A LOW-COST AND RESILIENT FLOW MONITORING FRAMEWORK IN SDN

by

Fangye Tang

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

at

Dalhousie University
Halifax, Nova Scotia
July 2019

© Copyright by Fangye Tang, 2019

Table of Contents

List of Tables

List of Figures

Abstract

Network measurement and monitoring are essential for learning the current network state and act accordingly. Moreover, Software-Defined Networking (SDN) makes the measurement and monitoring more accessible and flexible. However, existing measurement schemes in SDN suffer from high measurement cost due to a fix sampling rate while monitoring all the data plane elements. In addition, existing monitoring schemes are not resilient in the presence of communication link and node failures. Therefore, in this thesis, we propose a low-cost and resilient flow monitoring framework in SDN.

We first propose a low-cost measurement algorithm, which reduces the measurement cost by aggregating flows at a subset of switches. Next, we define a model to optimization the measurement cost and accuracy. Furthermore, we observe that link and node failures can impact measurement accuracy. Thus, we propose a resilient monitoring framework called ReMon. In particular, we propose three algorithms to recover from node and link failures, which are implemented in the SDN controller. Then, we update the measurement scheme after recovering from a failure. The experimental results show that the proposed solutions outperform their counterparts in terms of measurement and computation cost, accuracy, recovery time, and memory usage.

Acknowledgements

I want to thank my supervisor, Dr. Israat Haque, for her support to complete this thesis. I feel lucky to have the chance to work on research for the first time. Next, I would like to thank my parents for supporting me for my graduate studies. I could not complete this work without their support. Finally, I would like to thank my lab partners (Meysam, Dipon, and Moyeen) for constructive discussions on various research issues.

Chapter 1

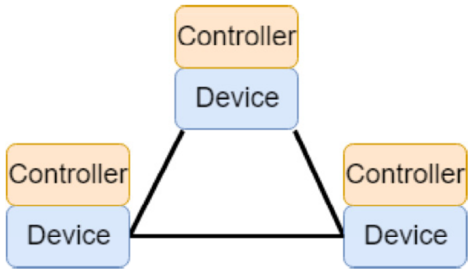
Introduction

Network measurement and monitoring are essential for efficient traffic engineering and Quality of Service (QoS). The purpose of it is to monitor network usage and performance and check for slow or failing systems. The system then notifies the network administrators of any performance issues or outages. Thus, a resilient, accurate, and low-cost monitoring framework is essential. This thesis presents the design and implementation of a flow-based monitoring framework in Software-Defined Network (SDN).

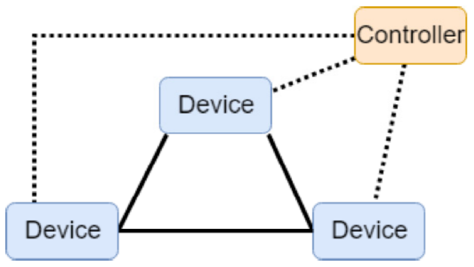
1.1 Background and Motivation

In traditional network architecture, the control plane (that decides how to handle network traffic) and the data plane (that forwards traffic according to the rules that control plane assigned) are equipped inside the network devices (routers and switches), which is shown in Figure 1.1(a). It makes the network configuration complex as the network administrators have to manually configure individual network devices using different specific commands. Furthermore, it reduces flexibility because of such lack of automatic reconfiguration.

Software-Defined Network (SDN) [11] is a new networking approach that physically separates the network control plane from the data plane, and a programmable controller can control several devices, which is shown in Figure 1.1(b). The controller can use a protocol like OpenFlow [22] to contact the data plane elements. Network administrators can easily configure switches by running predefined script in the controller, and it will automatically apply to each switch. One of the main advantages of SDN is its flow-based packet forwarding. The flow is traffic traveling from a source to a destination, and each flow has a corresponding flow rule, which is installed in a flow table of a switch. Besides, each flow has its counters. Thus, the network measurement is more convenient because the network administrators need to get the statistics of the counters from the monitored data plane switches.



(a) Traditional network architecture



(b) SDN architecture

Figure 1.1: The architecture of traditional network and SDN.

Typically, there are two types of network measurement in SDN: *active* and *passive* [33]. The active measurement injects a probe packet in the network. This probe packet will traverse through a flow and learn the necessary network state of this flow. On the other hand, the passive measurement directly read the counters based on a sampling rate. The passive measurement does not insert any probe packets in the network but requires full access and control on the devices. The active measurement enables demand-based statistics gathering, where the accuracy depends on the probing frequency. The higher the rate, the better the accuracy at the price of measurement cost.

A hybrid measurement combines both the active and passive measurements. It can either send probe packets with a fixed polling frequency or read the counters with a fixed sampling rate. For instance, in the case of the measurement tool sFlow [8], a monitoring agent is installed at each switch to send the measured traffic to the monitoring controller. Thus, the controller or collector can poll statistics at a regular interval, or an agent can push the statistics after observing a configured number of flows.

Once we decide on the measurement scheme, we can measure a flow more than once at multiple switches or once at a single switch. Sampling a single flow multiple times may consume extra resources of the monitored switches. Furthermore, recent work states that it is enough to measure a flow once at a single switch to achieve the required accuracy [28]. However, a single switch can carry more than a single flow, in such case, it is beneficial to measure the aggregated flows from a subset of switches instead of monitoring every switch from a monitored network. Thus, it is essential to decide on which switches to monitor.

FlowCover [28] proposes a flow statistics aggregation heuristic to minimize the measurement cost without degrading the measurement accuracy. It assigns a weight to each switch based on the number of installed flows. Then, FlowCover selects switches with the highest weight until all the flows are covered. Thus, FlowCover reduces the measurement cost by reducing the number of monitoring switches. However, our initial experimental results reveal that the time complexity of FlowCover is high, which can impact the measurement scalability. We furthermore observe that the location of the chosen switch along the route of a monitored flow has an impact on the measured accuracy (e.g., throughput). This critical measurement criterion is missing in FlowCover. Thus, we need a flow monitoring algorithm that offers high measurement accuracy while reduces both the measurement cost and computation time.

The polling or sampling frequency also has an impact on the measurement accuracy. For active measurement, the higher the polling frequency, the better the accuracy at the price of measurement cost. On the other hand, the higher sampling rate in the passive measurement consumes more resources of the monitored switches. Thus, it is important to choose an optimal polling or sampling rate. However, existing works mostly rely on a fixed rate and impact both the measurement accuracy and network health. There are a few exceptions, where the authors define an on-demand polling/sampling rate. For instance, Payless [13] proposes a Markov process based algorithm uses two consecutive sampling rates to predict the subsequent rate. Sampling-On-Demand (SOD) [15] designs a model to decide which switches to be sampled at what rate to maximize the measurement accuracy. However, our initial experiment shows that while using Payless, the measurement accuracy drops when utilization changes. On the other hand, the measurement cost of SOD is high as it does not consider the measurement cost while optimizing the accuracy based on the chosen sampling rate. Thus, we demand a sampling approach that can dynamically adjust its sampling rate to offer high accuracy and low cost irrespective of the utilization changes.

Furthermore, we observe the fact that link and node failures can impact the measurement accuracy while we monitor a subset of switches. In particular, after recovering from a failure, the affected flows change their route; thus, the chosen set of switches may not be the right set to cover the entire flow set. Therefore, it is not only essential to recover from failure as soon as possible, but also necessary to update the set of monitoring switches. However, no measurement and monitoring scheme addresses this issue.

There are two types of failure recovery approaches: *reactive* and *protective* [16] while using SDN. In a reactive scheme, a switch contacts the controller once detecting link failure. The controller calculates a new route and installs new flows at the affected switches. In the protective scheme, backup routes are installed to switches before a failure occurs. Thus, switches can locally detect a failure and redirect the affected traffic to the backup route to reduce the failure recovery time because it does not need to communicate the controller. Fast Failure Group (FFG) [36] is a protective failure recovery scheme available in the OpenFlow.

However, FFG requires a backup route from the affected switch to a destination. There are network topologies that do not have such property. Thus, we can use Crankback approach [35]. In Crankback approach, affected packets backtrack through the primary route until reach a switch that has a backup path. All the affected packets follow this packet-by-packet backtracking, which introduces significant delay. Thus, we need a failure recovery scheme that can quickly recover from a failure in any network topologies as long as the topology is connected. In addition, we need to update the set of flow monitoring switches after recovering from the failure.

1.2 Objectives

The primary objectives of this thesis are as follows based on the above research gaps:

- Design and implement a new flow monitoring algorithm to reduce the measurement and computation cost while offering high measurement accuracy. Compare and contrast the proposed algorithm with its counterparts FlowCover and the baseline approach. In the baseline approach, we need to monitor all the available switches.

- Design and implement a model to optimize the polling frequency to achieve high accuracy while reducing the measurement cost. Then, compare its performance with existing solutions.
- Design and implement a resilient flow monitoring framework in the presence of link and node failures. Compare the performance of the proposed framework with its counterparts.
- Implement proposed algorithms and frameworks in Mininet [4] emulator using two real topologies. Evaluate the optimization model using IBM CPLEX optimizer [1].

1.3 Contributions

The primary contributions of this thesis are listed below:

- We propose a new flow monitoring algorithm, called *Weighted Assisted Selecting (WAS)*, which reduces 30% measurement cost compared to the baseline approach. It furthermore reduces more than 90% computation cost compared to FlowCover and maintains the same level of measurement cost.
- We propose a new optimization model that balances the measurement accuracy and cost. The model evaluation results show that our approach saves more than 50% cost compared to SOD and maintains the same level of accuracy. Next, we implement a heuristic in Mininet, and the results show that our approach has higher accuracy compared to Payless and saves 50% and 45% cost compared to SOD and Payless, respectively.
- We design two new algorithms: *Anchor Assisted Recovery (AAR)* and *Weight Assisted Recovery (WAR)* to recover from multiple link failures. The experimental results show that AAR and WAR reduce more than 40% and 50% of recovery time compared to Crankback and restoration approaches, respectively. Furthermore, WAR and the proposed node failure algorithm *Node Recovery with Destination (NRD))* reduce memory usage at the switches compared to Crankback. Finally, we update the flow monitoring algorithm, WAS, after recovering from failures to retain the measurement accuracy.

1.4 Thesis Organization

The remaining of the thesis is organized as follows. Chapter 2 first presents the necessary background to understand the proposed work. Then, we discuss on some prior work of measurement and failure recovery in SDN. We illustrate WAS and optimization model in Chapter 3. In the same chapter, we furthermore present the evaluation results. In Chapter 4, we provide different link and node failure recovery algorithms and corresponding evaluation results. At last, we give a concluding remark and the future research plan in Chapter 5.

Chapter 2

Background and Related Work

In this chapter, we present the necessary background to understand the proposed design and prior work. Section 2.1 provides the background on Software-Defined Network including architecture and key concepts. Section 2.2 reviews different types of network measurements. For each type, we briefly present several prior works. In section 2.4, we first introduce sFlow in details and analyze its drawbacks. After that, we present several prior works that consider adjusting the polling frequency/sampling rate. Section 2.5 presents two types of failure recovery schemes in SDN. For each type, we also present several prior works and analyze their drawbacks.

2.1 Software-Defined Network (SDN)

In a traditional network, it comprises some end-hosts that are connected to each other through a set of devices. The network devices could be switches, routers, or firewalls. Mostly, there are several different manufacturers, for example Cisco and Dell. Each manufacturer has a complex and proprietary operating system. The variety of such operating systems in network devices adds extra work to configure devices. Besides, the multi-vendor environments require network administrators with a high level of expertise. Hence, the network administrators need to have a wide knowledge of different configuration interfaces, features, and commands on multi-vendor devices. Furthermore, if we want to apply any change in the network, the network administrators have to log in to each device and modify the configuration using different commands. In a large network, this procedure creates a significant overhead and reduce flexibility [17]. As a result, the new network architecture should be easy to operate and flexible. Figure 1.1(a) presents an architecture of a traditional network.

Software-Defined Network (SDN) [11] is a new networking approach that enables the network to be intelligently and centrally controlled. SDN decouples the control plane from the data plane. A control plane has a global view of the network and it controls multiple data-plane elements (router or switch). Besides, the controller in SDN is programmable. Thus, once we decide the controller, network administrators only need the knowledge of this controller. The controller will decide the operation of each traffic and install a rule to data plane elements. The data plane elements are several network devices. They simply forward the traffic based on the decision of the controller [19]. In other words, the controller works as a "brain" of the network. "Brain" will decide the action and each device just follow the instruction, which is like a "muscle memory". Figure 1.1(b) presents the architecture of SDN.

2.1.1 SDN architecture

Figure 2.1 presents the architecture of SDN in details. Typically, SDN has three layers: *Infrastructure Layer*, *Control Layer*, and *Application Layer*. *Infrastructure Layer* consists of several network devices and end-hosts that are programmable. The middle layer is where the controller is placed, called *Control Layer*. The communication between devices and controller(s) occurs via an open interface named *Southbound API*. The top layer consists of several network application, called *Application Layer*. The communication between these application and SDN controllers occurs via another open interface, called *Northbound API*.

Infrastructure Layer

Infrastructure Layer hosts all network devices. A network device could be a physical/virtual switch, a router or a firewall. In the Software-Defined Network, such devices are just simple forwarding device and follow the instructions of the controller. Each device should support the same Southbound API so that the controller can simply run one script and apply to each device [22].

Southbound API

Southbound API is one of the most important components in SDN. It establishes a connection so that the controller(s) and devices can communicate with each other.

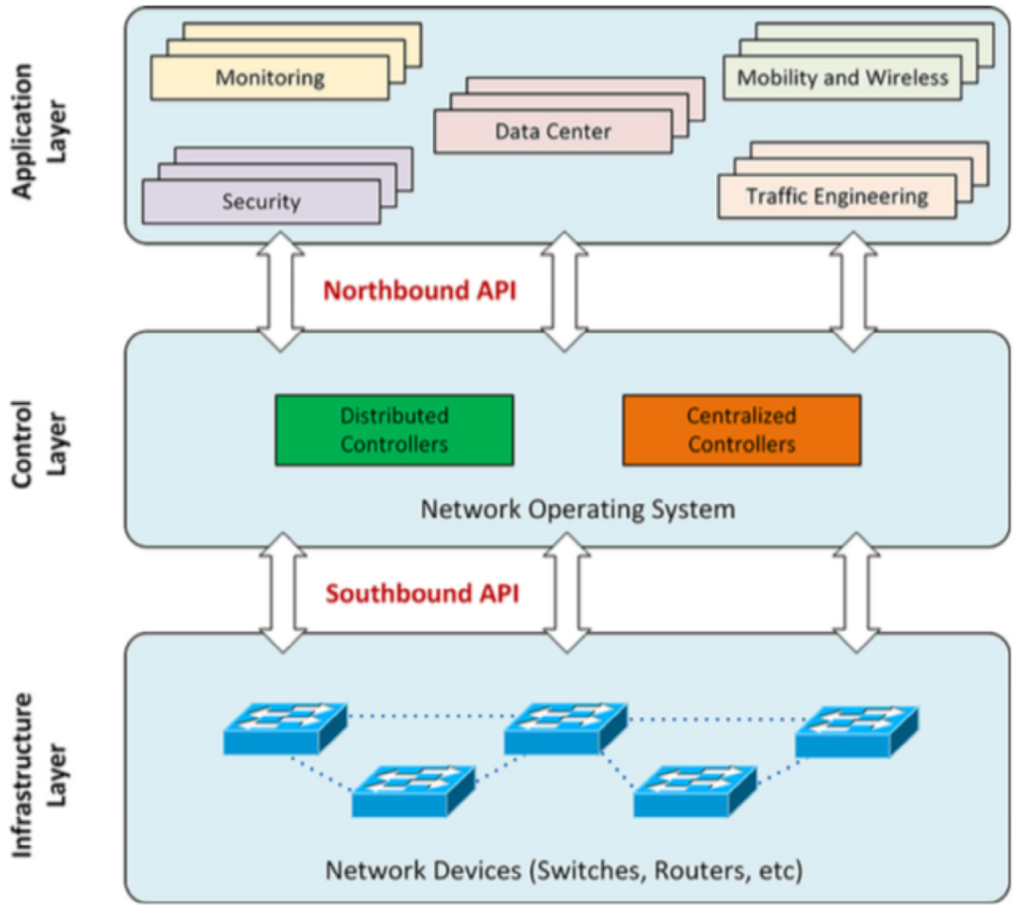


Figure 2.1: SDN architecture.

Typically, a Southbound API is a protocol. One mostly accepted and deployed Southbound API is OpenFlow [25]. To date, almost all of SDN controllers support OpenFlow and most of the devices also support OpenFlow. Open Virtual Switch (OVS) [5] is a widely used virtual switch. Besides, CISCO and Juniper also have their hardware switches that support OpenFlow.

Control Layer

The control layer is the heart of the SDN environment. It takes responsibility for configuring and managing the forwarding rules. Typically, the SDN controller can be grouped into two categories: and *a centralized controller* and *distributed controller*. The centralized controller normally has one controller, which takes care of all switches. The drawback is obvious: a single point of failure, performance, and scalability. The distributed controllers have several controllers. Each of them takes care of a subset of devices. Each distributed controller can also communicate with each other. If a controller fails, other controllers can take over the switches of that failed controller. But the problem of distributed controllers is high deployment cost and complex synchronization [19].

Northbound API

Northbound API is the programming interface that establishes a communication between Control Layer and Application Layer. Essentially, it is a RESTful API that runs on the controller and hosts application.

Application Layer

Application Layer consists of several tasks, including measurement and monitoring. Network administrators also can covey their requirements in this layer. For instance, network administrators once decide the controller, they can write a script and run in the controller. When the controller gets the script, it installs the rules to devices based on the feedback parameters from devices. This is the basic procedure in the SDN environment.

