path. In this case, we send the packet to the crankback path and inform the switch that appears in the crankback path about the failure with proper tagging. Such tagging helps that switch to separate the flow

as a failure affected flow. We notify the collector to let it know about the failure and the direction where the packet moves. This helps the collector to identify the location to get the statistics for that flow.

In the final scenario, we can find the current switch monitors the flow of the packet and link failure does not force the packet to move to the crankback path. In this case, the current switch can continue monitoring the packet. Because this kind of monitoring does not add extra operation cost as the flow is already being monitored.

Per Flow Stats Manager: this module uses the queue from the *packet collector* and pop a packet for further processing. It extracts the packet header and creates a tuple with source IP address, destination IP address, source MAC address, destination MAC address, and port number. That tuple forms the flow key. The flow key acts as a flow identifier to locate, insert, or update a flow statistics quickly. We use that flow key to store the flow information in a hash structure. Once we add any data in the hash structure we update an aggregated statistics such as average length, total packets and store them in that hash structure. Such pre-aggregation provides a quick view about that flow.

Furthermore, We calculate and maintain the packet transmission rate at per-flow stats manager. That transmission rate helps to determine the congestion level at a port. Before insertion of every new flow key, *Per flow stats manager* consults with *Link Failure Handler* and *Resource Limit Handler* modules. If it finds any resource limitation information from the resource limit handler it does not collect the statistics for that packet. Instead, *Resource Limit Handler* module tags the packet and sends it to the next capable switch for monitoring.

In the case of a link failure, if the current switch monitors the flow of the packet and link failure does not force the packet to move to the crankback path, this module stores the statistics for the packet. Otherwise, *Link Failure Handler* tags the packet for monitoring in the next capable switch.

Stats Exporter: This module sends the per-flow statistics to *stats collector* either at a periodic interval or *stats collector* can query for the flow information based on the configuration. This module sends the information to the *stats collector* in the compressed format to save the communication overhead.

Stats Collector: This module sits either in the control plane or management

plane. It collects the per-flow information from the *Stats Exporter* and does further processing before sending it to the monitoring application. This module can be configured to receive periodic information from *Stats Exporter* or it can send an on-demand query to the *Stats Exporter* to gather information.

5.1.1 cMon Algorithm

We present the overall operation of the functional module in the cMon algorithm 5.1. Our cMon algorithm 5.1, runs in a parallel process for every interface i of a switch. Recall that, when a packet comes to an interface *packet collector* module collects that packet and preserve in a queue P_{queue}. We perform a pop operation and pick the packet P for further processing. When a switches sense that a packet comes to a failure affected path P_{fail} then $FailTagPacketForMonitorNext$ module tags that packet so that next switch that is capable of monitoring the packet, can collect the statistics about that packet.

There is a possibility that switches resource usage $S_{resource}$ can cross the threshold limit of S_T. Current packet overhead, $P_{overhead}$, observed from the monitoring data, M_{data}, can also add to $S_{resource}$ and cross S_T. In such a resource constraint scenario, cMon leaves the packet for the next switch to monitor and notifies the collector about that resource limit issue so that the collector can get those monitoring data from the next switch. While handling resource limit issue, if cMon finds that the flow was being monitored in the current switch, it removes the flow key F_{key} from the monitored flow list M_{FL} as it will not monitor this flow further.

The normal scenario comes when there is no failure affected packet or there is no resource limit issue. In that case, two scenarios can happen. In the first scenario, F_{key} can be found in the M_{FL}, which indicates that the switch currently monitors the flow for the current packet. In this case, we can simply aggregate the packet information into the existing list.

In the other scenario, a packet can be new to the switch. In this case, again there can be three scenarios. First of all, a packet can come to the switch after it is affected by a failure. In that case, cMon notifies the collector that it is monitoring the packet that has experienced a failure. In another scenario, a packet can come after suffering from a resource limit issue.

Algorithm 3 cMon Algorithm

Input:

Interface: i where, $i \in I$

Current Flow Monitor List : M_{FL} where, $M_{FL} \in F$

Switch Resource Threshold: S_T

1: $P_{queue} \leftarrow PacketCollector()$
2: $P \leftarrow Pop(P_{queue})$
3: **if** $LinkFailAffectsMonitoring(P)$ **then**
4: $FailTagPacketForMonitorNext(P)$
5: $collector->notifyFail()$
6: **else**
7: $F_{key} \leftarrow BuildFlowKey(P)$
8: $M_{data} \leftarrow ExtractMonitorData(P)$
9: $P_{overhead} \leftarrow MonitorOverhead(M_{data})$
10: **if** $(S_{resource} + P_{overhead}) > S_T$ **then**
11: $TagPacketForMonitorNext(P)$
12: $collector->notifyResLimit()$
13: **if** $F_{key} \in M_{FL}$ **then**
14: $removeFlowKey(F_{key}, M_{FL})$
15: **end if**
16: **else if** $F_{key} \in M_{FL}$ **then**
17: $SaveStats(P)$
18: **else**
19: **if** $P->contains(fail_{tag})$ **then**
20: $collector->notifyFailSave()$
21: **else if** $P->contains(resLimit_{tag})$ **then**
22: $collector->notifyResLimitSave()$
23: **end if**
24: $AddFlowKey(F_{key}, M_{FL})$
25: $SaveStats(P)$
26: **end if**
27: **end if**

In this case, cMon also notifies the collector that the current switch will monitor the packet that has suffered from the resource limit issue. The tag information sent to the collector through the notification message helps the collector to properly aggregate the data. In the final scenario, a packet can come to the switch without having any tag. Such packets are new packets that come to a switch for the first time. In all of the three cases, we save the required statistics of the packet and add the F_{key} into the M_{FL} so that subsequent packet data can be aggregated with the existing data for that flow.

5.2 Evaluation Setup

We use the same experimental setup from the section 4.2 and evaluate the performance of cMon in USNET 4.3 and Darkstrand 4.4 topology. We measure accuracy and memory usages of cMon, Exact-Match [36] and FlowStat [18] to show its supremacy. We also measure the average throughput to show the impact of cMon in the overall network performance. For these measurements we only consider ICMP [78] and iPef [77] based UDP traffic.

To measure the memory usage we exchange ICMP traffic between all source-destination pair of USNET and Darkstrand topology. This kind of traffic exchange installs flow rules for all pair at every switch. It also enhances the compression ratio of flow rules due to having more common destinations. Given that, at cMon, we only compress rule based on the common destination address. For the FlowStat we make sure only one switch holds the exact-match rules along the path of the flow. During the measurement of memory usages, we count the total flow rules in all the switches.