2.1.2 OpenFlow

In this section, we provide a summary of the OpenFlow protocol and introduce how an OpenFlow switch works as we use both the OpenFlow protocol and the OpenFlow protocol enabled switches.

OpenFlow is first proposed in 2008 at Stanford University [22]. To date, OpenFlow has several versions from 1.0 to 1.5. As a Southbound API, it establishes a secure channel that enables communication between the SDN controller and OpenFlow switches. They communicate with several messages. Since we use OpenFlow 1.3 in this thesis, we just consider OpenFlow 1.3. There are three types of OpenFlow messages:

- **Controller-to-Switch:** manage or inspect the state of a switch.

- **Asynchronous (Switch-to-Controller):** generated by a switch and used to update the controller about network events.
- **Symmetric (both-way):** created by a switch or controller and does require a response. For example, HELLO message.

There are two messages that we used in this thesis: *Packet_in* and *Flow_removed*, which are asynchronous messages. A Packet_in message is triggered when a new flow enters a switch and there are no flow rules installed. Then, the switch sends a *Packet_in* message to the controller. The controller then decides the action for this flow. This is the basic traffic forwarding procedure in SDN. Figure 2.2 shows the workflow of a Packet_in message. A Flow_removed message is triggered when a flow rule is expired. Switch informs the controller that a flow rule is expired along with its counters. Hence, the controller gets statistics about this flow. Thus, we can use these two OpenFlow protocols for flow measurement in SDN.

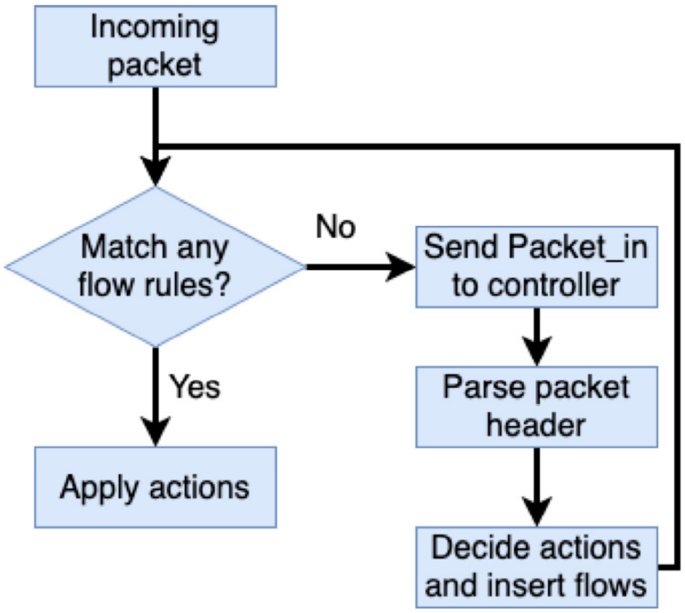


Figure 2.2: The workflow of a *Packet_in* message.

2.1.3 OpenFlow Switch

In the previous section, we illustrated the basic traffic forwarding procedure in SDN. In this section, we present the procedure follows in a switch.

Figure 2.3 presents the architecture of OpenFlow switches. Each OpenFlow switch contains a set of flow tables and each flow table contains a set of flow rules. Each flow rule entry consists of three components: a *match field*, an *action*, and its *statistics*.

- **match field:** used to match against packets. This consists of the ingress port, packet headers and optional metadata specified by programmers. A packet is matched to a flow rule only if all match elements are matched
- **action:** represents which action will be executed if a packet is matched, e.g., forward packet to port(s), drop the packet, or forward to the controller.
- **Statistics:** contains a counter that records statistics of this flow, e.g., the number of packets and the number of bytes traversed.

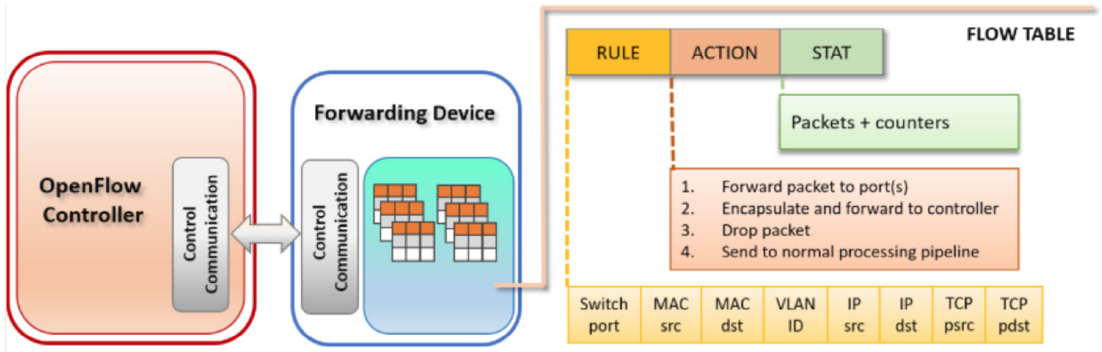


Figure 2.3: OpenFlow switch architecture.

2.2 Network Measurement Schemes

In this section, we discuss the measurement schemes in SDN. We briefly introduce two types of measurements and present some prior work related to each type. After that, we discuss hybrid schemes and also review some existing work.

2.2.1 Active Measurement Schemes

Active measurement monitors a flow by injecting probe packets into the network, where the measurement accuracy depends on the probing frequency. The higher the rate, the better the accuracy at the price of measurement cost. The *ping* application is one simple example of an active measurement that uses ICMP packets to measure the end-to-end connection status and round-trip-time.

SDN-Mon [26] is an active monitoring framework, i.e., inserts probe packets to the network to monitor a single flow from a switch. It uses a bloom filter to check whether an incoming probe packet is collected or not. If it is collected, ignore that packet; otherwise, collect it and add corresponding flow to the filter. Thus, SDN-Mon needs to check every new flow entry to decide on the monitoring flow. In our WAS, we first aggregate all flow entries by their source-destination. Thus, we do not need to filter probe packets as SDN-Mon.

[34] indicates that flow-based measurement consumes too many resources (bandwidth, CPU) because of the fine-grained monitoring demands. OpenNetMon (ONM) [34] proposes an online flow monitoring approach for throughput, packet loss, and delay using probe packets. Unlike other monitoring work, OpenNetMon needs to poll each flow entries' destination switch. There are two reasons: 1) it reduces resource consumption and 2) polling the destination switch can gain better throughput accuracy (we will illustrate in Chapter 3). However, OpenNetMon still monitors all flow entries overall source-destination pairs, which may not scale with the increasing number of flows and switches.

RFlow [18] is another active measurement scheme for WLAN. The authors mention that the per-flow monitoring leads to low accuracy in long-term monitoring and high overhead in short-term monitoring. Thus, they deploy a set of collectors, called RFlow local agent, to gather the flow statistics. The local agent can verify long-term and short-term flow by subtracting the current time from the time the flows are entered. If the difference is greater than a threshold, it means this flow entry is short-term monitoring. Thus, RFlow decreases the polling frequency to achieve lower communication cost. RFlow may balance between the accuracy and overhead in long-term and short-term measurement, but the solution needs to access every switch.

2.2.2 Passive Measurement Schemes

The passive measurement does not insert any probe packets in the network, but it requires full access and control on the devices. It just receives measured statistics from a set of configured switches. Thus, passive measurement reduces measurement overhead at the cost of full access requirements on the monitoring switches. One example is to use OpenFlow message. When a flow is expired, switch sends a *flow_removed* message to the controller with its statistics.

OpenTM [32] is the first traffic matrix estimation system for OpenFlow networks. In addition to using *flow_removed* message, OpenTM periodically samples counters from switches. Hence, the total number of queries is bounded by the number of active flows in OpenTM. The authors then propose different polling algorithms to reduce the monitoring overhead. However, OpenTM samples all switches without considering the cost.

In [41], the authors indicate that there is no measurement work that measures network utilization without sending probe packets. Thus, they propose FlowSense (FS) with zero measurement cost by using only the OpenFlow messages. In FS, a flow can be in two states: *flow arrival* and *flow completion.* Each state can be indicated by OpenFlow messages (*Packet_in* and *Flow_Removed*), respectively. FlowSense uses *Packet_in* to record a flow entry's start time and *Flow_Removed* to record its finish time. By subtracting these two, FlowSense can get the lifetime. In addition, *Flow_Removed* can also bring the statistic of this flow entry. Once a *Flow_Removed* message received, FlowSense can update current network utilization. The dependency on the OpenFlow messages can impact the measurement accuracy that we will discuss in the next section.

Payless [13] is another passive network measurement framework for SDN. The authors point out the frequency of sampling switches determines the accuracy and network overhead. They propose an algorithm to adjust the sampling rate that we will be discussed in Section 2.4. In addition, to support such an algorithm, they also make a set of RESTful API to transfer the data to the application layer. Payless also uses standard OpenFlow messages and has the same drawback of depending on these messages.

2.2.3 Hybrid Measurement Schemes

A hybrid measurement combines active and passive measurements. It can either send probe packets with a fixed polling frequency or read statistics from counters with a fixed sampling rate.

sFlow [8] consists of a sFlow-controller and a set of sFlow-agents to offer both the active and passive measurements. The sFlow-controller actively polls the selected set of switches with a fixed polling frequency, whereas sFlow agents passively sample switches' counters with a fixed sampled rate. These days most of the switches come with sFlow support. Thus, in this thesis sFlow is chosen to monitor the switches.

We conduct a simple test to show the performance of sFlow compared to OpenFlow *flow_removed* message. We use iperf [3] to generate traffic for five source-destination pairs, each with an initial rate of one Mbps and double it at every ten seconds. Thus, the total network utilization is five Mbps initially. We also set the flow entry's hard timeout to five and fifteen seconds to show the different performance of OpenFlow. The measurement in the case of OpenFlow depends on it triggers only at *flow_removed* message received, which impacts the accuracy as we observe in Figure 2.4. OpenFlow can update the utilization sFlow only when a *flow_removed* message received. On the other hand, sFlow presents a good accuracy on the entire measurement period, but accuracy slightly changes at the points of utilization change.

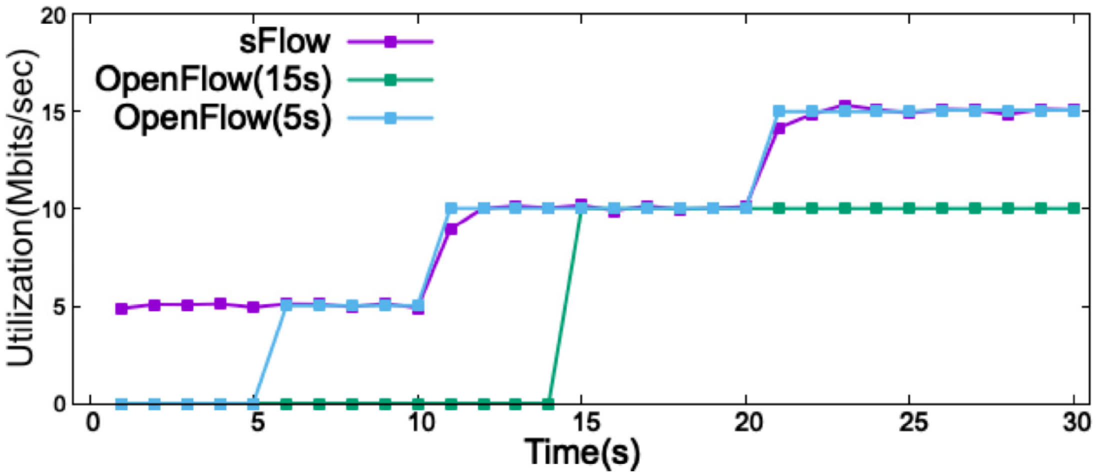


Figure 2.4: The measurement granularity of OpenFlow and sFlow [31].

There are some other works that propose low-cost algorithms to reduce the monitoring cost. Thus, we can use them either with the active/passive measurement. In this thesis, we group them with hybrid schemes.

FlowCover [28] reduces measurement cost by reducing the number of monitoring switches. CeMon [29] is a multi-controller version of FlowCover, i.e., the measurement task is distributed among a set of controllers. Partial Flow Statistics Collection (PFSC) [38] also collects flow statistics from a subset of switches such that the flow recall ratio on every switch reaches a predefined value while minimizing the number of queried switches. Lonely Flow First (LFF) [40] is another flow-based monitoring algorithm. The lonely flow is a flow entry which only passes through a single switch because it has a minimum measurement cost. Same as FlowCover, LFF also monitors a subset of switches but it considers switches with lonely flows first. We will discuss these works in details in Section 2.3.

2.2.4 Summary of Measurement Schemes

Table 2.1 presents a summary of previous network measurement solutions in SDN. It is apparent that there are more passive measurement schemes because of its low cost. Furthermore, recent works mostly choose hybrid schemes because it can further reduce the measurement cost. Hybrid measurement might become a trend as long as we can design a low-cost algorithm. In this thesis, we implement the top two most cited measurement schemes from each category and compare them with our work.

Table 2.1: The summary of types of network measurement schemes.

Active	**Passive**	**Hybrid**
SDN-Mon [26]	ABW [23]	sFlow [8]
OpenNetMon [34]	OpenTM [32]	LFF [40]
RFlow [18]	TMFramework [39]	FSBA [27]
	OpenMeasure [20]	CeMon [29]
	FlowSense [41]	PFSC [38]
	eOpenFlow [12]	FlowCover [28]
	Random Samplig [10]	
	Payless [13]	

Table 2.2 presents the performance of the top two most cited work from each type of measurement schemes and our work. The measurement cost of active schemes is significantly higher than the others. The passive schemes reduce the cost but just a little. On the other hand, the hybrid schemes with low-cost algorithm significantly reduce the cost, which we will discuss in details in Chapter 3. All three schemes are mostly accurate with only a little difference. The computation cost is the time complexity of the algorithm used in each work. FlowCover has a very high time

complexity that we mentioned above. In terms of memory cost, we measure the number of flow entries. Since SDN-Mon only monitors the destination switch of each flow entry, the memory cost is lower than other schemes. We also monitor a subset of switches as in FlowCover. All existing works measure the delay, throughput (TP), and network utilization (UTL) while FlowSense measures only the network utilization. We also consider the resiliency of each work. The resiliency indicates whether a work can measure in the presence of link and node failures. None of the prior work considers failure although it can affect the measurement. We will have further discussion on this issue in Section 2.5.

Table 2.2: The performance comparison of measurement schemes.

Types of Measurement	**Active**		**Passive**		**Hybrid**	
Frameworks	**SDN-Mon**	**ONM**	**FS**	**OpenTM**	**FlowCover**	**Our Work**
Measurement Cost	Very High	Very High	High	High	Low	Low
Accuracy	High	High	High	High	High	High
Computation Cost	$\mathcal{O}(nm)$	$\mathcal{O}(nm)$	$\mathcal{O}(nm)$	$\mathcal{O}(nm)$	$\mathcal{O}(n^2m)$	$\mathcal{O}(nm)$
Memory Cost	$< nm$	nm	nm	nm	$< nm$	$< nm$
Metrics	Delay TP UTL	Delay TP UTL	UTL	Delay TP UTL	Delay TP UTL	Delay TP UTL
Resiliency	No	No	No	No	No	Yes

2.3 Low-cost Measurement Methodology

In this section, we introduce three low-cost algorithms that are mentioned in Section 2.2.3.

A naive monitoring scheme in an SDN environment is to monitor all available switches. It measures each flow entry from every switch, which generates a large number of control packets and increases the overhead with the increasing number of flows and switches. The number of control packet will reach nm, where n is the number of switches and m is the number of flow entries at a switch. Thus, aggregation of flow statistics from a subset of switches can reduce the control overhead without degrading the measurement accuracy.

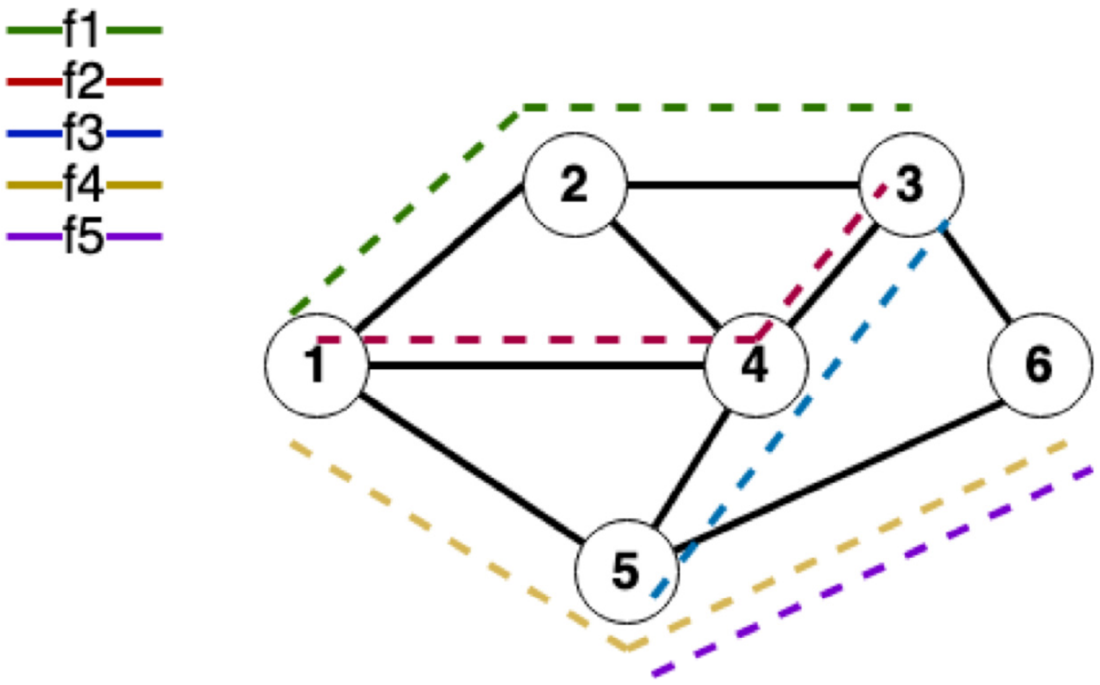


Figure 2.5: An example to illustrate the operations of low-cost algorithms.

FlowCover [28] proposes an algorithm to reduce the measurement cost by monitoring a subset of switches. It first assigns a weight to each switch proportional to the number of flows passing through it. The switches are then monitored according to their weight until all flow across the network is covered. In Figure 2.5, there are five flows in this example. The weighted switch to flow list is:$\{1 : 3, 2 : 1, 3 : 3, 4 : 2, 5 : 3, 6 : 2\}$. FlowCover first chooses switch 1 because it has the highest weight, which covers f_1, f_2, and f_4. After that, FlowCover updates the weighted list, the new is: 2:0,3:1,4:1,5:2,6:2. Thus, it selects switch 5 as the second monitoring switch. At this time, all flows are covered. Thus, the final set of monitoring switches include $\{1, 5\}$. The time complexity of FlowCover is $\mathcal{O}(n^2m)$ because FlowCover needs to go through all flow entries, assigns a weight, and picks the highest weighted switch. The number of control packets is at most nm as it monitors a subset of switches instead of all switches. However, the time complexity is of FlowCover is high. In this thesis, we propose an algorithm, called Weight Assisted Selecting (WAS, whose time complexity is reduced to $\mathcal{O}(nm)$ compared to FlowCover.

Lonely Flow First (LFF) [40] first monitors switches cover the lonely flows. In Figure 2.5, only switch 2 has a lonely flow. Thus, LFF monitors switch 2 first to cover flow f_2. After that, it chooses other switches as in FlowCover until all flows are covered. The weighted list is: $\{1 : 2, 3 : 2, 4 : 2, 5 : 3, 6 : 2\}$. Thus, LFF next selects switch 5 to cover f_3, f_4, and f_5. At this time, only f_2 is left, for which LFF can choose any one of switches. Thus, the final set of monitoring switches includes $\{2, 5, 1\}$. The time complexity of LFF is still $\mathcal{O}(n^2m)$. Although it chooses some switches first, the main operation is still the same as in FlowCover. The number of control packets of LFF is at most nm.

Partial Flow Statistics Collection (PFSC) [38] also collects flow statistics from a subset of switches. PFSC defines an optimization model that minimizes the number of monitoring switches such that the flow recall ratio on every switch reaches a predefined value β. The flow recall ratio is the ratio of the number of flows holding by a monitoring switch to the total number of flows. In Figure 2.5, we have five flows. Suppose we set $\beta = 0.5$, that means each switch must monitor at least 2.5 flows. In this case, only switch 1, 3 and 5 meet this criterion. Thus, PFSC needs to monitor these three switches. The time complexity of PFSC is $\mathcal{O}(nm)$ because it needs to go through all flow entries once and selects switches. However, there is an overlap if monitoring these three switches. Some flows are already covered by other switches.

2.4 Polling Schemes

Once we decide the type of measurements we will use and which switches need to be monitored, we need to choose the polling frequency/sampling rate. Unfortunately, none of the above works consider this issue except Payless. They all use a fixed polling frequency/sampling rate. However, there is a major drawback of a fixed rate. In this section, we first introduce the drawback of using a fixed rate. Then, we introduce prior work that adjusts the polling frequency/sampling rate.

The network measurement tool sFlow has some drawbacks. sFlow [8] combines both active and passive measurements. It sets a polling frequency which decides the frequency for sending a probe packet and a sampling rate which decides the rate for picking a packet out of some packets. However, sFlow provides a static measurement, which can cause inaccuracy and extra overhead because we need a lower frequency for high volume flows and higher frequency for low volume flows [24].

Payless [13] defines on-demand polling frequency using Markov-process based model. Once Payless receives statistics by *Flow_Removed* message, it calculates the difference between the current and previous counters. If the difference of byte count is above a threshold, it means there is high volume traffic in the network. Payless then decreases the sampling rate. The reason to decrease the rate, rather than increasing, is the cost. A high rate can get slightly better accuracy than the low rate at the price of high measurement cost [8].

OpenSample [30] leverages sFlow packet sampling to provide near-real-time measurements. It first determines the probability of each flow entry. The probability is based on the flow size (packets). The larger the size, the higher the probability. The probability then decides the polling frequency for this flow. Thus, the high volume flow will get a higher polling frequency, which is a conflict with Payless. The authors indicate that the accuracy is low if we use low polling frequency for the high volume of traffic. However, there are some works support Payless and the other supports OpenSample. In this thesis, we follow Payless because one of our goals is reducing the measurement cost.

Volley [24] is a violation likelihood-based approach for state monitoring in data centers. The goal of Volley is to detect a state violation (e.g., DDoS attacks). However, a fixed polling frequency can mis-detect violations because violations may occur between two polling operations. Thus, Volley dynamically adjusts polling frequency based on how likely a state violation will be detected: the higher the polling frequency, the higher the chance of occurring a state violation. Volley provides an algorithm to estimate the state violation. The state violation is estimated by two factors: the current sampled value and the changes between the two samples. When current samples value is low, a violation is less likely to occur. When the change between two sample values is large, a violation is more likely to occur.

Sampling-On-Demand (SOD) [15] is a new framework that provides adjusting sampling rate on-demand. SOD installs a sampling management module at each switch. This module allows the controller to determine sampling allocation. A sampling allocation is a mapping that indicates which flow should be sampled by which switch and at what rate. SOD designs an optimization model to indicate sampling allocation. The model maximizes the utility gained by deploying a sampling allocation such that all flow entries are covered and each switch does not exceed its capacity.

However, SOD does not consider measurement cost. In our model, we consider cost and the results show that our work can reduce 50% measurement cost compared to SOD.

Table 2.3 summarize previous polling schemes. Payless and Volley use Markov process, where they only consider the mean of two previous consecutive observations. OpenSample considers the volume of flows while both SOD and our work consider several attributes, including traffic volume, source-destination pairs and the capacity of switches. In addition, all work consider either the accuracy or the cost. In this thesis, we consider balancing both the cost and accuracy.

Table 2.3: The summary of the polling schemes

Polling Schemes	Parameter Considered	Adjust Polling Freq	Goal	Methodology Used
Payless	Changes between two samples	Yes	Cost	Markov Process
OpenSample	Packet size	Yes	Accuracy	Not Used
Volley	Changes between two samples	Yes	Accuracy	Markov Process
SOD	Attributes	Yes	Accuracy	MC-GAP
Our Work	Attributes	Yes	Accuracy and Cost	MC-GAP

2.5 Failure Recovery Schemes

As mentioned before, the resiliency is another important factor during monitoring. We conduct a simple test to show how failure can impact measurement accuracy. We use iperf to generate traffic for five source-destination pairs, each with a rate of one Mbps. We compare the measurement accuracy of our proposed monitoring scheme, WAS, with and without link failures, with and without failure recovery algorithm, which is shown in Figure 2.6. The results indicate that with a small number of link failures, we can have a reasonable accuracy as it may impact a small amount of traffic. However, the accuracy degrades with the increasing number of link failures. On the other hand, if we apply some failure recovery algorithms, we can hold a good accuracy even with several links failure. The accuracy only degrades with a lot of links failure. Thus, a proper failure recovery algorithm is essential during monitoring as it helps to increase accuracy.

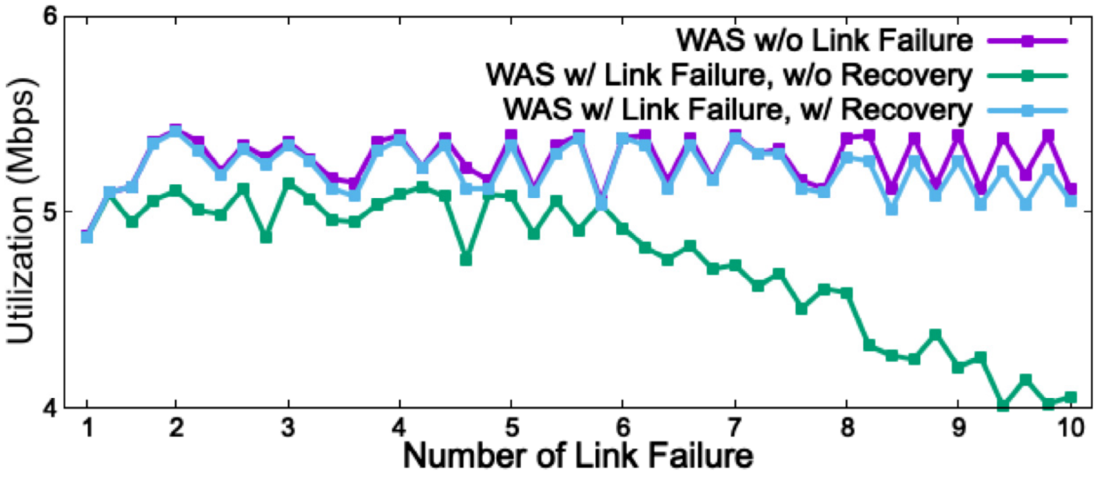


Figure 2.6: The link failure impact on the measurement accuracy.

Typically, there are two types of failure recovery schemes: *reactive* and *protective*. In reactive schemes, a switch contacts controller to get a new flow entry after detecting link failure. Thus, this on-demand recovery approach efficiently utilizes the memory of a switch at the cost of extra communication delay to the recovery time. On the other hand, protective scheme inserts flow entries for the available route before the actual failure occurs. Thus, a switch can locally recover from a failure without contacting the controller, which reduces the recovery time at the price of extra memory usage.

In SDN architecture, the OpenFlow protocol supports Fast-Failover Group (FFG) [25] to implement the failure recovery at the data plane. FFG is a widely used protective approach in SDN. Figure 2.7 presents the workflow of FFG. A switch maintains a Fast Failover Group table with several action buckets. Each bucket is associated with a port, and only a single bucket is executed at a time. The incoming packet will be sent out from a port if this port is alive. In the case of a link failure, the next active port and bucket are chosen to redirect the affected traffic. Nonetheless, if there is no such backup route, the alternative redirecting approach is essential.

For example, in Figure 2.8 there are two routes from Source to Destination. The green primary route (Source-A-B-Destination) and the red backup route ($Source - C - D - E - Destination$). Thus, a FFG table installed at switch $Source$ has two action buckets. The first one watches link one and the second one watches link four. Once link one between $Source$ and A fails, it forwards the packet to link four. Thus, the new route is $Source - C - D - E - Destination$. Suppose if link seven between

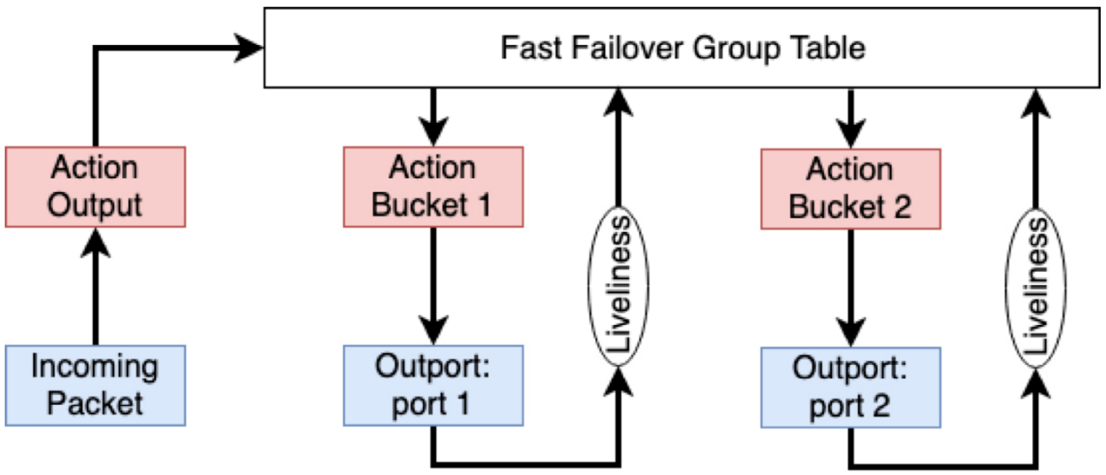


Figure 2.7: The workflow of Fast Failover Group.

switch *E* and *Destination* fails, there is no backup route available to redirect the traffic locally. In this case, Crankback is proposed.

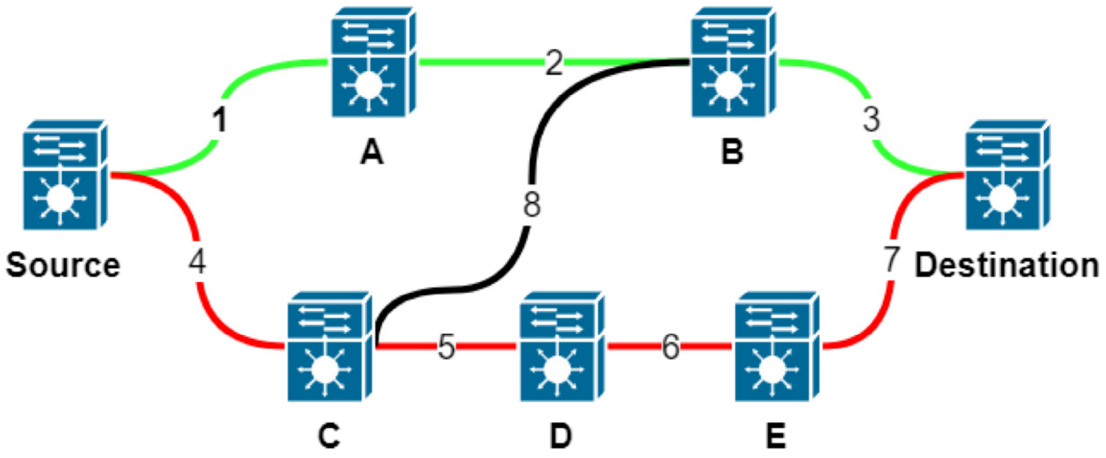


Figure 2.8: An example topology [31].

Crankback [35] is another link failure recovery approach that supports topologies without backup routes between every pair of switches. Crankback is another protective approach and can be thought as an extension of FFG. The affected packet traverses backward until finds a switch having a backup route towards the destination. The packet carries a unique tag to inform switches about the failure. All the subsequent packet for the same source-destination pair follows the alternative route.

In the previous example, once link seven fails, the packet is sent backward towards the Source. Once the packet reaches switch C, the packet is forwarded to link eight because C has a backup route. The new route is $Source - C - D - E - D - C - B - Destination$. Clearly, Crankback introduces extra delay and communication cost due to packet-by-packet backtracking. Furthermore, the source still keeps sending packets over the failed routes that backtrack to find an alternative path.

Neither FFG or Crankback can support multiple link failures as the former needs an available backup route and the latter supports only a limited number of routes (primary, backup, and Crankback routes). Thus, if a node experiences more than two link failure, it needs an alternative solution. Finally, in the case of a single-node failure, it is easy to update flow rules on the destination switches connected by the links originating from that failed node. For instance, in the example topology from Figure 2.8, if node B fails, A and C can use Crankback and FFG, respectively. It is also possible to deploy a controller assisted reactive recovery scheme to improve the recovery time in the case of Crankbacking.

Chapter 3

Measurement and Polling Schemes

In this chapter, we present a low-cost measurement algorithm and adjusting polling frequency/sampling rate scheme. Section 3.1 introduces the low-cost measurement algorithm: *Weight Assisted Selecting (WAS).* Section 3.2 presents the optimization model for the polling frequency/sampling rate. Section 3.3 explains the experimental setup. In section 3.4, we present and discuss the evaluation results.

3.1 Weight Assisted Selecting (WAS)

In this section, we present the low-cost measurement algorithm, called Weight Assisted Selecting (WAS). Suppose we have a network as a graph $G = (V, E)$, where E indicates the set of links and $V = \{v_1, v_2, ..., v_n\}$ is the set of switches. Hence, $n = |V|$ is the total number of switches and each switch carries m flows in average. We set a dictionary S which contains the shortest path for each source-destination pair. This process can be configured during the initial network setup. At the same time controller can maintain this route information. The WAS algorithm is presented in Algorithm 1.

In WAS, we first aggregate all flow entries based on their source-destination and record corresponding weight. The weight based on the number of flow entries shared by the same source-destination pair. After that, we sort all aggregated flow entries according to their weight. We start with the flows with the highest weight. Usually, it is enough to monitor a single switch along the path of a flow. Thus, it is possible to consider only the destination to check all flows that share the same source-destination pair. Then, we move to the next flow entry to pick a monitoring switch. We continue until all flows are covered. If the corresponding monitoring switch is already chosen, we move forward to the next group of flows.

For example, in Figure 2.5, there are five flows. In FlowCover, the final set of monitoring switches includes $\{1, 5\}$. However, WAS does not assign any weight to

Algorithm 1 Weight Assisted Selecting (WAS)

```
Input:
    Aggregated Flows: $S = \{(src, dst) : (src, ..., dst), ...\}$
Output:
    Monitored Switches: $P$
1: $P = []$
2: for each $(pair, path) \in S$ do
3:     $(src, dst) = pair$
4:     if $path \cap P = null$ then
5:         $P$.append($dst$)
6:     end if
7: end for
```

switches. WAS first aggregate all flow entries. Since there are no flows share same source-destination pair, we just skip this step. Next, WAS picks the destination switch if a newly installed flow is not monitored by any other switches and not aggregated. Thus, WAS first picks 3 for the flow f_1, which also covers the flow f_2 and f_3. Then, it selects 6 for the next monitoring switch, which furthermore covers the last two flows. The set of monitored switches for WAS is $\{3, 6\}$.