In another experiment, we measure the accuracy of cMon, Exact-Match, and FlowStat with a variable amount of link failure. In this case, we consider the same 60 source-destination pair from section 4.2 and exchange ICMP packets between them. We fail at least one link between 12 concurrent pairs at a time to simulate the effect of multiple link failure. In this evaluation, we count the total number of packets for cMon, Exact-Match and FlowStat. We divide the total number of packets correctly recognized by each scheme with the total number of packets that we actually exchanged to get the accuracy of each of monitoring schemes.

Finally, we measure the impact of cMon on network performance. In this experiment, we exchange different sized data among the 60 source-destination pair. For each data size, we measure the average throughput by following the same process of section 4.2.

5.3 Performance evaluation

We present the evaluation results of cMon in this section. To evaluate the performance of cMon we compare it against FlowStat [18] and Exact-Match [17]. As we cannot separate flows in the wildcard based forwarding strategies, we omit their results from our comparison. In subsection 5.3.1, we present the memory usage results to show the memory requirement of FlowStat and Exact-Match to provide per-flow statistics. After that, in subsection 5.3.2 we present the accuracy of them. Finally, we present the network throughput results to show the impact of cMon on the overall network performance.

5.3.1 Memory Usage

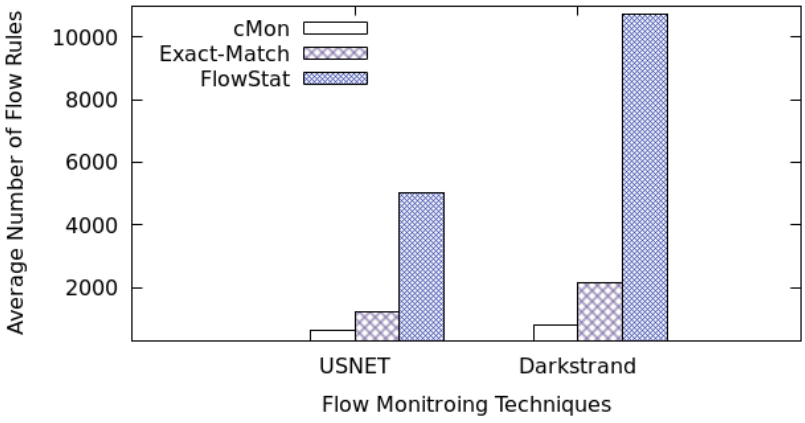


Figure 5.2: Impact On Memory Uses.

In Figure 5.2, we present the average memory requirement of cMon, FlowStat and Exact-Match. In cMon, we can fully use wildcard rules while gathering per-flow statistics. Whereas, FlowStat requires some exact match rule to gather per-flow statistics. Thus, cMon saves nearly 3 times more memory than FlowStat in Darkstrand topology. In USNET topology it saves almost double-flow rules compared to the FlowStat.

As Exact-Match installs flow rules for each flow, it consumes nearly 13 times more memory than the cMon in Darkstrand topology and 8 times more memory in USNET topology.

5.3.2 Impact of Link Failure on cMon Accuracy

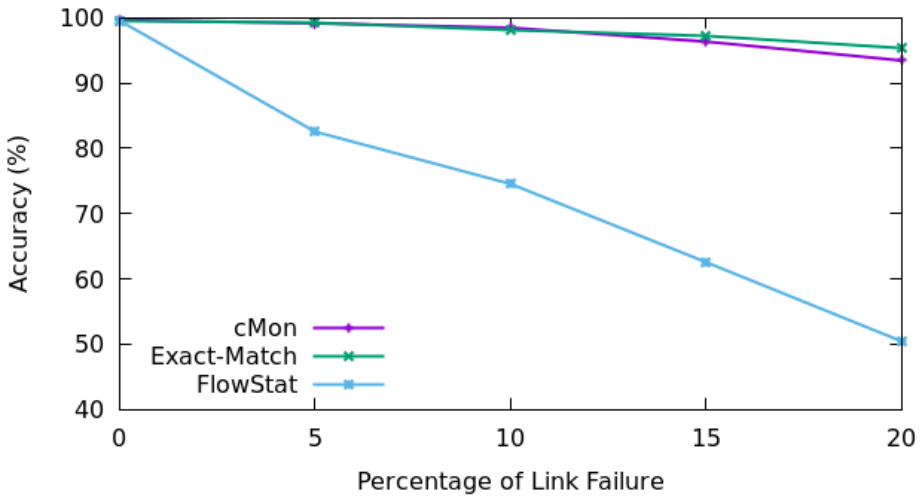


Figure 5.3: Impact of Link Failure on Per Flow Statistics.

In Figure 5.3, we present the results that highlight the impact of a link failure on the accuracy of cMon. When there is no link failure, cMon, Exact-Match, and FlowStat show almost 100% accuracy. But the accuracy of FlowStat rapidly falls near to the 50% when it experiences 20% link failure. Because it needs to go through a complex process to add exact-match rules. As Exact-Match has flow rules for each packet in the switch and cMon gathers almost every packet for monitoring, their performance remains almost consistent. The only performance drop they experience is mainly due to the packet drops during backup path restoration. Thus, we can conclude that cMon preserves network visibility even if the network suffers from link failure.

5.3.3 Impact on Network Performance

We measure the average throughput of the network to observe the overhead induced by cMon. Due to per-packet monitoring, cMon captures every packet and that has an impact on the performance of the overall network. Thus in Figure 5.4, we can see that cMon has degradation in its throughput than its counterparts.

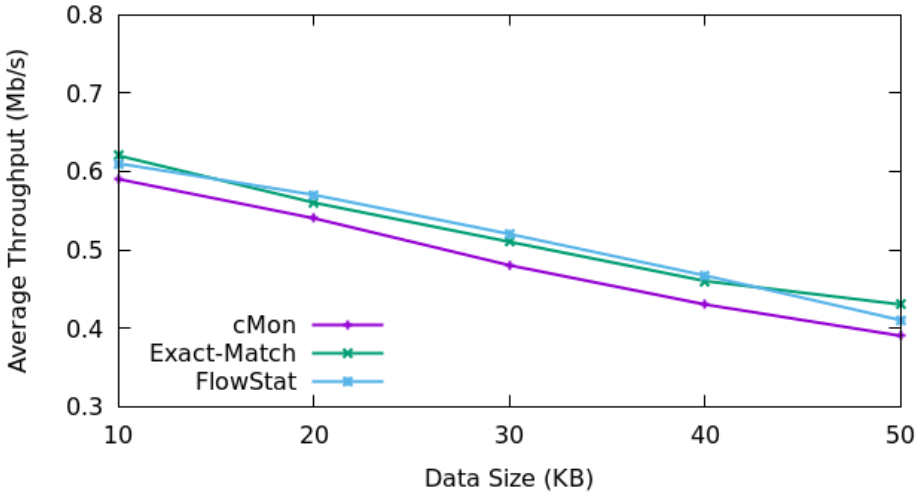


Figure 5.4: Impact of cMon on Network Throughput.