The main difference between WAS and FlowCover is that WAS considers the number of flow entries with same source-destination pair as a weight of a flow; whereas FlowCover considers the number of flow entries installed in a switch as a weight of a switch. Thus, the computation complexity of WAS is $\mathcal{O}(nm)$ because we only need to go through all flow entries once and the complexity of FlowCover is $\mathcal{O}(n^2m)$.

3.2 Adjusting Polling Frequency/Sampling Rate (APS)

In this section, we present our optimization model for adjusting the polling frequency/sampling rate. The goal of the model is to balance between the measurement accuracy and cost based on the associated weight. Our model has two versions: offline and online, which are presented in section 3.2.1 and section 3.2.2, respectively.

3.2.1 Offline APS

In the offline version, we assume all the flows have already entered the network. Since we already have a subset of monitoring switches from WAS, we can use those switches to get the list of available flows. Let S be the set of monitoring switches determined by WAS. For each switch $s \in S$, C_s is the sampling capacity of s in packets per second (pps) and R is a set of possible sampling rates supported by the switches. Also, let F be the set of flows be monitored. For each flow $f \in F$, d_f is the current packet rate in pps and r_f is the recommended sampling rate for a flow with rate d_f. Furthermore, P_f is the path of flow $f \in F$. We define an accuracy function A_{sfr}, which indicates the accuracy for the flow f at switch s using rate $r \in R$. We also define a resource consumption function RC_{sfr}, which indicates the resource consumption of the sampling flow f with rate r at switch s. Table 3.1 summarizes the symbols that we use in offline APS.

The model for the offline APS is presented below.

$$\text{maximize:} \sum_{s=1}^{|S|}\sum_{f=1}^{|F|}\sum_{r\in R} A_{sfr}x_{sfr} - \sum_{s=1}^{|S|}\sum_{f=1}^{|F|}\sum_{r\in R} RC_{sfr}x_{sfr} \tag{3.1}$$

$$\text{subject to} \sum_{f=1}^{|F|}\sum_{r\in R} d_f r \leq C_s, \text{for each } s \in S \tag{3.2}$$

$$\sum_{s=1}^{|S|}\sum_{r\in R} x_{sfr} \geq 1, \text{for each } f \in F \tag{3.3}$$

$$s \in P_f, \text{for each } f \in F, s \in S \tag{3.4}$$

$$r \geq r_f, \text{for each } f \in F \tag{3.5}$$

$$\sum_{r\in R} x_{sfr} = 1, \text{for each } f \in F, s \in S \tag{3.6}$$

In the above formulation, A_{sfr}, RC_{sfr}, and x_{sfr} are the accuracy function, the resource consumption function, and the decision variable, respectively. $x_{sfr} = 1$ if switch s is chosen to sample flow f with the rate r. The objective function is to find a feasible sampling rate r for each switch s that maximizes the total network accuracy and minimizes the cost, which is not considered in [15].

Equation (3.2) ensures that the total sampling rate required from each switch $s \in S$ does not exceed its maximum sampling capability C_s. Equation (3.3) says that

Table 3.1: List of symbols used in offline APS.

Symbol	**Definition**
S	Set of Monitoring Switches Determined by WAS
C_s	The Sampling Capacity of Switch s in Packets Per Second (pps)
R	Set of Possible Sampling Rates Supported by Switches
F	Set of Flows be Monitored
d_f	Current Packet Rate in pps for Flow f
r_f	Recommended Sampling Rate for rate d_f
P_f	Path of Flow f
A_{sfr}	Accuracy for Monitoring Flow f at switch s using rate r
RC_{sfr}	Resource Consumption for Monitoring Flow f at switch s using rate r
x_{sfr}	Decision Variable

each flow f should be sampled by at least one switch. Equation (3.4) ensures switch s should traverse in the corresponding flow and in the set of monitoring switches. Equation (3.5) says that the sampling rate r should at least exceed recommended sampling rate r_f for current packets per second d_f for each flow f. Equation (3.6) ensures that each switch can only choose exactly one sampling rate.

Definition 3.2.1. Multiple Configurations Generalized Assignment Problem (MC-GAP) model

MC-GAP [14] is an extension of Generalized Assignment Problem (GAP). GAP is a well-known Knapsack problem. The input of GAP is a set B of knapsacks and a set I of items. Each knapsack has a size, and each item has a size and a utility. The objective of GAP is to find a way to assign all items to a feasible knapsack, such that the utility is maximum.

MC-GAP extends GAP by associating multiple configurations with each item. The input of MC-GAP is a set B of knapsacks, a set I of items and a set C of configurations. Each knapsack also has a size, and size for each item in each knapsack using each configuration. The objective is to find a way to assign all times with a configuration to a feasible knapsack, such that the utility is maximum. In [14], MC-GAP is already proved as an NP-hard problem. The authors design a heuristic algorithm to solve it.

Theorem 3.2.1. *The offline adjusting polling frequency/sampling rate model is an NP-hard problem.*

Proof. We can prove the NP-hardness by showing offline APS model is a MC-GAP problem [14]. We can represent each switch as an MC-GAP knapsack whose size is equal to the sampling capacity. Each flow can be represented by a MC-GAP item, and each sampling rate as a MC-GAP configuration. The accuracy of sampling a flow at a switch using a sampling rate can be represented by the value in the MC-GAP utility of assigning an item to a knapsack with a configuration. □

Since offline model is NP-hard, we propose a heuristic algorithm to solve it efficiently based on the heuristic in [14]. The greedy algorithm is shown in Algorithm 2.

Algorithm 2 Greedy Select Polling Frequency

```
Input:
    Monitoring Switches: $S$
    Flows: $F$
    Supported Sampling Rate: $R$
1: $C \leftarrow []$
2: while $C \neq F$ do
3:     Find a switch $s \in S$ such that $a \sum_{f=1}^{|F|} A_{f(s,r)} - b \sum_{f=1}^{|F|} RC_{sfr}$ is maximum
4:     Record $r$ as sampling rate for switch $s$
5:     $C \leftarrow C \cup f$
6: end while
```

We assign a score to each switch, which equals to $a \sum_{f=1}^{|F|} A_{sfr} - b \sum_{f=1}^{|F|} RC_{sfr}$. The first sum is total accuracy obtained by a switch and the second sum is a total resource consumption. We maintain two ratios a and b for two sums, where $a + b = 1$. These two ratios are used to balance accuracy and cost. We consider accuracy and cost are the same important if both are equal to 0.5. We can easily focus on one feature by increasing its corresponding weight. We greedily choose a switch with maximum score and record the flows that are sampled in this switch. We continue the process until all flows are covered. The main loop iterates for $\mathcal{O}(n)$, where $n = |F|$. The maximum accuracy switch s can be found in $\mathcal{O}(\log m)$ time by a priority queue where $m = |S|$. Thus, the time complexity of algorithm 2 is $\mathcal{O}(n \log m)$.

3.2.2 Online APS

In the online version, flows are added one at a time into the network. This flow could be a newly installed flow or a modified flow because of a failure. When a new flow is added, the controller needs to decide whether to sample it, at which switch, and whether to change the sampling rate. Let f' be the newly added flow, $d_{f'}$ be the current packet rate (packets per second), $P_{f'}$ be the path of the new flow, and $r_{f'}$ be the recommended sampling rate for sampling a flow with d'_f. The accuracy function $A_{sf'r}$ indicates the accuracy for sampling new flow f' at switch s using rate $r \in R$ and a resource consumption function $RC_{sf'r}$ indicates the resource consumption for sampling flow f with rate r. Table 3.2 summarizes the symbols that we use in online APS.

Table 3.2: List of symbols used in offline APS.

Symbol	**Definition**
f'	Newly Added Flow
d'_f	Current Packet Rate in pps for New Flow f'
r'_f	Recommended Sampling Rate for rate d'_f
P'_f	Path of New Flow f'
$A_{sf'r}$	Accuracy for Monitoring New Flow f' at switch s using rate r
$RC_{sf'r}$	Resource Consumption for Monitoring New Flow f' at switch s using rate r
$x_{sf'r}$	Decision Variable

The online APS model can be formulated as:

$$\text{maximize:} \sum_{s=1}^{|S|} \sum_{r \in R} A_{sf'r} x_{sf'r} - \sum_{s=1}^{|S|} \sum_{r \in R} RC_{sf'r} x_{sf'r} \tag{3.7}$$

$$\text{subject to} \sum_{f=1}^{|F+1|} \sum_{r \in R} d_f r \leq C_s, \text{for each } s \in S \tag{3.8}$$

$$\sum_{s=1}^{|S|} \sum_{r \in R} x_{sf'r} \geq 1 \tag{3.9}$$

$$s \in P_{f'}, \text{for each } s \in S \tag{3.10}$$

$$r \geq r_{f'} \tag{3.11}$$

$$\sum_{r \in R} x_{sfr} = 1, \text{for each } f \in F \cup f', s \in S \tag{3.12}$$

This formulation is similar to the offline version. We consider new incoming flows instead of all flows in the offline version. The goal is to maximize the accuracy of sampling new flow f' and minimize resource consumption. Equation (3.8) guarantees that the total sampling rate required from each switch $s \in S$ does not exceed its maximum sampling capability C_s. Equation (3.9) ensures that new flow f' should be sampled by at least one switch. Equation (3.10) says that switch s should traverse in the path of new flow and in the set of monitoring switches. And Equation (3.11) ensures that the sampling rate r should at least exceed recommended sampling rate r'_f for current packets per second d'_f for new flow f'. Equation (3.12) says that each switch can only choose one sampling rate at the end.

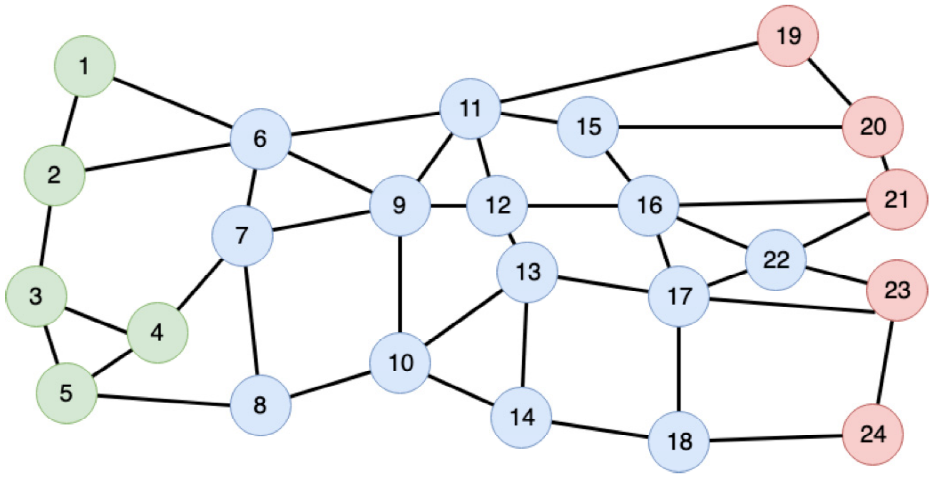
The online version can also be used when there is link failure. Since we recover from failure using some algorithms that we will talk about it later, we may add/modify some flows and adjust monitoring switches. Thus, we can apply this formulation in the presence of link failure. This online formulation is also NP-hard. In the heuristic, we choose a switch that can monitor the new flow with maximum score. The time complexity of the online version is $\mathcal{O}(\log m)$ because we just find a switch with maximum accuracy.

3.3 Evaluation Setup

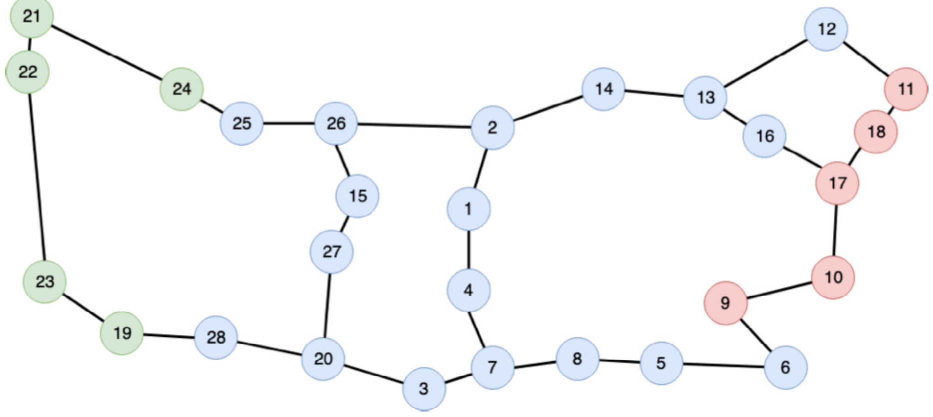
In this section, we illustrate the evaluation setup that we use to evaluate WAS and APS. We first evaluate the performance of WAS and compare the outcome with other prior work in two real topologies. Next, we compare the performance of APS with Sampling-On-Demand (SOD) and Payless.

To measure the performance of WAS, we implement it in two real topologies: USNET [9] and Darkstrand [2]. The USNET topology consists of 24 switches and 42 links as shown in Figure 3.1(a) [9]. The second topology (Darkstrand) is shown in Figure 3.1(b) [2], which consists of 28 switches and 31 links. We assume each switch connected with a host and each host can be a source or destination. Thus, USNET and Darkstrand can have at least 276 and 325 possible flows. USNET offers at least two flows between any source-destination pairs. However, it is not the case in Darkstrand. Thus, we can evaluate the performance of WAS in topologies with different properties.

We evaluate our framework and all algorithms in Mininet 2.2.2 emulator in an



(a) USNET



(b) Darkstrand

Figure 3.1: The USNET and Darkstrand topologies.

Oracle VirtualBox (VM) with an Intel Core i5 2.90 GHz (4 cores) CPU processor and 4GB RAM. The server runs in Ubuntu (64-bit) operating system. We choose Ryu [7] 4.30 controller as our single centralized controller in this thesis. And we can also choose several distributed controllers like ONOS in our future work. We use Open vSwitch [5] 2.11.90 as virtual switches. The data and control planes use OpenFlow 1.3 protocol to communicate.

We first evaluate the performance of WAS and compared with FlowCover in terms of monitoring accuracy. For the sake of completeness, we also consider the baseline approach, which monitoring all switches. In USNET topology, We randomly choose five source-destination pairs. Here, we choose a source from the source set (green nodes in Figure 3.1(a)) and a destination from the destination set (red nodes). We generate UDP traffic at a rate of 1 Mbps at the source node. We double this rate at every ten seconds and monitor the link utilization using sFlow. We will test the performance of TCP traffic in our future work. We set the sFlow sampling rate at switches as one every 200 packets and the witch polling interval at the controller as 20 seconds. Then, we evaluate the measurement cost in terms of number of monitoring switches. we assume the cost of monitoring a switch is constant. Thus, the number of monitoring switches can represents measurement cost. We also consider the USNET topology in the case of computation cost and accuracy. We measure the computation time as the computation cost and total packets received as accuracy. For each result, we calculate both the average and standard deviation. However, it is very small in some results and does not included in some figures.

To test the performance of APS model, we use iperf traffic received by sFlow. We compare the performance of APS with SOD and baseline approach in terms of the accuracy and resource consumption using IBM CPLEX optimizer [1]. The baseline approach is a static polling frequency set as default. We also try different a/b ratios to shows the balance between accuracy and cost. In addition, to show the performance of APS in the real Mininet environment, we implement our heuristic algorithm and compare the accuracy and resource consumption with Payless and SOD.

3.4 Discussion on the Evaluation Results

3.4.1 Performance of WAS

Figure 3.2 shows the accuracy of sFlow over time in USNET topology. The result shows that sFlow can measure link utilization accurately. However, sFlow also collects some additional control packets including ARP and LLDP packets. That is the reason why utilization monitored by sFlow is a little bit higher than real traffic. In addition, we observe slight discrepancy of sFlow when utilization changes [31].

Next, we present the performance of WAS. In Figure 3.3, we present the measurement cost of WAS in terms of the number of monitoring switches in USNET and Darkstrand. WAS just needs to monitor one additional switch compared to FlowCover in both topologies, where both approaches reduce around 30% measurement cost compared to the baseline approach [31].

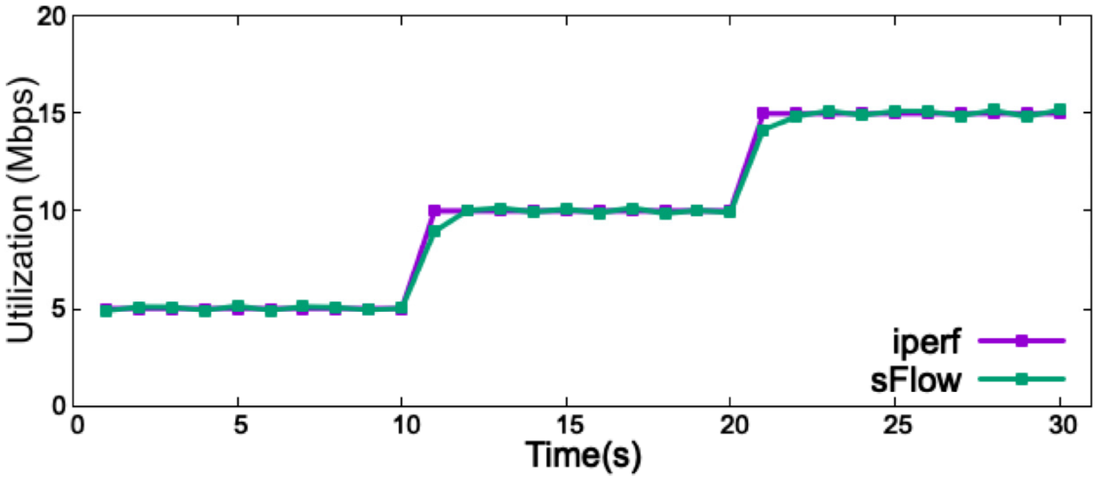


Figure 3.2: The measurement accuracy of sFlow.