But due to high standard error we cannot conclude on the performance bottleneck of cMon.

Chapter 6

Distributed Priority Aware Load Balancing

In this chapter, we present our distributed load balancing strategy, called DPAL, which can effectively distribute the load to alternate routes while higher priority flow demands the primary route. Section 6.1 presents DPAL design and architecture. We present evaluation setup at section 6.2. In section 6.3, we present and discuss performance evaluation result of DPAL.

6.1 DPAL Design and Architecture

In this section, we present the DPAL design and architecture. The Figure 6.1 shows the DPAL architecture with major functional modules. The *route planner* and *applications* modules reside in the application layer and coordinate with the control plane modules through the northbound API. In the control plane, DPAL has *statistics collector*, *topology control*, and *rule configuration* modules. The controller modules coordinate with the data plane using OpenFlow 1.3 protocol. The major functional modules of DPAL resides in the data plane. Among them, *congestion monitor*, *rule grabber*,*per flow statistics*,*reroute flow finder* and *rule changer* aids in congestion detection and rule redistribution to overcome congestion.

6.1.1 Management and Control Plane Modules

DPAL controller gathers underlying data plane topology with the *topology control* module. Recall that, when there is a change in underlying topology, the SDN switch notifies to the controller. Using the topology information, *route planner* generates a graph $G(V, E)$. Where V is the set of switches and E is the set of links. From the graph, *route planner* computes the best primary and alternative paths for a given source and destination pair using spanning structures [27]. After that it notifies *rule configuration* module about the primary and alternative paths. The *rule configuration* module prepares OpenFlow compatible corresponding flow rules for those routes.

After that, it installs such rules in the data plane OVS switches using OpenFlow 1.3 protocol.

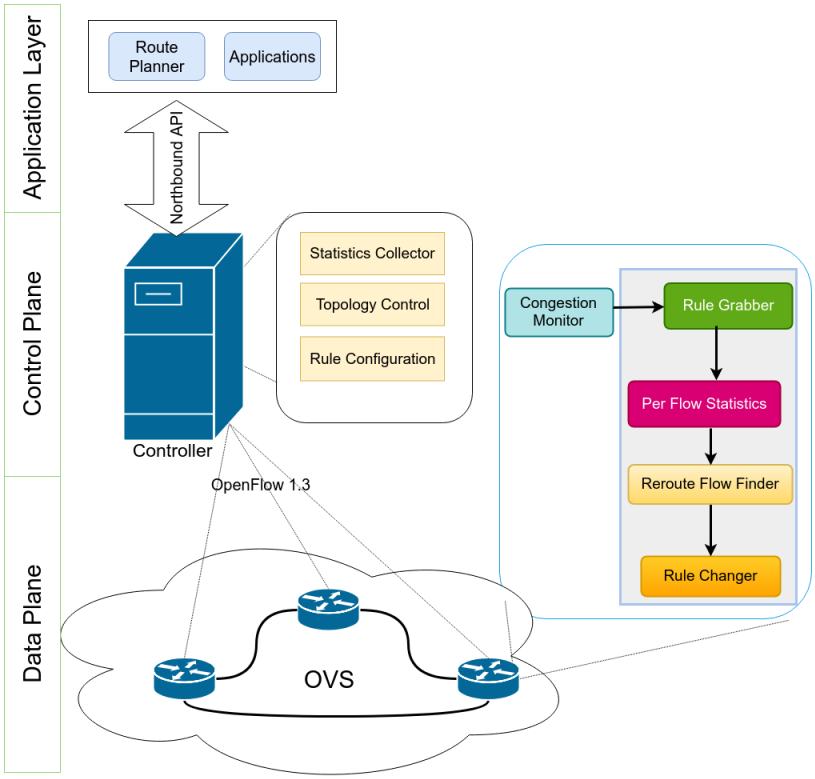


Figure 6.1: The Architecture of DPAL.

6.1.2 Data Plane Modules

The major functional modules of DPAL resides in the data plane. Initially, the *congestion monitor* module estimates the link capacity and periodically monitors the packet transmission rate at the port connected to that link. It uses link capacity information to agree on a transmission rate threshold. If the packet transmission rate at a port exceeds the threshold value then *congestion monitor* module calls the *rule grabber* module to pull the flow rules those deliver the packet to that congested link. Using these flow rules, the cMon *per flow statistics* module provides the per-flow transmission rate.

During the congestion period, DPAL considers the lowest priority flow for rerouting. It estimates the number of flow to be rerouted using *reroute flow finder*. The estimation approximates the number of flow removal by considering the transmission

rate of the flow. *reroute flow finder* grabs flow rule and checks whether reroute of that flow improves congestion of the port. If it does then it reroutes the flow to the alternative route. If it does not, it again grabs another flow and test for congestion improvement. It continues grabbing such flow until it finds congestion improvement. When a certain set of flow reroute improves the congestion state, *rule changer* changes the rules for those flow to reroute them in the alternate route. We present the overall operation of DPAL in the Flow Chart of Figure 6.2.

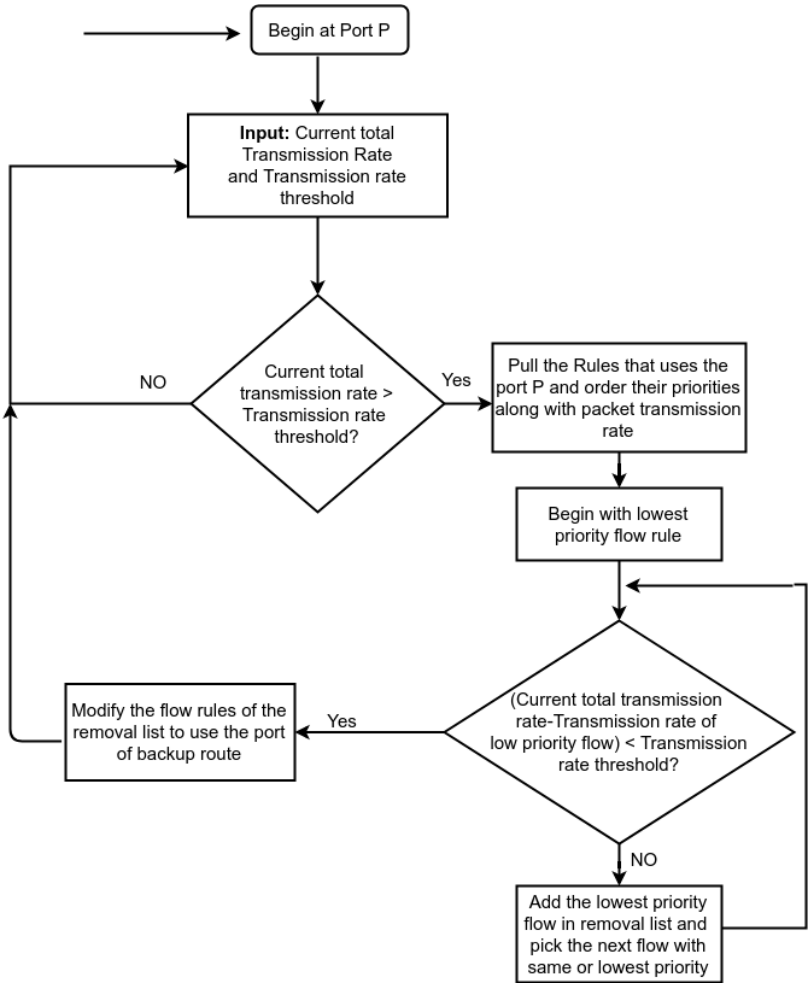


Figure 6.2: Flowchart Showing DPAL Operation.

6.1.3 Load Distribution Mechanism of DPAL

The algorithm 6.1.3 presents the load distribution operation of DPAL. The algorithm takes the transmission rate threshold, Tx_{thresh} and the current total transmission

rate on a port, Tx_r as input. Given that, initially DPAL *congestion monitor* module estimates Tx_{thresh} from the link capacity L_{cap}. When the total transmission rate Tx_r of a port exceeds the threshold value Tx_{thresh}, the algorithm grabs the currently installed flow rules priority P_L in ascending order.

Algorithm 4 DPAL Algorithm

Input:

Transmission Rate Threshold: Tx_{thresh} where, $Tx_{thresh} \in L_{cap}$

Current Total Transmission Rate: Tx_r

1: **if** $Tx_r > Tx_{thresh}$ **then**
2: $P_L \leftarrow GetSortedPriorityList()$
3: **for** $j \in P_L$ **do**
4: $F \leftarrow GetFlowRules(j)$
5: $F_{tx} \leftarrow GetTransmissionRate(F)$
6: $F_O \leftarrow SortFlowWithTxRate(F)$
7: $F_{stx} \leftarrow SortedTransmissionRate(F_{tx})$
8: $R_{FL} \leftarrow []$
9: $T_{xc} \leftarrow \emptyset$
10: **for** $k \in F_O$ **do**
11: $T_{xc} \leftarrow T_{xc} + F_{stx}[k]$
12: $R_{FL} = R_{FL} + k$
13: **if** $(Tx_r - T_{xc}) < Tx_{thresh}$ **then**
14: $Reroute(R_{FL})$
15: **end if**
16: **end for**
17: **end for**
18: **end if**

After that, the algorithm finds all flow rule F with the priority j, where $j \in P_L$. It also finds the transmission rate, F_{tx}, for those flow with priority j. Next, the algorithm sorts the flow rules in ascending order of transmission rate. According to the sorted flow rules order, F_O, it also creates a sorted flow transmission rate list F_{stx}.

Then, the algorithm starts approximation to find the reroute flows. To do so, it

at first begins with a flow rule k where $k \in F_O$. If it finds $Tx_r - T_{xc}$ has value greater than the Tx_{thresh}, then it adds up the transmission rate of the next flow with the T_{xc} and compares again $Tx_r - T_{xc}$ with Tx_{thresh}. This process continues until Tx_{thresh} value gets smaller than $Tx_r - T_{xc}$. As long as the Tx_{thresh} is larger, those packets whose transmission rate is considered for T_{xc} are stored in R_{FL} for rerouting. When the threshold value, Tx_{thresh}, drops then the rule changer module changes the rules stored in R_{FL}.

6.2 Evaluation Setup

In this section, we present and discuss the chosen emulation environment for the evaluation of DPAL. We use the same experimental environment of section 4.2. We consider asymmetric propagation delay for links to emulate congestion. While experimentation we inject both high priority and low priority flow. We consider both USNET (Figure 4.3) and Darkstrand (Figure 4.4) topology for the evaluation of DPAL.

During the experiment, we use 60 sources and destination pair from both of the topologies and transfer different sized real MP4 files among them with wget [79]. The chosen source and destination pair ensures that they have a longer primary route. We also randomly select twenty sources and destination pairs for higher priority data transfer.

We first evaluate the performance of DPAL in terms of average delay and average throughput for the higher priority flow. To do so, we maintain a list of source and destination pairs and mark their communication as higher priority data transfer. We exchange MP4 data among 60 source-destination pair and only measure the end-to-end transfer delay and throughput for the higher priority flows. Finally, we take the average to compute average delay and throughput.

To measure the average number of hop traversed by the higher priority flow, we count the path length from the moment of congestion control. We do this for all higher priority flows and later take the average.

To measure the overall throughput, we exchange iPerf [77] based UDP traffic between 60 sources and destination pair for a certain amount of time. During the

communication, we emulate congestion in the network by using asymmetric link bandwidth. We measure the throughput of each communicating pair at a periodic interval and take the average to get the average throughput at that time.

6.3 Performance Evaluation

In this section, we present the evaluation results of DPAL and compare its performance with Fast Switchover [13] and CLOVE [70]. Subsection 6.3.1 presents the average delay requirement for higher-priority flow. In subsection 6.3.2, we present average throughput results. We present the average number of hop traversed by higher priority flow in subsection 6.3.3. Finally, at subsection 6.3.4, we show the impact of DPAL on the overall network performance.