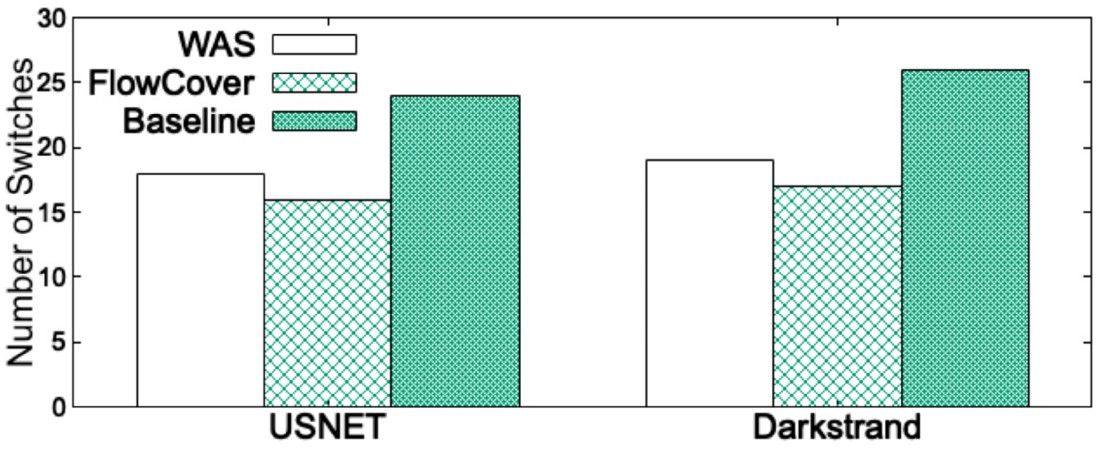


Figure 3.3: The measurement cost of different schemes.

However, the computation cost of FlowCover is significantly higher than that of WAS, which is shown in Figure 3.4. FlowCover spends over 25 ms in both topologies while WAS only takes 1 to 2 ms. However, 25 ms computation time is still acceptable. Thus, we evaluated the computation time in a larger topology called, DFN, with 58 switches and 87 links. This time, WAS takes 37 ms while FlowCover takes over 1100 ms. Thus, WAS reduces the similar amount of measurement cost with significantly less computation cost compared to FlowCover. In one word, WAS can reduce similar amount of measurement cost and significant lower computation cost compared to FlowCover [31].

Since FlowCover chooses a subset of monitoring switches based on the weight

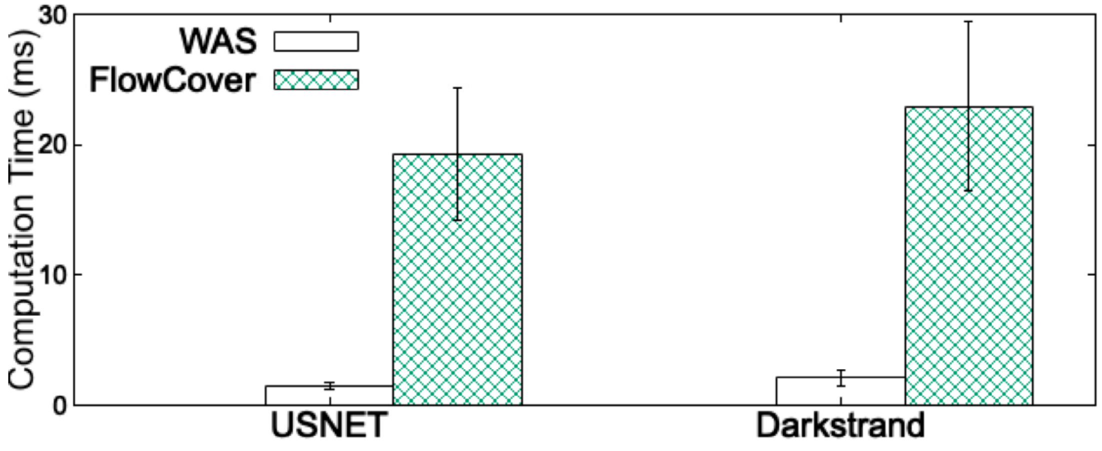


Figure 3.4: The average computation cost of WAS and FlowCover.

of a switch, that could impact the measurement accuracy. For example, suppose a source sends 100 packets to a destination and 98 of them reach the destination. If we monitor this flow at its source, the measured throughput is 100%. However, the actual throughput is $\frac{estimation}{actual} = \frac{98}{100} = 98\%$. However, WAS measures the monitored flows mostly in the destination across the chosen subset of switches, which enables better throughput measurement compared to FlowCover.

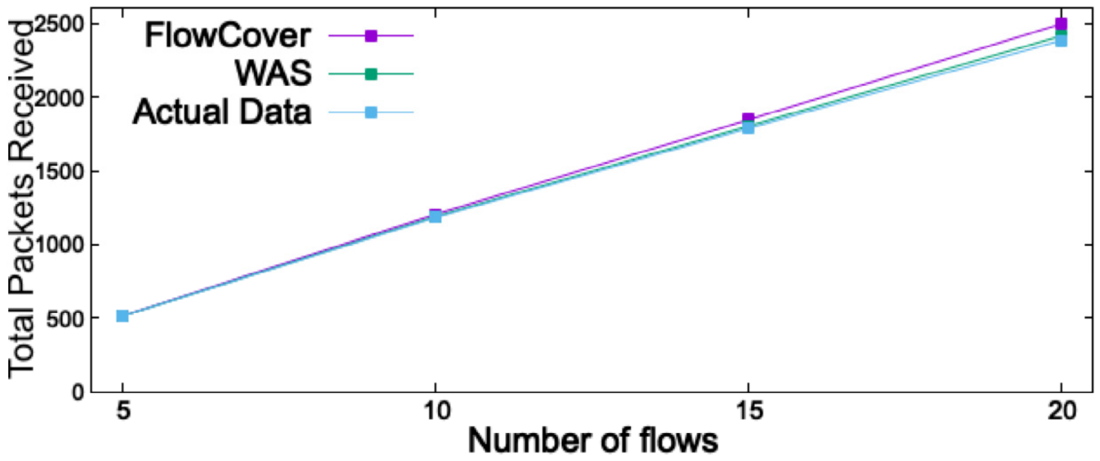


Figure 3.5: The accuracy of WAS and FlowCover.

Thus, we conduct a simple test in USNET topology with ten source-destination pairs. We set that each link has a 1% chance to lose packets. We increase the number of flows and record the total successfully received packets and compare them to the actual number of packets. Figure 3.5 shows that FlowCover measures extra packets compared to WAS that will impact the measured accuracy. Note that such a gap between the actual and the measured number of packets increases with the increasing number of flows in FlowCover. On the other hand, WAS is much closer to actual traffic while measuring all flows at the destination.

3.4.2 Performance of APS

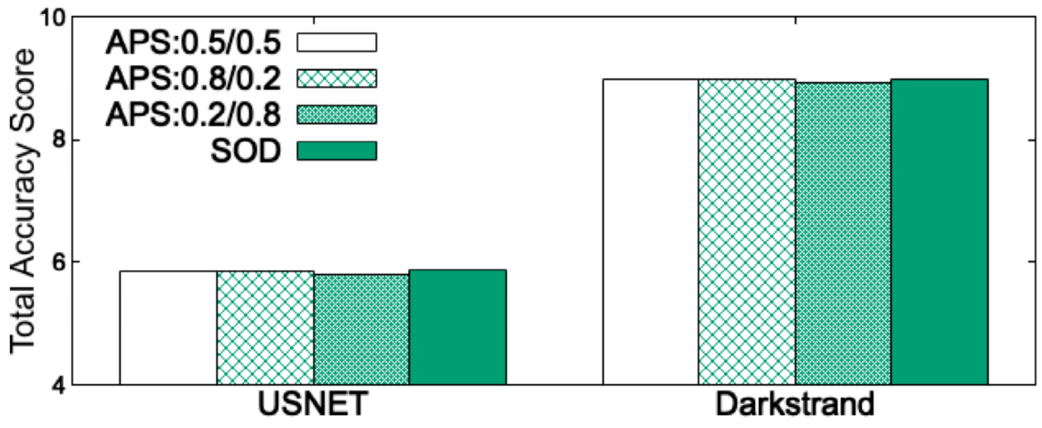


Figure 3.6: The accuracy score in USNET and Darkstrand.

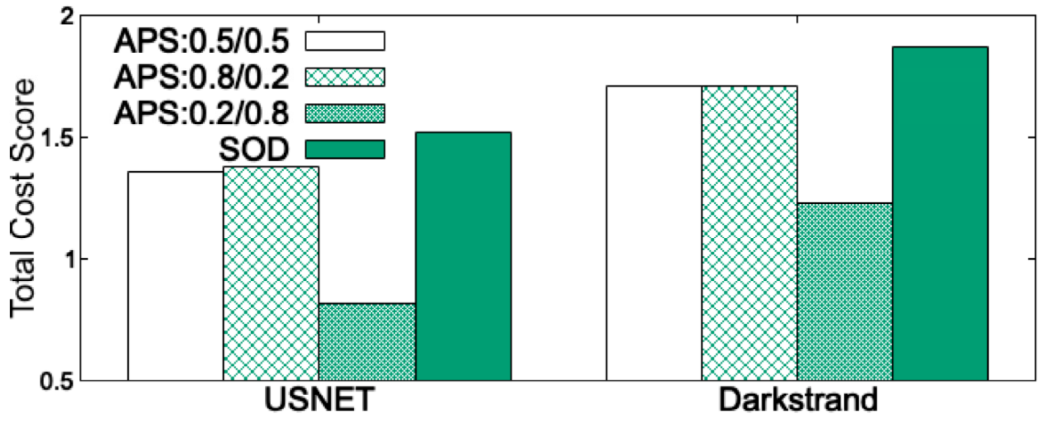


Figure 3.7: The total cost score in USNET and Darkstrand.

In this section, we first present the APS model evaluation results. Figure 3.6 presents the total accuracy score in USNET and Darkstrand. We set four different a/b ratios: 0.5/0.5 (considering accuracy and cost at the same level), 0.8/0.2 (more weight to the accuracy), 0.2/0.8(more weight to the cost), and Sampling-On-Demand (SOD) (only accuracy). The total accuracy score is the sum of accuracy that is obtained by sampling one flow at a switch. The result shows that the accuracy scores are similar for all methods. Sampling-On-Demand(SOD) has a little bit higher accuracy than others. However, the cost of SOD is very high, which is shown in Figure 3.7. The ratio 0.2/0.8 has a very low cost and obtain similar accuracy score compared to SOD. Besides, we test the ratio 0/1.0, but it is not accurate at all.

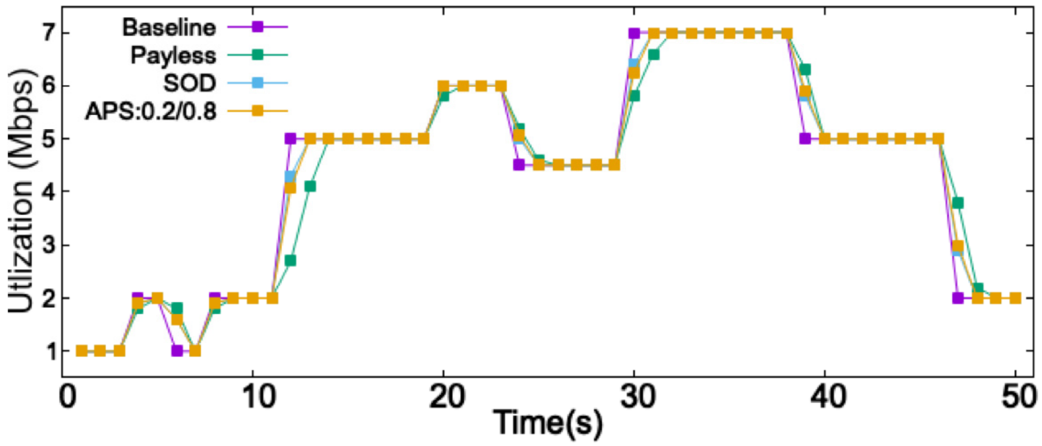


Figure 3.8: The real-time accuracy in USNET and Darkstrand.

At last, we implement the proposed heuristic in Mininet and compare with SOD and Payless in terms of accuracy and cost. We dynamically add/remove flows with an initial rate of 1 Mbps. Figure 3.8 presents the real-time utilization measured by three methods. The baseline is the actual traffic that we generate. All three methods show good accuracy most of the time. However, Payless is not accurate when a flow is added or removed. APS with ratio 0.2/0.8 is better and SOD is the best.

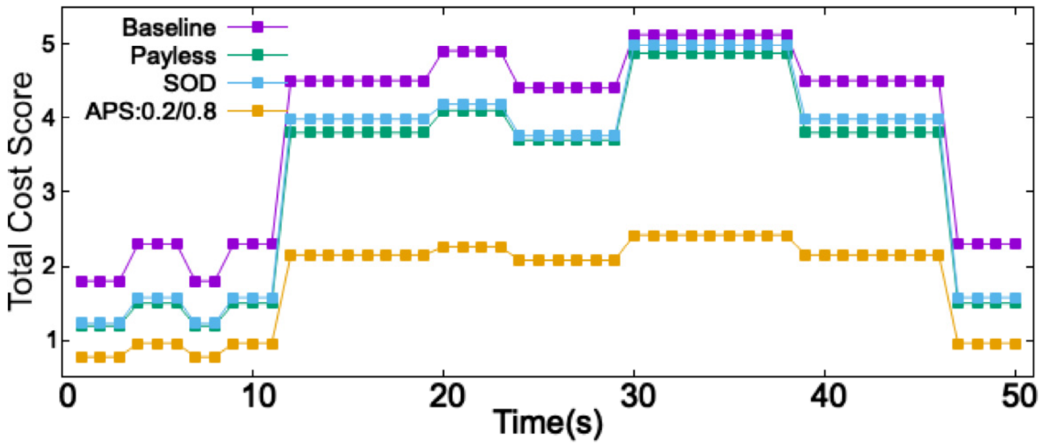


Figure 3.9: The real-time cost score in USNET and Darkstrand.

On the other hand, Figure 3.9 shows the real-time cost score. The baseline approach is sampling all monitoring switches with fix sampling rate. The cost of Payless is a little bit lower than SOD while APS with ratio 0.2/0.8 provides a very low cost. Both model evaluation and heuristic implementation results show that our APS can reduce roughly 40% measurement cost compared to Sampling-On-Demand and maintain similar accuracy. Thus, our solution provides a low-cost and high-accuracy dynamically adjusting sampling rate solution.

Chapter 4

A Resilient Flow Monitoring Framework

In this chapter, we present a resilient monitoring framework, called ReMon [31], that can successfully monitor traffic in the presence of link and node failure. Section 4.1 describes the architecture of ReMon. Section 4.2 presents our algorithms for recovering from link failure. Section 4.3 presents an algorithm to recover from node failure. In Section 4.4, we present the algorithm that updates WAS monitoring list because failure may change the list. Section 4.5 explains the experimental setup. In section 4.6, we present and discuss the evaluation results.

4.1 ReMon Architecture

In this section, we first provide an architectural overview of ReMon, which is presented in Figure 4.1. The software enabled switches are configured to initiate OpenFlow *Packet_in* and *Flow_Removed* messages. In the control plane of ReMon, we have a statistics gathering module (central sFlow collector) to collect these probe packets to learn the current network state. We use *Weight Assisted Selecting (WAS)* algorithm to determine a set of switches to be polled and sampled. Then, we deploy sFlow agents to query those chosen switches. Note that sFlow has two components; namely, sFlow-controller and sFlow-agent. The former one is deployed in the ReMon controller, and the latter one is in the switches. Once central sFlow collector receives statistics, it immediately reports to the sFlow controller. We use sFlow-RT as our sFlow controller. The sFlow controller then reports network status to the SDN controller or application layer through RESTful API so that the network administrators can view it.