6.3.1 Average Delay for Higher Priority Flow

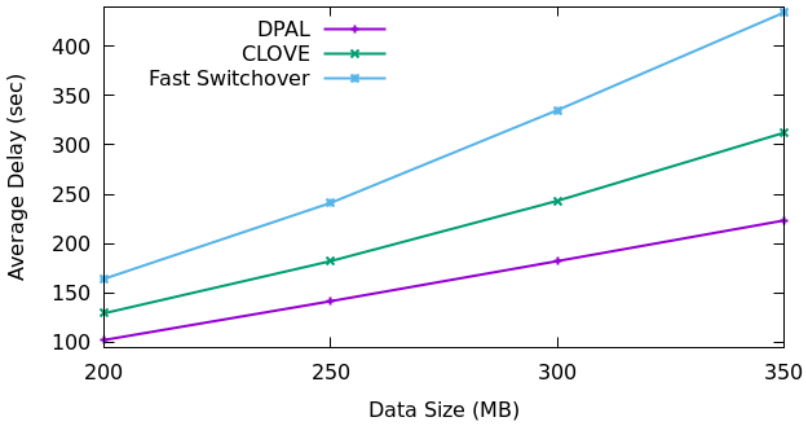


Figure 6.3: Average Delay For Higher Priority Flow in USNET Topology.

Figure 6.3 and 6.4 presents average delay results for the higher priority flow in both USNET and Darkstrand topology respectively. For the higher priority flow, DPAL experiences 49% less delay than the Fast Switchover and 28% less delay than CLOVE in USNET topology. In Fast Switchover, higher priority flow always uses the long route after congestion happens. Whereas in the CLOVE, route selection for higher priority flow happens in a round-robin fashion.

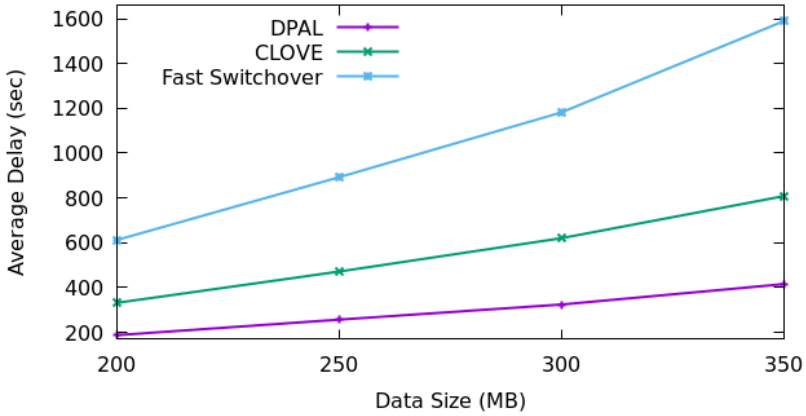


Figure 6.4: Average Delay For Higher Priority Flow in Darkstrand Topology.

In DPAL, higher priority flow always uses the shortest route. Thus, DPAL experiences less delay than CLOVE and Fast Switchover. Also, Fast Switchover Experiences more delay than CLOVE. As Darkstrand [80] has more nodes, it has source and destination pairs with longer routes. Thus, we further extended our experiment to observe the impact of Darkstrand topology on higher priority flow in a congestion scenario. Based on our evaluation result, DPAL shows nearly 74% delay improvement over Fast Switcher and 49% delay improvement over CLOVE in Darkstrand topology. Due to the same reason for the USNET evaluation, we achieve this performance gain in DPAL.

6.3.2 Average Throughput for Higher Priority Flow

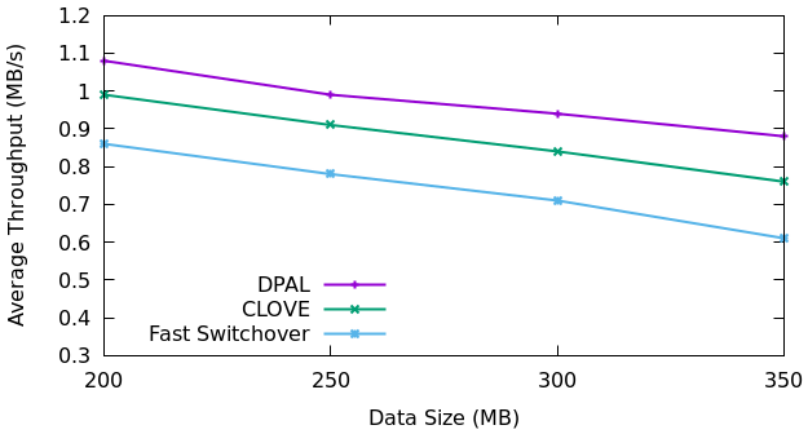


Figure 6.5: Average Throughput For Higher Priority Flow in USNET Topology.

We present the average throughput results for the higher priority flow in Figure 6.5 and 6.6 for USNET and Darkstrand topology, respectively. From the results, we can find that, in USNET topology, DPAL shows almost 31% better throughput than the Fast Switchover and that improvement reaches to almost 1.7 times in Darkstrand topology due to its longer route length than USNET topology. Recall that, higher priority flow always follows the shortest path in DPAL and Fast Switchover chooses the backup route for the higher priority flow. As CLOVE chooses the route for higher priority flow in a round-robin fashion, it shows nearly 38% and 20% degraded throughput than DPAL in Darkstrand and USNET topology respectively.

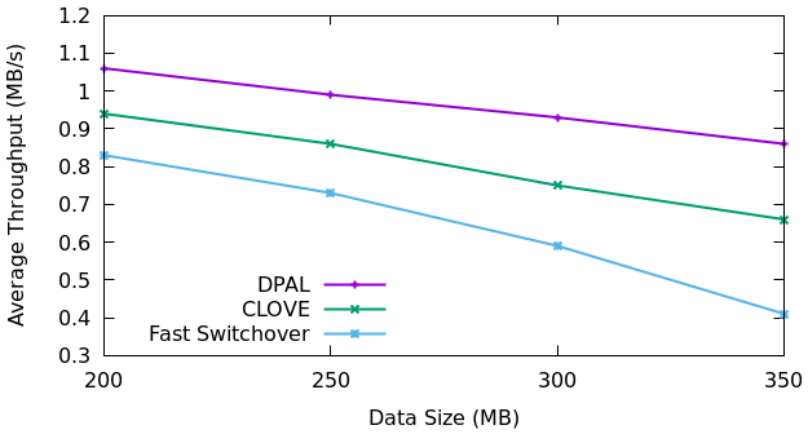


Figure 6.6: Average Throughput For Higher Priority Flow in Darkstrand Topology.