4.2 Link Failure Recovery Algorithms

In this section, we present two algorithms *Anchor Assisted Recovery (AAR)* and *Weight Assisted Recovery (WAR)*. These two algorithms can be used to recover from

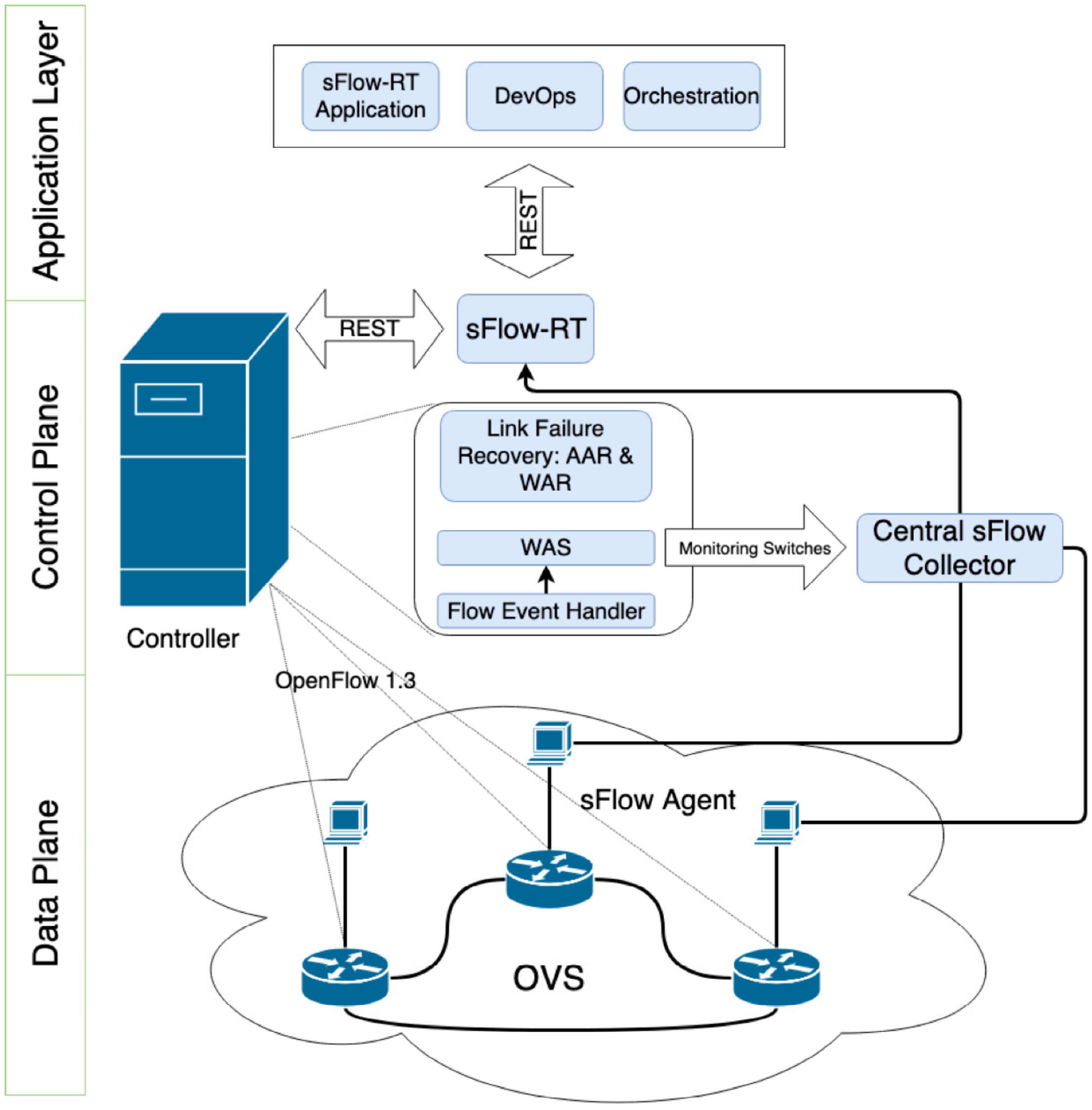


Figure 4.1: ReMon architecture.

a link failure if a switch does not have an alternative path towards a destination i.e., FFG does not work. These algorithms are hybrid as they combine both reactive and proactive recovery schemes. A switch reports a link failure to the controller after detecting it and the controller then proactively updates all routes affected by the failed link.

4.2.1 Anchor Assisted Recovery (AAR)

In AAR (Algorithm 3), an *anchor node* is a switch that has more than two neighbors. A switch with more neighbors more likely have alternative routes towards the destination. A switch will inform the controller if it cannot recover from a failure using FFG upon detecting a link failure. After that, the controller will recompute alternative routes only at anchor nodes and install corresponding flow entries. The anchor nodes can be configured during the network setup phase. At the same time controller can maintain a list of anchor nodes for each flow.

Algorithm 3 Anchor Assisted Recovery (AAR)

```
Input:
    Failure path: $P = (src, ..., s1, s2, ..., dst)$
    Anchor list of $(src, dst)$: $anchor\_list$
Output:
    Alternative path: $P$
    Affected switch: $switch$
1: for each $switch \in P[src : s1]$ do
2:     if $switch \in anchor\_list$ then
3:         if There is a new path $path_{new}$ to $dst$ then
4:             $path_{old} \leftarrow P[src : switch]$
5:             $P \leftarrow path_{old} + path_{new}$
6:             return $P, switch$
7:         end if
8:     end if
9: end for
```

For example, in Figure 2.8, the anchor node is C along the route $Source - C - D -$

$E - Destination$. Suppose link seven between E and $Destination$ fails, the affected packets will take an alternative route from the anchor node C. Thus, the new route will be $Source - C - B - Destination$ instead of $Source - C - D - E - D - C - B - Destination$ in the case of Crankback. Thus, in contrast to Crankback, using AAR a controller can insert new flow entries right after receiving a link failure event instead of waiting for the affected packets to be backtracked to an anchor node. The packets between the anchor and failed link still can use Crankback.

The computation complexity of AAR is $\mathcal{O}(n)$ for one route. Since a link failure may cause more than one path failure, the total complexity is $\mathcal{O}(n^3)$ because in the worst case, there can be $\mathcal{O}(n^2)$ source-destination pairs. The computation complexity of AAR and Crankback are same, except that the former one reduces the recovery time by exploiting the global network view and reactive alternative path computation.

4.2.2 Weight Assisted Recovery (WAR)

In WAR (Algorithm 4), we assign a weight to a switch based on its number of neighbors. Similar to AAR, the controller will recompute the alternative routes only at nodes with the highest weight. The controller can also assign weight during the network setup phase. Thus, after detecting a link failure, the controller can choose a set of switches with the highest weight to redirect the affected traffic through these chosen switches.

For example, in Figure 2.8, switch C and B have weight three, whereas others have weight two. Thus, in the case of a link failure, say link seven again, the affected packet will take the alternative route from C. Thus, AAR and WAR differ in terms of their switch selection approach. The computation complexity of WAR is same as AAR. In AAR and WAR, the idea is to take the help of the controller to find alternative routes for all possible flows affected by a single link failure instead of just relying on FFG or Crankback.

4.3 Node Failure Recovery

In this section, we present an algorithm (Algorithm 5) *Node Recovery with Destination (NRD)*, which recovers from node failure. Instead of checking all flow entries and updating all affected switches' flow table, NRD checks only a subset of flow entries.

Algorithm 4 Weight Assisted Recovery (WAR)

```
Input:
    Failure path: $P = (src, ..., s1, s2, ..., dst)$
    Weight list: $Weight = (w_1, w_2, ..., w_n)$
Output:
    Alternative path: $P$
    Affected switch:$switch$
 1: Update $Weight$
 2: Label all $switch \in P$ as $unchecked$
 3: while $\exists$ $unchecked$ $switch \in P[src : s1]$ do
 4:     find a $switch$ such that $Weight[switch]$ is maximum and $unchecked$
 5:     if There is a new path $path_{new}$ to $dst$ then
 6:         $path_{old} \leftarrow P[src : switch]$
 7:         $P \leftarrow path_{old} + path_{new}$
 8:         return $P, switch$
 9:     end if
10:     Label $switch$ as $checked$
11: end while
```

When there is a node failure, NRD first goes through dictionary P and gets all affected flow entries where the failed node is the source or destination. For each affected path, we consider its destination instead of both source and destination. After that we compute the new path using AAR or WAR. For this new path, we update a flow entry at a switch one-hop away from the failed node instead of updating at all previous switches.

In topology (shown in Figure 4.2), suppose switch B fails. All affected paths are: $f_1 : Source - Destination$, $f_2 : A - Destination$, $f_3 : C - Destination$, $f_4 : A - E$ and their reverse path. After failure recovery, the new flow entries are shown in Figure 4.3. Under the naive method, we have to add one flow entry at switch $Source, C, D$, and E for $Source - Destination$ pair. And for route $A - Destination$, we also need to insert new flow entry at switch $A, Source, C, D, E$. With the same idea, pair $C - Destination$ has to update flow entries at three switches and $A - E$ has to update four switches. We also need to consider the reverse route, so the updating flow entries

Algorithm 5 Node Recovery with Destination (NRD)

```
Input:
    Failure node: $Node$
    Affected path: $Path = (src, ..., Node, ..., dst)$
1:  Checking list: $Checking \leftarrow []$
2:  Remove corresponding flow entries at $sw \in Path$
3:  if $Node \in path$ then
4:      if $Node \neq src/dst$ then
5:          $sw_{prev} = Path[index(Node) - 1]$
6:          $path_{new}$ = AAR/WAR($(sw_{prev}, dst)$)
7:          while $sw \in path_{new}$ do $(src, dst)$
8:              if $(sw, dst) \in Checking$ then
9:                  Update flow entry at $sw \in path_{new}$
10:                 $Checking$.append($(sw, dst)$)
11:             end if
12:         end while
13:     end if
14: end if
```

should be doubled. As a result, the total cost is 32 added flow entries.

However, in NRD we only consider the destination. For pair $Source - Destination$, we have to add one flow entry at switch $Source, C, D$ and E which is same as above. But for pair $A - Destination$, we only need to add one new flow entry at switch A because $Source, C, D$, and E already have a flow entry to $Destination$, which was installed when updating pair $Source - Destination$. With the same idea, pair $C - Destination$ does not need to add any flow entries and $A - E$ need to add four flow entries. So the total cost is 18 adding flow entries. Such saving will increase when the size of the network increases because we will have more source-destination pairs.

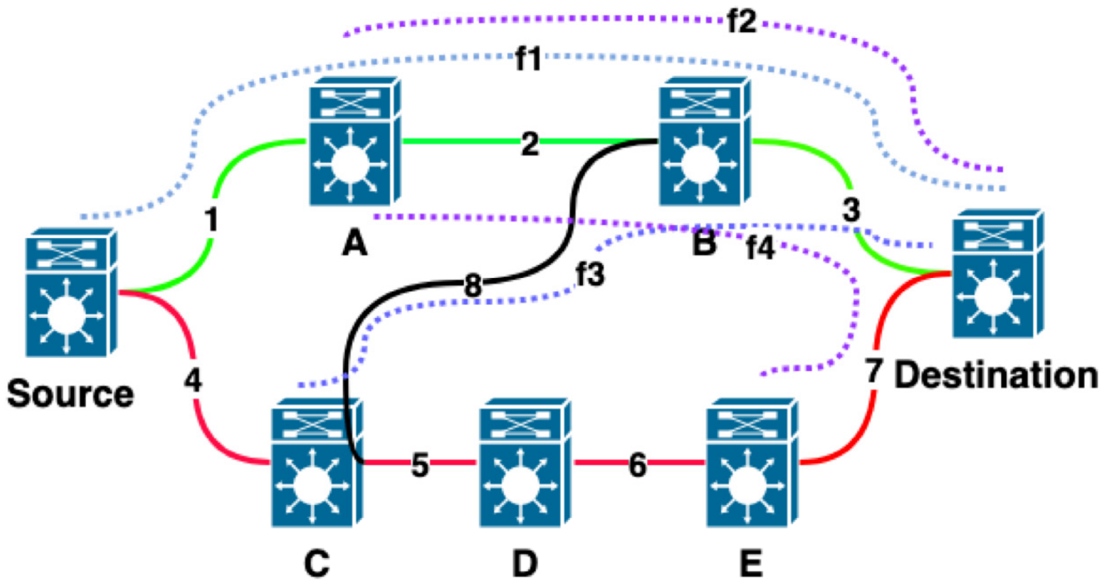


Figure 4.2: The node failure recovery example.

4.4 Updating WAS after link failure

The monitoring list needs to be updated after recovering from a link failure because we install new flow entries using AAR/WAR/NRD. In this section, we present an algorithm (Algorithm 6) to update WAS that dynamically update monitoring list rather than rerunning the WAS algorithm.

When we install a new flow entry upon detecting a link failure, this flow entry may need a new switch to monitor. Some switches, on the other hand, may no longer need to be monitored because of removing the failed flow entries. As a consequence, a link failure impacts the monitoring lists of WAS. Thus, we define Algorithm 6 to update WAS after installing flow entries.

Let us consider Figure 4.2. Suppose link 2 between A and B fails, which will impact all flow entries. We first compute new routes for the affected flows using AAR or WAR. The new routes are shown in Figure 4.3, except f_3. f_3 still use original route. Thus, we need to update the list of switches to be monitored. Suppose, the monitoring switches are $\{Destination, B, D\}$. Note that f_1, f_2, and $f3$ are covered by the switch $Destination$, $f4$ is covered by the switch D. Therefore, the current monitoring list remains unchanged. However, we furthermore need to check the endpoints of the failed link. If an endpoint is in the current monitoring list, we need to check whether

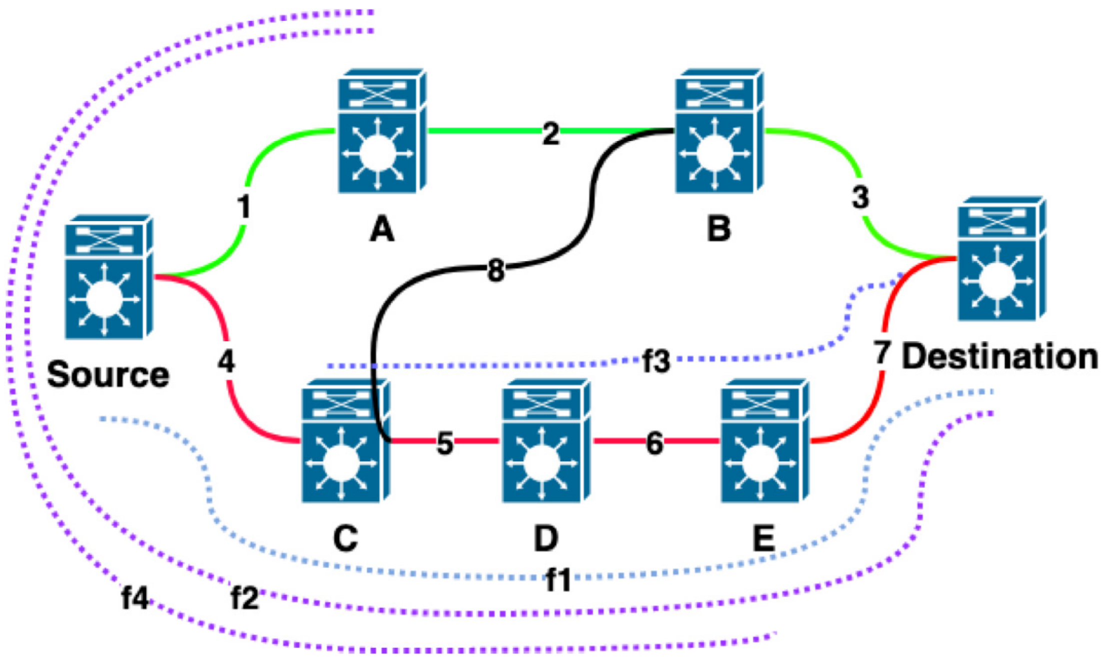


Figure 4.3: The node failure recovery example after recovery.

we need to update it. For instance, B is in the list, which only monitor f_3. However, f_3 also be monitored by *Destination*. Thus, B can be removed from the existing monitoring list.

4.5 Evaluation Setup

In this section, we illustrate the evaluation setup that we use to evaluate ReMon. We first evaluate the performance of AAR/WAR and NRD and compare the outcome with Crankback and reactive approach.

To evaluate the performance of our link failure recovery algorithms, we first measure the performance of single-link failures. We use the same USNET and DFN topologies as in Chapter 3 and iperf to generate UDP traffic at a rate of 1 Mbps between ten source-destination pairs. In Fig. 3.1, the green and red nodes are the source set and destination set, respectively. We randomly choose one from each set as a source-destination pair and repeat the process ten times. We first randomly fail one link from each pair and record their average recovery time. Then, we randomly fail five links from both the primary and the backup routes to get end-to-end delay and throughput compared to Crankback. At last, we evaluate how the structure of

Algorithm 6 Updating WAS

```
Input:
    Failure link: $Link_{fail} = (fw_1, fw_2)$
    Monitoring List: $M = (s_1, s_2, ..., s_n)$
Output:
    Updating Monitoring List: $M$
 1: while $\exists$ $path_{fail}$ contains $Link_{fail}$ do
 2:     $path_{new}$ = AAR/WAR($path_{fail}$)
 3:     if $\exists$ $switch \in path_{new}$ is monitored then
 4:         pass
 5:     else
 6:         $M$.append($path_{new}[destination]$)
 7:     end if
 8: end while
 9: if $\exists$ $flow \in fw_1$ or $fw_2$ and monitored only by $fw_1$ or $fw_2$ then
10:     pass
11: else
12:     remove $fw_1$ or $fw_2$ from $M$
13: end if
```

topology affects performance of link failure recovery. We randomly fail several links for each topology and record number of monitoring switches determined by WAS.

To test the performance of multiple-link and node failure recovery, we keep using ten testing pairs and choose failed nodes with the most number of neighbors. We keep failing links one by one for each node until all links; thus, the associated node is disconnected or failed. We record the failure recovery time, the memory usage in terms of the number of flow entries, and the operation cost in terms of the total number of added/removed flow entries. To enable network reachability, we increase the number of FFG rules , which equals to the number of neighbors instead of just two. Thus, we can recover from multiple-link failure using FFG. However, in ReMon we two FFG entries are enough.

4.6 Discussion on the Evaluation Results

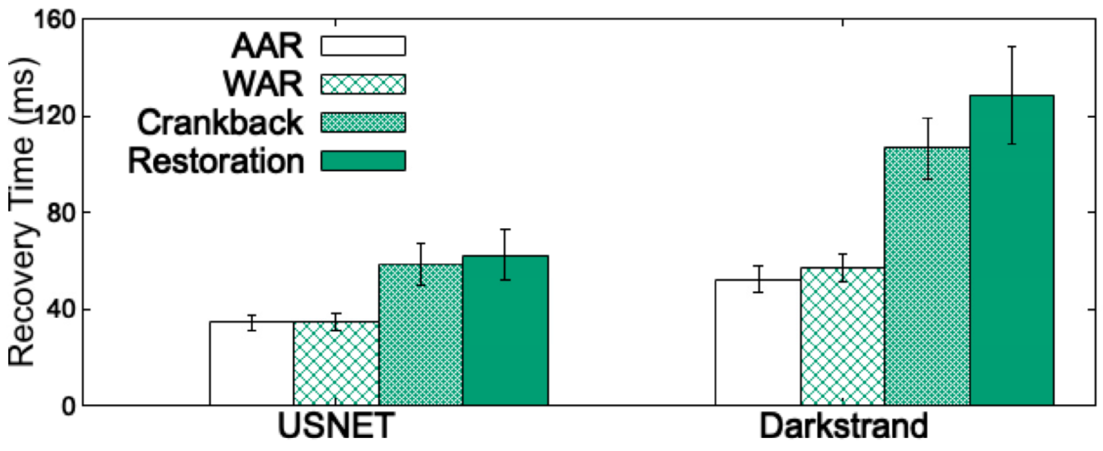


Figure 4.4: The total link failure recovery time in USNET and Darkstrand.