6.3.3 Average Hop Count for Higher Priority Flow

For DPAL, CLOVE and Fast Switcher, Figure 6.7 presents the average number of hop traversed by the higher priority flow in USNET and Darkstrand topology. As the source and destination in Darkstrand have longer routes than USNET topology, the average hop count in Darkstrand topology is longer. In Darkstrand topology, CLOVE experiences almost 1.5 times longer route than DPAL. Whereas it experiences a 25% longer route in USNET topology than DPAL. The results tell that, in the case of DPAL, higher priority flow moves through the primary path and CLOVE switches its path between primary and backup route. But compared to DPAL, Fast Switchover experiences nearly double the longer route in Darkstrand topology due to the usage of the backup path for the higher priority flow. Whereas in USNET it takes nearly

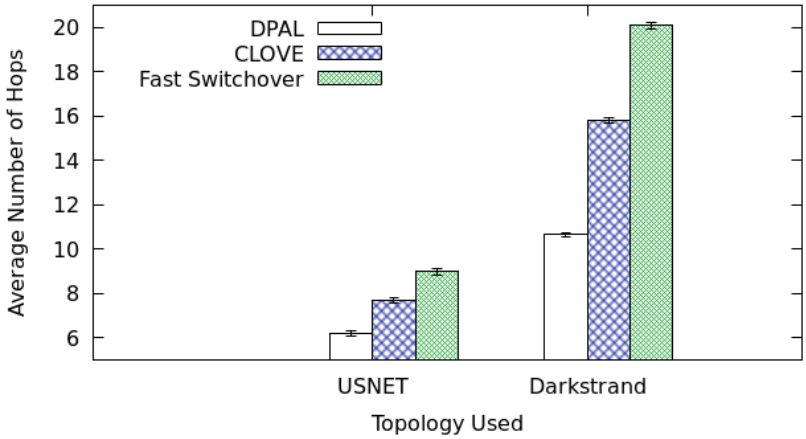


Figure 6.7: Average Hop Count For Higher Priority Flow.

46% longer route than DPAL due to the same reason.

6.3.4 Impact of DPAL on The Network Performance

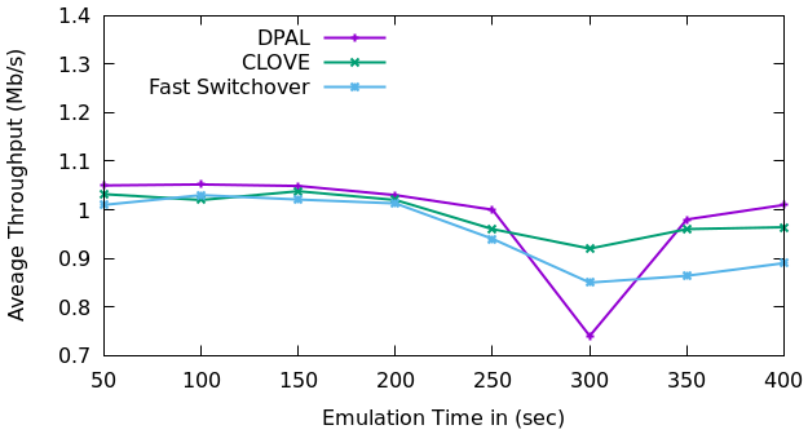


Figure 6.8: Average Throughput of the Network.

DPAL algorithm runs in the data plane switch and can impact the overall network performance. Thus, we run an experiment to see the overall network throughput to understand the impact of DPAL on the overall network performance. Figure 6.8 shows that, up to the congestion point (nearly 250 seconds emulation time as shown in the Figure 6.8) DPAL, CLOVE and Fast Switchover shows similar throughput because of following the similar route and routing mechanism. But when congestion happens, DPAL requires redistribution of low priority flow. Thus, it shows a sudden throughput degradation. Whereas, Fast Switchover incorporates controller decision

and requires a long time to control the congestion state. Thus, it shows degraded throughput beyond the congestion point. CLOVE, on the other hand, balances the load in a round-robin fashion. Thus, its performance does not degrade much. From the results of Figure 6.8, we can say that DPAL does not add much overhead in the overall network.

Chapter 7

Conclusions and Future Work

This thesis presents a local failure recovery mechanism called, SD-FAST, a local monitoring algorithm called cMon and a local load balancing mechanism called, DPAL. In this chapter, we will conclude this thesis in section 7.1 and provide some future research directions in section 7.2.

7.1 Conclusions

In this thesis, we have introduced a local failure recovery technique called SD-FAST. Not only it helps the switch to reroute the failure affected traffic in the edge-disjoint route, but also it can reroute the affected traffic in the crankback path. Moreover, SD-FAST can terminate crankback backtracking locally. That's why our experimental results show the 64% average end-to-end delay improvement of SD-FAST over CB approach and 73% average backtracking convergence delay improvement over the CBC approach. Moreover, SD-FAST does not affect regular traffic for providing failure recovery. Thus, the experimental results show the impact of pipeline processing on the performance of FFG, CB, and CBC.

After offering local failure recovery, we have found that the proactive installation consumes more memory and rule compression improves such an issue. But rule compression reduces the visibility of the network. Thus, we proposed another local algorithm called cMon, which can monitor the network even if the flow rules are compressed. The only competitive work of cMon that offers per-flow statistics with low memory overhead is FlowStat. As it also stores few exact-match rules in the flow table, in our evaluation results we have found the FlowStat still consumes 3 times more memory than cMon.

Finally, we have figured out that congestion impacts higher priority flow. None of the existing literature guarantees the shortest path for a higher-priority flow. Rather

than that, they reroute the higher priority flow in the alternate route to avoid congestion. Thus, we introduced DPAL to ensure the shortest path for a higher-priority flow. Our DPAL algorithm can effectively select the proper low priority flow for reroute to leave the primary route for higher-priority flow. The evaluation results of DPAL shows the 49% average end-to-end delay improvement of higher priority flow over its counterparts.

7.2 Future Work

In our future work, we plan to evaluate the performance of SD-FAST, cMon, and DPAL in real hardware switch. Also, we plan to implement all these approaches as the core part of the OVS switch. Our proposed algorithms currently do not consider VPN connections and encrypted packet. Thus, in our future work, we want to address the encrypted packet and encrypted connection issue. During our experiment, we have found a few packet loss in SD-FAST during failure recovery. While implementation in OVS we want to investigate and fix this issue. Moreover, we want to evaluate our proposed approaches with other real-world topology considering different types of traffic.

Bibliography

[1] M. Moyeen, F. Tang, D. Saha, and I. Haque, "Sd-fast: A packet rerouting architecture in sdn," in *Accepted in International Conference on Network and Service Management (CNSM 2019)*. IEEE, 2019.

[2] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking control of the enterprise," *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 4, pp. 1–12, Aug. 2007.