4.6.1 Performance of Single-Link Failure Recovery

We first evaluate the single-link failure recovery time of four methods: AAR, WAR, restoration, and Crankback both in USNET and Darkstrand. Since a single-link failure may impact multiple routes, the recovery time here is the total link failure recovery time is the time to update all affected routes. Figure 4.4 represents such total recovery time. In USNET, the recovery time of AAR and WAR is around 34 milliseconds. On the other hand, the restoration approach, which just recompute alternative route from controller, takes about 62 milliseconds, and the Crankback also takes the longer time. We observe similar performance trend in Darkstrand topology. Thus, AAR and WAR reduce more than 50% and 40% of the recovery time compared to the restoration and Crankback approaches, respectively [31].

Next, figrue 4.5 presents the memory usage of ReMon and Crankback approach. We use the total number of flow entries installed at switches to represent memory usage. Crankback needs to install flow entries for both primary and backup routes, and also for the backtracking rules. However, we just need primary routes pre-installed and insert backup routes if detecting failure. Thus, the memory usage of Crankback is significantly higher than that of ReMon [31].

Then, we compare the performance of AAR and WAR. We use the total number of switches checked in each algorithm to represent the failure recovery overhead. The

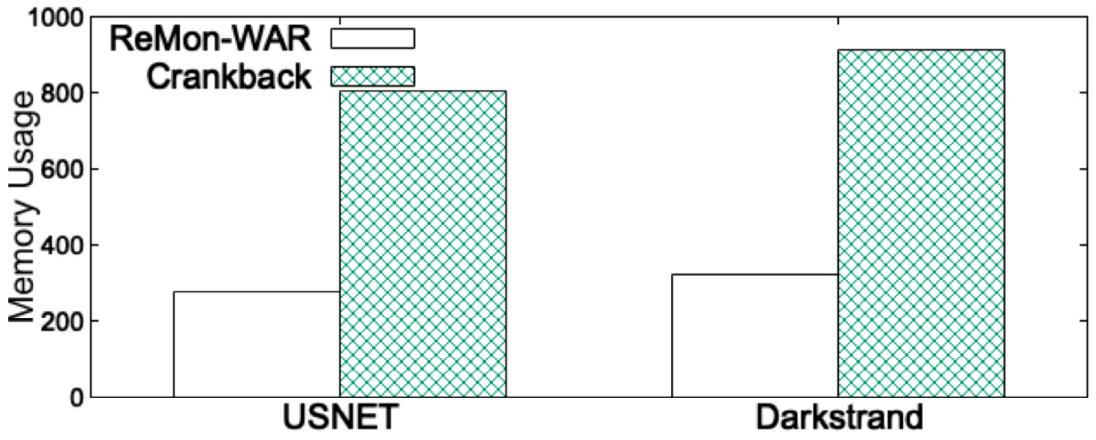


Figure 4.5: The total number of flow entries used in USNET and Darkstrand [31].

result is shown in Figure 4.6, where we show the results for Darkstrand topology because the performance is similar in USNET. The number of computation for WAR is significantly lower than AAR in Darkstrand. The reason is the structure of the topology. WAR only checks switches with higher weights, and those switches more likely have alternative routes towards a destination. However, an anchor node in AAR is a switch having more than two neighbors. Thus, AAR will need more time to find the set of anchor nodes. Therefore, we conclude that if a topology with large number of links, AAR and WAR will perform similarly. However, if the topology is a less-links and more-switches one, like Darkstrand, WAR will be useful [31].

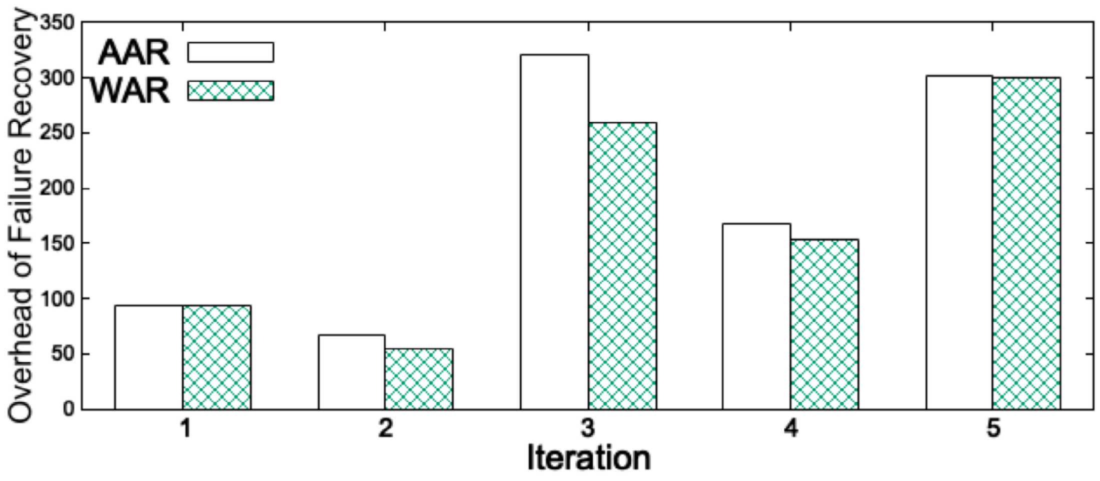
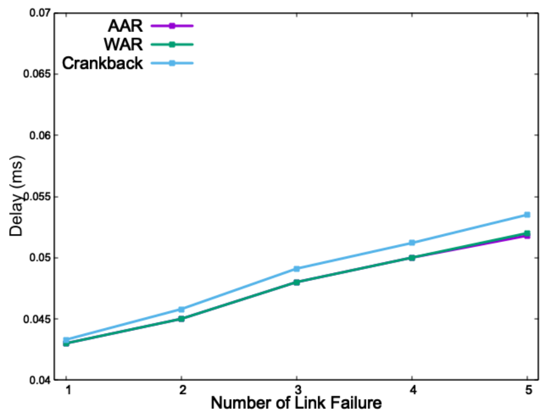
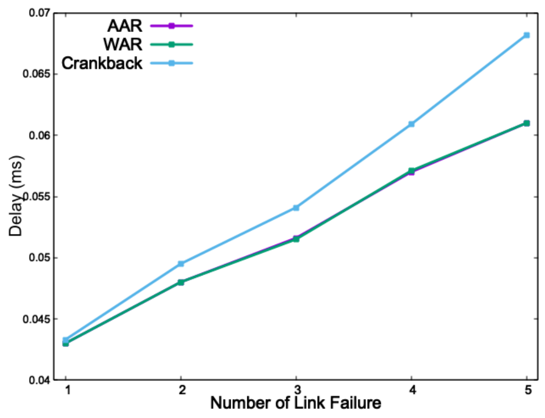


Figure 4.6: The performance comparison of AAR and WAR in Darkstrand [31].

AAR and WAR not only improve the resiliency but also improves the overall



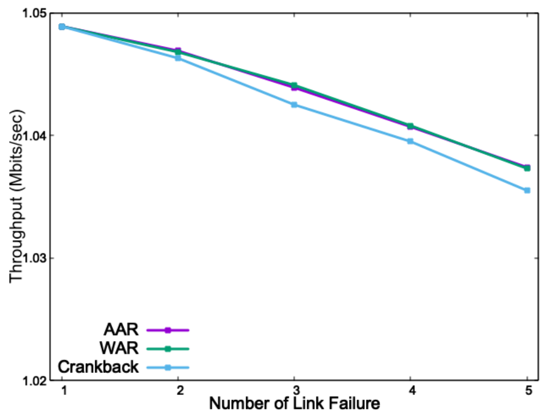
(a) USNET



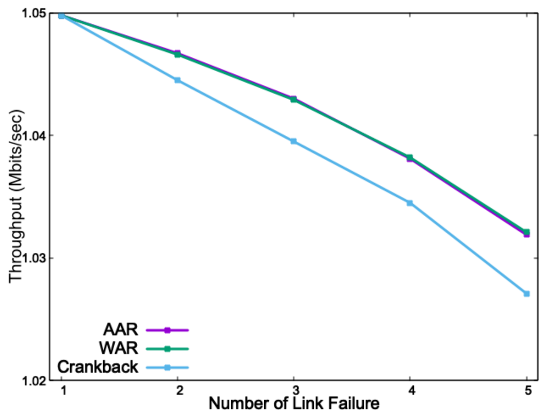
(b) Darkstrand

Figure 4.7: The average delay of different failure recovery schemes [31].

network throughput and delay. In figure 4.7, we present the average delay that AAR, WAR, and Crankback in USNET and Darkstrand topologies. Crankback has the highest delay in both topologies because it takes extra delay on backtrack process. Besides, all the affected switches will still use this backtracking, which is not the case in AAR and WAR. AAR and WAR just reconstruct all possible affected routes for given link failure. We observe similar results in the case of throughput, which is shown in Figure 4.8 [31].



(a) USNET



(b) Darkstrand

Figure 4.8: The average throughput of different failure recovery schemes.

Next, we evaluate how structure of the topology affects the performance of link failure recovery. Figure 4.9 shows the change in the total number of monitoring switches in USNET with links failure while using ReMon. We first randomly fail several links from around the perimeter of the USNET topology. Thus, the affected flows are likely redirected towards the center. Thus, multiple flows shared common routes and a common switch. We can just monitor this common switch and cover more flow entries than before. Hence, we see a decrease in the number of monitoring switches. Then, we start failing link also from the center of the topology. The flows then separate to different routes. It leads to an increase in the number of monitoring switches as flows are spreading out [31].

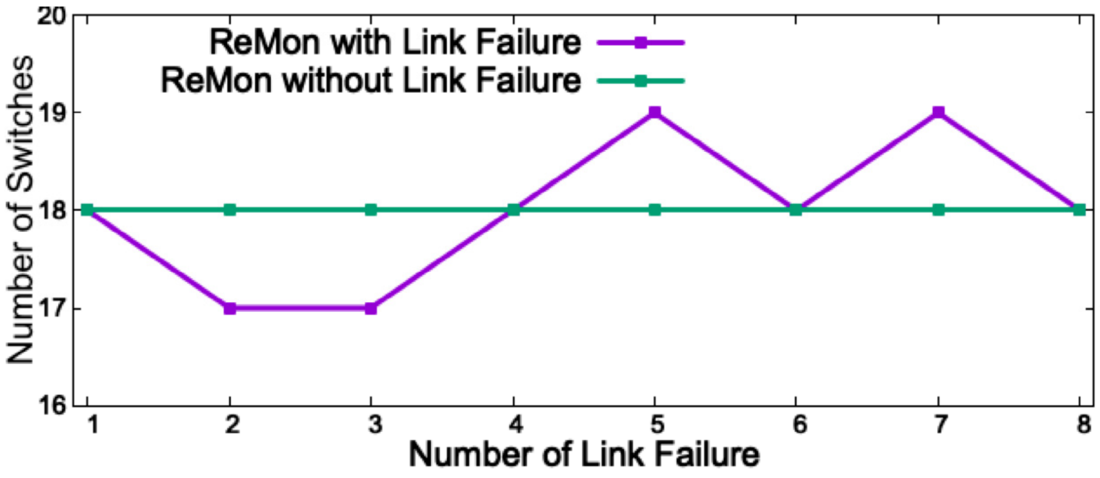


Figure 4.9: The link failure impact on the measurement cost in USNET.

Next, we evaluate the performance in Darkstrand, which is shown in Figure 4.10. Since we have to keep network connectivity, we just fail less number of links in Darkstrand compared to the USNET topology. The number of monitoring switches always decreases with the increasing number of link failures. The reason is, again, the structure of the Darkstrand topology. Darkstrand is a less-links and more-switches topology. Thus, with more link failure, the flows are more likely to be grouped to common routes and common switches. In that case, we can just monitor a less number of switches and cover more flow entries than before. Therefore, we conclude that the number of switches to be monitored will depend on the structure of a topology [31].

Recall that in USNET topology switches have alternative routes to any destination; thus, we can just use FFG to recover from a link failure. However, in Darkstrand due to the structure of topology, we have to follow Crankback approach, which may lead to more memory usage. On the other hand, the usage of link is much higher in Darkstrand than such in USNET. With more links fail, the usage of link will increase

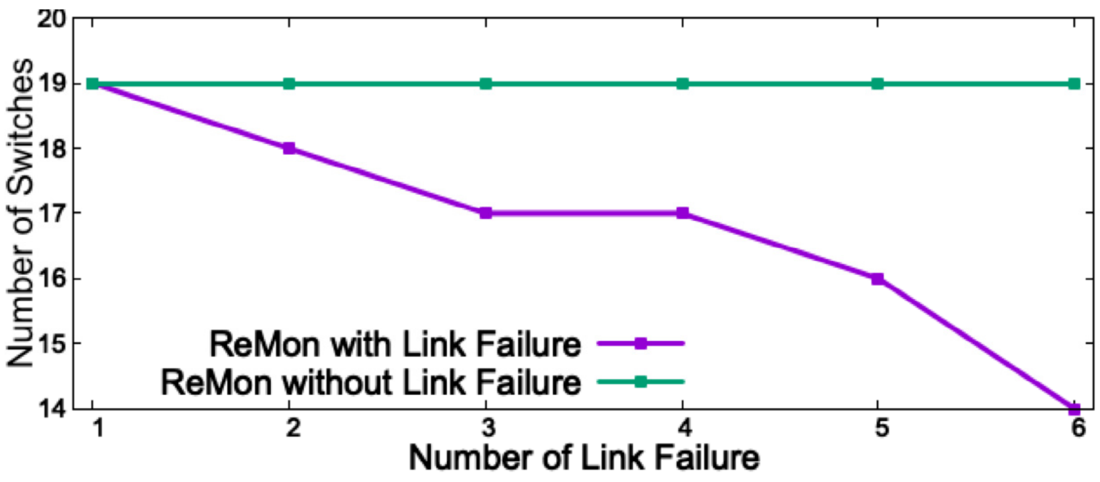


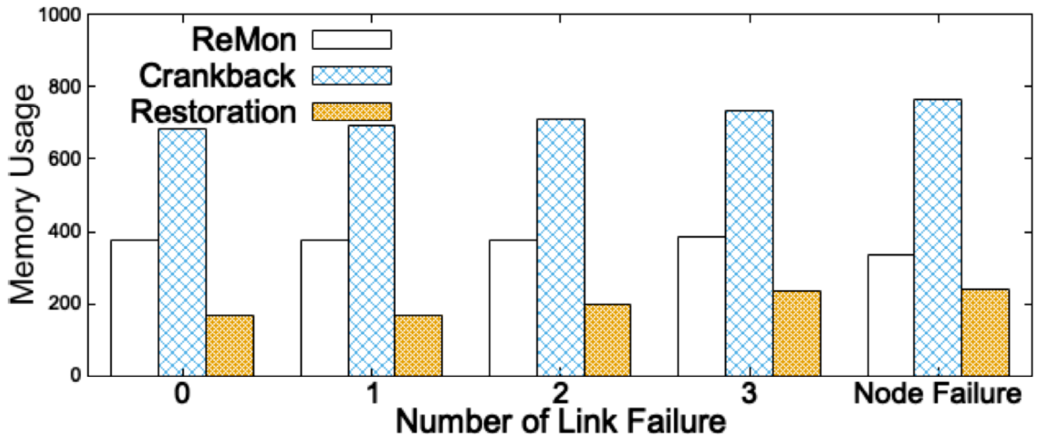
Figure 4.10: The link failure impact on measurement cost in Darkstrand [31].

rapidly. However, we assume each link has very high capacity; so it is not an issue in our evaluation. We will consider such capacity in the future when we deploy our works in real testbed.

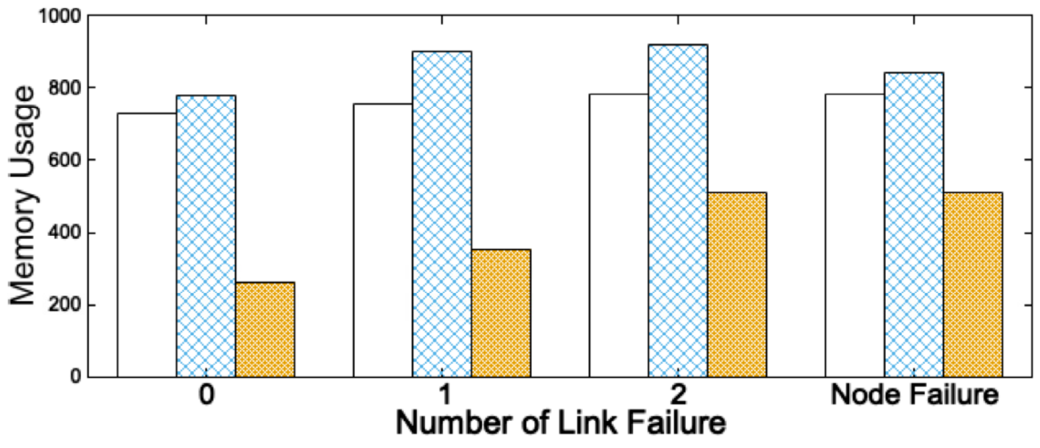
4.6.2 Performance of Multiple-Link and Node Failure Recovery

In this section, we present the performance of ReMon in the presence of multiple-link and node failures. Figure 4.11 presents the memory usage in different topology with multiple links failure. In USNET topology, Crankback costs too much memory because it needs to insert group entries to all possible action buckets to recover from multiple links failures. On the other hand, restoration takes much less memory because it only needs one flow entry. Our ReMon's cost is in between Crankback and reactive schemes. However, in Darkstrand topology, restoration has the worst memory usage with large number of links failure because it has to insert flow entries to almost every switches after a link failure. All three methods also show that the memory usage increases with the increasing number of link failure as we need to add new flow entries at backup switches. We also observe that the memory usage decreases for Crankback in Darkstrand in the case of node failure because such node failure can release a large number of flows.

More specifically, Figure 4.12 shows the operation cost with links failure. Restoration always has the highest operation cost because it needs to add new alternative flow entries and remove old one. However, it is not the case for ReMon and Crankback



(a) USNET

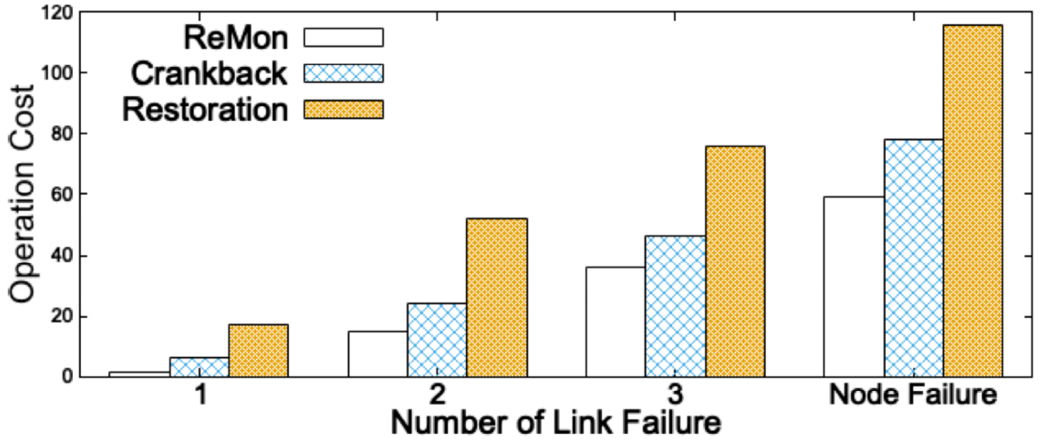


(b) Darkstrand

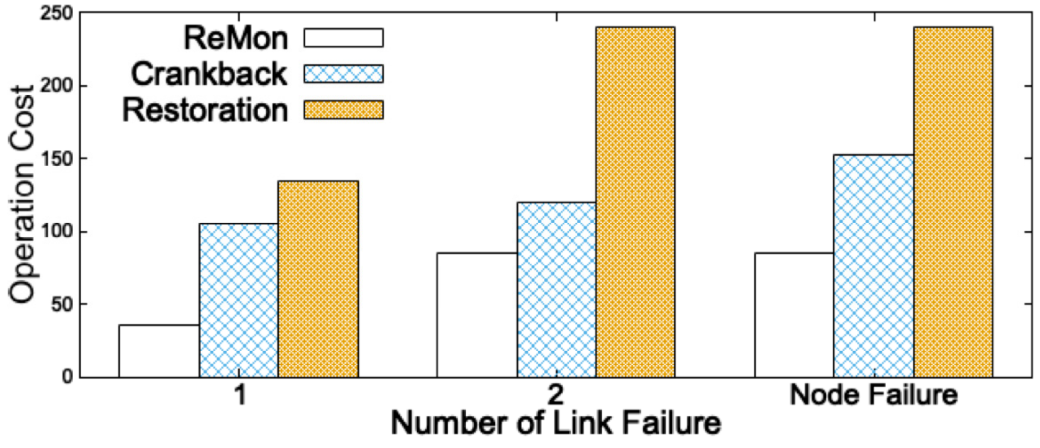
Figure 4.11: The memory usage with multiple-link and node failure.

because they do not need to remove old flow entries. Those old flow entries are FFG rules used to recover. The situation gets worse in Darkstrand topology because of its structure.

In Figure 4.13, the recovery time for node failure is least if using Crankback, which gives a similar result when measuring single link failure (shown in Figure 4.4). Restoration always takes the longest time to recover and Crankback takes a little bit higher than ReMon because of extra delay for Crankback's backtracking.



(a) USNET



(b) Darkstrand

Figure 4.12: The operation cost with multiple-link and node failure.

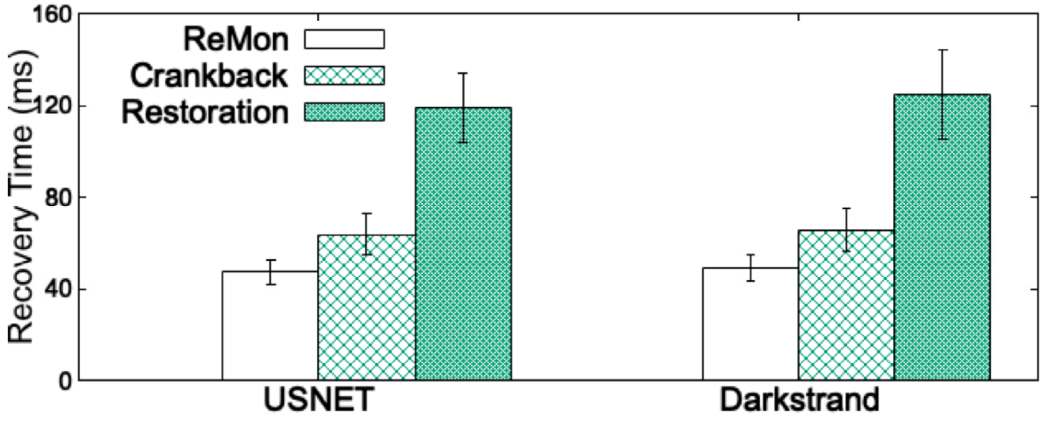


Figure 4.13: The average node failure recovery time in USNET and Darkstrand.

Chapter 5

Conclusions

This thesis presents a low-cost and resilient monitoring framework in SDN. Section 5.1 reviews a summary of this thesis. Section 5.2 discusses the conclusions of this thesis. At last, future work is presented in Section 5.3.

5.1 Thesis Summary

In Chapter 1, we described the importance of measurement and monitoring. We briefly described the difference between the traditional and the SDN network architecture. We highlighted the drawbacks of traditional network architecture and how SDN overcomes them. Next, we introduced two types of network measurement in SDN and analyzed their pros and cons. Then, we presented the importance of polling or sampling frequency and two types of failure recovery approaches. At last, we introduced our objectives and highlighted our primary contributions.

In Chapter 2, we first provided the background needed to understand this thesis. We described the SDN architecture and introduced the OpenFlow switch and protocol. Then, we described active and passive measurement schemes. We presented three prior work for each type. Next, we described sFlow (a hybrid scheme that combines both active and passive schemes) and explained the advantages of sFlow. Then, we introduced a low-cost measurement algorithm and explained the benefit of using such an algorithm. We furthermore presented the importance of adjusting the polling frequency/sampling rate. We explained four prior work related to the sampling rate. Finally, we presented the fact that failure can impact the accuracy of the measurement. We further introduced two types of failure recovery approaches in SDN.

Chapter 3 presented our low-cost measurement (WAS) and polling frequency/sampling rate selection (APS) algorithms. We introduced WAS algorithm with an example. Next, we introduced our optimization model and a heuristic for both the offline and online versions. In addition, we described how we implement our algorithms and corresponding experimental setup. At last, we presented the performance of our algorithms. We evaluated the accuracy and measurement cost of WAS and APS.

Chapter 4 presented our failure recovery model, called ReMon, which consists of a set of algorithms. We first described the architecture of ReMon. Next, we presented two link failure recovery algorithms: AAR and WAR and a node failure recovery algorithm, NRD. Then, we introduced the algorithm that updates the monitoring list once recover from a failure. Finally, we presented the performance of ReMon. We evaluated the recovery time, memory usage, end-to-end delay and throughput of ReMon. We compared the memory usage and operation cost of NRD with other approaches.

5.2 Conclusions

In this thesis, we illustrated the importance of low-cost and resilient monitoring framework. We proposed several algorithms to achieve these two goals.

The experimental evaluation showed that the proposed low-cost measurement algorithm (WAS) can reduce 30% measurement cost compared to the baseline approach and significantly lower the computation cost compared to FlowCover. We further noticed that the accuracy of WAS is better than FlowCover because we considered destination switches while monitor flows. The evaluation indicated that our polling frequency/sampling rate solution (APS) can further reduce 50% measurement cost and maintained the same level of accuracy compared to Sampling-On-Demand (SOD).

Finally, our experimental evaluation on failure recovery model (ReMon) showed that ReMon had better performance that Crankback and reactive method in terms of recovery time, memory usage, and network status for single-link failure recovery. ReMon saved 50% recovery time, 30% memory usage and 60% operation cost compared to its counterparts. ReMon also presented a better network status in terms of end-to-end delay and throughput compared to Crankback. We also indicated how the structure of topology can impact the performance of failure recovery. At last, we showed the performance of ReMon in multi-link and node failure recovery. The results showed that ReMon saved 30% memory usage compared to Crankback and 50% of operation cost compared to the reactive approach.

5.3 Future Work

In this work, we focus on designing and implementing a low-cost and resilient monitoring framework in a virtual mininet environment. However, we have several avenues for future research. Below are a few recommendations for future research directions:

- **Real Testbed:** In future, we will focus on the implementation of our work in a real testbed. We can prepare several CISCO Nexus 9000 Series Switches as our physical switches. We will test the performance of our work in physical devices and compare them with the Mininet performance.

- **P4 Based Measurement:** We will focus on the implementation of our work into a P4-based environment. Programming Protocol-independent Packet Processors (P4) [6] is a new programming language in SDN, which was first introduced in 2013. It enables both control plane and data plane programmability while standard SDN only allows control plane programmability. Thus, P4 is a *target-independent* and *protocol-independent* protocol while OpenFlow is just a *target-independent* but not *protocol-independent*.

  - **Target-independent** means the programmer can describe packet processing functionality independently, e.g. OpenFlow allows the programmer to define the action of each flow as long as the switch supports OpenFlow.

  - **Protocol-independent** means devices should not be tied to any specific network protocols. Obviously, OpenFlow does not fit this because OpenFlow has to define the flows based on an existing protocol, e.g. OpenFlow has to define the matching field for each flow entry, including IP address, MAC address, etc. On the other hand, P4 enables a programmer to customize matching fields and actions.

  There are few prior works that implemented monitoring framework using P4. UnivMon [21] presents a flow monitoring framework in P4 environment. The author design a high accurate flow-based monitoring framework. INT [37] also propose a P4-based monitoring system for the ONOS controller. The authors design a packet-level monitoring framework with a graphical user interface. Clearly, both works just design a solution for monitoring. They do not consider cost and resiliency. Thus, it can be a challenge to implement all our algorithms into P4 environment.

Bibliography

[1] CPLEX Optimizer. https://www.ibm.com/analytics/cplex-optimizer.

[2] Darkstrand Topology. http://www.topology-zoo.org/maps/Darkstrand.jpg.

[3] iperf - the ultimate speed test tool for tcp, udp and sctp. https://iperf.fr/.

[4] Mininet. http://mininet.org/.

[5] Open vSwitch. https://www.openvswitch.org/.

[6] P4 language consortium. https://p4.org/.

[7] Ryu SDN Framework. https://osrg.github.io/ryu/.

[8] sFlow-RT. https://sflow-rt.com/index.php.

[9] USNET topology. https://www.researchgate.net/figure/Network-topologies-aNSFNET-bUSNET_fig3_321952636.

[10] Zdravko Bozakov, Amr Rizk, Divyashri Bhat, and Michael Zink. Measurement-based flow characterization in centrally controlled networks. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. IEEE, July 2016.

[11] Martn Casado, Michael J. Freedman, Justin Pettit, Jiangying Luo, Natasha Gude, Nick McKeown, and Scott Shenker. Rethinking enterprise network control. *IEEE/ACM Transactions on Networking (TON)*, 17(4):1270–1283, August 2009.

[12] Guang Cheng and Yongning Tang. eopenflow: Software defined sampling via a highly adoptable openflow extension. In *2017 IEEE International Conference on Communications (ICC)*. IEEE, May 2017.

[13] Shihabur Rahman Chowdhury, Md. Faizul Bari, Reaz Ahmed, and Raouf Boutaba. Payless: A low cost network monitoring framework for software defined networks. In *2014 IEEE Network Operations and Management Symposium (NOMS)*. IEEE, June 2014.

[14] Reuven Cohen and Guy Grebla. Joint scheduling and fast cell selection in ofdma wireless networks. *IEEE/ACM Transactions on Networking*, 23(1):114–125, February 2015.

[15] Reuven Cohen and Evgeny Moroshko. Sampling-on-demand in sdn. *IEEE/ACM Transactions on Networking*, 26(6):2612–2622, December 2018.

[16] Paulo Csar da Rocha Fonseca and Edjard Souza Mota. A survey on fault management in software-defined networks. *IEEE Communications Surveys & Tutorials*, 19(4):2284–2321, June 2017.

[17] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to sdn: an intellectual history of programmable networks. *ACM SIGCOMM Computer Communication Review*, 44(2):87–98, April 2014.

[18] RhongHo Jang, DongGyu Cho, Youngtae Noh, and DaeHun Nyang. Rflow: An sdn-based wlan monitoring and management framework. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*. IEEE, May 2017.

[19] Diego Kreutz, F. M. V. Ramos, Paulo Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Reference : Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.

[20] Chang Liu, Mehdi Malboubi, and Chen-Nee Chuah. Openmeasure: Adaptive flow measurement & inference with online learning in sdn. In *IEEE INFOCOM 2016 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, April 2016.

[21] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *SIGCOMM '16 Proceedings of the 2016 ACM SIGCOMM Conference*, pages 101–114, August 2016.

[22] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, April 2008.

[23] Peter Megyesi, Alessio Botta, Gluseppe Aceto, Antonio Pescape, and Sandor Molnar. Available bandwidth measurement in software defined networks. In *31st ACM/SIGAPP Symposium on Applied Computing*. ACM, April 2016.

[24] Shicong Meng, Arun K. Lyengar, Isabelle M. Rouvellou, and Ling Liu. Volley: Violation likelihood based state monitoring for datacenters. In *2013 IEEE 33rd International Conference on Distributed Computing Systems*. IEEE, July 2013.

[25] Open Networking Foundation. Openflow switch specification, September 2012.

[26] Xuan Thien Phan and Kensuke Fukuda. Sdn-mon: Fine-grained traffic monitoring framework in software-defined networks. *Journal of Information Processing*, 25:182–190, February 2017.

[27] Marco Polverini, Andrea Baiocchi, Antonio Cianfrani, Alfonso Iacovazzi, and Marco Listanti. The power of sdn to improve the estimation of the isp traffic matrix through the flow spread concept. *IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS*, 34(6):1904–1913, June 2016.

[28] Zhiyang Su, Ting Wang, Yu Xia, and Mounir Hamdi. Flowcover: Low-cost flow monitoring scheme in software defined networks. In *2014 IEEE Global Communications Conference*. IEEE, December 2014.

[29] Zhiyang Su, Ting Wang, Yu Xia, and Mounir Hamdi. Cemon: A cost-effective flow monitoring system in software defined networks. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 92(1):101–115, December 2015.

[30] Junho Suh, Ted Taekyoung Kwon, Colin Dixon, Wes Felter, and John Carter. Opensample: A low-latency, sampling-based measurement platform for commodity sdn. In *2014 IEEE 34th International Conference on Distributed Computing Systems*. IEEE, July 2014.

[31] Fangye Tang and Israat Haque. Remon: A resilient flow monitoring framework. In *IEEE/IFIP TMA 2019*, June 2019.

[32] Amin Tootoonchian, Monia Ghobadi, and Yashar Ganjali. Opentm: Traffic matrix estimator for openflow networks. *PAM2013: Passive and Active Measurement*, 6032:201–210, 2010.

[33] Pang-Wei Tsai, Chun-Wei Tsai, Chia-Wei Hsu, and Chu-Sing Yang. Network monitoring in software-defined networking: A review. *IEEE Systems Journal*, 12(4):3958–3969, Dec 2018.

[34] Niels L. M. van Adrichem, Christian Doerr, and Fernando A. Kuipers. Opennetmon: Network monitoring in openflow software-defined networks. In *2014 IEEE Network Operations and Management Symposium (NOMS)*. IEEE, May 2014.

[35] Niels L. M. van Adrichem, Benjamin J. van Asten, and Fernando A. Kuipers. Fast recovery in software-defined networks. In *EWSDN '14 Proceedings of the 2014 Third European Workshop on Software Defined Networks*, September 2014.

[36] Niels L.M. van Adrichem, Benjamin J. van Asten, and Fernando A. Kuipers. Fast recovery in software-defined networks. In *2014 Third European Workshop on Software Defined Networks*. IEEE, September 2014.

[37] Nguyen Van Tu, Jonghwan Hyun, and James Won-Ki Hong. Towards onos-based sdn monitoring using in-band network telemetry. In *2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, September 2017.

[38] Hongli Xu, Xiang-Yang Li, Liusheng Huang, Yang Du, and Zichun Liu. Partial flow statistics collection for load-balanced routing in software defined networks. *Computer Networks*, 122(1):43–55, July 2017.

[39] Tian Yang, Weiwei Chen, and Chin-Tau Lea. An sdn-based traffic matrix estimation framework. *IEEE Transactions on Network and Service Management*, pages 1–1, August 2018.

[40] Ze Yang and Kwan L. Yeung. An efficient flow monitoring scheme for sdn networks. In *2017 IEEE 30th Canadian Conference on Electrical and Computer Engineering (CCECE)*. IEEE, April 2017.

[41] Curtis Yu, Cristian Lumezanu, Yueping Zhang, Vishal Singh, Guofei Jiang, and Harsha V. Madhyastha. Flowsense: Monitoring network utilization with zero measurement cost. *PAM2013: Passive and Active Measurement*, 7799:31–41, 2013.