[3] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. Gude, N. McKeown, and S. Shenker, "Rethinking enterprise network control," *IEEE/ACM Transactions on Networking (TON)*, vol. 17, no. 4, pp. 1270–1283, August 2009.

[4] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[5] S. Jain *et al.*, "B4: Experience with a globally-deployed software defined wan," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 3–14, Aug. 2013.

[6] M. Darianian, C. Williamson, and I. Haque, "Experimental evaluation of two openflow controllers," in *T2017 IEEE 25th International Conference on Network Protocols (ICNP)*, October 2017.

[7] I. Haque and N. Abu-Ghazaleh, "Wireless software defined networking: a survey and taxonomy," *IEEE Communications Surveys and Tutorials*, 2016, (forthcoming).

[8] V. Kolar, I. Haque, V. P. Munishwar, and N. B. Abu-Ghazaleh, "CTCV: A protocol for coordinated transport of correlated video in smart camera networks," in *ICNP*. IEEE Computer Society, 2016, pp. 1–10.

[9] P. Fonseca and E. Mota, "A survey on fault management in software-defined networks," *IEEE Communications Surveys and Tutorials*, vol. 19, no. 4, pp. 2284–2321, 2017.

[10] I. F. Akyildiz, A. Lee, P. Wang, M. Luo, and W. Chou, "Research challenges for traffic engineering in software defined networks," *IEEE Network*, vol. 30, no. 3, pp. 52–58, June 2016.

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

[12] A. Ghannami and C. Shao, "Efficient fast recovery mechanism in software-defined networks: multipath routing approach," in *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)*. IEEE, 2016, pp. 432–435.

[13] 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 *ICC*. IEEE, 2016, pp. 1–6.

[14] "Bidirectional Forwarding Detection (BFD) PRotocol." [Online]. Available: https://tools.ietf.org/html/rfc5880 (Date last accessed on 22-July-2019).

[15] S. M. Salam, A. Sajassi, and S. Henderson, "Connectivity fault management (cfm) auto-provisioning using virtual private lan service (vpls) auto-discovery," Feb. 26 2013, uS Patent 8,385,353.

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

[17] F. Tang and I. Haque, "Remon: A resilient flow monitoring framework," in *2019 Network Traffic Measurement and Analysis Conference (TMA)*. IEEE, 2019.

[18] S. Bera, S. Misra, and A. Jamalipour, "Flowstat: Adaptive flow-rule placement for per-flow statistics in sdn," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 3, pp. 530–539, 2019.

[19] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, G. Varghese *et al.*, "Conga: Distributed congestion-aware load balancing for datacenters," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 503–514.

[20] K.-T. Foerster, Y.-A. Pignolet, S. Schmid, and G. Tredan, "Local fast failover routing with low stretch," *ACM SIGCOMM Computer Communication Review*, vol. 48, no. 1, pp. 35–41, 2018.

[21] "Mininet." [Online]. Available: http://mininet.org (Date last accessed on 22-July-2019).

[22] "Ryu Controller." [Online]. Available: http://osrg.github.com/ryu/ (Date last accessed on 22-July-2019).

[23] "Open vSwitch." [Online]. Available: http://openvswitch.org/ (Date last accessed on 22-July-2019).

[24] S. Sezer, S. Scott-Hayward, P. K. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, and N. Rao, "Are we ready for sdn? implementation challenges for software-defined networks," *IEEE Communications Magazine*, vol. 51, no. 7, pp. 36–43, 2013.

[25] O. Salman, I. H. Elhajj, A. Kayssi, and A. Chehab, "Sdn controllers: A comparative study," in *2016 18th Mediterranean Electrotechnical Conference (MELECON)*. IEEE, 2016, pp. 1–6.

[26] M. Karakus and A. Durresi, "Economic analysis of software defined networking (sdn) under various network failure scenarios," in *ICC 2019-2019 IEEE International Conference on Communications (ICC)*. IEEE, 2019, pp. 1–6.

[27] I. Haque and M. Moyeen, "Revive: A reliable software defined data plane failure recovery scheme," in *2018 14th International Conference on Network and Service Management (CNSM)*. IEEE, 2018, pp. 268–274.

[28] M. Rifai, N. Huin, C. Caillouet, F. Giroire, D. Lopez-Pacheco, J. Moulierac, and G. Urvoy-Keller, "Too many sdn rules? compress them with minnie," in *2015 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2015, pp. 1–7.

[29] S. Shirali-Shahreza and Y. Ganjali, "Rewiflow: Restricted wildcard openflow rules," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 5, pp. 29–35, 2015.

[30] V. Muthumanikandan and C. Valliyammai, "Link failure recovery using shortest path fast rerouting technique in sdn," *Wireless Personal Communications*, vol. 97, no. 2, pp. 2475–2495, 2017.

[31] 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)*. IEEE, 2011, pp. 164–171.

[32] D. Staessens, S. Sharma, D. Colle, M. Pickavet, and P. Demeester, "Software defined networking: Meeting carrier grade requirements," in *18th IEEE Workshop on Local and Metropolitan Area Networks (LANMAN)*. IEEE, 2011.

[33] Z. Cheng, X. Zhang, Y. Li, S. Yu, R. Lin, and L. He, "Congestion-aware local reroute for fast failure recovery in software-defined networks," *IEEE/OSA Journal of Optical Communications and Networking*, vol. 9, no. 11, pp. 934–944, 2017.

[34] M. Kuźniar, P. Perešíni, N. Vasić, M. Canini, and D. Kostić, "Automatic failure recovery for software-defined networks," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013, pp. 159–160.

[35] A. Xie, X. Wang, G. Maier, and S. Lu, "Designing a disaster-resilient network with software defined networking," *arXiv preprint arXiv:1602.06686*, 2016.

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

[37] A. Sgambelluri, A. Giorgetti, F. Cugini, F. Paolucci, and P. Castoldi, "Openflow-based segment protection in ethernet networks," *Journal of Optical Communications and Networking*, vol. 5, no. 9, pp. 1066–1075, 2013.

[38] A. Sgambelluri *et al.*, "Effective flow protection in openflow rings," in *National Fiber Optic Engineers Conference*. Optical Society of America, 2013, pp. JTh2A–01.

[39] Y.-Z. Liao and S.-C. Tsai, "Fast failover with hierarchical disjoint paths in sdn," in *2018 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2018, pp. 1–7.

[40] D. Merling, W. Braun, and M. Menth, "Efficient data plane protection for sdn," in *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*. IEEE, 2018, pp. 10–18.

[41] W. Braun and M. Menth, "Scalable resilience for software-defined networking using loop-free alternates with loop detection," in *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2015, pp. 1–6.

[42] Z. Zhu, Q. Li, M. Xu, Z. Song, and S. Xia, "A customized and cost-efficient backup scheme in software-defined networks," in *Network Protocols (ICNP), 2017 IEEE 25th International Conference on*. IEEE, 2017, pp. 1–6.

[43] 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*. ACM, 2016, p. 9.

[44] 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*. ACM, 2013, p. 26.

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

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

[47] H. Liaoruo, S. Qingguo, and S. Wenjuan, "A source routing based link protection method for link failure in sdn," in *Computer and Communications (ICCC), 2016 2nd IEEE International Conference on*, 2016, pp. 2588–2594.

[48] T. Holterbach, S. Vissicchio, A. Dainotti, and L. Vanbever, "Swift: Predictive fast reroute," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 460–473.

[49] T. Holterbach, E. C. Molero, M. Apostolaki, A. Dainotti, S. Vissicchio, and L. Vanbever, "Blink: Fast connectivity recovery entirely in the data plane," in *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, 2019, pp. 161–176.

[50] "Barefoot tofino." [Online]. Available: https://www.barefootnetworks.com/ (Date last accessed on 22-July-2019).

[51] N. L. Van Adrichem, C. Doerr, and F. A. Kuipers, "Opennetmon: Network monitoring in openflow software-defined networks," in *2014 IEEE Network Operations and Management Symposium (NOMS)*. IEEE, 2014, pp. 1–8.

[52] N. Grover, N. Agarwal, and K. Kataoka, "liteflow: Lightweight and distributed flow monitoring platform for sdn," in *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2015, pp. 1–9.

[53] P. Tammana, R. Agarwal, and M. Lee, "Distributed network monitoring and debugging with switchpointer," in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 2018, pp. 453–456.

[54] P. Tammana, R. Agarwal, and M. Lee, "Simplifying datacenter network debugging with pathdump," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 233–248.

[55] A. Tootoonchian, M. Ghobadi, and Y. Ganjali, "Opentm: traffic matrix estimator for openflow networks," in *International Conference on Passive and Active Network Measurement*. Springer, 2010, pp. 201–210.

[56] C. Yu, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and H. V. Madhyastha, "Flowsense: Monitoring network utilization with zero measurement cost," in *International Conference on Passive and Active Network Measurement*. Springer, 2013, pp. 31–41.

[57] B. Claise, "Cisco systems netflow services export version 9," 2004.

[58] J. Suh, T. T. Kwon, C. Dixon, W. Felter, and J. Carter, "Opensample: A low-latency, sampling-based measurement platform for commodity sdn," in *2014 IEEE 34th International Conference on Distributed Computing Systems*. IEEE, 2014, pp. 228–237.

[59] Z. Su, T. Wang, Y. Xia, and M. Hamdi, "Flowcover: Low-cost flow monitoring scheme in software defined networks," in *2014 IEEE Global Communications Conference*. IEEE, 2014, pp. 1956–1961.

[60] Y. Li, R. Miao, C. Kim, and M. Yu, "Flowradar: A better netflow for data centers," in *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, 2016, pp. 311–324.

[61] Y.-J. Chen, Y.-H. Shen, and L.-C. Wang, "Traffic-aware load balancing for m2m networks using sdn," in *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*. IEEE, 2014, pp. 668–671.

[62] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, A. Vahdat *et al.*, "Hedera: dynamic flow scheduling for data center networks." in *Nsdi*, vol. 10, no. 2010, 2010.

[63] S. Wang, J. Zhang, T. Huang, T. Pan, J. Liu, and Y. Liu, "Fdalb: Flow distribution aware load balancing for datacenter networks," in *2016 IEEE/ACM 24th International Symposium on Quality of Service (IWQoS)*. IEEE, 2016, pp. 1–2.

[64] C. E. Hopps and D. Thaler, "Multipath issues in unicast and multicast next-hop selection," 2000.

[65] T. Benson, A. Anand, A. Akella, and M. Zhang, "Microte: Fine grained traffic engineering for data centers," in *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*. ACM, 2011, p. 8.

[66] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A centralized zero-queue datacenter network," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 307–318.

[67] F. Fan, B. Hu, and K. L. Yeung, "Routing in black box: Modularized load balancing for multipath data center networks," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 1639–1647.

[68] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, G. Varghese *et al.*, "Conga: Distributed congestion-aware load balancing for datacenters," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 503–514.

[69] M. Mahalingam, "Vxlan: A framework for overlaying virtualized layer 2 networks over layer 3 networks," *draft-mahalingam-dutt-dcopSVXLAN-00. txt*, 2011.

[70] N. Katta, M. Hira, A. Ghag, C. Kim, I. Keslassy, and J. Rexford, "Clove: How i learned to stop worrying about the core and love the edge," in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*. ACM, 2016, pp. 155–161.

[71] J. Hwang, J. Yoo, S.-H. Lee, and H.-W. Jin, "Scalable congestion control protocol based on sdn in data center networks," in *2015 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2015, pp. 1–6.

[72] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "Hula: Scalable load balancing using programmable data planes," in *Proceedings of the Symposium on SDN Research*. ACM, 2016, p. 10.

[73] A. Kabbani, B. Vamanan, J. Hasan, and F. Duchene, "Flowbender: Flow-level adaptive routing for improved latency and throughput in datacenter networks," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. ACM, 2014, pp. 149–160.

[74] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz, "Detail: reducing the flow completion time tail in datacenter networks," in *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. ACM, 2012, pp. 139–150.

[75] A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella, "On the impact of packet spraying in data center networks," in *2013 Proceedings IEEE INFOCOM*. IEEE, 2013, pp. 2130–2138.

[76] N. G. Duffield, F. L. Presti, V. Paxson, and D. Towsley, "Inferring link loss using striped unicast probes," in *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*, vol. 2. IEEE, 2001, pp. 915–923.

[77] "iPerf." [Online]. Available: https://iperf.fr/iperf-doc.php (Date last accessed on 22-July-2019).

[78] T. Chin, M. Rahouti, and K. Xiong, "End-to-end delay minimization approaches using software-defined networking," in *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*. ACM, 2017, pp. 184–189.

[79] "wget." [Online]. Available: https://www.gnu.org/software/wget/ (Date last accessed on 22-July-2019).

[80] "Darkstrand." [Online]. Available: http://www.topology-zoo.org/dataset.html (Date last accessed on 22-July-2019).